

DirectMusic

This documentation covers the following Microsoft® DirectMusic® application programming interface (API) topics:

- About DirectMusic
- Why Use DirectMusic?
- DirectMusic Architecture
- DirectMusic Essentials
- DirectMusic Tutorials
- DirectMusic Reference
- DirectMusic Samples

About DirectMusic

DirectMusic is the musical component of the Microsoft® DirectX® API. Unlike the Microsoft® DirectSound® API, which is for the capture and playback of digital sound samples, DirectMusic works with message-based musical data, which is converted into wave samples, either in hardware or in a software synthesizer. The default software implementation uses the Microsoft® Software Synthesizer to create wave samples that are then streamed to DirectSound. Instrument voices are synthesized from samples according to the downloadable sounds (DLS) standard.

As well as supporting input in Musical Instrument Digital Interface (MIDI) format, DirectMusic can compose music at run time. This music is not algorithmically generated, but is based on elements authored by a human composer. (The authoring tool, Microsoft® DirectMusic® Producer, is documented separately.) The music is performed with variations and can respond dynamically to program events.

Like other components of DirectX, DirectMusic is based on the Component Object Model (COM).

DirectMusic delivers full functionality on Microsoft® Windows® 95, Microsoft® Windows® 98, and Microsoft® Windows® 2000. However, hardware acceleration is available only on Windows 2000 and Windows 98 Second Edition.

Why Use DirectMusic?

The DirectMusic API addresses fundamental requirements for delivering music on the Windows platform:

- Consistent playback experience. By using downloadable sounds, an application can be sure that musical instruments will sound the same on all computers and can perform with instruments of its own design.
- Jitter-free timing. Playback of MIDI-generated music has timing accuracy within two milliseconds.
- Extensibility. DirectMusic does not restrict vendors to a base-level feature set.
- Hardware acceleration. On Windows 98 Second Edition and Windows 2000, DirectMusic can stream its output to DLS-capable hardware synthesizers.
- Software emulation. Computers without hardware acceleration are fully DirectMusic-capable with minimum impact on performance.

In addition, DirectMusic provides important features for easing application development and for enriching the user's experience:

- Generic mechanism for loading and performing musical segments, regardless of the performance technology. DirectMusic supports standard MIDI files, authored music segments, and third-party technology equally.
- Multiple performances. More than one piece of music can be played at once, with completely separate timing, instrument sets, and so on.
- More than 16 MIDI channels. By mapping performance channels to channel groups, DirectMusic breaks through the 16-channel limitation and makes it possible for any number of voices to be played simultaneously, up to the limits of the synthesizer.
- Automated management of DLS instruments.
- Dynamic and interactive playback. In combination with DirectMusic Producer, the DirectMusic performance engine can be used to create dynamic musical soundtracks based on stored compositional material. The music does not assume its final form until it is about to be played and can respond to program events.
- Synchronization of all music playback through the use of a master clock.

DirectMusic Architecture

This section introduces the components of DirectMusic. The following topics are discussed:

- Core and Performance Layers
- Overview of DirectMusic Objects and Interfaces (C/C++)
- Overview of DirectMusic Classes (Visual Basic)
- Overview of DirectMusic Data Flow
- DirectMusic Messages
- Downloadable Sounds
- Microsoft Software Synthesizer

For information about how to implement these components in your application, see DirectMusic Essentials.

Core and Performance Layers

The DirectMusic core layer manages timing and ports and provides services for managing DLS collections. It supports buffered, time-stamped MIDI input and output. By default, DirectMusic itself sequences the MIDI data.

The core layer also includes the Microsoft Software Synthesizer, which functions as the hardware emulation layer for DLS support. If the user's computer does not have DLS hardware acceleration, the software synthesizer uses DLS to synthesize wave output from the sequenced MIDI data. Vendors can also create their own software synthesizers.

The DirectMusic performance layer is responsible for the higher-level aspects of music playback, including the loading and playback of MIDI files and the composition of music, based on elements authored in DirectMusic Producer or a similar application.

[C++]

The interfaces and related API elements for the core layer are declared in the Dmusicc.h header file, and those for the performance layer are in the Dmusicl.h header file.

[Visual Basic]

DirectX for Visual Basic exposes primarily the performance layer. For specialized applications that need to work with multiple ports, manipulate DLS data at a low level, or create their own data tracks, use C++.

Overview of DirectMusic Objects and Interfaces

[Visual Basic]

The information in this section pertains only to applications written in C++. See Overview of DirectMusic Classes.

[C++]

In DirectMusic, it is helpful to make a distinction between objects and their COM interfaces because many objects have multiple interfaces. The object that supports the

IDirectMusicCollection interface, for example, also supports the standard COM **IPersistStream** interface, as well as the **IDirectMusicObject** interface.

In this documentation, DirectMusic objects are referred to by the name of their principal or unique interface, but without the initial *I*; thus the object represented by **IDirectMusicCollection** is called the DirectMusicCollection object. Objects are also referred to by short names, such as *collection*, *performance*, *segment*, and *track*.

Interface pointers are often used as pointers to their objects so that these objects can be accessed through the methods of other interfaces. In fact, one interface, **IDirectMusicDownloadedInstrument**, has no unique methods of its own and is used only as a parameter to the methods of other interfaces.

DirectMusic consists of many COM objects and interfaces that are related to one another in rather complex ways. However, they can be divided into categories, according to their broad functionality, as follows:

- Core Objects and Interfaces
 - Loader Objects and Interfaces
 - Instrument Objects and Interfaces
 - Tool Objects and Interfaces
 - Performance Objects and Interfaces
 - Composition Objects and Interfaces
 - Synthesizer Objects and Interfaces
-

Core Objects and Interfaces

[\[Visual Basic\]](#)

The information in this topic pertains only to applications written in C++. See Overview of DirectMusic Classes.

[\[C++\]](#)

The core objects handle the basic needs of DirectMusic: input, output, and timing. These objects are usually managed by other objects, such as DirectMusicPerformance, and you might never need to use their interfaces directly.

DirectMusic Object

The DirectMusic object, represented by the **IDirectMusic** interface, is used for creating ports and buffers, for connecting with DirectSound, and for setting up a master clock. There should not be more than one instance of this object per application.

Many applications never need to use the **IDirectMusic** interface directly. In this respect, it differs from other DirectX base interfaces. For example, **IDirectSound** is the starting point for every DirectSound application, performing essential tasks such as setting the cooperative level and creating sound buffers. In DirectMusic, most initialization is handled by other objects, such as **DirectMusicPerformance**, and these objects are created independently by direct calls to **CoCreateInstance**.

Port

Each device that sends or receives music data is encapsulated in a **DirectMusicPort** object. The methods of the **IDirectMusicPort** interface allow direct manipulation of the port, but most applications do not need to use these methods because the port is managed by the performance. For example, you assign channels to a port through the **DirectMusicPerformance** object so that data in those channels is correctly routed.

Buffer

The **IDirectMusicBuffer** interface represents the data currently ready to be played by the port (or read from the port). Most applications do not deal directly with the buffer object, but methods are available to manipulate its contents directly, if necessary.

Thru

The **IDirectMusicThru** interface is used to set up direct transmission of data from a capture port to another port.

Reference Clock

Objects that implement the **IReferenceClock** interface represent the master clock that synchronizes all DirectMusic activity and the latency clock of a port.

Loader Objects and Interfaces

[\[Visual Basic\]](#)

The information in this topic pertains only to applications written in C++. See [Overview of DirectMusic Classes](#).

[\[C++\]](#)

Certain types of objects, such as **DirectMusicCollection** and **DirectMusicStyle**, have to be loaded (typically from a file) before they can be incorporated into a music performance. Others, such as **DirectMusicSegment**, can be either loaded or constructed at run time. The interfaces introduced in this section are essential for loading.

Loader

The **DirectMusicLoader** object, through its **IDirectMusicLoader** interface, manages the enumeration, caching, and loading of objects.

Stream

Data being read from a file or resource is represented by a stream object. Most applications do not have to deal directly with streams, which are created and managed by the loader. The stream object implements the following two interfaces:

- **IStream** streams the data from a file or resource and passes it to the object being loaded, which parses it by using its own implementation of **IPersistStream**.
- **IDirectMusicGetLoader** has a single method that returns a pointer to the **DirectMusicLoader** object that created the stream. It is used when a reference to another object is found in a stream and the object being loaded needs to call the loader to load the referred to object.

For a closer look at the use of these interfaces in the loading process, see **DirectMusicLoader**.

Object

Every object in **DirectMusic** that represents a file or resource supports the **IDirectMusicObject** interface, which is used as a generic pointer by the loader. When an application has obtained a pointer to this interface, the **IDirectMusicObject::QueryInterface** method can be used to obtain a pointer to the object's own unique interface, such as **IDirectMusicCollection** or **IDirectMusicStyle**. However, you can usually obtain the interface that you need from the call to **IDirectMusicLoader::GetObject**.

The methods of **IDirectMusicObject** are used internally by the loader for identifying objects.

Instrument Objects and Interfaces

[\[Visual Basic\]](#)

The information in this topic pertains only to applications written in C++. See **Overview of DirectMusic Classes**.

[\[C++\]](#)

An instrument is an object that represents a basic musical timbre or other sound. For all ports except legacy hardware MIDI ports, each instrument is associated with its own set of downloadable sounds (DLS), which must be downloaded to the port before the instrument can be used.

Downloadable sounds can be handled at three levels of abstraction. At the highest level, you load a band from a file and let the band object handle the DLS downloading for the instruments. At the next level, you directly access individual instruments in a collection and download them to a port. At the lowest level, you work with the DLS data itself.

The following objects and interfaces are used for managing instruments.

Collection

Instruments are stored in a `DirectMusicCollection` object, which represents an instance of a DLS file. Once the `DirectMusicCollection` object has been loaded, the **IDirectMusicCollection** interface can be used to enumerate instruments in the collection and to obtain a pointer to an instrument that has a given MIDI patch number.

Instrument

An instrument from a collection is represented by a pointer to the **IDirectMusicInstrument** interface. This pointer can be passed to the **IDirectMusicPerformance::DownloadInstrument** or the **IDirectMusicPort::DownloadInstrument** method to download DLS data to a port.

Once an instrument has been downloaded, it is represented by an **IDirectMusicDownloadedInstrument** interface pointer. This pointer is used only to unload the instrument by calling **IDirectMusicPort::UnloadInstrument**.

Applications that need to download their own DLS data for an instrument (such as collection-editing tools) use the methods of the **IDirectMusicPortDownload** interface (implemented by the port object) to get that data to the synthesizer. When this interface is used to allocate a buffer for instrument data, an **IDirectMusicDownload** interface pointer is returned. The single method of this interface can be used to obtain a pointer to the buffer itself.

Band

The `DirectMusicBand` object represents a set of instruments and MIDI program changes for a musical segment. The band is created in an authoring tool and can be loaded separately from a file, or it can be part of an authored segment or style.

The **IDirectMusicBand** interface can be used to download and unload bands. It also has a method for creating a secondary segment from a band. This segment can be played by the performance to effect program changes.

The key differences between a collection and a band are as follows:

- A collection is a group of instruments available for use in the playback of any segment. A band is a group of instruments that actually plays a particular segment.

- Instruments in a collection contain DLS data defining their timbre. Instruments in a band contain no such data but are linked to instruments in one or more collections.
 - An instrument from a collection is not inherently associated with any particular performance channel (PChannel) of a segment. A band assigns the patch number of an instrument to each PChannel in a segment and assigns a voice priority to the channel.
 - A collection does not contain any information about how the instruments are to be played. A band contains settings for the volume, pan, and transposition of each instrument.
-

Tool Objects and Interfaces

[\[Visual Basic\]](#)

The information in this topic pertains only to applications written in C++. See Overview of DirectMusic Classes.

[\[C++\]](#)

Tools are objects that intercept musical messages and process them before they are passed on to the port. All tools (except the output tool implemented by DirectMusic) are application-defined.

Tool

The **IDirectMusicTool** interface represents a single tool. The methods of this interface are implemented by the application or DLL to define the tool's functionality.

Graph

Tools are collected in a graph, represented by the **IDirectMusicGraph** interface, which is implemented by both the segment and the performance object. The interface is used for directing messages from one tool to the next, as well as for adding tools to the graph, retrieving pointers to individual tools, and shutting down the graph.

Performance Objects and Interfaces

[\[Visual Basic\]](#)

The information in this topic pertains only to applications written in C++. See Overview of DirectMusic Classes.

[C++]

The following objects and interfaces are used in the playback of musical data. With a total of over 75 methods, the interfaces in this group play a major role in any DirectMusic application.

Performance

The **DirectMusicPerformance** object is the overall manager of music playback. Through the **IDirectMusicPerformance** interface, it adds and removes ports, downloads instruments, attaches graphs (collections of tools), manages event notification for multiple segments, and plays segments.

Segment

A **DirectMusicSegment** object represents a chunk of data, most often a piece of music, contained in one or more tracks. Typically, a segment is either loaded from a file or created at run time by a **DirectMusicComposer** object. To play the segment, the application passes the **IDirectMusicSegment** interface pointer to the **IDirectMusicPerformance::PlaySegment** method.

Methods of the **IDirectMusicSegment** interface are used to manage timing and looping, event notification, tracks, tool graphs, and various other parameters of the segment.

An instance of a segment that is playing is represented by another interface, **IDirectMusicSegmentState**. Methods of this interface return information about the state of playback, and a pointer to this interface is used by the performance to stop or remove the segment instance.

Track

A chunk of timed data of a particular kind is represented by a **DirectMusicTrack** object, more simply referred to as a track. Methods of the **IDirectMusicTrack** interface can be used to set and retrieve data, play the data, and set notifications. Most applications do not use this interface directly because tracks are normally handled through the methods of the **DirectMusicSegment** object that contains them.

Note

A **DirectMusicTrack** is not the same thing as an instrument track. A **DirectMusicTrack** represents any kind of timed data, such as MIDI messages, a chord progression, or band changes.

Composition Objects and Interfaces

[Visual Basic]

The information in this topic pertains only to applications written in C++. See Overview of DirectMusic Classes.

[C++]

The objects and interfaces in this category are used in the real-time composition of music. Except for the composer itself, they represent data loaded from a file created in an application such as DirectMusic Producer. For a closer look at the role of each object, see Music Composition.

Composer

Methods of the **IDirectMusicComposer** interface allow an application to compose musical segments and transitions, using chord maps, styles, and templates created by a human author.

Style

Styles contain basic information about a piece of music, including note patterns. Styles often form part of authored segments, in which they do most of their work behind the scenes. They can also be used to compose entirely new segments at run time. Styles are represented by the **IDirectMusicStyle** interface.

Chord map

A DirectMusicChordMap object represents a collection of chords and pathways used by DirectMusicComposer in determining the chord progression in a piece of music.

The **IDirectMusicChordMap** interface is obtained for a DirectMusicChordMap object loaded from a file. A pointer to this interface is passed to the methods of **IDirectMusicComposer** so that a segment or transition can be composed at run time, using the authored chord map. You can also change the chord pattern of an existing segment by applying a new chord map.

Template

Templates are a special type of DirectMusicSegment object. They are never played directly, but are used by the DirectMusicComposer in the real-time construction of segments based on styles and chord maps.

Synthesizer Objects and Interfaces

[Visual Basic]

The information in this topic pertains only to applications written in C++. See Overview of DirectMusic Classes.

[C++]

The synthesizer is responsible for converting MIDI messages into waveform data and streaming this to the wave output device. Although DirectMusic comes with its own software synthesizer, it allows the implementation of custom synthesizers. It also allows output to be directed to different devices. Information on these topics is contained in the DirectX Driver Development Kit (DDK).

Synthesizer

A synthesizer implemented by an application is represented by the **IDirectMusicSynth** interface. Most applications do not use this interface, and it is not documented in the DirectX SDK.

Synth Sink

The wave stream to which the synthesizer is sending data—for example, DirectSound or Microsoft® Win32® waveform audio—is represented by an **IDirectMusicSynthSink** interface. Most applications do not use this interface, and it is not documented in the DirectX SDK.

Overview of DirectMusic Classes

[C++]

The information in this section pertains only to applications written in Visual Basic. For C++ applications, see Overview of DirectMusic Objects and Interfaces.

[Visual Basic]

The DirectMusic classes can be grouped according to their broad functionality, as follows:

- Loader Class
 - Performance Classes
 - Composition Classes
 - Instrument Classes
-

Loader Class

[C++]

The information in this topic pertains only to applications written in Visual Basic. For C++ applications, see Overview of DirectMusic Objects and Interfaces.

[Visual Basic]

The first step in playing music is to load data from a file or resource. An object of the **DirectMusicLoader** class is used for loading bands, collections, segments, and styles. All file input and output and parsing of the data is handled by DirectMusic.

Each of the **DirectMusicLoader** methods (other than **SetSearchDirectory**) returns an object representing the data. Other objects referred to by the primary object being loaded are also made available. For example, if you load a style by using **DirectMusicLoader.LoadStyle** and that style contains a reference to a band, the band object can be retrieved at any time by calling a method of the style object.

Performance Classes

[C++]

The information in this topic pertains only to applications written in Visual Basic. For C++ applications, see Overview of DirectMusic Objects and Interfaces.

[Visual Basic]

The **DirectMusicPerformance** class is at the heart of any DirectMusic application. Most applications have a single object of this class, which is responsible for managing timing and ports, changing global musical parameters such as the master tempo, and playing musical segments.

Each individual piece of music, such as a MIDI file or a segment authored in an application such as DirectMusic Producer, is represented by an object of the **DirectMusicSegment** class. Methods of this class can be used to set or retrieve musical parameters for the segment and to manage the DLS instruments associated with it. A special kind of **DirectMusicSegment**, the template, is used in run-time composition of playable segments.

When a **DirectMusicSegment** object is passed to **DirectMusicPerformance.PlaySegment**, the method returns an object of the **DirectMusicSegmentState** class that represents the playing instance of the segment. Methods of this object can be used to obtain information such as the time at which it started playing. The **DirectMusicSegmentState** is also passed to **DirectMusicPerformance** to determine whether the segment is currently playing or to stop playback.

Composition Classes

[C++]

The information in this topic pertains only to applications written in Visual Basic. For C++ applications, see Overview of DirectMusic Objects and Interfaces.

[Visual Basic]

Much of the music played in DirectMusic applications is fully authored either as a MIDI file or as a segment. However, DirectMusic also allows applications to compose music at run time, based on elements such as styles and templates.

The **DirectMusicComposer** class has methods that create **DirectMusicSegment** objects representing either templates, playable segments based on a chord map or template, or transitions based on other segments. The composer object can also be used to change the chord map of an existing segment.

The **DirectMusicChordMap** class represents a chord map loaded from a file or resource, which provides **DirectMusicComposer** with the information that it needs to compose chord progressions.

An object of the **DirectMusicStyle** class is also based on data from a file or resource, either as a separate entity or as part of an authored segment. It represents a set of musical patterns used in composition and can be passed to methods of **DirectMusicComposer**. Its own methods can be used to retrieve other elements that might have been determined in the style, such as motifs and bands.

Instrument Classes

[C++]

The information in this topic pertains only to applications written in Visual Basic. For C++ applications, see Overview of DirectMusic Objects and Interfaces.

[Visual Basic]

The **DirectMusicBand** class represents a set of instruments (with their volume, pan, and transposition) mapped to the parts in a piece of music. Band changes determined in a style take place automatically when a segment based on that style is played, but you can also obtain a **DirectMusicBand** object from a style or from a separate file or resource and create a **DirectMusicSegment** from it. This segment can then be played to effect program changes.

The **DirectMusicCollection** class represents a DLS collection. Most applications do not need to work directly with collections because the instrument data is downloaded to the port when a segment is played or when a band is downloaded.

The key differences between a collection and a band are as follows:

- A collection is a group of instruments available for use in the playback of any segment. A band is a group of instruments that actually plays a particular segment.
 - Instruments in a collection contain DLS data defining their timbre. Instruments in a band contain no such data, but are linked to instruments in one or more collections.
 - An instrument from a collection is not inherently associated with any particular performance channel (PChannel) of a segment. A band assigns the patch number of an instrument to each PChannel in a segment and assigns a voice priority to the channel.
 - A collection does not contain any information about how the instruments are to be played. A band contains settings for the volume, pan, and transposition of each instrument.
-

Overview of DirectMusic Data Flow

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

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

Note

Any of these data sources can be stored in the application as a resource.

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

A segment comprises several tracks, each containing timed data of a particular kind—for example, notes or tempo changes.

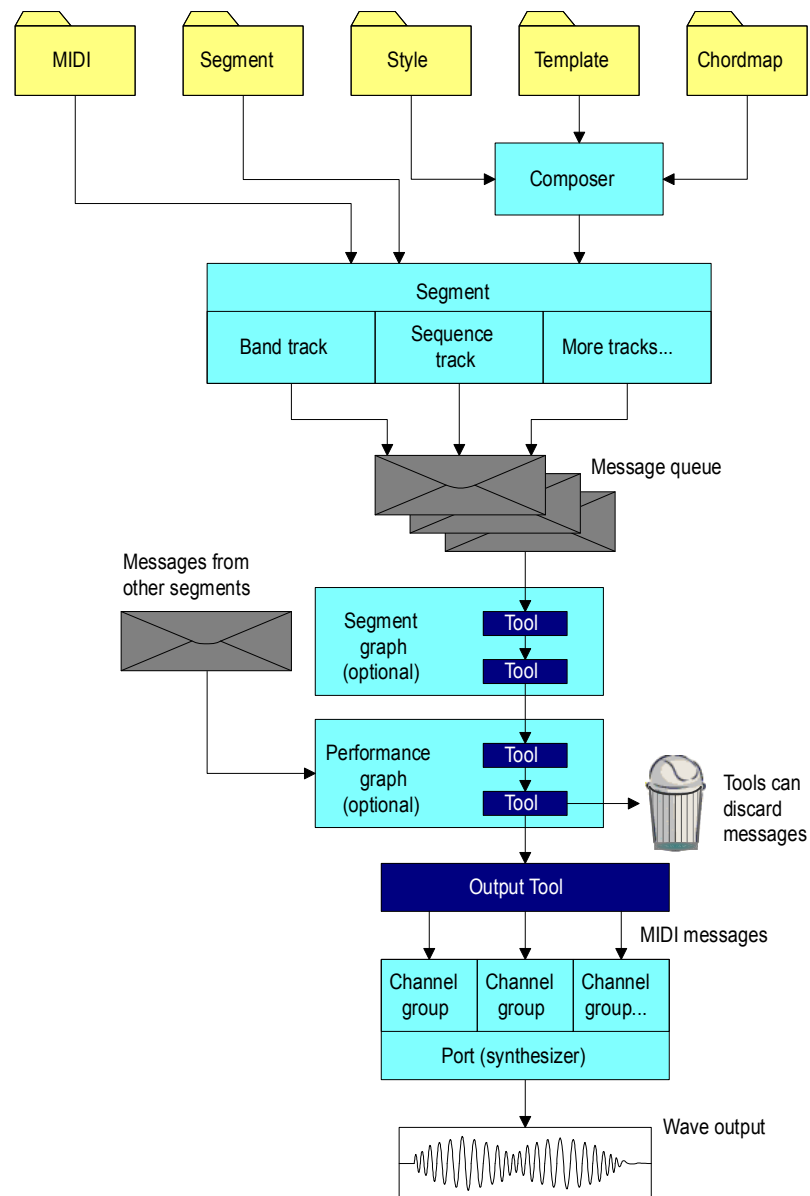
Most tracks generate messages when the segment is played by the performance, and the performance dispatches the messages to any application-defined tools, which can modify messages and pass them on, delete messages, and send new messages. Tools are grouped in segment graphs that process only messages from their own segments, and a performance graph that accepts messages from all segments.

[\[Visual Basic\]](#)

Application-defined tools are not supported by DirectX for Visual Basic.

Finally, the messages are delivered to the output tool, which converts the data to MIDI format before passing it to a port. Channel-specific MIDI messages are directed to the appropriate channel group on the port. The port synthesizes a sound wave that is streamed to a wave output device (normally a DirectSound buffer).

The following illustration shows how musical data gets from files to the wave output device. For the sake of simplicity, only a single segment is shown. This segment gets its data from only one of the three possible sources shown: a MIDI file, an authored segment file, or component files combined by the DirectMusicComposer object.



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

For information on how to implement the process illustrated in the illustration, see the DirectMusic Essentials section, in particular the topics DirectMusic Loader and Playing Music.

For more about segment and component files, see Music Composition. [\[C++.Visual Basic\]](#)

DirectMusic Messages

Musical data passes through the DirectMusic performance engine in the form of messages. Most DirectMusic applications do not work directly with messages, but a basic knowledge of their structure can help you understand how DirectMusic works.

DirectMusic works with 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 before being passed to the performance. Final output to the synthesizer is also in the form of MIDI messages.

The following topics give more information about messages and how they are routed:

- Channels
- Message Creation and Delivery
- Performance Message Types
- MIDI Messages

Channels

A channel is a destination for a message that is specific to one part in the performance. For example, a channel might receive a note-on message that causes the instrument on that channel to make a sound, or a program-change message that assigns a different instrument to that part. (See MIDI Channel Messages.)

Under the MIDI 1.0 standard, there are 16 MIDI channels, meaning that no more than 16 instruments can be playing at one time. To support this standard but at the same time make more channels available to applications, DirectMusic creates channel groups. Up to 65,536 channel groups can exist at one time, each containing 16 channels, for a total of over one million channels. A particular port can be assigned any number of channel groups, up to its capability to support them. Legacy 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 instrument in a DirectMusic performance has a unique performance channel, or *PChannel*. The PChannel 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 PChannel.

[C++]

The number of notes that can be played simultaneously is limited by the number of voices available on the port. (This number can be determined from the **dwMaxVoices** member of the **DMUS_PORTCAPS** structure.)

[\[Visual Basic\]](#)

The number of notes that can be played simultaneously is limited by the number of voices available on the port. (This number can be determined from the **IMaxVoices** member of the **DMUS_PORTCAPS** type.)

A voice is a set of resources dedicated to the synthesis of a single note being played on a channel. If there are more notes playing than there are available voices, one or more notes must be suppressed by the synthesizer. The choice is determined by the priority of the voice currently playing the note, which is based in turn on the priority of the channel. By default, channels are ranked according to their index value, except that channel 10, the MIDI percussion channel, is ranked highest.

[\[C++\]](#)

Applications and synthesizers can set their own channel priorities. For more information, see the Remarks for **IDirectMusicPort::GetChannelPriority**. See also **DMUS_CHANNEL_PRIORITY_PMSG**.

Message Creation and Delivery

When a segment is played, most of its tracks generate messages containing information about events that are to take place during playback. (For more information, see Tracks.)

A few tracks send more than one kind of message. For example, a style track sends note messages and time signature messages. In such cases, an application can disable certain kinds of messages in the track. For more information, see Setting and Retrieving Track Parameters.

Applications can also place messages in the queue directly. You might do this, for example, to change the tempo.

[\[C++\]](#)

For sample code, see **IDirectMusicPerformance::SendPMsg**.

[\[Visual Basic\]](#)

For sample code, see the DLSEffects sample.

The performance engine determines when each message is to be processed in real time (reference time). In the case of channel messages, the performance also determines to what PChannel the message is directed. This information, along with other data—including the message type, its source track, and pointers to the first graph and tool that are to receive the message—are stored in the message.

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, in which they are processed in order of time stamp.

Notes

There is no guarantee that messages with the same time stamp will be processed in any particular order.

Tempo messages tell the performance how to convert music time to reference time. Time-signature messages are purely informational, because the time signature is built into the segment and cannot be changed.

[C++]

Messages are first sent to any tools in the segment's graph, and then to tools in the performance's graph. (The application is responsible for creating either or both of these graphs and defining the tools. There is no default graph.)

The first tool in a graph processes the message, and then, if it wants to pass it on, has the graph stamp the message with a pointer to the next tool. For more information, see Tutorial 2: Using Tools.

At this point, the graph 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 graph has finished stamping and flagging the message.

Ultimately, unless a message has been discarded, it arrives at the DirectMusic output tool, which converts all the data that it receives into standard MIDI messages and delivers these to the synthesizer through the port buffer.

Performance Message Types

[C++]

Messages are stored in various structures derived from **DMUS_PMSG**. Because C does not support inheritance, the members of **DMUS_PMSG** are included in the declaration for each message type as the **DMUS_PMSG_PART** macro. These members contain data common to all messages, including the type of the message, time stamps, the performance channel to which the message is directed, and what graph and tool are next in line to process the message. The other members contain data that is unique to the message type.

The following message structures are defined:

DMUS_PMSG	Simple message with no additional parameters.
DMUS_CHANNEL_PRIORITY_PMSG	Channel-priority change. See Channels.
DMUS_CURVE_PMSG	Curve.
DMUS_MIDI_PMSG	Any MIDI message that does not have a unique message type—for example, a control change.
DMUS_NOTE_PMSG	Music note. (Includes duration, so MIDI note-on and note-off messages are combined in this type.)
DMUS_NOTIFICATION_PMSG	Notification. See Notification and Event Handling.
DMUS_PATCH_PMSG	MIDI patch change.
DMUS_SYSEX_PMSG	MIDI system-exclusive message.
DMUS_TEMPO_PMSG	Tempo change.
DMUS_TIMESIG_PMSG	Time-signature change.
DMUS_TRANSPOSE_PMSG	Transposition.

[Visual Basic]

DirectX for Visual Basic does not give applications direct access to message structures. Instead, it provides the following methods for sending particular kinds of messages:

DirectMusicPerformance.SendCurvePMSG	MIDI curve.
DirectMusicPerformance.SendNotePMSG	Musical note.
DirectMusicPerformance.SendPatchPMSG	Patch change.
DirectMusicPerformance.SendMIDIPMSG	Miscellaneous MIDI events.
DirectMusicPerformance.SendTempoPMSG	Tempo change.
DirectMusicPerformance.SendTimeSigPMSG	Time-signature change.

DirectMusicPerformance.SendTransposePMSG Transposition change.

MIDI Messages

This section gives an overview of standard MIDI messages and how such messages, typically streamed from a MIDI file, are handled by DirectMusic. Most applications do not deal directly with MIDI messages because the loader and the performance manage all the details of playback.

MIDI input is converted into performance message format before being routed through tools by the performance. The output tool converts the data back into the standard MIDI message format before passing it to the synthesizer.

Note

There is no guarantee that MIDI messages will be processed in the same order in which they occur in the file. DirectMusic messages are delivered in order of time stamp, and two MIDI messages with identical time stamps might not be delivered in the expected order. Be sure, in authoring MIDI content, not to give events simultaneous time stamps if they must take place sequentially. For example, do not place a program change at the same time as a note that depends on the program change.

MIDI messages consist of a status byte, usually followed by 1 or 2 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. This is called the running status. Data bytes are recognizable because the high bit is always clear, whereas in status bytes, it is always set.

The time at which MIDI events are streamed from a file 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.

MIDI messages are divided into two main categories:

- MIDI Channel Messages
- MIDI System Messages

MIDI Channel Messages

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

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

A mode message determines how a channel deals with subsequent voice messages. For example, a mode message might instruct the channel to remain silent, ignoring all note-on messages until further notice.

Most channel messages are voice messages: they instruct the channel to begin or stop playing a note or to modify the note in some way, or they change the instrument by assigning a different MIDI patch number to the channel.

Voice messages are of the following types:

Voice message	Purpose
Note-on	Play a note.
Note-off	Stop playing the note.
Control change	Modify the tone with data from a pedal, lever, or other device; also used for miscellaneous controls such as volume and bank select.
Program (patch) change	Select an instrument for the channel by assigning a patch number.
Aftertouch	Modify an individual note, or all notes on the channel, according to the aftertouch of a key.
Pitch bend change	Modify the pitch of all notes played on the channel.

These descriptions apply to standard MIDI messages, not MIDI data that has been converted to performance message format. For example, a pair of MIDI messages to start and stop a note are combined by DirectMusic into a single message giving the duration of the note. DirectMusic messages also contain much additional information about the timing and routing of the message.

MIDI notes

The data bytes of a note-on message represent the pitch and velocity. In most cases, a pitch value of 0 represents C below subcontra C (called C0 in MIDI notation), 12 represents subcontra C (or C1), 60 is middle C (or C5), and so on. For drum kits, the data byte represents a particular drum sound instead. For example, as long as the General MIDI (GM) percussion key map is being adhered to, a value of 60 represents a high bongo sound. Channel 10 is reserved for drum kits, so the synthesizer knows that note-on messages on that channel are to be treated differently than on other channels.

For information on how DirectMusic converts to and from MIDI notes, see [Music Values and MIDI Notes](#).

Program changes

Program changes and patch numbers are very important 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. (Of course, the actual sound produced at the speakers depends on how the instrument is synthesized.)

Bank selection

Because a single data byte is used to select the patch number in a program change and only 7 bits in each data byte of a MIDI message are significant, a program change can select from a maximum of 128 instruments. To provide a greater choice, the MIDI specification allows for the use of up to 16,384 instrument banks, each containing up to 128 instruments.

To select an instrument from a different bank, the MIDI sequencer must first send a control change message called *bank select*. The 2 data bytes of this message are referred to as the most significant byte (MSB) and least significant byte (LSB), and they are combined to identify a bank. Once the bank has been selected, each subsequent program change selects an instrument from that bank.

DirectMusic patch numbers

[Visual Basic]

This topic pertains only to applications written in C++. DirectX for Visual Basic does not allow applications to change patch numbers except by using the **DirectMusicPerformance.SendPatchPMSG** method.

[C++]

In DirectMusic, the instrument patch number is not the 7-bit MIDI patch number, but a 32-bit value that packs the MIDI patch number together with the MSB and LSB of the bank select and a 1-bit flag for a drum kit. This extended patch number is returned by the **IDirectMusicCollection::EnumInstrument**, **IDirectMusicCollection::GetInstrument**, and **IDirectMusicInstrument::GetPatch** methods. It can be changed for an instrument by using the **IDirectMusicInstrument::SetPatch** method.

The organization of DirectMusic patch values is as follows:

Bits	Purpose
0-7	MIDI patch number (bit 7 is always 0)
8-15	LSB bank select (bit 15 is always 0)
16-23	MSB bank select (bit 23 is always 0)
24-30	Unused
31	Flag for drum kit

MIDI System Messages

System messages are not exclusive to any channel. They are of three kinds, as shown in the following table.

Message type	Purpose
System common	Miscellaneous commands and data.
System exclusive	Equipment-specific commands and data.
System real-time	Synchronization of clock-based MIDI equipment.

Unlike other MIDI messages, system-exclusive messages can contain any number of data bytes. After transmitting the data, the sequencer sends a system-common message called an EOX, which signals the end of the system-exclusive message.

[C++]

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

[Visual Basic]

DirectX for Visual Basic does not give applications direct access to system-exclusive messages.

Downloadable Sounds

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

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

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

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

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

A DLS instrument is created from one or more digital samples, typically representing single pitches, which are then modified by the synthesizer to create other pitches. Multiple samples are used to make the instrument sound realistic over a wide range of pitches. When a DLS instrument is downloaded, each sample is assigned to a certain range of pitches, called a *region*. Usually, there are no more than 16 regions.

In addition, samples can be given an articulation, which defines things like attack (how quickly a note reaches full volume), decay (how quickly it falls away from full volume), and other characteristics that make the sound more like that produced by a real instrument.

Downloadable sounds are stored in instrument collections, from which they are downloaded to the synthesizer.

DLS instruments are assigned patch numbers and respond to MIDI messages as do other MIDI instruments. However, a DLS instrument does not have to belong to the General MIDI set. In fact, it does not have to represent a musical instrument at all. Any sound, even a fragment of speech or a fully composed measure of music, can be turned into a DLS instrument.

For more information on DLS collections and how instruments are created, see the documentation for DirectMusic Producer. For a guide to incorporating DLS in your applications, see Using Downloadable Sounds.

Microsoft Software Synthesizer

The Microsoft Software Synthesizer is supplied with DirectMusic and is the default port when hardware DLS is not available or is not supported by the operating system. The synthesizer creates a waveform based on a stream of MIDI messages, using instrument timbres synthesized from DLS samples. By default, the samples are from the Roland GS collection, which is also part of the DirectMusic installation. The synthesizer sends its output to the DirectSound mixer.

Note

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

The Microsoft Software Synthesizer includes reverberation capabilities, which are on by default. The Waves TrueVerb reverberation technology is licensed to Microsoft as the SimpleVerb implementation.

DirectMusic Essentials

This section gives practical information on how to implement DirectMusic in applications. For a more general overview, see DirectMusic Architecture.

The following topics are discussed:

- Building DirectMusic Projects
- Debugging DirectMusic Projects
- Integrating DirectMusic and DirectSound
- Using Ports
- DirectMusic Loader
- DirectMusic File Format
- Using Downloadable Sounds
- Playing Music
- Music Parameters
- Capturing Music
- DirectMusic Tools
- Music Composition

Building DirectMusic Projects

[\[Visual Basic\]](#)

This topic pertains only to application development in C++.

[\[C++\]](#)

Unlike other components of DirectX, the DirectMusic API is completely COM-based and does not contain any library functions, such as helper functions to create COM objects. Therefore, there is no Dmusic.lib file to link to during the build.

Most projects must include the Dmusicc.h and Dmusicci.h header files, which contain declarations for the core and performance layers, respectively, and Dmerror.h, which contains return values. Dmusicf.h contains file formats and is needed only for applications such as music authoring tools that work directly with files and do not rely solely on the loaders built into DirectMusic.

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

DirectMusic uses the multithreading capabilities of the Windows 32-bit operating system. Multithreading allows DirectMusic to generate, process, and synthesize music in the background while your application is accomplishing other tasks. Develop your project with multithreading in mind. At least be sure to link with the multithreaded libraries.

Also be sure that your application has access to the GUIDs used by DirectMusic. For more information, see [Compiling DirectX Samples and Other DirectX Applications](#).

Debugging DirectMusic Projects

[\[Visual Basic\]](#)

This topic pertains only to application development in C++.

[\[C++\]](#)

The DirectMusic dynamic-link libraries (DLLs) installed with the debug version of the DirectX software development kit generate information in the debug output window as the application is running. By default, all available information is shown.

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 the least detailed debugging information and 5 the most. If there is no entry in Win.ini, the debug output is at level 0. You can focus on problems in a particular component by setting higher values for the other components. You can also change the debug level by using the DirectX property sheet in Control Panel. However, this method sets the same value for all DirectMusic components.

See also [Debugging DirectX Applications](#).

Integrating DirectMusic and DirectSound

The Microsoft Software Synthesizer and other synthesizers normally stream their output wave data to a DirectSound buffer. DirectMusic is capable of handling all the details of linking to DirectSound without any action on your part. When it creates or links to a DirectSound object, DirectMusic ensures that the primary buffer format matches that of the highest format among all DirectMusic ports.

[C++]

In an application that uses only music files for its soundtrack and does not require DirectSound for playing wave files or resources, the DirectSound object is typically created when the DirectMusic performance is initialized. This is shown in the following example, in which *pPerf* is a pointer to the **IDirectMusicPerformance** interface:

```
pPerf->Init(NULL, NULL, hwnd);
```

In this example, the first NULL specifies that the DirectMusic object is to be created and managed internally, the second NULL specifies the same for the DirectSound object, and *hwnd* is the handle of the controlling window for DirectSound.

Note

It is a good idea to supply the top-level application window handle when requesting that DirectMusic create the DirectSound object. See the Remarks for **IDirectMusicPerformance::Init** and **IDirectSound::SetCooperativeLevel**.

[Visual Basic]

In an application that uses only music files for its soundtrack and does not require DirectSound for playing wave files or resources, the **DirectSound** object is typically created when the DirectMusic performance is initialized. This is shown in the following example, where *Perf* is a **DirectMusicPerformance** object:

```
Perf.Init(Nothing, hwnd);
```

In this example, *Nothing* specifies that the **DirectSound** object is to be created and managed internally, and *hwnd* is the handle of the controlling window for DirectSound.

Note

It is a good idea to supply the top-level application window handle when requesting that DirectMusic create the **DirectSound** object. See the Remarks for **DirectMusicPerformance.Init** and **DirectSound.SetCooperativeLevel**.

More information is contained in the following topics:

- Setting the DirectSound Object
- Setting the DirectSound Buffer Object

Setting the DirectSound Object

[C++]

Allowing DirectMusic to create and manage the DirectSound object works for applications that are not using DirectSound independently. However, if your application is using DirectSound to play wave data from a source other than the DirectMusic synthesizer, you must ensure that the same **IDirectSound** interface is used by DirectMusic. Create the DirectSound object first; then pass the interface pointer to DirectMusic. This can be done in the call to **IDirectMusicPerformance::Init**, as in the following example, in which *pPerf* is the **IDirectMusicPerformance** and *pDS* is a pointer to **IDirectSound**:

```
pPerf->Init(NULL, pDS, NULL);
```

Note

When an **IDirectSound** pointer is passed to **IDirectMusicPerformance::Init**, the third parameter, the window handle, is ignored. The application is responsible for setting the controlling window by calling

IDirectSound::SetCooperativeLevel.

Set the DSSCL_PRIORITY cooperative level for any DirectSound object to be used with DirectMusic. If you set the DSSCL_NORMAL cooperative level, DirectMusic cannot upgrade the primary buffer format.

If you create the DirectMusic object by using **CoCreateInstance**, rather than letting the performance do it for you, you must also explicitly link it to an existing **IDirectSound** interface. This is done by using the **IDirectMusic::SetDirectSound** method.

You can also use the **IDirectMusicPort::SetDirectSound** method to assign a different DirectSound object to a port. Do this when different ports are on different audio devices, each represented by its own **IDirectSound** interface. The DirectSound object for a port cannot be changed once the port has been activated.

When a port that requires DirectSound is activated, it takes the first available **IDirectSound** interface pointer from the following list:

1. The **IDirectSound** interface passed to **IDirectMusicPort::SetDirectSound**.
2. The **IDirectSound** interface passed to **IDirectMusic::SetDirectSound**.
3. The **IDirectSound** interface created by DirectMusic if NULL was passed to **IDirectMusic::SetDirectSound**.

When DirectSound is in emulation mode, it has exclusive use of the audio device. An application should release DirectSound whenever it loses the focus to allow other applications to use the device. Typically, this is done in response to a WM_ACTIVATE message.

If DirectMusic created the DirectSound object, it automatically releases the object when all ports are deactivated (and creates a new one when the first port using DirectMusic is reactivated). However, if you created the DirectSound object yourself, you are responsible for releasing it. This is done by using the **IDirectMusic::SetDirectSound** or the **IDirectMusicPort::SetDirectSound** method, passing NULL as the *pDirectSound* parameter.

[Visual Basic]

Allowing DirectMusic to create and manage the **DirectSound** object works for applications that are not using DirectSound independently. However, if your application is using DirectSound to play wave data from a source other than the DirectMusic synthesizer, you must ensure that the same **DirectSound** object is used by DirectMusic. Create the **DirectSound** object first; then pass it to DirectMusic. This is done in the call to **DirectMusicPerformance.Init**, as in the following example, in which *Perf* is the **DirectMusicPerformance** object and *DS* is the **DirectSound** object:

```
Perf.Init(DS, 0);
```

Note

When a **DirectSound** object is passed to **DirectMusicPerformance.Init**, the second parameter, the window handle, is ignored. The application is responsible for setting the controlling window by calling

DirectSound.SetCooperativeLevel.

Set the DSSCL_PRIORITY cooperative level for any **DirectSound** object to be used with DirectMusic. If you set the DSSCL_NORMAL cooperative level, DirectMusic cannot upgrade the primary buffer format.

Setting the DirectSound Buffer Object

[Visual Basic]

This topic pertains only to applications written in C++. DirectX for Visual Basic does not support custom DirectSound buffers for DirectMusic output.

[C++]

When DirectMusic is linked to DirectSound, it creates and manages a secondary DirectSound buffer for the wave output from each port, in a format matching that of the port. You can override the default behavior and ensure that the data is streamed to a different buffer by using the **IDirectMusicPort::SetDirectSound** method. You might do this, for example, to have 3-D effects on the sound buffer. (See the 3DMusic Sample.) You might even create multiple instances of the synthesizer port, each with

its own DirectSound 3-D buffer, to place different instruments at different points in space.

The buffer that you pass to **IDirectMusicPort::SetDirectSound** must be a secondary streaming buffer with a matching format. Get information about the wave format and recommended buffer size by calling the **IDirectMusicPort::GetFormat** method.

DirectMusic does not attempt to upgrade the primary buffer when you pass your own **IDirectSoundBuffer** to **IDirectMusicPort::SetDirectSound**.

Using Ports

A port is a device that sends or receives musical data. It can correspond to a hardware device, a software synthesizer, or a software filter.

[C++]

Each port in a DirectMusic application is represented by an **IDirectMusicPort** interface. Methods of this interface are used to retrieve information about the device, manage the memory on the device, download and unload DLS instruments, read incoming data, and cue playback buffers.

Every performance must have at least one port. If you want to use a port other than the default port to set up special parameters for the default port, first set up a **DMUS_PORTPARAMS** structure. You do not 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 **IDirectMusic::CreatePort** method.

The following C++ code example demonstrates how an object might be created for the default port, setting five channel groups on the port, assuming that *pDirectMusic* is a valid **IDirectMusic** pointer.

```
IDirectMusicPort* pPort;
DMUS_PORTPARAMS dmos;

ZeroMemory( &dmos, sizeof(DMUS_PORTPARAMS) );
dmos.dwSize = sizeof(DMUS_PORTPARAMS);
dmos.dwValidParams = DMUS_PORTPARAMS_CHANNELGROUPS;
dmos.dwChannelGroups = 5;
HRESULT hr = pDirectMusic->CreatePort( GUID_NULL, &dmos,
    &pPort, NULL )
```

Once you have a port, activate it by calling **IDirectMusic::Activate** or **IDirectMusicPort::Activate** and attach it to the performance by using the **IDirectMusicPerformance::AddPort** method.

When you add a port to a performance, assign a block of PChannels to it by calling the **IDirectMusicPerformance::AssignPChannelBlock** method. The only time this

is not necessary is when you add the default port by passing NULL to **IDirectMusicPerformance::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 **IDirectMusicPerformance::AssignPChannelBlock** and **IDirectMusicPerformance::AssignPChannel** methods.

[\[Visual Basic\]](#)

DirectX for Visual Basic supports only a single port for each performance, and all PChannels are mapped to this port. Choose the port after initializing the performance by using **DirectMusicPerformance.SetPort**. To use the default port, pass -1 as the *index* parameter.

Available ports are automatically enumerated when you initialize the performance. You can look for a particular port or capabilities by using the **DirectMusicPerformance.GetPortCount**, **DirectMusicPerformance.GetPortCaps**, and **DirectMusicPerformance.GetPortName** methods. The following code example looks for the first software synthesizer and sets this as the port for the performance, with support for up to 16 channel groups:

```
' perf is a DirectMusicPerformance object.
Dim X As Integer
Dim portcaps As DMUS_PORTCAPS

perf.Init(Nothing, Me.hWnd)
For X = 1 to perf.GetPortCount
    Call perf.GetPortCaps(X, portcaps)
    If portcaps.lFlags And DMUS_PC_SOFTWARESYNTH Then
        perf.SetPort(X, 16)
    Exit For
    End If
Next X
' If no port was set, set the default port, or take other action.
```

More information about ports is contained in the following topics:

- Default Port
- Legacy Ports
- Port Property Sets

Default Port

Under Windows 95 and versions of Windows 98 prior to Windows 98 Second Edition, and always when hardware that supports DLS is not available, the Microsoft Software Synthesizer is the default port. Under later versions of Windows 98 and under Windows 2000, a hardware synthesizer might be the default port.

[C++]

If you want your application to use the default port, you do not have to call the **IDirectMusic::CreatePort** method before adding the port to the performance. Instead, you can pass NULL to **IDirectMusicPerformance::AddPort**.

Obtain the default port by a call to **IDirectMusic::GetDefaultPort**, and then check its capabilities by using the **IDirectMusicPort::GetCaps** method. If the port does not meet the needs of your application, use the **IDirectMusic::EnumPort** method to find the Microsoft Software Synthesizer or another port.

[Visual Basic]

A performance uses the default port if -1 is passed as the index parameter to **DirectMusicPerformance.SetPort**. If you do this, however, you have no way of checking the port's capabilities. If your application needs a port with particular capabilities, examine the ports in the collection created by DirectMusic when the performance is initialized. For more information, see Using Ports.

Legacy Ports

Under Windows 95 and Windows 98, DirectMusic supports legacy ports—that is, it sequences output data to a MIDI device that uses FM or hardware wave-table synthesis.

Legacy ports have the following restrictions in DirectMusic:

- Not supported on Windows 2000.
- No support for downloadable sounds.
- Master volume cannot be changed. (This can be done for other ports by setting a global parameter. See Setting and Retrieving Global Parameters.)
- There is only one channel group. (See Channels.)

Most applications do not need to consider the presence of legacy ports because the Microsoft Software Synthesizer produces consistent results on all output devices capable of playing waveform audio.

[C++]

You can recognize a legacy port by the `DMUS_PORT_WINMM_DRIVER` flag in the **dwType** member of the **DMUS_PORTCAPS** structure returned by the **IDirectMusic::EnumPort** or the **IDirectMusicPort::GetCaps** method.

Legacy ports also differ from others in that the application determines whether channels on the legacy port are shared between logical ports by setting the **fShare** member of the **DMUS_PORTPARAMS** structure. This member is relevant only for ports that have the `DMUS_PC_SHAREABLE` flag in the **dwFlags** member of **DMUS_PORTCAPS**, as is always the case for legacy ports.

[\[Visual Basic\]](#)

You can recognize a legacy port by the `DMUS_PORT_WINMM_DRIVER` flag in the **IType** member of the **DMUS_PORTCAPS** type returned by **DirectMusicPerformance.GetPortCaps**.

Port Property Sets

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++. DirectX for Visual Basic does not support port property sets.

[\[C++\]](#)

Through property sets, DirectMusic provides unlimited support for new features in hardware and drivers. A property set is associated with a particular port.

Hardware vendors define new capabilities as properties and publish the specification for these properties, including GUIDs. You can use the **IKsControl::KsProperty** method to find out whether a property is available, and then to set and retrieve values for that property. Obtain the **IKsControl** interface for a port by calling the **IDirectMusicPort::QueryInterface** method, passing `IID_IKsControl` as the interface identifier.

A property set is represented by a GUID, and each item within the set is represented by a zero-based index. The meaning of the indexed items for a GUID never changes. For a list of the property sets supported by DirectMusic, see **KSPROPERTY**.

All property sets predefined by DirectMusic have only one item, usually at index 0. However, the full definition of kernel-streaming (KS) properties is supported, and vendors are free to create property sets with any number of items and instances, and data of any size.

Routing of the property item request to the port varies, depending on the port implementation. No properties are supported by ports that represent DirectMusic

emulation over the Win32 handle-based multimedia calls (the **midiOut** and **midiIn** functions).

The following code example uses the **IKsControl::KsProperty** method to determine if the port supports General MIDI in hardware:

```

BOOL IsGMSupported(IDirectMusicPort *pPort)
{
    HRESULT hr;
    IKsControl *pControl;
    KSPROPERTY ksp;
    DWORD dwFlags;
    ULONG cb;
    BOOL flsSupported;

    // Query for an IKsControl interface.
    hr = pPort->QueryInterface(IID_IKsControl, (void**)&pControl);
    if (FAILED(hr))
    {
        // Port does not support properties; assume no GM support.
        return FALSE;
    }

    // Ask about GM.
    ksp.Set = GUID_DMUS_PROP_GM_Hardware;
    ksp.Id = 0;
    ksp.Flags = KSPROPERTY_TYPE_BASIC SUPPORT;
    hr = pControl->KsProperty(&ksp, sizeof(ksp),
        &dwFlags, sizeof(dwFlags), &cb);
    flsSupported = FALSE;
    if (SUCCEEDED(hr) || (cb >= sizeof(dwFlags)))
    {
        // Set is supported.
        flsSupported = (BOOL)(dwFlags & KSPROPERTY_TYPE_GET);
    }
    pControl->Release();
    return flsSupported;
}

```

The following code example shows how a property can be changed. In this case, the reverberation properties of the software synthesizer are set to those contained in a **DMUS_WAVES_REVERB_PARAMS** structure.

```

/* Assume that m_* variables have been initialized to valid values.
   m_pPort is a pointer to IDirectMusicPort. */

```

```

DMUS_WAVES_REVERB_PARAMS Params;

```

```

Params.fInGain = m_fReverbIn;
Params.fHighFreqRTRatio = m_fReverbHigh;
Params.fReverbMix = m_fReverbMix;
Params.fReverbTime = m_fReverbTime;

IKsControl *pControl;
if (m_pPort)
{
    // Query for IKsControl interface.
    HRESULT hr = m_pPort->QueryInterface(IID_IKsControl,
        (void**)&pControl);
    if (SUCCEEDED(hr))
    {
        KSPROPERTY ksp;
        ULONG cb;

        ZeroMemory(&ksp, sizeof(ksp));
        ksp.Set = GUID_DMUS_PROP_WavesReverb;
        ksp.Id = 0;
        ksp.Flags = KSPROPERTY_TYPE_SET;

        pControl->KsProperty(&ksp,
            sizeof(ksp),
            (LPVOID)&Params,
            sizeof(Params),
            &cb);
        pControl->Release();
    }
}

```

The following code example turns on the reverb effect for the port represented by *m_pPort*:

```

DWORD dwEffects = 0;
IKsControl *pControl;
HRESULT hr = m_pPort->QueryInterface(IID_IKsControl,
    (void**)&pControl);
if (SUCCEEDED(hr))
{
    KSPROPERTY ksp;
    ULONG cb;

    ZeroMemory(&ksp, sizeof(ksp));
    dwEffects = DMUS_EFFECT_REVERB;
    ksp.Set = GUID_DMUS_PROP_Effects;
    ksp.Id = 0;

```

```

ksp.Flags = KSPROPERTY_TYPE_SET;

pControl->KsProperty(&ksp,
    sizeof(ksp),
    (LPVOID)&dwEffects,
    sizeof(dwEffects),
    &cb);
pControl->Release();
}

```

The final example shows how you might turn off reverb, leaving any other effects intact:

```

DWORD dwEffects = 0;
IKsControl *pControl;
HRESULT hr = m_pPort->QueryInterface(IID_IKsControl,
    (void**)&pControl);
if (SUCCEEDED(hr))
{
    KSPROPERTY ksp;
    ULONG cb;

    ZeroMemory(&ksp, sizeof(ksp));
    ksp.Set = GUID_DMUS_PROP_Effects;
    ksp.Id = 0;
    ksp.Flags = KSPROPERTY_TYPE_GET;

    pControl->KsProperty(&ksp,
        sizeof(ksp),
        (LPVOID)&dwEffects,
        sizeof(dwEffects),
        &cb);

    ZeroMemory(&ksp, sizeof(ksp));
    dwEffects = dwEffects & ~DMUS_EFFECT_REVERB;
    ksp.Set = GUID_DMUS_PROP_Effects;
    ksp.Id = 0;
    ksp.Flags = KSPROPERTY_TYPE_SET;

    pControl->KsProperty(&ksp,
        sizeof(ksp),
        (LPVOID)&dwEffects,
        sizeof(dwEffects),
        &cb);

    pControl->Release();
}

```

}

DirectMusic Loader

[C++]

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

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

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

Objects that are referred to by other objects are loaded transparently. For example, suppose a style being loaded from a DirectMusic Producer file contains a reference to a band whose data is in another file. When the style's **IPersistStream** finds the reference, it obtains the **IDirectMusicGetLoader** interface from the **IStream** that passed it the data stream. Using this interface, it obtains a pointer to the **DirectMusicLoader**. Then it calls **IDirectMusicLoader::GetObject** to load the band.

[Visual Basic]

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

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

However, if your application has occasion to load a segment more than once, you should be aware that because of the caching system used internally by DirectMusic, you might get back the same segment object, with the same settings such as start and loop points and any connection to a DLS collection. To ensure that this does not happen, you should release the **DirectMusicLoader** object by setting it to Nothing, and then create a new one, before reloading the segment.

Objects that are referred to by other objects are loaded transparently. For example, suppose a style being loaded from a DirectMusic Producer file contains a reference to a band whose data is in another file. DirectMusic loads the band automatically, and the application retrieves it by using the **DirectMusicStyle.GetBand** method.

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

- Setting the Loader's Search Directory
- Scanning a Directory for Objects
- Enumerating Objects
- Loading an Object from a File
- Loading an Object from a Resource
- Getting Object Descriptors
- Caching Objects
- Setting Objects
- Custom Loading

Setting the Loader's Search Directory

[C++]

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

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

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

Having called this function, the application can now load a style by file name, without including the full path, as in the following code example:

```
HRESULT myLoadStyleFromPath (  
    IDirectMusicStyle **ppIStyle, // Receives a pointer to a style
```

```
IDirectMusicLoader *pLoader) // Loader already created
{
    HRESULT hr;
    DMUS_OBJECTDESC Desc;

    ZeroMemory(&Desc, sizeof(DMUS_OBJECTDESC);
    Desc.dwSize = sizeof(DMUS_OBJECTDESC);
    wcscpy(Desc.wszFileName, L"polka.sty"); // Short file name
    Desc.guidClass = CLSID_DirectMusicStyle; // Object class
    Desc.dwValidData = DMUS_OBJ_CLASS | DMUS_OBJ_FILENAME;

    hr = pLoader->GetObject(&Desc,
        IID_IDirectMusicStyle, (void **) ppIStyle);
    return hr;
}
```

[Visual Basic]

By default, the loader looks for file objects in the current directory unless a full path has been passed to the load method. Using the

DirectMusicLoader.SetSearchDirectory method, you can set a different default path for the **DirectMusicLoader.LoadBand**, **DirectMusicLoader.LoadCollection**, **DirectMusicLoader.LoadSegment**, and **DirectMusicLoader.LoadStyle** methods.

Scanning a Directory for Objects

[Visual Basic]

This topic pertains only to applications written in C++.

[C++]

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

The method compiles a list of all matching files and uses the

IDirectMusicObject::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

It is always a good idea to call **IDirectMusicLoader::ScanDirectory** before loading any objects. Even though you might be loading objects explicitly by file name, those objects could contain references to other objects not identified by file name, and the loader would not be able to find these referenced objects if **ScanDirectory** was not called on every directory in which the objects might be.

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

Note

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

For an example, see Enumerating Objects.

Enumerating Objects

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

[\[C++\]](#)

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

Note

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

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

```
void myListStyles(
    IDirectMusicLoader *pLoader)

{
    HRESULT hr = pLoader->SetSearchDirectory(
        CLSID_DirectMusicStyle,
```

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

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

Loading an Object from a File

[\[C++\]](#)

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

```
IDirectMusicLoader* m_pLoader;

CoInitialize(NULL);
HRESULT hr = CoCreateInstance(
```

```
CLSID_DMLoader,  
NULL,  
CLSCTX_INPROC,  
IID_IDirectMusicLoader,  
(void**)&m_pLoader);
```

Then, describe the object, and call the **IDirectMusicLoader::GetObject** method to load it and obtain the desired interface.

The following code example loads a style from disk and returns a pointer to it in the variable addressed by the parameter.

```
void myLoadStyle(  
    IDirectMusicStyle **pplStyle)  
{  
    IDirectMusicLoader *pLoader;    // Loader interface  
  
    /* Normally you would create the loader once and use it for the  
    duration of the application. This reduces overhead and takes  
    advantage of the loader's ability to cache objects. However,  
    this example creates it dynamically and throws it  
    away once the style is loaded. */  
  
    CoCreateInstance(  
        CLSID_DirectMusicLoader, NULL,  
        CLSCTX_INPROC,  
        IID_IDirectMusicLoader,  
        (void **) &pLoader);  
  
    if (pLoader)  
    {  
        DMUS_OBJECTDESC Desc;  
  
        // Start by initializing Desc with the file name and  
        // class GUID for the style object.  
  
        wcscpy(Desc.wszFileName, L"c:\\mymusic\\funky\\polka.sty");  
        Desc.guidClass = CLSID_DirectMusicStyle;  
        Desc.dwSize = sizeof (DMUS_OBJECTDESC);  
        Desc.dwValidData = DMUS_OBJ_CLASS |  
            DMUS_OBJ_FILENAME |  
            DMUS_OBJ_FULLPATH;  
  
        pLoader->GetObject(&Desc, IID_IDirectMusicStyle,  
            (void **) pplStyle);  
        pLoader->Release();  
    }  
}
```

```
}
```

This code example identifies the file by a full path name and indicates this by setting the `DMUS_OBJ_FULLPATH` flag. If you have previously set the search directory, you can use the short name of the file without full path information. For an example, see [Setting the Loader's Search Directory](#).

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

[\[Visual Basic\]](#)

To load an object, first create a **DirectMusicLoader** object. Then, call one of the following methods:

- **DirectMusicLoader.LoadBand**
- **DirectMusicLoader.LoadCollection**
- **DirectMusicLoader.LoadSegment**
- **DirectMusicLoader.LoadStyle**

Pass in either a simple file name or a full path for the file that contains the desired object. Each of these methods returns an instance of the appropriate class.

The following code example, in which *objDX* is a **DirectX7** object, loads a segment from a file in the current search directory:

```
Dim objDMLoader As DirectMusicLoader
Dim objSeg As DirectMusicSegment

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

See also [Loading an Object from a Resource](#).

Loading an Object from a Resource

[\[C++\]](#)

Loading an object from a resource, or from some other location in memory, is done like loading an object from a file. (See [Loading an Object from a File](#).) In this case, 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 **lMemLength** members respectively, of the **DMUS_OBJECTDESC** structure. You must also set the **DMUS_OBJ_MEMORY** flag in the **dwFlags** member.

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

The following code example loads a MIDI file from a resource into a segment:

```
HRESULT LoadMidi(HMODULE hMod, WORD ResourceID)
{
    HRESULT          hr;
    DMUS_OBJECTDESC  ObjDesc;
    IDirectMusicSegment* pSegment = NULL;

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

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

    hr = m_pDXPerformance->m_pLoader->GetObject(
        &ObjDesc, IID_IDirectMusicSegment2,
        (void**) &m_pSegment );

    return hr;
}
```

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

[Visual Basic]

Loading an object from a resource, or from some other location in memory, is done like loading an object from a file. (See Loading an Object from a File.) The **DirectMusicLoader.LoadBandFromResource**, **DirectMusicLoader.LoadChordMapFromResource**,

DirectMusicLoader.LoadCollectionFromResource, **DirectMusicLoader.LoadSegmentFromResource**, and **DirectMusicLoader.LoadStyleFromResource** methods each take a module name and resource identifier as parameters and return an instance of the appropriate class.

The following resource types are recognized by the loader:

"DMBAND"	LoadBandFromResource
"DMCHORD"	LoadChordmapFromResource
"DMCOLL"	LoadCollectionFromResource
"DMSEG"	LoadSegmentFromResource
"DMSTYLE"	LoadStyleFromResource

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

The following code example, in which *loader* is the **DirectMusicLoader** object and *perf* is the **DirectMusicPerformance**, loads and plays a MIDI file stored as a "DMSEG" resource in the executable:

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

Getting Object Descriptors

[Visual Basic]

This topic pertains only to applications written in C++.

[C++]

Once you have loaded an object, you can use its **IDirectMusicObject** interface to retrieve information about it in a **DMUS_OBJECTDESC** structure. You must first obtain the **IDirectMusicObject** interface for the object.

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

```
/* It is assumed that pStyle is a valid pointer to an
   IDirectMusicStyle interface. */

if (pStyle)
{
    IDirectMusicObject *pIObject;
```

```
DMUS_OBJECTDESC Desc;

if (SUCCEEDED(pStyle->QueryInterface(IID_IDirectMusicObject,
    (void **) &pObject))
{
    if (SUCCEEDED(pObject->GetDescriptor(&Desc))
    {
        if (Desc.dwValidData & DMUS_OBJ_NAME)
        {
            TRACE("Style name is %S\n", Desc.wszName);
        }
    }
    pObject->Release();
}
}
```

Caching Objects

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

[\[C++\]](#)

When an object is cached, the same interface pointer is always returned by the **IDirectMusicLoader::GetObject** method.

Caching is used extensively in the file-loading process to resolve links to objects. If an object is not found in the cache, it has to be reloaded, even if it already exists. For example, two segments could refer to the same style. When the first segment loads, it calls the loader to get the style. The style in turn creates a style object, loads it from disk, stores a pointer to the style in the cache, and returns it to the segment. If caching is enabled, when the second segment loads, it asks for the style, and the loader immediately returns it. Then, both segments point to the same style. If caching is disabled, the second segment's request for the style causes a duplicate style to be loaded from the file. This is very inefficient.

Another example: **IDirectMusicBand** counts on the loader to keep the General MIDI DLS collection cached. Every time it comes across a GM instrument, it gets the collection from the loader by requesting it with `GUID_DefaultGMCollection`. 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.

By default, caching is enabled for all object classes. You can disable caching for an object class, or for all objects, by using the **IDirectMusicLoader::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 **IDirectMusicLoader::ClearCache** method.

To cache a single object, pass it to the **IDirectMusicLoader::CacheObject** method. You can remove it from the cache, ensuring that it will be loaded again on the next call to **GetObject**, by using the **IDirectMusicLoader::ReleaseObject** method.

Call **ReleaseObject** on a cached object, particularly a segment, before destroying the object by calling its own **Release** method. If you do not, a copy of the object remains in the cache, along with certain state information. In the case of a segment, any instance that you later create will be loaded from the cache, and its start point and loop points will be the same as they were when the previous instance was destroyed.

With judicious use of **CacheObject**, **ReleaseObject**, and **EnableCache**, you can have the objects that you do not need released, while the others remain in the cache.

Setting Objects

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

[\[C++\]](#)

Sometimes it is desirable to tell the loader where to get an object, without actually loading that object, so that the loader can retrieve it if the object is later referred to 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 **SetObject** method takes as a parameter a **DMUS_OBJECTDESC** structure that contains two key pieces of information:

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

The following code example 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)
```



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

The following code could be used to assign a name to a MIDI file stored as a resource of type MIDI:

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

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

Custom Loading

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++. DirectX for Visual Basic supports only loading of standard DirectMusic objects.

[C++]

Specialized applications might create their own object types that encapsulate data from a file or resource. It can be convenient to have the DirectMusic loader handle the loading of such objects. This is especially true if the custom object is referred to by other objects or contains references to other objects.

To implement a loading mechanism that takes advantage of the DirectMusic loader, take the following steps:

- Register the object class so that it can be found by **IDirectMusicLoader::GetObject**.
- Implement the **IDirectMusicObject** interface on the object so that the loader can get the information that it needs to find and cache it.
- Implement the **IPersistStream** interface on the object, with full functionality in the **IPersistStream::Load** method. This allows you to parse the data that you obtain through calls on the **IStream** interface passed by the DirectMusicLoader.
- In the implementation of **Load**, ensure that references to other objects are dealt with by querying the **IStream** for the **IDirectMusicGetLoader** interface, then calling **IDirectMusicGetLoader::GetLoader** to obtain a pointer to the DirectMusicLoader that created the stream. Once you have this pointer, call **IDirectMusicLoader::GetObject** to load the new object.

An application might need to manage file input itself—for example, if all objects are stored in a special compressed resource file. The application can create its own loader by creating an object that supports the **IDirectMusicLoader** interface, with the **IDirectMusicLoader::GetObject** method implemented. All other methods are optional. This implementation of the loader must also create its own stream object that has both the **IStream** and the **IDirectMusicGetLoader** interfaces.

DirectMusic File Format

[Visual Basic]

This section pertains only to applications written in C++. DirectX for Visual Basic does not support file parsing, which is always handled transparently by the **DirectMusicLoader** object.

[C++]

This section describes the format of music files created in an application such as DirectMusic Producer and read by DirectMusic when

IDirectMusicLoader::GetObject is called. Most applications do not 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 information about RIFF files and how DirectMusic data is stored:

- About RIFF
- RIFF Notation
- DirectMusic File Chunks

For a reference to data structures used in DirectMusic files, see File Structures.

About RIFF

The basic building block of a RIFF file is a chunk. A chunk is a logical unit of data. Each chunk contains the following fields:

- A four-character code (**FOURCC**) specifying the chunk identifier. Conventionally, this is uppercase for registered chunk types, and lowercase otherwise.
- A **DWORD** value specifying the size of the data member in the chunk.
- The data.

A chunk contained in another chunk is a subchunk. The only chunks allowed to contain subchunks are those with a chunk identifier of RIFF or LIST.

The first chunk in a file must be identified as RIFF. All other chunks in the file are subchunks of the RIFF chunk.

RIFF chunks include an additional field in the first 4 bytes of the data field. This additional field provides the form type of the chunk. The form type is a four-character code identifying the format of the data stored in the file. For example, DirectMusic styles have the form type DMST.

LIST chunks also include an additional field in the first 4 bytes of the data field. This additional 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.

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 a little confusing. You might expect the chunk labeled <part-list>, for example, to be a list of parts. In fact, it is a list of the elements of a "part" chunk, which describes a single part.

Note

Every four-character code used in DirectMusic files has a corresponding macro in Dmusicf.h. For example, the **FOURCC** for DMST is returned by the **DMUS_FOURCC_STYLE_FORM** macro .

For more information on RIFF files in general, see Resource Interchange File Format Services in the Platform SDK documentation.

RIFF Notation

The descriptions of DirectMusic files in the following sections use a subset of the conventional notation for RIFF files. The principal parts of this notation are shown in the following table:

Notation	Description
<element>	File element labeled "element", or of type element .
[<element>]	Optional file element.
<element>...	One or more copies of the specified element.
[<element>]...	Zero or more copies of the specified element.
name, 'name', NAME, or 'NAME'	FOURCC identifier of a form type, list type, or chunk.
// Comment	Comment.

Labels are used only in the notation, not in the files themselves. The label <chek-ck> refers to a chunk with a unique **FOURCC** identifier and format. Wherever a chunk of this kind occurs in the notation, the same label is used.

The data or subelements associated with a label are described as follows:

```
<chek-ck> -> cheh( <DMUS_IO_CHORDENTRY> )
```

This notation shows that the chunk labeled <chek-ck> consists of the **FOURCC** identifier "chek" followed by a **DMUS_IO_CHORDENTRY** structure. Of course, a **DWORD** showing the size of the data must precede the data, as it does in any RIFF chunk. The presence of this data-size element is assumed and 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>... )
```

DirectMusic File Chunks

The following sections describe the format of chunks used in DirectMusic RIFF files:

- Common Chunks
- Band Form
- Chord-map Form
- Segment Form
- Style Form
- Tool Form
- Tool Graph Form
- Track Form

Common Chunks

The following chunks occur within various list chunks and forms.

GUID Chunk

```
<guid-ck> -> guid( <GUID> )
```

This is the GUID identifier of the element.

Version Chunk

```
<vers-ck> -> vers( <DMUS_IO_VERSION> )
```

This chunk contains version information for the element.

UNFO Chunk

```
<UNFO-list> -> LIST( 'UNFO'  
    <unfo-text-ck>...  
    )
```

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.

Reference List Chunk

The reference list chunk contains information about a reference to an object in another file. For example, a band object might contain a reference to a DLS collection in a separate file.

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
)

```

The data begins with a header that includes information about the object being referred to:

```
<refh-ck> -> refh( <DMUS_IO_REFERENCE> )
```

All other chunks are optional. The GUID and version chunks were described previously. The notation for the others is as follows:

```

<date-ck> -> date( <FILETIME> )
<name-ck> -> name( <WCHAR>... ) // Null-terminated string
<file-ck> -> file( <WCHAR>... ) // Null-terminated string
<catg-ck> -> catg( <WCHAR>... ) // Null-terminated string

```

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
  <lbin-list>  // List of instruments
)

```

For the first three chunks, which are optional, see Common Chunks.

The data is contained in a list of lists:

```

<lbin-list> -> LIST( 'lbin'
  <lbin-list>...
)

```

Each instrument is described in a list that has the following format:

```

<lbin-list> -> LIST( 'lbin'
  <bins-ck>
  [<DMRF-list>]
)

```

Within the instrument list, the following chunk contains a header describing the instrument:

```
<bins-ck> -> bins( <DMUS_IO_INSTRUMENT> )
```

The instrument list can also contain <DMRF-list>, which is a reference to a DLS file. See Common Chunks.

Chord-map Form

The following notation shows the organization of the top-level chunk, or form, of a chord-map file:

```
RIFF( 'DMPR'
    <perh-ck> // Chord-map header chunk
    [ <guid-ck> ] // GUID chunk
    [ <vers-ck> ] // Version chunk
    [ <UNFO-list> ] // UNFO list
    <chdt-ck> // Chord data chunk
    <chpl-list> // Chord palette
    <cmap-list> // Chord graph
    <spsq-list> // Signpost list
)
```

Each of the items—with the exception of the GUID chunk, version chunk, and UNFO list—is required. See Common Chunks

The required chunks and their subchunks are as follows:

```
<perh-ck> -> perh( <DMUS_IO_CHORDMAP> )
```

This is the basic header information for a chord map.

```
<chdt-ck> -> chdt(
    <WORD> // Size of DMUS_IO_CHORDMAP_SUBCHORD
    <DMUS_IO_CHORDMAP_SUBCHORD>...
)
```

The <chdt-ck> chunk contains a **WORD**, indicating the number of bytes per subchord followed by an array of unique subchords. The subchord identifiers referred to in other parts of this file all correspond directly to an index into this array.

```
<chpl-list> -> LIST( 'chpl'
    <chrd-list>...
)
```

This list contains the chord palette. Currently 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 the chord's name and a list of identifiers for its subchords.

```
<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>. Currently a maximum of four chords is supported.

```
<cmap-list> -> LIST( 'cmap'
  <choe-list>...
)
```

The "cmap" list contains the entire chord connection graph for the chord map. The bulk of the data for the chord map resides in this chunk.

```
<choe-list> -> LIST( 'choe'
  <ch eh-ck> // Chord entry data
  <chrd-list> // Chord data; see above.
  <ncsq-ck> // Next chord list
)
```

The "choe" 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.

```
<ch eh-ck> -> ch eh( <DMUS_IO_CHORDENTRY> )
```

This is the chord entry header. The identifier in the structure is the identifier for the chord connection graph, not a subchord identifier.

```
<ncsq-ck> -> ncsq (
  <WORD> // Size of DMUS_IO_NEXTCHORD
  <DMUS_IO_NEXTCHORD>...
)
```

The "ncsq" chunk contains data that connects one chord in the connection graph to another. Each chord in the connection graph is represented by a 16-bit identifier.

```
<spsq-list> -> LIST( 'spsq'
  <spst-list>...
)
```


The "spsq" list contains data for each of the signposts.

```
<spsq-list> -> LIST( 'spsq'  
  <spsq-ck>  
  <chrd-list> // Chord data; see above.  
  [ <cade-list> ]  
)
```

The "spst" list contains data for a single signpost, consisting of a header, chord information, and optional cadence chords.

```
<spst-list> -> LIST( 'spst'  
  <chrd-list>...  
)
```

The "cade" list contains the chord information for cadence chords. Currently, there is support for up to two cadence chords in this list. Any additional chords or other information is ignored.

```
<spsq-ck> -> spsq(  
  <DMUS_IO_CHORDMAP_SIGNPOST>  
)
```

Finally, the "spsh" chunk contains the signpost data.

Segment Form

The following notation shows the organization of the top-level chunk of a segment file:

```
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  
  <trkl-list> // List of tracks  
  [<DMTG-form>] // Optional tool graph  
)
```

The individual chunks and their subchunks are as follows:

```
<segh-ck> -> segh( <DMUS_IO_SEGMENT_HEADER> )
```

This chunk contains the basic header information for a segment. For the next three chunks, see Common Chunks.

Next, comes the track list. Each track is encapsulated in a Track Form.

```
<trkl-list> -> LIST( 'trkl'  
  <DMTK-form>...
```

)

Finally, the segment form can contain a Tool Graph Form.

Style Form

The following notation shows the organization of the top-level chunk of a style file:

```
RIFF( 'DMST'
  <styh-ck>    // Style header chunk
  <guid-ck>    // Unique identifier
  [<UNFO-list>] // Name, author, copyright information, comments
  [<vers-ck>]  // Version chunk
  <part-list>... // List of parts in the style, used by patterns
  <pttn-list>... // List of patterns in the style
  <DMBD-form>... // List of bands in the style
  [<motf-list>] // List of motifs in the style
  [<prrf-list>] // List of chord-map references in the style
)
```

The individual chunks and their subchunks are as follows:

```
<styh-ck> -> styh( <DMUS_IO_STYLE> )
```

This chunk contains the basic header information for a style. For the next three chunks, see Common Chunks.

Next, comes a chunk for each musical part in the style:

```
<part-list> -> LIST('part'
  <prth-ck>    // Part header chunk
  [<UNFO-list>]
  [<note-ck>]  // List of notes in the part
  [<crve-ck>]  // List of curves in the part
)
```

The part list includes a header, an optional UNFO chunk, and a list of notes and curves, as shown in the following example. (For the UNFO list, see Common Chunks.)

```
<prth-ck> -> prth( <DMUS_IO_STYLEPART> )

<note-ck> -> note(
  < DWORD > // Size of DMUS_IO_STYLENOTE
  < DMUS_IO_STYLENOTE >...
)

<crve-ck> -> crve(
  < DWORD > // Size of DMUS_IO_STYLECURVE
```

```

    < DMUS_IO_STYLECURVE >...
)

```

After the part-list chunk comes the pattern-list chunk:

```

<pttn-list> -> LIST( 'pttn'
    <ptnh-ck>    // Pattern-header chunk
    <rhtm-ck>    // List of rhythms for chord matching
    [<UNFO-list>]
    [<mtfs-ck>]  // Motif settings chunk
    <pref-list>... // List of part reference IDs
)

```

The pattern list consists of the following subchunks. (For the optional UNFO list, see Common Chunks.)

```

<ptnh-ck> -> ptnh(
    < DMUS_IO_PATTERN >
)

```

```

<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**.

```

<mtfs-ck> -> mtfs(
    < DMUS_IO_MOTIFSETTINGS >
)

```

```

<pref-list> -> LIST( 'pref'
    <prfc-ck> // Part reference chunk
)

```

The "pref" chunk in turn consists of an array of part references:

```

<prfc-ck> -> prfc(
    < DMUS_IO_PARTREF >
)

```

The last chunk in the style form is the list of chord-map references:

```

<prrf-list> -> LIST( 'prrf'
    <DMRF-list>...
)

```

For more information on <DMRF-list>, see Common Chunks.

Tool Form

The tool form contains information about tools. Tools can be embedded in a Tool Graph Form or stored as separate files.

```
<DMTL-form> -> RIFF( 'DMTL'
    <tolh-ck>
    [<guid-ck>] // GUID for tool object instance
    [<vers-ck>] // Version information
    [<UNFO-list>] // Name, author, copyright information, comments
    [<data>] // Tool data
)
```

The tool header chunk is as follows:

```
<tolh-ck> -> tolh( <DMUS_IO_TOOL_HEADER> )
```

For the next three elements, which are optional, see Common Chunks.

The <data> element consists of a chunk of the type identified in the **DMUS_IO_TOOL_HEADER**. The format of this chunk depends on the definition of the tool. It can be a list or a chunk.

Tool Graph Form

A tool-graph chunk can occur either as a top-level form or as a subchunk of a segment form.

```
RIFF( 'DMTG'
    [<guid-ck>] // GUID for tool graph
    [<vers-ck>] // Optional version information
    [<UNFO-list>] // Name, author, copyright information, comments
    <toll-list> // List of tools
)
```

For the first three elements, which are optional, see Common Chunks.

The main and only required part of the tool-graph chunk is the tool list:

```
<toll-list> -> LIST( 'toll'
    <DMTL-form>...
)
```

For more information on the <DMTL-form> chunk, see Tool Form.

Track Form

The track form contains information about a single track. It can be embedded in a Segment Form or stored in its own file.

```

<DMTK-form> -> RIFF( 'DMTK'
    <trkh-ck>
    [<guid-ck>] // GUID for track object instance
    [<vers-ck>] // Version information
    [<UNFO-list>] // Name, author, copyright information, comments
    [<data>] // Track data
)

```

The subchunks of the form are as follows:

```

<trkh-ck> -> trkh(
    <DMUS_IO_TRACK_HEADER>
)

```

This chunk contains the basic header information for a track.

For the next three elements, which are optional, see Common Chunks.

The last element in the track form is the data for the track. The chunk type used for the data is identified in the **DMUS_IO_TRACK_HEADER** structure. The following standard track chunks are defined:

- Band Track Form
- Chord Track List
- Chord-map Track List
- Command Track Chunk
- Mute Track Chunk
- Sequence Track List
- Signpost Track Chunk
- Style Track List
- Sysex Track Chunk
- Tempo Track Chunk
- Time Signature Track Chunk

Band Track Form

The band track form can be a top-level form but is also found as the data part of a Track Form. It is organized as follows:

```

RIFF( 'DMBT'
    [<bdth-ck>] // Band track header
    [<guid-ck>] // GUID for band track
    [<vers-ck>] // Optional version information
    [<UNFO-list>] // Name, author, copyright information, comments
    <lbdl-list> // List of band lists
)

```

The subchunks of the form are as follows:

```
<bnth-ck> -> bdth( <DMUS_IO_BAND_TRACK_HEADER> )
```

This optional chunk contains header information for a band track. The only data in the structure is a flag for automatic downloading.

For the next three elements, which are optional, see Common Chunks.

The last chunk contains one or more bands:

```
<lbdl-list> -> LIST( 'lbdl'  
    <lbdn-list>...  
    )
```

Each band is encapsulated in a list of the following type:

```
<lbdn-list> -> LIST( 'lbdn'  
    <bdlh-ck>  
    <DMBD-form>  
    )
```

The band list begins with a header:

```
<bdlh-ck> -> ( <DMUS_IO_BAND_ITEM_HEADER> )
```

The header is followed by a Band Form containing information about the instruments in the band.

Chord Track List

The chord track list contains chord data for a Track Form. It is organized as follows:

```
<cord-list> -> LIST( 'cord'  
    <crdh-ck>    // Header  
    <crdb-ck>    // Chord body chunk  
    )
```

The subchunks are as follows:

```
<crdh-ck> -> crdh ( <DWORD> )
```

The header is a **DWORD** containing the chord root in the upper 8 bits and the scale in the lower 24 bits. For an explanation of what these bits represent, see

DMUS_IO_SUBCHORD.

The body of data for the chord track list consists of information about a chord change and the component subchords:

```
<crdb-ck> -> crdb(  
    <DWORD>          // Size of DMUS_IO_CHORD
```

```
<DMUS_IO_CHORD>
<DWORD>          // Number of subchords
<DWORD>          // Size of DMUS_IO_SUBCHORD
<DMUS_IO_SUBCHORD>...
)
```

Chord-map Track List

The chord-map track list contains data for a Track Form. It is organized as follows:

```
<pfr-list> -> LIST('pfr'
  <pfrf-list>...
)
```

The data consists of one or more lists containing time stamps and references to chord maps:

```
<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 Common Chunks.

Command Track Chunk

The command track chunk contains data for a Track Form. It is organized as follows:

```
<cmnd-ck> -> cmnd(
  <DWORD>  //Size of DMUS_IO_COMMAND
  <DMUS_IO_COMMAND>...
)
```

Mute Track Chunk

The mute track chunk contains data for a Track Form. It is organized as follows:

```
<mute-ck> -> mute(
  <DWORD>  //Size of DMUS_IO_MUTE
  <DMUS_IO_MUTE>...
)
```

Sequence Track List

The sequence track list contains data for a Track Form. It is organized as follows:

```
<seqt-list> -> LIST('seqt'  
    <evtl-ck>  
    <curl-ckt>  
)
```

The list contains two chunks, one for sequence items and one for curve items:

```
<evtl-ck> -> evtl(  
    <DWORD>    // Size of DMUS_IO_SEQ_ITEM  
    <DMUS_IO_SEQ_ITEM>...  
)  
  
<curl-ck> -> curl(  
    <DWORD>    // Size of DMUS_IO_CURVE_ITEM  
    <DMUS_IO_CURVE_ITEM>...  
)
```

Signpost Track Chunk

The signpost track chunk contains data for a Track Form. It is organized as follows:

```
<sgnp-list> -> sgnp(  
    <DWORD>    // Size of DMUS_IO_SIGNPOST  
    <DMUS_IO_SIGNPOST>...  
)
```

Style Track List

The style track list contains data for a Track Form. It is organized as follows:

```
<str-list> -> LIST('strf'  
    <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>  
)
```

For information on <DMRF-list>, see Common Chunks.

Sysex Track Chunk

The sysex track chunk contains data for a Track Form. It is an array of sysex items, as shown in the following notation:

```
<syex-ck> -> syex(  
    <DMUS_IO_SYSEX_ITEM>
```



```
<BYTE>... // Data  
)...
```

Tempo Track Chunk

The tempo track chunk contains data for a Track Form. It is organized as follows:

```
<tetr-ck> -> tetr(  
    <DWORD> // Size of DMUS_IO_TEMPO_ITEM  
    <DMUS_IO_TEMPO_ITEM>...  
)
```

Time Signature Track Chunk

The time signature track chunk contains data for a Track Form. It is organized as follows:

```
<tims-ck> -> tims(  
    <DWORD> // Size of DMUS_IO_TIMESIGNATURE_ITEM  
    <DMUS_IO_TIMESIGNATURE_ITEM>...  
)
```

Using Downloadable Sounds

This section covers the use of downloadable sounds (DLS) in DirectMusic. For an architectural overview of DLS, see Downloadable Sounds.

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

[C++]

For specialized DirectMusic applications that do their own DLS management, two steps must be taken: loading the instrument collection and downloading instrument data to a port.

These steps are covered in the following sections:

- Loading a Collection
- Working with Instruments
- Playing a MIDI File with Custom Instruments

Applications that allow the editing of instruments and collections must work with DLS data at an even lower level. This topic is covered in the following section:

- Low-Level DLS

[\[Visual Basic\]](#)

More information about working with collections is presented in the following topics:

- Loading a Collection
 - Playing a MIDI File with Custom Instruments
-

Loading a Collection

[\[C++\]](#)

The simplest way to load a collection from a file is to use the **IDirectMusicLoader** interface. (For more information, see Loading an Object from a File.) Once you have obtained a pointer to the **IDirectMusicCollection** interface, you have access to all the instruments in the collection. At this point, though, none of them have actually been downloaded to a port. In fact, no instrument data is loaded into memory until it is needed.

The collection is a COM object that supports the **IDirectMusicCollection**, **IDirectMusicObject**, and **IPersistStream** interfaces. **IPersistStream** is a standard COM interface providing methods for saving and loading objects that use a simple serial stream for their storage needs.

IDirectMusicCollection does not load the entire collection when **IPersistStream::Load** is called. Typically, objects supporting **IPersistStream** load all the persistent data in the stream and do not use the **IStream** pointer outside the **Load** method. However, supporting **IPersistStream::Load** in this manner in **IDirectMusicCollection** would mean that the entire DLS collection would have to be loaded into memory even if only a single instrument in the collection was to be used. Instead, **IDirectMusicCollection** saves the **IStream** pointer, and later uses it to load only the data for instruments that are downloaded to a DirectMusic port. **IDirectMusicCollection** assumes that the data stream provided through the **IPersistStream** interface is in the DLS file format.

The following code example does manually what the **IDirectMusicLoader::GetObject** method does automatically: it creates a collection object and loads a stream into it:

```
HRESULT myLoadCollectionFromStream(  
    IStream *pIStream,    // Stream created from a file  
    IDirectMusicCollection **ppICollection )  
  
{  
    HRESULT hr;  
  
    hr = CoCreateInstance(CLSID_DirectMusicCollection,
```

```

        NULL,
        CLSCTX_INPROC,
        IID_IDirectMusicCollection,
        (void **)pplCollection);
if (SUCCEEDED(hr))
{
    IPersistStream* plPersistStream;
    hr = *pplCollection->QueryInterface(
        IID_IPersistStream, (void**)&plPersistStream);
    if (SUCCEEDED(hr))
    {
        hr = plPersistStream->Load(plStream);
        plPersistStream->Release();
    }
}
return hr;
}

```

The following code example uses the DirectMusicLoader to load the collection by file name:

```

HRESULT myLoadCollectionByName(
    IDirectMusicLoader *plLoader,
    char *pszFileName,
    IDirectMusicCollection **pplCollection)
{
    HRESULT hr;
    DMUS_OBJECTDESC Desc;           // Descriptor

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

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

    hr = plLoader->GetObject(&Desc,
        IID_IDirectMusicCollection,
        (void **) pplCollection);
    return hr;
}

```

To load the standard GM/GS set, pass `GUID_DefaultGMCollection` to the loader in the **guidObject** member of the **DMUS_OBJECTDESC** structure. If you intend to use the loader to access this object more than once, make sure that caching is enabled (as it is by default) so that you do not create another copy of the GM collection each time that you request it.

Note

The GM/GS Sound Set cannot be altered. For more information, see the Copyright Warning.

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

```
HRESULT myGetGMCollection(
    IDirectMusicLoader *pLoader,
    IDirectMusicCollection **ppICollection)
{
    HRESULT hr;
    DMUS_OBJECTDESC desc;

    desc.dwSize = sizeof(DMUS_OBJECTDESC);
    desc.guidClass = CLSID_DirectMusicCollection;
    desc.guidObject = GUID_DefaultGMCollection;
    desc.dwValidData = (DMUS_OBJ_CLASS | DMUS_OBJ_OBJECT);
    hr = pLoader->GetObject(&desc, IID_IDirectMusicCollection,
        (void **) ppICollection);
    return hr;
}
```

[Visual Basic]

To load an instrument collection, call the **DirectMusicLoader.LoadCollection** or the **DirectMusicLoader.LoadCollectionFromResource** method. Each of these methods returns a **DirectMusicCollection** object. This object has no methods, and its sole function is as a parameter to **DirectMusicSegment.ConnectToCollection**. Call that method to associate the collection with a segment, and then download the instruments by calling **DirectMusicSegment.Download**.

These steps are necessary only when you want to use a collection other than the default one. Normally, when you call **DirectMusicSegment.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 contain a custom collection or is a MIDI file. When you download a band, all DLS data needed by the instruments in that band is downloaded. See Using Bands.

Working with Instruments

[Visual Basic]

This topic pertains only to applications written in C++. DirectX for Visual Basic does not allow applications to work with individual instruments from a collection.

[C++]

Once a **DirectMusicCollection** is created and loaded from a collection file, you can retrieve the patch number and name of all the available instruments by using the **IDirectMusicCollection::EnumInstrument** method.

The following code example enumerates all instruments in a collection and displays their names and patch numbers.

```
void myListInstruments(
    IDirectMusicCollection *pCollection)

{
    HRESULT hr = S_OK;
    DWORD dwPatch;
    WCHAR wszName[MAX_PATH];
    DWORD dwIndex;
    for (dwIndex = 0; hr == S_OK; dwIndex++)
    {
        hr = pCollection->EnumInstrument(
            dwIndex, &dwPatch, wszName, MAX_PATH);
        if (hr == S_OK)
        {
            printf("Patch %lx is %S\n",dwPatch,wszName);
        }
    }
}
```

You can obtain a pointer to a specific instrument by passing its patch number to the **IDirectMusicCollection::GetInstrument** method.

After obtaining an instrument, you can change its patch number by using the **IDirectMusicInstrument::SetPatch** method.

Loading a collection and retrieving the instruments is only the first step in making the instruments available. You must next download them to the port.

To download an instrument to a port, pass an **IDirectMusicInstrument** interface pointer to the **IDirectMusicPort::DownloadInstrument** method. This method makes the DLS data available on the port; it does not actually associate the instrument with any particular performance.

You can also download an instrument by using the **IDirectMusicPerformance::DownloadInstrument** method. In addition to

downloading the DLS data, this method assigns the instrument to a particular performance channel.

To save memory, only waves and articulation required for given ranges of notes are downloaded. For example, for a bassoon you might specify that only data for the note range from low C through middle B is to be downloaded. Only the data for the regions falling within that range would be 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 **IDirectMusicPort::DownloadInstrument** method. Typically, only a single range of notes is specified, but you can specify multiple ranges. If you pass NULL instead of a pointer to an array, the data for all regions is downloaded.

```
HRESULT myDownload(
    IDirectMusicCollection *pCollection, // DLS collection
    IDirectMusicPort *pPort,           // Destination port
    IDirectMusicDownloadedInstrument **ppDLInstrument,
    DWORD dwPatch,                     // Requested instrument
    DWORD dwLowNote,                   // Low note of range
    DWORD dwHighNote)                  // High note of range
{
    HRESULT hr;
    IDirectMusicInstrument* pInstrument;
    hr = pCollection->GetInstrument(dwPatch, &pInstrument);
    if (SUCCEEDED(hr))
    {
        DMUS_NOTERANGE NoteRange[1]; // Optional note range
        NoteRange[0].dwLowNote = dwLowNote;
        NoteRange[0].dwHighNote = dwHighNote;
        hr = pPort->DownloadInstrument(pInstrument,
            ppDLInstrument,
            NoteRange, // Array of ranges
            1);        // Number of elements in the array
        pInstrument->Release();
    }
    return hr;
}
```

The **DownloadInstrument** method returns a pointer to the **IDirectMusicDownloadedInstrument** interface. This pointer has just one purpose: to identify the instrument in a subsequent call to the **IDirectMusicPort::UnloadInstrument** method, which unloads the instance of the instrument on a particular port. (The **DirectMusicCollection** is not bound to any specific port. You can download different instruments to different ports or download a single instrument to multiple ports.)

The following code example downloads an instrument, and then unloads it, which illustrates how the **IDirectMusicDownloadedInstrument** pointer can be used:

```
HRESULT myFickleDownload(
    IDirectMusicInstrument* pInstrument,
    IDirectMusicPort *pPort,
    DWORD dwPatch)

{
    HRESULT hr;
    IDirectMusicDownloadedInstrument * pDLInstrument;
    hr = pPort->DownloadInstrument(
        pInstrument, &pDLInstrument,
        NULL, 0);
    if (SUCCEEDED(hr))
    {
        pPort->UnloadInstrument(pDLInstrument);
        pDLInstrument->Release();
    }
    return hr;
}
```

The **IDirectMusicBand::Download** method automates the downloading of all instruments in a band. You supply a pointer to a performance, and the method downloads each instrument to the appropriate port attached to that performance.

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 play a MIDI file, using instruments from any collection.

[C++]

To do so, first load the collection, as described in Loading a Collection, obtaining a pointer to the **IDirectMusicCollection** interface. Then call

IDirectMusicSegment::SetParam on the MIDI segment to establish a connection between the segment and the collection.

The following code example shows how the connection is made:

```
/* Assume that pSegment was created from a MIDI file
   and that pLoadedCollection is a valid IDirectMusicCollection
   pointer. */

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

As with any other collection, instruments to be used in playing a MIDI file must be downloaded before the segment is played, unless automatic downloading has been enabled. (For more information on automatic downloading, see [Setting and Retrieving Global Parameters and Downloading and Unloading Bands](#).) If instruments are not being downloaded automatically, you must download them by calling **IDirectMusicSegment::SetParam**, as in the following code example:

```
/* pSegment is an IDirectMusicSegment pointer, and pPerformance
   is a valid pointer to IDirectMusicPerformance. */

pSegment->SetParam(GUID_Download, 0xFFFFFFFF, 0, 0, (void *) pPerformance);
```

For more information on downloading by using **SetParam**, see [Setting and Retrieving Track Parameters](#).

[\[Visual Basic\]](#)

To do so, load the collection, associate it with the segment, based on the MIDI file, and download the collection as you would with any other segment. For more information, see [Loading a Collection](#).

Note

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.

Low-Level DLS

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++. DirectX for Visual Basic does not support low-level manipulation of DLS data.

[\[C++\]](#)

If you are writing a DirectMusic application that edits DLS collections, you must be able to download instrument data to the synthesizer without encapsulating it in a `DirectMusicInstrument` object.

Working with DLS data requires knowledge of the DLS specification and file structure. For detailed information on these topics, contact the MIDI Manufacturers Association.

To download raw instrument data, you must first get a pointer to the **IDirectMusicPortDownload** interface, as shown in the following code example, in which it is assumed that *pIPort* is a valid pointer to an **IDirectMusicPort** interface:

```
IDirectMusicPortDownload **pplDownloadPort;

HRESULT hr = pIPort->QueryInterface(IID_IDirectMusicPortDownload,
    (void **) pplDownloadPort);
```

If the **HRESULT** is not **S_OK**, the port does not support DLS downloading.

Next, identify the buffers that must be prepared and downloaded. To send an instrument to the synthesizer, create one instrument buffer that represents the entire instrument definition with all the regions and articulations, and a series of wave buffers, one for each wave that the instrument refers to for its regions.

Each buffer must be tagged with a unique identifier. Identifiers are used to resolve linkages between buffers, in particular the links between regions and waves. Tally the number of buffers that you need to download, and call

IDirectMusicPortDownload::GetDLId to allocate a range of identifiers. For example, if you are downloading an instrument with three waves, you must download four buffers in all. Therefore, request a set of four identifiers.

For each buffer, calculate the size needed; then call

IDirectMusicPortDownload::AllocateBuffer to allocate it. This method returns an **IDirectMusicDownload** interface representing the buffer. Call **IDirectMusicDownload::GetBuffer** to access the memory.

Note

There are two methods called **GetBuffer**:

IDirectMusicPortDownload::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 each buffer. 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_INSTRUMENT** for an instrument or **DMUS_DOWNLOADINFO_WAVE** for a wave.
- Set the **dwDLId** member to one of the unique identifiers that you obtained by using **IDirectMusicPortDownload::GetDLId**.
- Set the **dwNumOffsetTableEntries** member to the number of entries in the **DMUS_OFFSETTABLE** structure.
- Set the **cbSize** member to the size of the download chunk, including **DMUS_DOWNLOADINFO** and **DMUS_OFFSETTABLE**.

The **DMUS_DOWNLOADINFO** structure is always followed by a **DMUS_OFFSETTABLE** structure. This offset table is used to manage all links within the data. Whenever a structure in the data refers to another structure, it addresses it with an integer index, instead of a pointer. For every structure within the data that can be referred to, there is a unique index. The **DMUS_OFFSETTABLE** translates this integer index into a byte offset into the data.

The instrument or wave data follows the **DMUS_OFFSETTABLE**. If the download is an instrument, the data starts with the **DMUS_INSTRUMENT** structure. Otherwise, it starts with the **DMUS_WAVE** structure.

The instrument data that follows the **DMUS_INSTRUMENT** structure is organized in the following structures:

- **DMUS_ARTICPARAMS**
- **DMUS ARTICULATION**
- **DMUS_COPYRIGHT**
- **DMUS_EXTENSIONCHUNK**
- **DMUS_INSTRUMENT**
- **DMUS_NOTERANGE**
- **DMUS_REGION**

The wave data pointed to by the **DMUS_WAVE** structure is organized in a **DMUS_WAVEDATA** structure.

When the buffers are all ready, download them by using **IDirectMusicPortDownload::Download**. Download the wave buffers first so that they are in place and can be referred to when the instrument is downloaded.

Once the buffers have been downloaded, the synthesizer is ready to play the instrument. The memory in the buffer is no longer accessible.

Later, when done playing the instrument, unload the buffers, and release them. First, unload the instrument buffer, then all the wave buffers. To unload, call **IDirectMusicPortDownload::Unload** and pass it the **IDirectMusicDownload** objects. Then, release each buffer with a call to **IDirectMusicDownload::Release**.

To update an instrument that has already been downloaded, you cannot write over the previously downloaded buffer. Instead, replace the instrument, but not the waves. To do this, call **IDirectMusicPortDownload::AllocateBuffer** to allocate a new **IDirectMusicDownload** interface with a buffer of the correct size. Be sure to generate a new identifier for the buffer with a call to **IDirectMusicPortDownload::GetDLId**. Write the new articulation information into the buffer; then download it. Then unload the previously downloaded buffer with a call to **IDirectMusicPortDownload::Unload**.

To update a wave buffer, take one extra step. Create both a new wave buffer and an updated instrument buffer that refers to it. Download the new wave, then the new instrument. Then unload the old instrument, followed by the old wave.

Playing Music

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

The following topics are discussed:

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

Creating the Performance

[C++]

The manager of music playback is the performance object, which does most of the work of getting music from the source to the output buffer. It performs the following tasks:

- Adding ports
- Assigning instruments to channels
- Downloading instrument data to the synthesizer
- Playing and stopping segments
- Dispatching messages
- Managing tools and timing

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

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

```
IDirectMusicPerformance* pPerf;

if (FAILED(CoCreateInstance(
    CLSID_DirectMusicPerformance,
    NULL,
    CLSCTX_INPROC,
    IID_IDirectMusicPerformance,
    (void**)&pPerf
```

```
    )))  
{  
    pPerf = NULL;  
}
```

Once the performance is created, it must be initialized. An important part of initialization is the creation of a **DirectMusic** object. You can create a **DirectMusic** object by passing **CLSID_DirectMusic** to **CoCreateInstance**, and then passing the **IDirectMusic** interface pointer to **IDirectMusicPerformance::Init**. However, in most cases, it is more convenient to have **Init** create the **DirectMusic** object. You can also choose whether or not to retrieve a pointer to the **IDirectMusic** interface, depending on how much control you need over ports and the master clock. If you intend to use only the default synthesizer and the default master clock, you probably do not need access to the methods of **IDirectMusic**; in this case, you would pass **NULL** to **Init**.

The following code example initializes the performance, retrieves a pointer to **IDirectMusic**, and creates an **IDirectSound** interface initialized with the application window handle:

```
IDirectMusic* pDirectMusic;  
  
if (SUCCEEDED(pPerf->Init(&pDirectMusic,  
    NULL,    // Create a DirectSound object.  
    hWnd     // Application window handle  
    )))  
{  
    // Performance initialized  
}
```

[Visual Basic]

The manager of music playback is the performance object, which does most of the work of getting music from the source to the output buffer. It performs the following tasks:

- Enumerating ports and setting the active port
- Playing and stopping segments
- Dispatching messages
- Managing the timing

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

The follow code example, in which *objDX* is a **DirectX7** object, creates a performance with its own **DirectSound** object, initializes it, and sets it to use the default port with one channel group:

```
Dim objDMPPerformance as DirectMusicPerformance

Set objDMPPerformance = objDX.DirectMusicPerformanceCreate
Call objDMPPerformance.Init(Nothing, 0)
Call objDMPPerformance.SetPort(-1, 1)
```

For more information on setting up the performance, see Integrating DirectMusic and DirectSound and Using Ports.

Segments

[C++]

The basic chunk of data in DirectMusic is called a segment. A segment is represented by an **IDirectMusicSegment** interface. You can create a segment in any of the following ways:

- Load a file or resource object that supports the **IDirectMusicSegment** interface. For more information, see DirectMusic Loader.
- Get a motif from a style by using the **IDirectMusicStyle::GetMotif** method.
- Use methods of the **IDirectMusicComposer** interface to create a composition or transition at run time. See Overview of Programming for Composition and Using Transitions.
- Make a copy of an existing segment by using the **IDirectMusicSegment::Clone** method.
- Construct a segment from existing tracks. Create a segment object by calling **CoCreateInstance**, and then add tracks by calling **IDirectMusicSegment::InsertTrack**.
- Use the **IDirectMusicBand::CreateSegment** method. This creates a special type of secondary segment that is used only for making band changes. See Making Band Changes Programmatically.

Each segment consists of one or more *tracks*, each represented by an **IDirectMusicTrack** 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 do not need to use this interface because the tracks are managed through the segment object. For more information, see Tracks.

[Visual Basic]

The basic chunk of data in DirectMusic is called a segment. A segment is represented by a **DirectMusicSegment** object. You can create a segment in any of the following ways:

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

Segments can serve different purposes. The following are the kinds of segments that you are most likely to use:

- Primary segment. A piece of music such as a MIDI file, a segment authored in DirectMusic Producer, or a segment formed at run time by the composer object.
 - Motif segment. A short piece of music to be played over the primary segment. Motifs are part of a style object, but you can also create short secondary segments from other sources and play them over the primary segment like motifs.
 - Band segment. A set of instruments and instrument settings for the various channels in the performance. The application can play a band segment to execute changes in the band performing the music.
 - Template segment. A guide to chord progressions, groove levels, and embellishments, used in conjunction with a style and chord map to compose music at run time. Unlike other segments, a template segment is never played directly by an application; instead, it is passed to the composer object to be used in creating a musical segment.
-

[\[C++\]](#)

The playback of segments is controlled by the **IDirectMusicPerformance** interface and begins with a call to **IDirectMusicPerformance::PlaySegment**. An instance of a playing segment is represented by an **IDirectMusicSegmentState** interface, which can be used to retrieve information about the current state of the segment.

[\[Visual Basic\]](#)

The playback of segments is controlled by the **DirectMusicPerformance** object and begins with a call to **DirectMusicPerformance.PlaySegment**. An instance of a playing segment is represented by a **DirectMusicSegmentState** object, which can be used to retrieve information about the current state of the segment.

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

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

Secondary segments do not normally alter the performance of the primary segment. For example, a secondary segment can be based on a different style without affecting the style of the primary segment.

[C++]

However, a secondary segment can be designated as a *control segment*, in which case it takes over the task of responding to **IDirectMusicPerformance::SetParam** and **IDirectMusicPerformance::GetParam** calls, which normally go to tracks in the primary segment. Thus a control segment might control the current chord or groove level. For more information, see Tracks.

For more information on playing secondary segments as control segments, see **DMUS_SEGF_FLAGS**.

[Visual Basic]

However, a secondary segment can be designated a *control segment*, in which case it takes over the task of responding to certain calls such as **DirectMusicPerformance.GetCommand** and would also control some aspects of the music, such as the current chord or groove level.

For more information on playing secondary segments as control segments, see **CONST_DMUS_SEGF_FLAGS**.

For more information on segment playback, see Segment Timing.

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 a piece of music.

[Visual Basic]

DirectX for Visual Basic does not provide applications with access to individual tracks. All playback is handled at the segment level.

[C++]

Each track is represented by an **IDirectMusicTrack** interface. The methods of this interface are called by the performance, and most applications do not need to use them directly.

When an application calls **IDirectMusicPerformance::PlaySegment**, DirectMusic calls the **IDirectMusicTrack::Play** method on the segment's tracks. Most tracks respond by immediately generating time-stamped messages containing data that is valid for the part of the segment that is being played. These messages are placed in a queue. (See Message Creation and Delivery for more information about what happens after that.)

A few tracks do not actively generate messages in response to **IDirectMusicTrack::Play**, but instead do their work by responding to requests for information that come from the performance or other tracks in the form of a **GetParam** call. (See Setting and Retrieving Track Parameters.)

The following list shows the standard track types implemented by DirectMusic, with a brief description of each one. For a list of the methods supported by the standard tracks, see the reference for **IDirectMusicTrack**.

- **Band**. Downloads DLS data to the performance. Sends messages of type **DMUS_PATCH_PMSG**, **DMUS_TRANSPOSE_PMSG**, **DMUS_CHANNEL_PRIORITY_PMSG**, and **DMUS_MIDI_PMSG** (for volume and pan). Used in segments based on MIDI files and styles.
- **Chord**. Used to convert music values (as stored in patterns) to MIDI values. Sends messages of type **DMUS_NOTIFICATION_PMSG** (for **GUID_NOTIFICATION_CHORD** notifications).
- **Chord map**. Used in template segments to compose chord tracks.
- **Command**. Used in template segments to compose chord tracks, and in style segments to determine which patterns is played. Sends messages of type **DMUS_NOTIFICATION_PMSG** (for **GUID_NOTIFICATION_COMMAND** notifications).
- **Motif**. Used to play motifs to accompany other segments. Sends messages of type **DMUS_CURVE_PMSG**, **DMUS_NOTE_PMSG**, and **DMUS_NOTIFICATION_PMSG** (for **GUID_NOTIFICATION_MEASUREANDBEAT** notifications).
- **Mute**. Used with either style-based or MIDI-based segments. Allows PChannels to be remapped to other PChannels or to be muted.
- **Sequence**. Sends sequence messages of type **DMUS_NOTE_PMSG** and **DMUS_MIDI_PMSG**. Used in segments based on MIDI files. Also sends messages of type **DMUS_CURVE_PMSG** for segments saved in the .sgt format.

- **Signpost.** Used in template segments to compose chord tracks.
 - **Style.** Fundamental track for segments based on styles. Sends messages of type **DMUS_TIMESIG_PMSG**, **DMUS_CURVE_PMSG**, **DMUS_NOTE_PMSG**, and **DMUS_NOTIFICATION_PMSG** (for **GUID_NOTIFICATION_MEASUREANDBEAT** notifications).
 - **SysEx.** Sends system-exclusive messages of type **DMUS_SYSEX_PMSG**. Used in segments based on MIDI files.
 - **Tempo.** Sets the tempo of the performance by sending messages of type **DMUS_TEMPO_PMSG**.
 - **Time Signature.** Sends messages of type **DMUS_TIMESIG_PMSG** as well as **GUID_NOTIFICATION_MEASUREANDBEAT** notifications. Used in segments based on MIDI files.
-

Using Bands

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

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

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

[C++]

You can retrieve a pointer to the default band by using the **IDirectMusicStyle::GetDefaultBand** method.

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

[\[Visual Basic\]](#)

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

Loading a band from its own file is like loading any other object in DirectMusic. For more information, see DirectMusic Loader.

More information about bands is contained in the following topics:

- Downloading and Unloading Bands
- Making Band Changes Programmatically
- Ensuring Timely Band Changes

Downloading and Unloading Bands

Before a band can be used, the instruments that it refers to must be downloaded to the performance. This step maps the instruments to PChannels and downloads the DLS data to the port.

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

Note

Automatic downloading should be used only when the timing of segment starts is not critical. Repeated loading and unloading of instruments takes time and can cause serious degradation of performance in complex musical environments. Be aware also that automatic unloading, which is part of the automatic downloading mechanism, can 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. Therefore, there is no instrument on that channel, and the channel plays silence.

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

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

- Automatically Downloading Bands
- Manually Downloading Bands
- Unloading Bands

Automatically Downloading Bands

[C++]

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

- Call the **IDirectMusicPerformance::SetParam** method for the GUID_PerfAutoDownload parameter. See Setting and Retrieving Global Parameters and the following example.
- Call the **IDirectMusicSegment::SetParam** or the **IDirectMusicTrack::SetParam** method for the GUID_Enable_Auto_Download parameter. See Setting and Retrieving Track Parameters.

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

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

[Visual Basic]

You can turn on automatic downloading of bands for the entire performance by using the **DirectMusicPerformance.SetMasterAutoDownload** method, or for an individual segment by using the **DirectMusicSegment.SetAutoDownloadEnable** method.

Manually Downloading Bands

[C++]

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

- Obtain an **IDirectMusicBand** interface from a loaded object, and call the **IDirectMusicBand::Download** method. (See the following example.) This is the best way to download the band if you want to unload it after any segments using the band have been released.
- Call the **IDirectMusicSegment::SetParam** method with the GUID_Download parameter to download the band in the segment's first band track. You can also use **IDirectMusicPerformance::SetParam** to set this parameter on the primary

segment, or **IDirectMusicTrack::SetParam** to set it directly on a band track.
See Setting and Retrieving Track Parameters.

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 down to the synthesizer.

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

```
HRESULT myDownloadBand(
    IDirectMusicLoader *pLoader,    // Loader interface
    IDirectMusicBand **ppBand,     // To retrieve pointer
    IDirectMusicPerformance *pPerf, // Performance to use band
    WCHAR *pwszFile)              // File to load

{
    HRESULT hr;
    DMUS_OBJECTDESC Desc;          // Descriptor

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

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

    hr = pLoader->GetObject(&Desc, IID_IDirectMusicBand,
        (void **) ppBand);
    if (SUCCEEDED(hr))
    {

        // Download the band via the performance.
        hr = (*ppBand)->Download(pPerf);

    }
    return hr;
}
```

[\[Visual Basic\]](#)

You can manually download a band by obtaining a **DirectMusicBand** object from a file, a resource, or a **DirectMusicStyle** object, and then using its **DirectMusicBand.Download** method.

Unloading Bands

[C++]

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

- Call the **IDirectMusicBand::Unload** method.
- Call the **IDirectMusicSegment::SetParam** method for the `GUID_Unload` parameter to unload the band in the segment's band track. You can also use **IDirectMusicPerformance::SetParam** to set this parameter on the primary segment, or **IDirectMusicTrack::SetParam** to set it directly on the band track. See Setting and Retrieving Track Parameters. For an example, see Tutorial 1, Step 6: Shut Down DirectMusic.

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

[Visual Basic]

Bands take up memory, so they should be unloaded when they are no longer in use. If you have enabled automatic downloading of bands, the bands associated with a segment are unloaded automatically when the segment ends. Otherwise, you can manually unload a band by using the **DirectMusicBand.Unload** method.

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

Making Band Changes Programmatically

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

[C++]

First, create a secondary segment with a call to the **IDirectMusicBand::CreateSegment** method, and then play that segment by calling **IDirectMusicPerformance::PlaySegment**. Typically, you would use `DMUS_SEGF_MEASURE` or `DMUS_SEGF_GRID` (see **DMUS_SEGF_FLAGS**) in the *dwFlags* parameter to ensure that the band change takes effect on an appropriate boundary.

The following code example creates a segment from a band and plays it:

/ It is presumed that automatic downloading is turned on or that the application has called pBand->Download. */*

```

HRESULT myPlayBand(
    IDirectMusicBand *pBand,      // Pointer to a band object
    IDirectMusicPerformance *pPerf, // Performance to use the band
    REFERENCE_TIME rfTime,        // Time to play at
    DWORD dwFlags)                // Performance flags

{
    IDirectMusicSegment *pSegment;
    HRESULT hr = pBand->CreateSegment(&pSegment);
    if (SUCCEEDED(hr))
    {
        hr = pPerf->PlaySegment(pSegment,
                                dwFlags | DMUS_SEGF_SECONDARY,
                                rfTime,
                                NULL);
        pSegment->Release();
    }
    return hr;
}

```

[Visual Basic]

First, create a segment by using **DirectMusicBand.CreateSegment**, and then play that segment by calling **DirectMusicPerformance.PlaySegment**. Typically, you would use **DMUS_SEGF_MEASURE** or **DMUS_SEGF_GRID** (see **CONST_DMUS_SEGF_FLAGS**) in the *lFlags* parameter to ensure that the band change takes effect on an appropriate boundary.

A performance can be playing instruments from more than one band at a time. For example, suppose your application is playing a primary segment using one band, and then plays a motif from a style that has a different band. As long as the instruments in the first band are mapped to different PChannels than the instruments in the second, no conflict arises. However, motif segments do not normally have their own band tracks, so you might get silence from the motif's PChannels unless you first create a band segment and play it. (It is possible to add a band track to a motif segment, but creating a separate band segment is easier.)

Ensuring Timely Band Changes

A consideration in playing band segments is the randomness in the timing of notes played by a style track. For instance, a note that is on measure 1, beat 1 might actually

play somewhat earlier or later than the actual beat boundary. The band segment is not aware of this; therefore, some of the notes might play with the incorrect instrument.

[C++]

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

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

REFERENCE_TIME rtResolved;
MUSIC_TIME mtResolved;

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

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

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

/* Play the style segment on the measure boundary. */

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

[Visual Basic]

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

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

Dim ctResolved As Long
Dim mtResolved As Long

ctResolved = perf.GetResolvedTime(ctResolved, DMUS_SEGF_MEASURE)
```

```
mtResolved = perf.ClockToMusicTime(ctResolved)
```

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

```
mtResolved = mtResolved - 31
```

```
Call perf.PlaySegment(bandSeg, 0, mtResolved)
```

```
' Play the style segment on the measure boundary.
```

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

Note

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

Timing

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

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

Master Clock

[\[Visual Basic\]](#)

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

[\[C++\]](#)

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

The master clock is a high-resolution timer that is shared by all processes, devices, and applications that are using DirectMusic. The clock is used to synchronize all

music 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 **IDirectMusic::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 **IDirectMusic::EnumMasterClock** method. Once you have obtained the GUID of the device that you want to use as the master clock, you pass this to the **IDirectMusic::SetMasterClock** method.

Clock Time vs. Music Time

[C++]

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

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

The **IDirectMusicPerformance::AdjustTime** method can be used to make small changes to the performance time. Most applications do not 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 **IDirectMusicPerformance::MusicToReferenceTime** and **IDirectMusicPerformance::ReferenceToMusicTime** methods.

When a segment is cued to play by a call to **IDirectMusicPerformance::PlaySegment** 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 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**), the 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 passed to **PlaySegment** 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 plays at its designated time, perhaps interrupting another segment.

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

[Visual Basic]

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

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

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

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

If the tempo is changed before the segment starts playing, the start time can be affected, or the segment might not start on the desired boundary. In the first case, in which the conversion to music time is made immediately, the start time (in reference time) is advanced if the tempo speeds up and delayed if the tempo slows down. In the second case, in which conversion is made at start time, a change in tempo can mean

that the segment does not start at correct resolution boundaries. For example, if the segment is supposed to start on a measure boundary (as indicated in the *IFlags* parameter of **PlaySegment**), the start time (in clock time) is calculated when the segment is cued. However, if the tempo then changes, a measure boundary might not fall at that time.

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

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

Changing the Tempo

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

[C++]

There are two ways to do so: by sending a message and by setting a track parameter.

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

```
/* Assume that pIDMSegment is a valid IDirectMusicSegment and
   IDMPPerformance is a valid IDirectMusicPerformance. */

// Disable tempo track in segment so that it does not reset the tempo.
pIDMSegment->SetParam( GUID_DisableTempo, 0xFFFF,0,0, NULL );

DMUS_TEMPO_PMSG* pTempo;

if( SUCCEEDED(pIDMPPerformance->AllocPMsg(
    sizeof(DMUS_TEMPO_PMSG), (DMUS_PMSG**)&pTempo)))
{
    // Queue the tempo event.
    ZeroMemory(pTempo, sizeof(DMUS_TEMPO_PMSG));
```

```
pTempo->dwSize = sizeof(DMUS_TEMPO_PMSG);  
pTempo->dblTempo = 100;  
pTempo->dwFlags = DMUS_PMSGF_REFTIME;  
pTempo->dwType = DMUS_PMSGT_TEMPO;  
pIDMPerformance->SendPMsg((DMUS_PMSG*)pTempo);  
}
```

The following code example shows how to change the tempo parameter. For more information, see [Setting and Retrieving Track Parameters](#).

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

[\[Visual Basic\]](#)

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

The master tempo is a factor by which all tempos in the performance are multiplied. For example, if you set the master tempo by calling

DirectMusicPerformance.SetMasterTempo with a parameter of 0.75, and then play a segment that has a tempo of 120 beats per minute, the segment plays with a tempo of 90.

Sending a tempo by using the **DirectMusicPerformance.SendTempoPMSG** method changes the tempo at the time for which the message is stamped. The new tempo is valid until another tempo message is sent, either directly by the application or by a segment that is being played. The tempo value can be modified by the master tempo.

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

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

Prepare Time

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

For an illustration, see [Latency and Bumper Time](#).

[\[C++\]](#)

The queue is like a gas tank that is constantly being topped off by calls to **IDirectMusicTrack::Play**. Each time the performance calls **Play**, it calculates the end time for that call by adding the prepare time to the current time. If the current time is 10,000 milliseconds (or the equivalent in `REFERENCE_TIME` units) and the prepare time is the default 1000 ms, the end time is 11,000—that is, all new messages that are to play up to time 11,000 must be prepared and placed in the queue.

The size of the queue can be changed by calling the **IDirectMusicPerformance::SetPrepareTime** method, and the current size can be retrieved by using **IDirectMusicPerformance::GetPrepareTime**.

Most applications do not 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 **IDirectMusicPerformance::PlaySegment**.

[\[Visual Basic\]](#)

The queue is like a gas tank that is constantly being topped off. Each time the performance calls on the segment to send messages, it calculates the end time for that call by adding the prepare time to the current time. If the current time is 10,000 milliseconds and the prepare time is the default 1000 ms, the end time is 11,000—that is, all new messages that are to play up to time 11,000 must be prepared and placed in the queue.

The size of the queue can be changed by calling the **DirectMusicPerformance.SetPrepareTime** method, and the current size can be retrieved by using **DirectMusicPerformance.GetPrepareTime**.

Most applications do not 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 **DirectMusicPerformance.PlaySegment**.

Normally, if you set a start time of "now" for the segment, the performance invalidates any messages currently in the queue. Any tracks that are still valid at this point (for example, tracks of secondary segments that continue to play despite the introduction of a new primary segment) then have to resend their messages, taking into account any changes made to the musical environment by the new segment. To avoid unnecessary processing and to ensure continuity (for example, to ensure that a long note or nonmusical DLS sound is not cut short), you can use the `DMUS_SEGF_AFTERPREPARETIME` flag to specify that the segment is not to start playing until all messages currently in the queue have been processed and passed to the port buffer. So, for example, if messages up to time 10,000 are in the queue and the current time is 9,000, a segment cued to play immediately starts playing just after the 10,000 ms mark.

For more information, see [Segment Timing](#).

Latency and Bumper Time

[C++]

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

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

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

IDirectMusicPerformance::GetBumperLength.

Latency time and bumper time can be combined. Suppose an event is supposed 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.

Once a message has been placed in the port buffer, it no longer belongs to the performance and cannot be stopped from playing by using the **IDirectMusicPerformance::Invalidate** or the **IDirectMusicPerformance::Stop** method. 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 **IDirectMusicPerformance::GetQueueTime** method.

[Visual Basic]

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

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

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

Latency time and bumper time can be combined. Suppose an event is supposed 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.

Once a message has been placed in the port buffer, it no longer belongs to the performance and cannot be stopped from playing by using the **DirectMusicPerformance.Invalidate** or the **DirectMusicPerformance.Stop**

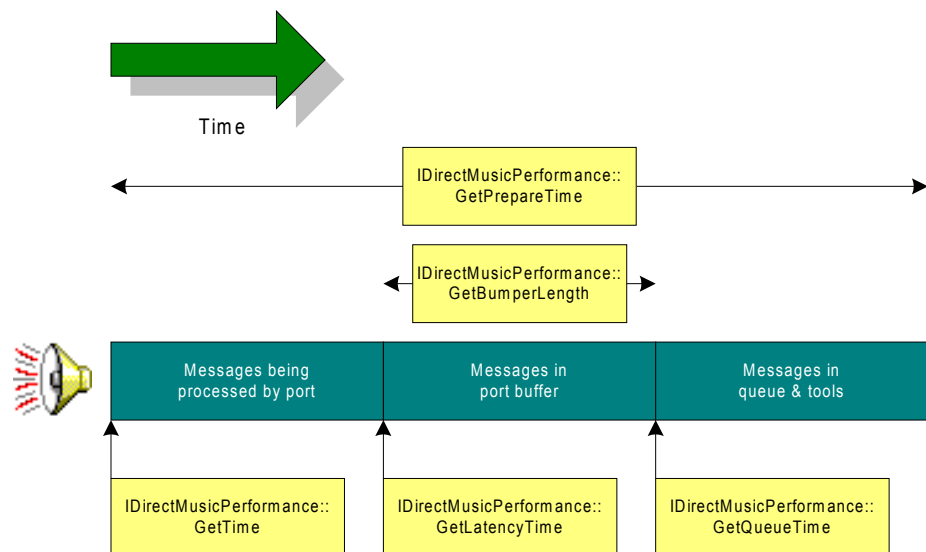
method. The first message that can be invalidated has a time stamp equal to or greater than the current time plus the latency and bumper time. This value can be retrieved by using the **DirectMusicPerformance.GetQueueTime** method.

The following illustration 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. Prepare time is only an approximation of the total timespan of messages in the queue at any given moment.

[Visual Basic]

The methods in the illustration are from the C++ API, but most have the same names in DirectX for Visual Basic. For example,

IDirectMusicPerformance::GetPrepareTime is equivalent to **DirectMusicPerformance.GetPrepareTime**. The **GetTime** method in the illustration is equivalent to **DirectMusicPerformance.GetClockTime**.



Segment Timing

[C++]

Segments play from the beginning unless a start point is set by a call to the **IDirectMusicSegment::SetStartPoint** method. If a repeat count is set by using **IDirectMusicSegment::SetRepeats**, the entire segment repeats that number of times, unless a loop has been defined by a call to **IDirectMusicSegment::SetLoopPoints**. In this case, only the part of the segment between the loop points repeats.

The performance time at which the segment starts playing is determined by two parameters of **IDirectMusicPerformance::PlaySegment**:

- 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 time at which the segment starts depends on the type of segment. If it is a primary segment or a control segment, the earliest start time is at queue (or flush) time. If it is a noncontrol secondary segment, the earliest start is at latency time. For more information on queue time and latency time, see Latency and Bumper Time.
 - The *dwFlags* parameter determines how soon after the scheduled time the segment starts playing, depending on the rhythm of the currently playing segment. Usually, you will want to wait for an appropriate moment before introducing a new segment, a transition, or a motif. You control the delay by setting one of the following **DMUS_SEGF_FLAGS**:
-

[Visual Basic]

Segments play from the beginning unless a start point is set by a call to the **DirectMusicSegment.SetStartPoint** method. If a repeat count is set by using **DirectMusicSegment.SetRepeats**, the entire segment repeats that number of times, unless a loop has been defined by a call to **DirectMusicSegment.SetLoopPoints**. In this case, only the part of the segment between the loop points repeats.

The performance time at which the segment starts playing is determined by two parameters of **DirectMusicPerformance.PlaySegment**:

- The *startTime* parameter sets the earliest time at which the segment can start playing. If *startTime* is 0, this time is as soon as possible. The time at which the segment starts depends on the type of segment. If it is a primary segment or a control segment, the earliest start time is at queue (or flush) time. If it is a noncontrol secondary segment, the earliest start is at latency time. For more information on queue time and latency time, see Latency and Bumper Time.
 - The *lFlags* parameter determines how soon after the scheduled time the segment starts playing, depending on the rhythm of the currently playing segment. Usually, you will want to wait for an appropriate moment before introducing a new segment, a transition, or a motif. You control the delay by setting one of the following **CONST_DMUS_SEGF_FLAGS**:
-

DMUS_SEGF_AFTERPREPARETIME

Play at the earliest start time plus the prepare time. This ensures that any messages from the currently playing segment that have already been queued to the port are not invalidated. This saves processing time and also ensures that any motifs continue to play smoothly over a transition from one primary segment to another.

DMUS_SEGF_GRID

Play on a grid boundary. A grid is a subdivision of a beat. The time signature (authored into the style) determines how many grids each beat is divided into.

DMUS_SEGF_BEAT

Play on a beat.

DMUS_SEGF_MEASURE

Play at the beginning of a measure.

DMUS_SEGF_DEFAULT

Use the cued segment's default boundary.

If none of these flags is set, the segment starts playing at exactly the earliest start time.

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

Notification and Event Handling

From time to time, your application might need to respond to a music 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. Get the desired information by asking DirectMusic to notify you when a certain type of event has taken place.

[C++]

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

```
/* It is assumed that pPerformance is a valid
   IDirectMusicPerformance pointer. */

GUID guid = GUID_NOTIFICATION_SEGMENT;
// C syntax:
pPerformance->AddNotificationType(&guid);
// C++ syntax:
pPerformance->AddNotificationType(guid);
```

You can also add notification types for a particular segment by using the **IDirectMusicSegment::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.

Notifications are sent in the form of **DMUS_NOTIFICATION_PMSG** message structures. You can poll for any pending notification messages within the Windows message loop by calling the **IDirectMusicPerformance::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 Win32 **CreateEvent** function. Typically, you would create an autoreset event with a call such as the following:

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

After creating the event, assign the handle to the performance by passing it to the **IDirectMusicPerformance::SetNotificationHandle** method. You can use the second parameter of this method to change the default time that DirectMusic holds 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 **IDirectMusicPerformance** interface:

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

The following code example runs repeatedly in its own thread, checking for signaled events and retrieving notification messages.

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

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

This thread is run as follows:

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

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

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

It is not necessary to create an event to retrieve notification messages in your application's message loop. As long as you have requested notifications by calling the **IDirectMusicPerformance::AddNotificationType** method, the performance sends messages that can be retrieved by calling **IDirectMusicPerformance::GetNotificationPMsg**.

More than one message can 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.

You must free any message that you retrieve, by calling the **IDirectMusicPerformance::FreePMsg** method.

[\[Visual Basic\]](#)

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

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

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

To have DirectMusic signal events, use **DirectX7.CreateEvent** to obtain an event handle, and then pass this handle to **DirectMusicPerformance.SetNotificationHandle**. The module that you pass to **CreateEvent** must implement the **DirectXEvent** class and must also provide an implementation of the **DirectXEvent.DXCallback** method, which is called by DirectMusic whenever an event is signaled.

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

```
' DX is a DirectX7 object; perf is a DirectMusicPerformance object.  
Dim hEvent As Long  
hEvent = DX.CreateEvent(frmMain)  
Call perf.SetNotificationHandle(hEvent)
```

The form module contains code similar to the following, which looks for a message indicating that the segment has finished playing:

```
Implements DirectXEvent

Private Sub DirectXEvent_DXCallback(ByVal eventid As Long)

    Dim GotMSG As Boolean
    Dim PMsg As DMUS_NOTIFICATION_PMSG

    Do
        GotMSG = gobjDMPPerformance.GetNotificationPMSG(PMsg)
        If GotMSG Then
            If PMsg.lNotificationOption = DMUS_NOTIFICATION_SEGEND Then
                ' Segment has finished playing.
            End If
        End If
    Loop Until Not GotMSG

End Sub
```

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

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

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

Music Parameters

[\[Visual Basic\]](#)

In DirectX for Visual Basic, music parameters are set and retrieved by using various methods of **DirectMusicPerformance** and **DirectMusicSegment**.

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

[C++]

DirectMusic lets you control many aspects of track behavior by changing parameters during execution, using one of the following **SetParam** methods:

- **IDirectMusicPerformance::SetParam** sets data on a specific track within the current control segment of this 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 **DMUS_SEGF_FLAGS**.
- **IDirectMusicSegment::SetParam** sets data on a specific track within this segment.
- **IDirectMusicTrack::SetParam** sets data on this track.

In addition, the **IDirectMusicPerformance::SetGlobalParam** method allows 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 **IDirectMusicPerformance::Invalidate** method. This method causes all tracks to resend messages from the specified point forward.

More information about parameters is contained in the following topics:

- Setting and Retrieving Track Parameters
 - Setting and Retrieving Global Parameters
-

Setting and Retrieving Track Parameters

[Visual Basic]

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

[C++]

Set and retrieve track parameters by using the **SetParam** and **GetParam** methods of either the performance, the segment, or the track. When calling one of these methods on the performance or segment, you can identify the track by setting the *dwGroupBits* and *dwIndex* parameters. However, in most cases, you can let DirectMusic find the appropriate track for you. For more information, see *Identifying the Track*.

In some cases, you must specify the time within the track at which the change is to take effect or for which the parameter is to be retrieved. To see whether this value is used for a particular track parameter, see [Track Parameter Types](#).

For a few parameters, a call to **SetParam** turns a feature on or off, and no data is needed. When setting other parameters, however, you must also supply a structure or variable containing the data. Only parameters with associated data can be retrieved, so when using **GetParam**, you must always supply a pointer to an appropriate variable or structure to receive the data.

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

More information is given in the following topics:

- [Identifying the Track](#)
 - [Track Parameter Types](#)
 - [Disabling and Enabling Messages](#)
-

Identifying the Track

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

[\[C++\]](#)

When you set or retrieve a parameter by using **IDirectMusicTrack::SetParam** or **IDirectMusicTrack::GetParam**, the parameter is associated with the track on which the method is called. However, when you call **IDirectMusicPerformance::SetParam**, **IDirectMusicPerformance::GetParam**, **IDirectMusicSegment::SetParam**, or **IDirectMusicSegment::GetParam**, DirectMusic needs to find the appropriate track.

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

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

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

Every track belongs to one or more groups, each group being represented by a bit in the *dwGroupBits* parameter of one of the methods under discussion. (The track is assigned to a group or groups when it is inserted in the performance. See

IDirectMusicSegment::InsertTrack. In the case of segments loaded from a file, track groups are assigned by the author of the segment.)

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

Suppose a segment contains the following tracks:

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

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

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

Track Parameter Types

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

[\[C++\]](#)

The track parameter that is being set or retrieved by one of the **SetParam** or **GetParam** methods 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 calling **GetParam**.

The predefined parameters are listed in the following table, which links you to tables with more detailed information about each parameter. In the detailed tables, *pParam* and *mtTime* refer to the parameters of the various **GetParam** and **SetParam** methods. Although parameters are always associated with particular track types, you will usually call the method on the segment or the performance and let DirectMusic find the appropriate track. (See Identifying the Track.)

GUID_ChordParam

GUID_EnableTimeSig

GUID_Clear_All_Bands	GUID_IDirectMusicBand
GUID_CommandParam	GUID_IDirectMusicChordMap
GUID_CommandParam2	GUID_IDirectMusicStyle
GUID_ConnectToDLSCollection	GUID_MuteParam
GUID_Disable_Auto_Download	GUID_RhythmParam
GUID_DisableTempo	GUID_SeedVariations
GUID_DisableTimeSig	GUID_StandardMIDIFile
GUID_Download	GUID_TempoParam
GUID_Enable_Auto_Download	GUID_TimeSignature
GUID_EnableTempo	GUID_Unload

GUID_ChordParam

[C++]

Track type	Chord.
Purpose	Set or retrieve a chord change.
Data type (*pParam)	DMUS_CHORD_PARAM.
mtTime	The time, in track time, at which to add the chord to the track, or the time at or directly after the chord to be retrieved from the track.

[Visual Basic]

This topic pertains only to applications written in C++.

GUID_Clear_All_Bands

[C++]

Track type	Band.
Purpose	Clear all bands from the track.
Data type (*pParam)	None.
mtTime	Not used.

[Visual Basic]

This topic pertains only to applications written in C++.

GUID_CommandParam

[C++]

Track type	Command.
Purpose	Set or retrieve a groove or embellishment command.
Data type (*pParam)	DMUS_COMMAND_PARAM2 .
<i>mtTime</i>	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.

[Visual Basic]

This topic pertains only to applications written in C++.

GUID_CommandParam2

[C++]

Track type	Command.
Purpose	Set or retrieve a groove or embellishment command.
Data type (*pParam)	DMUS_COMMAND_PARAM2 . The mtTime member of this structure gives the actual time of the command.
<i>mtTime</i>	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.

[Visual Basic]

This topic pertains only to applications written in C++.

GUID_ConnectToDLSCollection

[C++]

Track type	Band.
Purpose	Connect a band to a DLS collection. See Playing a MIDI File with Custom Instruments.
Data type (*pParam)	IDirectMusicCollection pointer.
<i>mtTime</i>	Not used.

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

GUID_Disable_Auto_Download

[\[C++\]](#)

Track type	Band.
Purpose	Disable automatic downloading of instruments. See Using Bands.
Data type (*pParam)	None.
mtTime	Not used.

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

GUID_DisableTempo

[\[C++\]](#)

Track type	Tempo.
Purpose	Disable tempo messages. See Disabling and Enabling Messages.
Data type (*pParam)	None.
mtTime	Not used.

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

GUID_DisableTimeSig

[\[C++\]](#)

Track type	Time signature, style, and motif.
Purpose	Disable time-signature messages. See Disabling and Enabling Messages.
Data type (*pParam)	None.
mtTime	Not used.

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

GUID_Download

[\[C++\]](#)

Track type	Band.
Purpose	Download instrument data for the track's bands. See Playing a MIDI File with Custom Instruments . See also GUID_Unload .
Data type (*pParam)	IDirectMusicPerformance pointer.
mtTime	Not used.

Remarks

Setting this parameter by using **IDirectMusicSegment::SetParam** downloads only the instruments in the first band track. To download instruments in the other tracks, iterate through the band tracks by using **IDirectMusicSegment::GetTrack**, and call **IDirectMusicTrack::SetParam** on each track.

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

GUID_Enable_Auto_Download

[\[C++\]](#)

Track type	Band.
Purpose	Enable automatic downloading of instruments. See Using Bands .
Data type (*pParam)	None.
mtTime	Not used.

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

GUID_EnableTempo

[C++]

Track type	Tempo.
Purpose	Enable tempo messages. See Disabling and Enabling Messages.
Data type (*pParam)	None.
mtTime	Not used.

[Visual Basic]

This topic pertains only to applications written in C++.

GUID_EnableTimeSig

[C++]

Track type	Time signature, style, and motif.
Purpose	Disable time-signature messages. See Disabling and Enabling Messages.
Data type (*pParam)	None.
mtTime	Not used.

[Visual Basic]

This topic pertains only to applications written in C++.

GUID_IDirectMusicBand

[C++]

Track type	Band.
Purpose	Set a band.
Data type (*pParam)	IDirectMusicBand.
mtTime	The time, in track time, at which to add the band to the track.

[Visual Basic]

This topic pertains only to applications written in C++.

GUID_IDirectMusicChordMap

[C++]

Track type	Chord map.
Purpose	Set or retrieve the chord map.
Data type (*pParam)	IDirectMusicChordMap pointer (SetParam) or address of a variable to receive this pointer (GetParam).
mtTime	The time, in track time, at which to add the chord map to the track, or the time at or directly after the chord map to be retrieved from the track.

[Visual Basic]

This topic pertains only to applications written in C++.

GUID_IDirectMusicStyle

[C++]

Track type	Style.
Purpose	Set or retrieve the style.
Data type (*pParam)	IDirectMusicStyle pointer (SetParam) or address of a variable to receive this pointer (GetParam).
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.

[Visual Basic]

This topic pertains only to applications written in C++.

GUID_MuteParam

[C++]

Track type	Mute.
Purpose	Set or retrieve channel-mapping information.
Data type (*pParam)	DMUS_MUTE_PARAM . The dwPChannel member must be initialized before this structure is passed to GetParam .
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.

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

GUID_RhythmParam

[\[C++\]](#)

Track type	Chord.
Purpose	Retrieve the rhythm pattern for a sequence of chords stored in a measure in the track.
Data type (*pParam)	DMUS_RHYTHM_PARAM . The TimeSig member must be initialized before this structure is passed to GetParam .
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.

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

GUID_SeedVariations

[\[C++\]](#)

Track type	Style and motif.
Purpose	A nonzero value seeds the random number generator for variation selection. A value of 0 reverts to the default behavior of getting the seed from the system clock.
Data type (*pParam)	Long .
mtTime	Not used.

Remarks

Setting this parameter to nonzero is useful for testing since 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, but each time the entire segment is replayed from the beginning, each loop sounds the same as the first time the segment was played.

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

GUID_StandardMIDIFile

[\[C++\]](#)

Track type	Band.
Purpose	Ensure that a standard MIDI file (one not authored specifically for DirectMusic) plays correctly.
Data type (*pParam)	None.
<i>mtTime</i>	Not used.

Note

This parameter must be set for any segment based on a standard MIDI file before any instruments are downloaded.

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

GUID_TempoParam

[\[C++\]](#)

Track type	Tempo.
Purpose	Set or retrieve the tempo.
Data type (*pParam)	DMUS_TEMPO_PARAM . For SetParam , the mtTime member of the structure is ignored. For GetParam , the mtTime member receives the offset of the tempo change from the requested time and is always 0 or less.
<i>mtTime</i>	The time, in track time, at which to set the tempo, or the time at or directly after the tempo change to retrieve.

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

GUID_TimeSignature

[\[C++\]](#)

Track type	Time signature and style.
Purpose	Retrieve the time signature.
Data type (*pParam)	DMUS_TIMESIGNATURE . The mtTime member receives the offset of the time signature change from the requested time and is always 0 or less.
<i>mtTime</i>	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.

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

GUID_Unload

[\[C++\]](#)

Track type	Band.
Purpose	Unload instrument data for the track's bands. See also GUID_Download .
Data type (*pParam)	IDirectMusicPerformance pointer.
<i>mtTime</i>	Not used.

[\[Visual Basic\]](#)

This topic pertains only to applications written in C++.

Setting and Retrieving Global Parameters

[\[C++\]](#)

By using the **IDirectMusicPerformance::SetGlobalParam** and **IDirectMusicPerformance::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>) and Data (<i>*pParam</i>)	Description
GUID_PerfAutoDownload BOOL	This parameter controls whether instruments are automatically downloaded when a segment is played. By default, it is off. See Downloading and Unloading Bands.
GUID_PerfMasterGrooveLevel char	The master groove level is a value that is always added to the groove level established by the command track. The resulting value is adjusted, if necessary, to fall within the range from 1 through 100.
GUID_PerfMasterTempo float	The master tempo is a scaling factor that is applied to the tempo by the final output tool. By default, it is 1. A value of 0.5 would halve the tempo, and a value of 2.0 would double it. This value can be set in the range from DMUS_MASTERTEMPO_MIN through DMUS_MASTERTEMPO_MAX.
GUID_PerfMasterVolume long	The master volume is an amplification or attenuation factor, in hundredths of a decibel, applied to the default volume of the entire performance. The range of permitted values is determined by the port. Legacy hardware MIDI ports do not support changing master volume.

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

When a parameter is set, the performance allocates memory for the data in a linked list of items identified by GUID. The data can be retrieved by a call to **IDirectMusicPerformance::GetGlobalParam**. Even predefined parameters have to be set before they can be retrieved. **GetGlobalParam** fails if **SetGlobalParam** has never been called on the parameter.

[\[Visual Basic\]](#)

The **DirectMusicPerformance** class has the following methods for setting and retrieving global parameters, which affect the entire performance:

GetMasterAutoDownload SetMasterAutoDownload	This parameter controls whether instruments are automatically downloaded when a segment is played. By default, it is off. See <i>Downloading and Unloading Bands</i> .
GetMasterGrooveLevel SetMasterGrooveLevel	The master groove level is a value that is always added to the groove level established by the command track. The resulting value is adjusted, if necessary, to fall within the range from 1 through 100.
GetMasterTempo SetMasterTempo	The master tempo is a scaling factor that is applied to the tempo by the final output tool. By default, it is 1. A value of 0.5 would halve the tempo, and a value of 2.0 would double it. This value can be set in the range from 0.25 through 2.0.
GetMasterVolume SetMasterVolume	The master volume is an amplification or attenuation factor, in hundredths of a decibel, applied to the default volume of the entire performance. The range of permitted values is determined by the port. Legacy hardware MIDI ports do not support changing master volume.

Disabling and Enabling Messages

[C++]

By setting the `GUID_DisableTempo` and `GUID_DisableTimeSig` parameters on a track, you can disable the generation of **DMUS_TEMPO_PMSG** and **DMUS_TIMESIG_PMSG** messages, respectively. You might want to do this, for example, when you have set the tempo dynamically and do not want the primary segment to send tempo messages.

To re-enable messages, call one of the **SetParam** methods with `GUID_EnableTempo` and `GUID_EnableTimeSig` as the *rguidType* parameter. You can also set these parameters to force a segment to send tempo messages, even though it is not the control segment, or to cause a secondary segment to send time-signature messages. (For more information on control segments, see *Segments and DMUS_SEGF_FLAGS*.)

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

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

[Visual Basic]

By using the **DirectMusicSegment.SetTempoEnable** and **DirectMusicSegment.SetTimeSigEnable** methods, you can disable the generation of tempo and time-signature messages, respectively. You might want to do this, for example, when you have set the tempo dynamically and do not want the primary segment to send tempo messages.

You can also use **SetTempoEnable** to force a segment to send tempo messages, even though it is not the control segment, and **SetTimeSigEnable** to cause a secondary segment to send time signature messages. (For more information on control segments, see Segments and **DMUS_SEGF_FLAGS**.)

Capturing Music

[Visual Basic]

This topic pertains only to applications written in C++. DirectX for Visual Basic does not support music capture.

[C++]

Capturing MIDI messages from a device such as a keyboard is very easy in DirectMusic.

Typically, you would create a port for the capture device and use its **IDirectMusicPort::SetReadNotificationHandle** method to cause an event to be signaled whenever messages are available to be read. In response to the event, call the **IDirectMusicPort::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 **IDirectMusicBuffer::GetNextEvent** method. Each call retrieves a single event until no more are available, at which point **S_FALSE** is returned.

The following code example 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 IDirectMusic::CreateMusicBuffer. */
```

```
REFERENCE_TIME rt;
DWORD         dwGroup;
```

```
DWORD    cb;
BYTE     *pb;

DWORD dw = WaitForMultipleObjects(1, hEvent, FALSE, INFINITE);
for (;;)
{
    hr = pPort->Read(pBuffer);
    if (hr == S_FALSE)
    {
        break; // No more messages to read into the buffer
    }
    pBuffer->ResetReadPtr();
    for (;;)
    {
        hr = pBuffer->GetNextEvent(&rt, &dwGroup, &cb, &pb);
        if (hr == S_OK)
        {
            // pb points to the data structure for the message, and
            // you can do anything that you want with it.
        }
        else if (hr == S_FALSE)
        {
            break; // No more messages in the buffer
        }
    } // Done with the buffer
} // Done reading pending events
```

If you do not want to intercept messages, but simply want to send them from one port to another, you can use the **IDirectMusicThru** interface.

DirectMusic Tools

In DirectMusic, a tool is an object that intercepts messages and handles them in some way. The tool might alter the message, and then pass it on to the next tool, free the message, or send a new message based on information in the old one.

DirectMusic has an output tool that is normally the last to receive messages. It is this tool that converts performance messages to standard MIDI messages and streams them to the synthesizer. Other tools are implemented by the application or obtained from libraries.

[\[Visual Basic\]](#)

DirectX for Visual Basic does not support the use of application-defined or third-party tools.

[C++]

To implement a tool, you must first create an object that supports the **IDirectMusicTool** interface. The object's implementation of the **IDirectMusicTool** methods determines what messages are processed by the tool and what work is performed on these messages.

All tools other than the output tool are normally collected in graphs; even if your application is using only one other tool, you should create a **DirectMusicGraph** to contain it. Then add this graph to a segment or the performance. Graphs provide a convenient mechanism for directing messages from one tool to another.

When the performance engine is playing a segment, it allows each tool in the segment graph, and then each tool in the performance graph to process each message. After a tool processes a message, it should call the **IDirectMusicGraph::StampPMsg** method (obtaining the **IDirectMusicGraph** pointer from the **pGraph** member of the **DMUS_PMSG** structure) to stamp the message with a pointer to the next tool, if any, that is to receive it. Then, the tool puts the message back in the pipeline.

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 **IDirectMusicTool::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 **IDirectMusicTool**, be sure 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 graph passed into its **IDirectMusicTool::Init** method. If the tool fails to release this reference, there is a problem when the segment attempts to release the graph. Because the tool still has a reference to the graph, the graph is not fully released; and because the graph has a reference to the tool, the tool cannot be released either.

For more information on how to set up a tool, see Tutorial 2: Using Tools.

Music Composition

This section is an introduction to the composition engine of DirectMusic. You can use the engine to implement a dynamic musical program, using previously authored elements, such as styles, templates, and bands.

The following topics are discussed in this section:

- Overview of Music Authoring
- Music Files for Composition
- Overview of Programming for Composition

- How Music Varies During Playback
- Music Values and MIDI Notes
- Using Compositional Elements

Overview of Music Authoring

The author of compositional music elements uses a tool such as DirectMusic Producer to create the basic elements of each musical theme required by the application. For more information on the following topics, see the documentation for DirectMusic Producer.

- Authoring Styles
- Authoring Chord Maps
- Authoring Style-based Segments
- Authoring MIDI-based Segments
- Authoring Templates
- Authoring Bands

Note

In this section, the human composer is referred to as the author to avoid confusion with the composer object of DirectMusic.

Authoring Styles

A style is a basic definition of the music. It is a collection of patterns, along with a time signature (meter) and a tempo (beats per minute). A style can also contain one or more bands.

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

A motif is a special type of pattern designed to be played solo over the basic score. Typically, a motif would be used in an interactive application to mark an event: for example, the drumbeat that occurs in the DMDonuts sample whenever the ship hits the edge of the screen.

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

The author can create many variations for each pattern. Typically, he or she would do this by copying the pattern, and then making small changes to one or more parts. At run time, variations are chosen by the style playback mechanism. However, the author

can disable any variation for any chord—that is, specify that the variation must never be chosen when a certain chord is being played.

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

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

Authoring Chord Maps

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

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

The chord map is a road map of chord progressions. Within the chord-map designer, the author chooses chords that can express the desired musical feeling or personality. He or she then arranges these chords in a flowchart along a time line. (The time line is conceptually circular, so it keeps looping back to the beginning as long as that segment of music is being played.)

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

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

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

Note

DirectMusic allows up to `DMUS_MAXSUBCHORD` subchords in a chord, and this value is defined as 8. However, DirectMusic Producer currently allows authors to create a maximum of four subchords per chord.

Authoring Style-based Segments

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

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

Authoring MIDI-based Segments

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

Authoring Templates

A template is a segment, but unlike a style-based segment, it is not bound to a particular style and does not have a fixed chord progression.

Instead of a chord progression, the template has a series of signpost group markers along a time line. Signposts are chords that mark the beginning and end of regions in which variations in the chord progression are possible. When the author creates a signpost, he or she assigns it to a group.

The following happens later, either within the authoring tool or at run time, when the DirectMusicComposer generates a segment by combining the template with a particular style and chord map. Each time the engine encounters a pair of signpost group markers along the time line in the template, it looks in the chord map 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 chord map. If it is unable to find a path that works or there is no end signpost marker, the engine plays any chord from the group of the beginning signpost group marker.

The author might use templates to apply similar chord progressions, groove levels, and embellishments to different styles while composing segments. However, templates can also be combined with styles and chord maps by the DirectMusicComposer object at run time.

Authoring Bands

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

In a tool such as DirectMusic Producer, the author creates a band by assigning instruments to PChannels. These instruments can be from any DLS collection, and instruments from different sources can be mixed within a band. For example, a band might have a jazz guitar from the General MIDI set in part 1, a sixties organ from the Roland GS set in part 2, and an ethnic percussive instrument from a custom DLS collection in part 3. Each of these instruments is also given volume, pan, and transposition settings.

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

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

Music Files for Composition

When programming for DirectMusic composition, you will use a variety of files produced in a tool such as DirectMusic Producer. You load these elements into the application as COM objects and obtain interfaces to them. (See DirectMusic Loader.)

The following table summarizes the types of files that you will encounter.

[C++]

The Class GUID is the value that you put in the **guidClass** member of the **DMUS_OBJECTDESC** structure when loading the object.

Element	Class GUID	Interface	File type
Band	CLSID_DirectMusicBand	IDirectMusicBand	.bnd
DLS collection	CLSID_DirectMusicCollection	IDirectMusicCollection	.dls
Chord map	CLSID_DirectMusicChordMap	IDirectMusicChordMap	.cdm
Segment	CLSID_DirectMusicSegment	IDirectMusicSegment	.sgt
Style	CLSID_DirectMusicStyle	IDirectMusicStyle	.sty
Template	CLSID_DirectMusicSegment	IDirectMusicSegment	.tpl

[\[Visual Basic\]](#)

Element	DirectMusicLoader method	Class	File type
Band	LoadBand	DirectMusicBand	.bnd
DLS collection	LoadCollection	DirectMusicCollection	.dls
Chord map	LoadChordMap	DirectMusicChordMap	.cdm
Segment	LoadSegment	DirectMusicSegment	.sgt
Style	LoadStyle	DirectMusicStyle	.sty
Template	LoadSegment	DirectMusicSegment	.tpl

Note

Bands can be authored as part of a style, in which case they are automatically loaded when the style is loaded. Similarly, styles and bands can be authored into a segment, in which case you do not need separate files for those elements. Files can also contain references to other files. If a style contains a reference to a band, the band is automatically loaded when the style is, provided the loader can find the band file.

Overview of Programming for Composition

When you implement music composed at run time, you will use previously authored objects as building blocks. In consultation with the author or other content provider, you can choose to get the musical data in the form of small building blocks that offer you the greatest possible flexibility and variation at run time, or you can use larger prefabricated elements that define the form of the music more fully.

[\[C++\]](#)

Using the largest building blocks, you load highly structured segments (either style-based or MIDI-based) that contain everything that the performance needs to know about the music to play it. All you have to do is load the segment and query for the **IDirectMusicSegment** interface. You pass this interface pointer to the **IDirectMusicPerformance::PlaySegment** method. The style playback engine then selects pattern variations from the style and plays them according to a fixed chord progression—or, in the case of a MIDI-based segment, plays the MIDI sequence. Band changes are usually contained in the segment, as well.

[\[Visual Basic\]](#)

Using the largest building blocks, you load highly structured segments (either style-based or MIDI-based) that contain everything that the performance needs to know about the music to play it. All you have to do is load the segment by using the

DirectMusicLoader.LoadSegment or the **DirectMusicLoader.LoadSegmentFromResource** method. You then pass the returned **DirectMusicSegment** object to the **DirectMusicPerformance.PlaySegment** method. The style playback engine then selects pattern variations from the style and plays them according to a fixed chord progression—or, in the case of a MIDI-based segment, plays the MIDI sequence. Band changes are usually contained in the segment, as well.

If you want to use smaller building blocks, obtain the following elements:

- Chord maps, which are road maps of chord progressions.
- Styles, which define a basic melody and rhythm with variations, motifs, and embellishments.
- Template segments, which are structural plans that control various aspects of playback, including the length of the segment, whether it loops, where groove level changes and embellishment patterns are to be placed, and what types of chords in the chord map are to serve as signposts.

[C++]

You then construct a segment by combining any chord map, style, and template, using the **IDirectMusicComposer::ComposeSegmentFromTemplate** method.

To have even more flexibility in music composition at run time, create segments based on predefined shapes, rather than templates, using the **IDirectMusicComposer::ComposeSegmentFromShape** method. The shape is used in creating the command and signpost tracks, which control the choice of embellishment patterns, the chord progression, and the frequency of chord changes.

When playing segments, you can also control the band used to play the parts. Bands are typically authored right into styles and templates, but they might be supplied as separate files so that band changes can be made dynamically by the application. In this case, you must create a secondary segment containing only the band, using the **IDirectMusicBand::CreateSegment** method, and play this segment when it is time to assign instruments and instrument settings to the primary segment. For more information, see Using Bands.

[Visual Basic]

You then construct a segment by combining any chord map, style, and template, using the **DirectMusicComposer.ComposeSegmentFromTemplate** method.

To have even more flexibility in music composition at run time, create segments based on predefined shapes, rather than templates, using the **DirectMusicComposer.ComposeSegmentFromShape** method. The shape is used in creating the command and signpost tracks, which control the choice of embellishment patterns, the chord progression, and the frequency of chord changes.

When playing segments, you can also control the band used to play the parts. Bands are typically authored right into styles and templates, but they might be supplied as separate files so that band changes can be made dynamically by the application. In this case, you must create a secondary segment containing only the band, using the **DirectMusicBand.CreateSegment** method, and play this segment when it is time to assign instruments and instrument settings to the primary segment. For more information, see Using Bands.

How Music Varies During Playback

As DirectMusic plays a segment (other than a simple MIDI file or an authored segment based on a MIDI file), 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. Typically a style contains multiple patterns, which are selected in response to commands from the command track. For example, if the command track calls for a break embellishment to be played, the style playback engine selects a break pattern that is compatible with the current groove level. (The author specifies which groove levels are appropriate for each pattern.) If there is more than one suitable pattern, a random choice is made.
- 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 current 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 or template. The current groove level of a segment can 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, the underlying chord changes according to the progression in the chord track. The notes in the current pattern are automatically transposed to harmonize with the new chord.
- Variations in timing. The playback engine can introduce small random changes in the parameters of individual notes—when they begin and when they end.
- Band. The choice of instruments and instrument settings (volume, pan, and transposition) can be changed as the segment is playing, either by the band track within an authored segment or dynamically by the application. The application can change the band by creating a secondary segment based on a band object, then playing that segment.

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

For more information on how to change the music at run time, see [Music Parameters](#).

Music Values and MIDI Notes

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

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

[C++]

For an explanation of the data format of music values, see **DMUS_NOTE_PMSG**.

[Visual Basic]

For an explanation of the data format of music values, see **DMUS_NOTE_PMSG**.

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.

[Visual Basic]

For more information on the various play modes, see **CONST_DMUS_PLAYMODE_FLAGS**.

The subchord level is a value in the range from 0 through 31 that determines which subchords can be used in establishing the music value. A DirectMusic chord consists of one or more subchords, allowing for complex harmonies with multiple parallel chord progressions. Each subchord supports one or more levels. The author of the music defines the supported levels for each subchord.

When a segment is played, each note is encapsulated in a **DMUS_NOTE_PMSG** type. The **midiValue** member of this type holds the MIDI note value that would normally be sent to the synthesizer. The message also holds the original music value, as well as the play mode and subchord level that were used in transposition.

Music values and play modes are primarily of interest to tools, which are not supported by DirectX for Visual Basic. If you want to inject a note into the performance by using **DirectMusicPerformance.SendNotePMSG**, specify the desired **midiValue** in the **DMUS_NOTE_PMSG** type, and ignore the **musicValue**, **playModeFlags**, and **subChordLevel** members.

The following code example sends a message to play middle C immediately:

```
' perf is the DirectMusicPerformance.
Dim note As DMUS_NOTE_PMSG

note.midiValue = 60
note.mtDuration = 500
note.flags = DMUS_NOTEF_NOTEON
note.velocity = 127
Call perf.SendNotePMSG(0, DMUS_PMSGF_REFTIME, 1, note)
```

[C++]

For more information on the various play modes, see **DMUS_PLAYMODE_FLAGS**.

The subchord level is a value in the range from 0 through 31 that determines which subchords can be used in establishing the music value. A DirectMusic chord (as represented by a **DMUS_CHORD_PARAM** or **DMUS_CHORD_KEY** structure) consists of one or more subchords, allowing for complex harmonies with multiple parallel chord progressions. Each subchord supports one or more levels, as specified in the **dwLevels** member of the **DMUS_SUBCHORD** structure. The author of the music defines the supported levels for each subchord.

When a segment is played, each note is encapsulated in a **DMUS_NOTE_PMSG** structure. The **bMidiValue** member of this structure holds the MIDI note value that would normally be sent to the synthesizer. The message also holds the original music value, as well as the play mode and subchord level that were used in transposition. A tool can use this information to alter the note in any way it likes. For example, a tool could intercept a note that was transposed in a certain play mode, change the play mode, obtain a new MIDI note by using the **IDirectMusicPerformance::MusicToMIDI** method, and put the new value in the **bMidiValue** member of the message before passing it on.

The **MusicToMIDI** method is at the heart of the style playback mechanism of DirectMusic. It is called by the style track to convert its internal music values into notes, using the current chord in the chord track.

If a tool wants to reverse the process and obtain a new music value from a MIDI note using a different chord, play mode, or subchord level, it calls the **IDirectMusicPerformance::MIDIToMusic** method. The new music value can be placed in the **wMusicValue** member of the note message, in which it might be of use to other application-defined tools. However, the DirectMusic output tool ignores it and plays the note in **bMidiValue** as usual.

Using Compositional Elements

This section is a guide to incorporating music components into a DirectMusic application. It is presumed that you have a basic understanding of the purpose of each component and how it is represented by an object. If not, you might first want to read Composition Objects and Interfaces and Overview of Music Authoring.

You can incorporate DirectMusic into your applications without working with individual components, such as styles, chord maps, and templates. If you prefer, you can work with fully authored segments, or even with MIDI files. Using individual components simply gives you greater control over the performance at run time.

This section covers the following topics:

- Using Authored Segments
- Using Styles
- Using Motifs
- Using Chord Maps
- Using Templates
- Using Transitions

Using Authored Segments

An authored segment is a file or resource that contains all the data for a piece of music. It can be based on a MIDI file or on a style and a chord map. Unlike a simple MIDI file or resource, it can contain band changes and variations.

Note

A template is an authored segment as well, but it does not represent a self-contained piece of music. For more information, see Using Templates.

[C++]

Create a segment in your application by loading the segment as an object and obtaining the **IDirectMusicSegment** interface, as in the following code example, in which the segment is loaded from a DirectMusic Producer file:

```
/* It is assumed that pLoader is a valid pointer to
   an IDirectMusicLoader interface and that the search
   directory has been properly set. */

DMUS_OBJECTDESC ObjectDescript;
IDirectMusicSegment* pSegment;

ObjectDescript.dwSize = sizeof(DMUS_OBJECTDESC);
ObjectDescript.idClass = CLSID_DirectMusicSegment;
```

```
strcpy(ObjectDescript.wszFileName, L"Dance.sgt");  
ObjectDescript.dwValidData = DMOBJ_CLASS | DMOBJ_PATH ;  
pLoader->GetObject(&ObjectDescript, IID_IDirectMusicSegment2,  
    (void**) pSegment)))
```

You can now pass *pSegment* to the **IDirectMusicPerformance::PlaySegment** method.

[Visual Basic]

Create a segment in your application by using the **DirectMusicLoader.LoadSegment** or the **DirectMusicLoader.LoadSegmentFromResource** method, as in the following code example, in which the segment is loaded from a DirectMusic Producer file:

```
' loader is the DirectMusicLoader object.  
Dim seg As DirectMusicSegment  
Set seg = loader.LoadSegment("Sample.sgt")
```

You can now pass *seg* to the **DirectMusicPerformance.PlaySegment** method.

Using Styles

The **DirectMusicStyle** 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 that are played in each instrument part.

You can obtain the **DirectMusicStyle** object from a style or segment file or from a resource. For more information, see **DirectMusic Loader and Music Files for Composition**.

A style by itself does not contain enough information to create a segment of music at run time. You need two other components: a chord map (map of chord progressions) and a command track to set the groove level and embellishments as the music plays. The command track can come from a template or be generated at run time from a shape. The chord map generally comes from a chord-map file or resource.

[C++]

To create a segment with a command track based on a template, call the **IDirectMusicComposer::ComposeSegmentFromTemplate** method. (See **Using Templates**.)

To create a segment based on a shape, call the **IDirectMusicComposer::ComposeSegmentFromShape** method. You supply pointers to the style and the chord map, and a variable to receive a pointer to the created segment. You also supply a rate of harmonic motion, which controls the

frequency of chord changes, and a shape constant, which determines the progression of groove levels and embellishments.

[Visual Basic]

To create a segment with a command track based on a template, call the **DirectMusicComposer.ComposeSegmentFromTemplate** method. (See Using Templates.)

To create a segment based on a shape, call the **DirectMusicComposer.ComposeSegmentFromShape** method. You supply pointers to the style and the chord map, and a variable to receive a pointer to the created segment. You also supply a rate of harmonic motion, which controls the frequency of chord changes, and a shape constant, which determines the progression of groove levels and embellishments.

Using Motifs

A motif is a special kind of pattern in a style intended to be played over the basic style pattern, typically in response to an interactive event. Although a motif can be as complex as any other pattern, even containing variations and multiple instrument parts, usually it is a short, simple musical figure that sounds good against a variety of background patterns. It might also be a sound effect played by a custom DLS instrument or instruments.

[C++]

All the motifs authored into a style become available to you as soon as you have loaded that style. To get a particular motif ready for playback, call the **IDirectMusicStyle::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 **IDirectMusicStyle::EnumMotif** method.
- A pointer to receive the **IDirectMusicSegment** interface to the segment object to be created by the method.

The following code example obtains and plays the motif whose name is passed in as *pwszMotifName*.

```
void PlayMotif(IDirectMusicPerformance* pPerf,
               IDirectMusicStyle* pStyle,
               WCHAR* pwszMotifName)
{
    IDirectMusicSegment* pSeg;

    // Get the motif segment from the style. Check for S_OK
```

```
// specifically because GetMotif() returns S_FALSE if it
// does not find the motif.

if (S_OK == pStyle->GetMotif(pwszMotifName, &pSeg))
{
    /* Play the segment. DMUS_SEGF_BEAT means play on the next beat if
    there is a segment currently playing. DMUS_SEGF_SECONDARY means
    play the segment as a secondary segment, which plays over
    the currently playing primary segment. The 0 indicates to
    play now. The final NULL means do not return an
    IDirectMusicSegmentState* in the last parameter
    because you don't need to track the state of playback. */

    pPerf->PlaySegment(pSeg,
        DMUS_SEGF_BEAT | DMUS_SEGF_SECONDARY,
        0,
        NULL);
    pSeg->Release();
}
}
```

Note that *pSeg* is played as a secondary segment because a motif is normally played over a primary segment. You cannot play a motif as a primary segment because it does not have a chord track or band track. If you do want to play a motif against silence, create a primary segment from a style that has only blank patterns, and keep that segment playing while you play the motif.

[\[Visual Basic\]](#)

All the motifs authored into a style become available to you as soon as you have loaded that style. To get a particular motif ready for playback, call the **DirectMusicStyle.GetMotif** method, passing in the name of the motif. You might know this from the documentation for the style, or you can obtain it from an index value by using the **DirectMusicStyle.GetMotifName** method.

The following code example obtains and plays the first motif in the style:

```
' style is a DirectMusicStyle.
' perf is the DirectMusicPerformance.
Dim MotifName As String
Dim segMotif As DirectMusicSegment

MotifName = style.GetMotifName(0)
Set segMotif = style.GetMotif(MotifName)
Call perf.PlaySegment(segMotif, DMUS_SEGF_SECONDARY, 0)
```

Note that *segMotif* is played as a secondary segment because a motif is normally played over a primary segment. You cannot play a motif as a primary segment because it does not have a chord track or band track. If you want to play a motif against silence, create a primary segment from a style that has only blank patterns, and keep that segment playing while you play the motif.

Using Chord Maps

A chord-map object represents a collection of chords that provides the foundation of the harmonic structure and the mood of the music. A chord map 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 authored segments, applications do not normally need to concern themselves with chord maps. The chord map is used at the authoring stage to create a fixed chord progression. However, chord maps can be used to compose segments at run time and to alter the chord progression of existing segments.

You obtain the `DirectMusicChordMap` object from a chord-map file or resource. For more information, see `DirectMusic Loader` and `Music Files for Composition`.

[C++]

If a chord map has been authored into a style, you can retrieve a pointer to its **IDirectMusicChordMap** interface by passing its name (assigned by the author) to the **IDirectMusicStyle::GetChordMap** method. You can also use the **IDirectMusicStyle::EnumChordMap** method to search for a particular chord map, or the **IDirectMusicStyle::GetDefaultChordMap** method to obtain a pointer to the default chord map for the style.

Note

DirectMusic Producer does not support authoring chord maps into style files.

You set the chord map for a composition when you create a segment by using either **IDirectMusicComposer::ComposeSegmentFromTemplate** or **IDirectMusicComposer::ComposeSegmentFromShape**. For more information, see `Using Styles`.

Once a segment has been created, you can change its chord map by calling the **IDirectMusicComposer::ChangeChordMap** method. This changes the mood of the music without altering its basic rhythm and melody.

Every chord map has an underlying scale, consisting of 24 tones. You can determine the tones of the scale by using the **IDirectMusicChordMap::GetScale** method. The lower 24 bits of the variable pointed to by the *pdwScale* parameter of this method are set or clear, depending on whether the corresponding tone is part of the scale. The upper 8 bits give the root of the scale as an integer in the range from 0 through 23 (low C to middle B).

[Visual Basic]

You set the chord map for a composition when you create a segment by using either **DirectMusicComposer.ComposeSegmentFromTemplate** or **DirectMusicComposer.ComposeSegmentFromShape**. For more information, see Using Styles.

Once a segment has been created, you can change its chord map by calling the **DirectMusicComposer.ChangeChordMap** method. This changes the mood of the music without altering its basic rhythm and melody.

Using Templates

A template is a special kind of 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 provides the command track, which controls changes in the groove level and the choice of embellishment patterns. It also prescribes how the chord map is used in composing the segment, by specifying from which signpost group each new chord must come.

[C++]

A template is represented by a **DirectMusicSegment** object.

There are two ways to obtain a template:

- Obtain one from a template file or resource. You load the file as a **DirectMusicObject** and query for the **IDirectMusicSegment** interface. For more information, see **DirectMusic Loader**.
- Create one from a shape, using the **IDirectMusicComposer::ComposeTemplateFromShape** method. You choose the length, the overall shape, whether intro and end embellishment patterns are to be played, and how long the ending is to be. You get back a pointer to the **IDirectMusicSegment** interface.

Once you have obtained a template segment object, you can pass it to the **IDirectMusicComposer::ComposeSegmentFromTemplate** method, along with pointers to a style and a chord map. 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 at the address given in the *ppSectionSeg* parameter. You pass this pointer to the **IDirectMusicPerformance::PlaySegment** method.

[Visual Basic]

A template is represented by a **DirectMusicSegment** object.

There are two ways to obtain a template object:

- Obtain one from a template file or resource by using **DirectMusicLoader.LoadSegment** or **DirectMusicLoader.LoadSegmentFromResource**.
- Create one from a shape, using the **DirectMusicComposer.ComposeTemplateFromShape** method. You choose the length, the overall shape, whether intro and end embellishment patterns are to be played, and how long the ending is to be.

Once you have obtained a template segment object, you can pass it to the **DirectMusicComposer.ComposeSegmentFromTemplate** method, along with pointers to a style and a chord map. You also supply a rate of harmonic motion, which sets the frequency of chord changes. The **ComposeSegmentFromTemplate** method creates a segment that can be passed to the **DirectMusicPerformance.PlaySegment** method.

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 create an intermediate or closing segment that provides an appropriate transition.

[C++]

There are two methods for composing transitional segments.

- The **IDirectMusicComposer::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 **IDirectMusicComposer::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.

Both these methods take a chord map, a command, and a set of flags as parameters.

- The chord map, as usual, is used to create a chord track that defines the chord progression in the segment.
- The command is one of the **DMUS_COMMANDT_TYPES** enumeration. It determines which type of pattern—either an ordinary groove pattern or one of the embellishments—is called for in the command track of the transitional segment. When the segment plays, an appropriate pattern is selected from the style.

- The flags are from **DMUS_COMPOSEF_FLAGS** and further define the transition, principally its timing. The **DMUS_COMPOSEF_MODULATE** flag can be used to cause the transition to move smoothly from one tonality to another; it cannot be used when there is no second segment because there can be no modulation to silence.
-

[Visual Basic]

There are two methods for composing transitional segments.

- The **DirectMusicComposer.AutoTransition** method, given a **DirectMusicPerformance**, creates a transition from the currently playing segment to a second segment of your choice, and then automatically cues the transitional segment and the second segment for playback. The transition begins playing immediately or on the next boundary, as specified in the *IFlags* parameter. Optionally, the second segment can be *Nothing* so that the transition is to silence.
- The **DirectMusicComposer.ComposeTransition** method composes a transition from any point in one segment to the beginning of a second segment, or to silence, and returns a **DirectMusicSegment** object so that the application can play the transition.

Both these methods take a chord map, a command, and a set of flags as parameters.

- The chord map, as usual, is used to create a chord track that defines the chord progression in the segment.
 - The command is one of the **CONST_DMUS_COMMANDT_TYPES** enumeration. It determines which type of pattern—either an ordinary groove pattern or one of the embellishments—is called for in the command track of the transitional segment. When the segment plays, an appropriate pattern is selected from the style.
 - The flags are from **CONST_DMUS_COMPOSEF_FLAGS** and further define the transition, principally its timing. The **DMUS_COMPOSEF_MODULATE** flag can be used to cause the transition to move smoothly from one tonality to another; it cannot be used when there is no second segment, because there can be no modulation to silence.
-

Transitions are normally a single measure in length. There are two exceptions: when the **DMUS_COMPOSEF_LONG** flag is included and when the command is **DMUS_COMMANDT_END** and the end embellishment in the style is more than one measure long.

DirectMusic Reference

This section contains reference information for the application programming interface (API) elements provided by Microsoft® DirectMusic® in C/C++ and Microsoft® Visual Basic®. Reference material is organized by language:

- DirectMusic C/C++ Reference
- DirectMusic Visual Basic Reference

DirectMusic C/C++ Reference

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

- Interfaces
- Messages
- Structures
- File Structures
- DLS Structures
- Enumerated Types
- Return Values

Interfaces

This section contains references for the following DirectMusic interfaces:

- **IDirectMusic**
- **IDirectMusicBand**
- **IDirectMusicBuffer**
- **IDirectMusicChordMap**
- **IDirectMusicCollection**
- **IDirectMusicComposer**
- **IDirectMusicDownload**
- **IDirectMusicDownloadedInstrument**
- **IDirectMusicGetLoader**
- **IDirectMusicGraph**
- **IDirectMusicInstrument**
- **IDirectMusicLoader**
- **IDirectMusicObject**
- **IDirectMusicPerformance**
- **IDirectMusicPort**

- **IDirectMusicPortDownload**
- **IDirectMusicSegment**
- **IDirectMusicSegmentState**
- **IDirectMusicStyle**
- **IDirectMusicThru**
- **IDirectMusicTool**
- **IDirectMusicTrack**
- **IKsControl**
- **IReferenceClock**

IDirectMusic

The **IDirectMusic** interface provides methods for managing buffers, ports, and the master clock. There should not be more than one instance of this interface per application.

Note

There is no helper function to create this interface. Applications use the COM **CoCreateInstance** function or the **IDirectMusicPerformance::Init** method to create a DirectMusic object.

The methods of the **IDirectMusic** interface can be organized into the following groups:

Activation	Activate
Buffers	CreateMusicBuffer
Linkage	SetDirectSound
Ports	CreatePort
	EnumPort
	GetDefaultPort
Timing	EnumMasterClock
	GetMasterClock
	SetMasterClock

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusic::Activate

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

```
HRESULT Activate(  
    BOOL fEnable  
);
```

Parameters

fEnable

Switch to activate (TRUE) or deactivate (FALSE) all port objects created in this instance of DirectMusic.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DSERR_NODRIVER, indicating that no sound driver is present.

Remarks

Applications should call **IDirectMusic::Activate**(FALSE) when they lose input focus if they do not need to play music in the background. This allows another application that has the input focus to have access to the ports. Once the application has input focus again, it should call **Activate**(TRUE) to enable all its allocated ports.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicPort::Activate

IDirectMusic::CreateMusicBuffer

The **IDirectMusic::CreateMusicBuffer** method creates a **DirectMusicBuffer** object to hold music messages being sequenced to the port. Most applications do not need to call this method directly because buffer management is handled by the performance when a port is added.

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

Parameters

pBufferDesc

Address of the **DMUS_BUFFERDESC** structure that contains the description of the music buffer to be created. The application must initialize the **dwSize** member of this structure before passing the pointer. See Remarks.

ppBuffer

Address of a variable to receive the **IDirectMusicBuffer** interface pointer.

pUnkOuter

Address of the controlling object's **IUnknown** interface for COM aggregation. Aggregation is not currently supported, so this value must be set to **NULL**.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

- E_INVALIDARG**
- E_NOAGGREGATION**
- E_NOINTERFACE**
- E_OUTOFMEMORY**
- E_POINTER**

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in **dmusicc.h**.

IDirectMusic::CreatePort

The **IDirectMusic::CreatePort** method is used to create an object for a particular DirectMusic port.

```
HRESULT CreatePort(
    REFCLSID rclsidPort,
    LPDMUS_PORTPARAMS pPortParams,
    LPDIRECTMUSICPORT *ppPort,
    LPUNKNOWN pUnkOuter
);
```

Parameters

rclsidPort

Reference to (C++) or address of (C) the GUID that identifies the port for which the **IDirectMusicPort** interface is to be created. The GUID is retrieved through the **IDirectMusic::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_PORTPARAMS** structure that contains parameters for the port. The **dwSize** member of this structure must be initialized before the method is called.

ppPort

Address of a variable to receive an **IDirectMusicPort** interface pointer.

pUnkOuter

Address of the controlling object's **IUnknown** interface for COM aggregation. Aggregation is not currently supported, so this value must be NULL.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if a requested parameter is not available.

If it fails, the method can return one of the following error values:

```
DMUS_E_DSOUND_NOT_SET
E_INVALIDARG
E_NOAGGREGATION
E_NOINTERFACE
E_OUTOFMEMORY
E_POINTER
```

Remarks

By default, the port is inactive when it is created. It must be activated by a call to **IDirectMusic::Activate** or **IDirectMusicPort::Activate**.

If not all parameters could be obtained, the **DMUS_PORTPARAMS** structure is changed as follows to match the available parameters of the port.

On entry, the **dwValidParams** member of the structure indicates which members in the structure are valid. If the flag is not set for a member of the structure, a default value is set for that parameter when the port is created.

On return, the flags in **dwValidParams** show which port parameters were set. If a particular parameter was not requested but was set to the default, that flag is added to those passed in.

If the port supports a specified parameter but the given value for the parameter is out of range, the parameter value in **pPortParams* is changed. In this case, the flag in **dwValidParams** remains set, but **S_FALSE** is returned to indicate that the value has been changed.

The following code example shows how an application can request reverb capabilities and determine if they were obtained. (For an alternative way of checking and setting port properties, see Port Property Sets.)

```
DMUS_PORTPARAMS params;

ZeroMemory(&params, sizeof(params));
params.dwSize = sizeof(params);
params.dwValidParams = DMUS_PORTPARAMS_EFFECTS;
params.dwEffectFlags = DMUS_EFFECT_REVERB;
HRESULT hr = pDirectMusic->CreatePort(guidPort, &params,
    &port, NULL);
if (SUCCEEDED(hr))
{
    fGotReverb = TRUE;
    if (hr == S_FALSE)
    {
        if (!(params.dwValidParams & DMUS_PORTPARAMS_EFFECTS))
        {
            // Device does not support any effects.
            fGotReverb = FALSE;
        }
        else if (!(params.dwEffectFlags & DMUS_EFFECT_REVERB))
        {
            // Device understands effects,
            // but could not allocate reverb.
            fGotReverb = FALSE;
        }
    }
}
```

```
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusic::EnumMasterClock

The **IDirectMusic::EnumMasterClock** method is used to enumerate and get the description of the clocks that DirectMusic can use as the master clock. Each time that it is called, this method retrieves information about a single clock.

```
HRESULT EnumMasterClock(  
    DWORD dwIndex,  
    LPDMUS_CLOCKINFO lpClockInfo  
);
```

Parameters

dwIndex

Index of the clock for which the description is to be returned. This parameter should be 0 on the first call, and then incremented in each subsequent call until S_FALSE is returned.

lpClockInfo

Address of a **DMUS_CLOCKINFO** structure to receive the description of the clock. The application must initialize the **dwSize** member of this structure before passing the pointer.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if there is no clock with that index number.

If it fails, the method can return one of the following error values:

```
E_INVALIDARG  
E_NOINTERFACE  
E_POINTER
```

Remarks

Applications should not rely on or store the index number of a clock. Rebooting or adding and removing hardware could cause the index number of a clock to change.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusic::SetMasterClock, **IDirectMusic::GetMasterClock**

IDirectMusic::EnumPort

The **IDirectMusic::EnumPort** method is used to enumerate and get the capabilities of the DirectMusic ports connected to the system. Each time it is called, this method retrieves information about a single port.

```
HRESULT EnumPort(  
    DWORD dwIndex,  
    LPDMUS_PORTCAPS pPortCaps  
);
```

Parameters

dwIndex

Index of the port for which the capabilities are to be returned. This parameter should be 0 on the first call, and then incremented in each subsequent call until S_FALSE is returned.

pPortCaps

Address of the **DMUS_PORTCAPS** structure to receive the capabilities of the port. The **dwSize** member of this structure must be initialized before the pointer is passed.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if there is no port with that index value.

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_NOINTERFACE
E_POINTER

Remarks

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

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusic::GetDefaultPort

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

```
HRESULT GetDefaultPort(  
    LPGUID pguidPort  
);
```

Parameters

pguidPort

Address of a variable to receive the default port GUID.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

Default Port

IDirectMusic::GetMasterClock

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

```
HRESULT GetMasterClock(  
    LPGUID pguidClock,  
    IReferenceClock **ppReferenceClock  
);
```

Parameters

pguidClock

Address of a variable to receive the master clock's GUID. The application can pass NULL if this value is not desired.

ppReferenceClock

Address of a variable to receive the **IReferenceClock** interface pointer for this clock. The application can pass NULL if this value is not desired.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_NOINTERFACE
E_POINTER

Remarks

The **IReferenceClock** interface pointer must be released once the application has finished using the interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusic.h.

See Also

IDirectMusic::SetMasterClock

IDirectMusic::SetDirectSound

The **IDirectMusic::SetDirectSound** method connects DirectMusic to a DirectSound object for wave output.

```
HRESULT SetDirectSound(  
    LPGUID pDirectSound,  
    HWND hWnd  
);
```

Parameters

pDirectSound

Address of the **IDirectSound** interface to use for output. If this parameter is NULL, the method creates a DirectSound object and sets the DSSCL_PRIORITY cooperative level. (See Remarks.) If this parameter contains an **IDirectSound** pointer, the caller is responsible for setting the cooperative level.

hWnd

Window handle to the DirectSound object created by this call. If this value is NULL, the current foreground window is set as the focus window. See Remarks.

If *pDirectSound* is a valid interface, this parameter is ignored because it is the caller's responsibility to supply a valid window handle in the call to

IDirectSound::SetCooperativeLevel.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DMUS_E_DSOUND_ALREADY_SET.

Remarks

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

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

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

If you created the DirectSound object yourself, you can release it by calling this method with NULL in the *pDirectSound* parameter after deactivating all ports. (It is an error to call **SetDirectSound** on an active port.)

You can pass NULL in the *hWnd* parameter to pass the current foreground window handle to DirectSound. However, it is not wise to assume that the application window is in the foreground during initialization. In general, the top-level application window handle should be passed to DirectMusic, DirectSound, and DirectDraw. See the Remarks for **IDirectSound::SetCooperativeLevel**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in *dmusicc.h*.

See Also

Integrating DirectMusic and DirectSound

IDirectMusic::SetMasterClock

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

```
HRESULT SetMasterClock(  
    REFGUID rguidClock  
);
```

Parameters

rguidClock

Reference to (C++) or address of (C) the GUID that identifies the clock to set as the master clock for DirectMusic. This parameter must be a GUID returned by the **IDirectMusic::EnumMasterClock** method.

Return Values

If the method succeeds, the return value is *S_OK*.

If it fails, the method can return *DMUS_E_PORTS_OPEN*.

Remarks

If another running application is also using DirectMusic, it is not possible to change the master clock until that application is shut down.

Most applications do not need to call **SetMasterClock**. It should not be called unless there is a need to synchronize tightly with a hardware timer other than the system clock.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusic.h.

See Also

IDirectMusic::GetMasterClock, **IDirectMusic::EnumMasterClock**

IDirectMusicBand

The **IDirectMusicBand** interface represents a DirectMusic band object. A band is used to set the instrument choices and mixer settings for a set of PChannels. For an overview, see Using Bands.

Bands can come from several places. They can be stored directly in their own files or embedded in a style's band list or a segment's band track.

The DirectMusicBand object also supports the **IPersistStream** and **IDirectMusicObject** interfaces for loading its data.

The **IDirectMusicBand** interface has the following methods:

Segment creation	CreateSegment
Instrument data	Download
	Unload

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

The **LPDMUS_BAND** type is defined as a pointer to the **IDirectMusicBand** interface.

```
typedef IDirectMusicBand __RPC_FAR *LPDMUS_BAND;
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicBand::CreateSegment

The **IDirectMusicBand::CreateSegment** method creates a **DirectMusicSegment** object that can be used to perform the volume, pan, transposition, and patch change commands in the band dynamically, using the **IDirectMusicPerformance::PlaySegment** method.

```
HRESULT CreateSegment(
    IDirectMusicSegment** ppSegment
);
```

Parameters

ppSegment

Address of a variable to receive a pointer to the created segment.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

- E_FAIL**
- E_OUTOFMEMORY**
- E_POINTER**

Remarks

For an example of creating a segment from a band, see [Making Band Changes Programmatically](#).

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in **dmusic1.h**.

IDirectMusicBand::Download

The **IDirectMusicBand::Download** method downloads the DLS data for instruments in the band to a performance object. The method downloads each instrument in the band by calling the **IDirectMusicPerformance::DownloadInstrument** method. **DownloadInstrument**, in turn, uses the **PChannel** of the instrument to find the appropriate port, and then calls the **IDirectMusicPort::DownloadInstrument** method on that port.

Once a band has been downloaded, the instruments in the band can be selected, either individually with program-change MIDI messages, or all at once by playing a band segment created through a call to the **IDirectMusicBand::CreateSegment** method.

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

Parameters

pPerformance

Performance to which instruments are to be downloaded. The performance manages the mapping of PChannels to DirectMusic ports.

Return Values

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

If it fails, the method can return one of the following error values:

```
DMUS_E_NOT_INIT  
E_OUTOFMEMORY  
E_POINTER
```

Remarks

Because a downloaded band uses synthesizer resources, it should be unloaded when no longer needed by using the **IDirectMusicBand::Unload** method.

In the current version of DirectMusic, 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 up to an initialized port. Since this is a complete failure, there is no need to call **IDirectMusicBand::Unload** later.

If the download partially succeeds, DMUS_S_PARTIALDOWNLOAD is returned. This means that some of the instruments successfully downloaded while others did not. This usually occurs because of programming error setting up the performance and port. The best way to find the problem is to set debug traces to 1 for Dmime.dll, Dmband.dll, and Dmsynth.dll. (See Debugging DirectMusic Projects.)

The following are some common causes of a partial download:

- The band has instruments on PChannels that have not been set up on the performance (by using **IDirectMusicPerformance::AssignPChannelBlock**).

- The band has instruments on PChannels 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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicBand::Unload

IDirectMusicBand::Unload

The **IDirectMusicBand::Unload** method unloads the DLS data for instruments in the band previously downloaded by **IDirectMusicBand::Download**.

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

Parameters

pPerformance

Performance from which to unload instruments.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicBand::Download, **IDirectMusicPort::UnloadInstrument**

IDirectMusicBuffer

The **IDirectMusicBuffer** interface represents a buffer containing time-stamped data (typically in the form of MIDI messages) to be sequenced by a port. Unlike a segment, the buffer contains a small amount of data (typically less than 200 milliseconds) over which the application has control at event granularity.

Unless your application is doing its own sequencing, you do not need to use the methods of this interface.

Buffer objects are completely independent of port objects until the buffer is passed to the port by a call to the **IDirectMusicPort::PlayBuffer** or the **IDirectMusicPort::Read** method. The application is then free to reuse the buffer.

The methods of the **IDirectMusicBuffer** interface can be organized in the following groups:

Data	Flush
	GetNextEvent
	GetRawBufferPtr
	PackStructured
	PackUnstructured
	ResetReadPtr
Parameters	GetBufferFormat
	GetMaxBytes
	GetUsedBytes
	SetUsedBytes
Time	GetStartTime
	SetStartTime
	TotalTime

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

The **LPDIRECTMUSICBUFFER** type is defined as a pointer to the **IDirectMusicBuffer** interface:

```
typedef IDirectMusicBuffer *LPDIRECTMUSICBUFFER;
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicBuffer::Flush

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

```
HRESULT Flush();
```

Parameters

None.

Return Values

The method always returns S_OK.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicBuffer::GetBufferFormat

The **IDirectMusicBuffer::GetBufferFormat** method retrieves the GUID representing the buffer format.

```
HRESULT GetBufferFormat(  
    LPGUID pGuidFormat  
);
```

Parameters

pGuidFormat

Address of a variable to receive the GUID of the buffer format.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

If the format was not specified when the buffer was created, KSDATAFORMAT_SUBTYPE_DIRECTMUSIC is returned in **pGuidFormat*.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusic::CreateMusicBuffer, DMUS_EVENTHEADER

IDirectMusicBuffer::GetMaxBytes

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

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

Parameters

pcb

Address of a variable to contain the maximum number of bytes that the buffer can hold.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicBuffer::GetNextEvent

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

```
HRESULT GetNextEvent(  
    LREFERENCE_TIME prt,  
    LPDWORD pdwChannelGroup,  
    LPDWORD pdwLength,  
    LPBYTE* ppData  
);
```

Parameters

prt

Address of a variable to receive the time of the message.

pdwChannelGroup

Address of a variable to receive the channel group of the message.

pdwLength

Address of a variable to receive the length, in bytes, of the message.

ppData

Address of a variable to receive a pointer to the message data.

Return Values

If the method succeeds, the return value is **S_OK**, or **S_FALSE** if there are no messages in the buffer.

If it fails, the method can return **E_POINTER**.

Remarks

Any of the passed pointers can be **NULL** if the item is not needed.

The pointer returned in *ppData* is valid only for the lifetime of the buffer object.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in **dmusicc.h**.

See Also

IDirectMusicBuffer::ResetReadPtr

IDirectMusicBuffer::GetRawBufferPtr

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

```
HRESULT GetRawBufferPtr(  
    LPBYTE* ppData  
);
```

Parameters

ppData

Address of a variable to receive a pointer to the buffer's data.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

This method returns a pointer to the raw data of the buffer. The format of the data depends on the implementation. The lifetime of the data is the same as the lifetime of the buffer object; therefore, the returned pointer should not be held after the next call to the **IDirectMusicBuffer::Release** method.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicBuffer::GetStartTime

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

```
HRESULT GetStartTime(  
    LPPREFERENCE_TIME ppt  
);
```

Parameters

ppt

Address of a variable to receive the start time.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_BUFFER_EMPTY
E_POINTER

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer::SetStartTime, IDirectMusicBuffer::TotalTime

IDirectMusicBuffer::GetUsedBytes

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

```
HRESULT GetUsedBytes(  
    LPDWORD pcb  
);
```

Parameters

pcb

Address of a variable to receive the number of used bytes.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer::SetUsedBytes

IDirectMusicBuffer::PackStructured

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

```
HRESULT PackStructured(  
    REFERENCE_TIME rt,  
    DWORD dwChannelGroup,  
    DWORD dwChannelMessage  
);
```

Parameters

rt

Absolute time of the message. See Remarks.

dwChannelGroup

Channel group to which the data belongs.

dwChannelMessage

Data (MIDI message) to pack.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

```
DMUS_E_INVALID_EVENT  
E_OUTOFMEMORY
```

Remarks

At least 32 bytes (the size of **DMUS_EVENTHEADER** plus *dwChannelMessage*) must be free in the buffer.

The *rt* parameter must contain the absolute time at which the data is to be sent to the port. To play a message immediately, retrieve the time from the latency clock, and use this as *rt*. See **IDirectMusicPort::GetLatencyClock**.

Messages stamped with the same time do not necessarily play in the same order in which they were placed in the buffer.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer::PackUnstructured

IDirectMusicBuffer::PackUnstructured

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

```
HRESULT PackUnstructured(  
    REFERENCE_TIME rt,  
    DWORD dwChannelGroup,  
    DWORD cb,  
    LPBYTE lpb  
);
```

Parameters

rt
Absolute time of the message.

dwChannelGroup
Channel group to which the message belongs.

cb
Size of the data, in bytes.

lpb
Pointer to the data.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_OUTOFMEMORY
E_POINTER

Remarks

This method can be used to send any kind of data to the port.

At least 28 bytes (the size of **DMUS_EVENTHEADER**) plus the size of the data, padded to a multiple of 4 bytes, must be free in the buffer. The buffer space required can be obtained by using the **DMUS_EVENT_SIZE**(*cb*) macro, where *cb* is the size of the data.

The *rt* parameter must contain the absolute time at which the data is to be sent to the port. To play a message immediately, retrieve the time from the latency clock, and use this as *rt*. See **IDirectMusicPort::GetLatencyClock**.

Messages stamped with the same time do not necessarily play in the same order in which they were placed in the buffer.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer::PackStructured

IDirectMusicBuffer::ResetReadPtr

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

HRESULT ResetReadPtr()

Parameters

None.

Return Values

The method always returns S_OK.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer::GetNextEvent

IDirectMusicBuffer::SetStartTime

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

```
HRESULT SetStartTime(  
    REFERENCE_TIME rt  
);
```

Parameters

rt

New start time for the buffer.

Return Values

The method always returns S_OK.

Remarks

Events already in the buffer are time stamped relative to the start time and play at the same offset from the new start time.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer::GetStartTime

IDirectMusicBuffer::SetUsedBytes

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

```
HRESULT SetUsedBytes(  
    DWORD cb  
);
```


Parameters

cb

Number of valid data bytes in the buffer.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DMUS_E_BUFFER_FULL.

Remarks

This method allows an application to repack a buffer manually. Normally, this should only be done 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 **IDirectMusicBuffer::GetMaxBytes** method.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer::GetUsedBytes

IDirectMusicBuffer::TotalTime

The **IDirectMusicBuffer::TotalTime** method returns the total time spanned by the data in the buffer.

```
HRESULT TotalTime(  
    LPREFERENCE_TIME prtTime  
);
```

Parameters

prtTime

Address of a variable to receive the total time spanned by the buffer, in units of 100 nanoseconds.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusic.h.

See Also

IDirectMusicBuffer::GetStartTime

IDirectMusicChordMap

The **IDirectMusicChordMap** interface represents a chord map. Chord maps provide the composer (represented by the **IDirectMusicComposer** interface) with the information needed to create chord progressions for segments to be composed. Chord maps can also be used to change the chords in an existing segment.

The **DirectMusicChordMap** object also supports the **IDirectMusicObject** and **IPersistStream** interfaces for loading its data.

The interface has the following method:

IDirectMusicChordMap **GetScale**

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicChordMap::GetScale

The **IDirectMusicChordMap::GetScale** method retrieves the scale associated with the chord map.

```
HRESULT GetScale(  
    DWORD* pdwScale  
);
```

Parameters

pdwScale

Address of a variable to receive the scale value.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be E_POINTER.

Remarks

The scale is defined by the bits in a **DWORD**, split into a scale pattern (lower 24 bits) and a root (upper 8 bits). For the scale pattern, the low bit (0x0001) is the lowest note in the scale, the next higher (0x0002) is a semitone higher, and so on for two octaves. The upper 8 bits give the root of the scale as an integer between 0 and 23 (low C to middle B).

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicCollection

The **IDirectMusicCollection** interface manages an instance of a DLS file. The collection provides methods to access instruments and download them to the synthesizer by means of the **IDirectMusicPort** interface.

The DirectMusicCollection object also supports the **IDirectMusicObject** and **IPersistStream** interfaces for loading its data.

For more information on how to work with collections, see Using Downloadable Sounds.

The **IDirectMusicCollection** interface has the following methods:

Instruments

EnumInstrument

GetInstrument

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusic.h.

IDirectMusicCollection::EnumInstrument

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

```
HRESULT EnumInstrument(
    DWORD dwIndex,
    DWORD* pdwPatch,
    LPWSTR pwszName,
    DWORD dwNameLen
);
```

Parameters

dwIndex

Index of the instrument in the collection.

pdwPatch

Address of a variable to receive the patch number.

pwszName

Address of a buffer to receive the instrument name. Can be NULL if the name is not wanted.

dwNameLen

Size of the instrument name buffer, in **WCHARs**.

Return Values

If the method succeeds, the return value is **S_OK**, or **S_FALSE** if there is no instrument with that index number.

If it fails, the method can return one of the following error values:

E_FAIL

E_OUTOFMEMORY
E_POINTER

Remarks

To enumerate all instruments in a collection, start with a *dwIndex* of 0 and increment until **EnumInstrument** returns S_FALSE.

The patch number returned in *pdwPatch* describes the full patch address, including the MIDI parameters for MSB and LSB bank select. For more information, see MIDI Channel Messages.

Although the ordering of the enumeration is consistent within one instance of a DLS collection, it has no relationship to the ordering of instruments in the file, their patch numbers, or their names.

For an example of instrument enumeration, see Working with Instruments.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicCollection::GetInstrument

The **IDirectMusicCollection::GetInstrument** method retrieves an instrument from a collection by its patch number.

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

Parameters

dwPatch

Instrument patch number.

ppInstrument

Address of a variable to receive a pointer to the **IDirectMusicInstrument** interface.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_INVALIDPATCH
 E_FAIL
 E_OUTOFMEMORY
 E_POINTER

Remarks

The patch number passed in *dwPatch* describes the full patch address, including the MIDI parameters for MSB and LSB bank select. MSB is shifted left 16 bits, and LSB is shifted left 8 bits. For more information, see MIDI Channel Messages.

In addition, the high bit must be set (0x80000000) if the instrument is specifically a drum kit intended to be played on MIDI channel 10. This is a special tag for DLS Level 1, which always puts drums on MIDI channel 10.

For an example of how this method is used, see Working with Instruments.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicComposer

The **IDirectMusicComposer** interface permits access to the composition engine. In addition to building new segments from templates and chord maps, the composer can generate transitions between different segments. It can also apply a chord map to an existing segment, thus altering the chord progression and the mood of the music.

For an overview, see Music Composition.

The methods of the **IDirectMusicComposer** interface can be grouped as follows:

Changing chord maps	ChangeChordMap
Composing ordinary segments	ComposeSegmentFromShape
	ComposeSegmentFromTemplate
Composing template segments	ComposeTemplateFromShape
Composing transition segments	AutoTransition
	ComposeTransition

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicComposer::AutoTransition

The **IDirectMusicComposer::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.

```
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. See

DMUS_COMMANDT_TYPES. If this value is

DMUS_COMMANDT_ENDANDINTRO, the method composes a segment containing both an ending to the primary segment and an introduction to *pToSeg*.

dwFlags

Composition options. See **DMUS_COMPOSEF_FLAGS**.

pChordMap

Chord map to be used when composing the transition.

ppTransSeg

Address of a variable to receive a pointer to the created segment. This value can be NULL, in which case the pointer is not returned.

ppToSegState

Address of a variable to receive 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 to receive 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 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), intro embellishments are not valid. If the currently playing segment is NULL or does not contain a style track, then fill, break, end, and groove embellishments are not valid. If no style track is available either in the currently playing segment or in the one represented by *pToSeg*, all embellishments are invalid, and no transition occurs. In that case, both *ppTransSeg* and *ppTransSegState* return NULL, but the method succeeds and cues the segment represented by *pToSeg*, if that pointer is not NULL.

The value in *pChordMap* can be NULL. If it is, an attempt is made to obtain a chord map from a chord-map track, first from *pToSeg*, and then from the performance's primary segment. If neither of these segments contains a chord-map track, the chord occurring at the current time in the primary segment is used as the chord in the transition.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicComposer::ComposeTransition, Using Transitions

IDirectMusicComposer::ChangeChordMap

The **IDirectMusicComposer::ChangeChordMap** method modifies the chords and scale pattern of an existing segment to reflect a new chord map.

```
HRESULT ChangeChordMap(
    IDirectMusicSegment* pSegment,
    BOOL fTrackScale,
    IDirectMusicChordMap* pChordMap
);
```

Parameters

pSegment

Segment in which to change the chord map.

fTrackScale

If TRUE, the method transposes all the chords to be relative to the root of the new chord map's scale, rather than leaving their roots as they were.

pChordMap

New chord map for the segment.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

The method can be called while the segment is playing.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicComposer::ComposeSegmentFromShape

The **IDirectMusicComposer::ComposeSegmentFromShape** method creates an original segment from a style and a chord map, based on a predefined shape. The

shape represents the way chords and embellishments occur over time across the segment.

```
HRESULT ComposeSegmentFromShape(
    IDirectMusicStyle* pStyle,
    WORD wNumMeasures,
    WORD wShape,
    WORD wActivity,
    BOOL fIntro,
    BOOL fEnd,
    IDirectMusicChordMap* pChordMap,
    IDirectMusicSegment** ppSegment
);
```

Parameters

pStyle

Style from which to compose the segment.

wNumMeasures

Length, in measures, of the segment to be composed.

wShape

Shape of the segment to be composed. Possible values are of the **DMUS_SHAPET_TYPES** enumerated type.

wActivity

Rate of harmonic motion. Valid values are from 0 through 3. Lower values mean more chord changes.

fIntro

TRUE if an introduction is to be composed for the segment.

fEnd

TRUE if an ending is to be composed for the segment.

pChordMap

Chord map from which to create the segment.

ppSegment

Address of a variable to receive a pointer to the created segment.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicComposer::ComposeSegmentFromTemplate,
IDirectMusicComposer::ComposeTemplateFromShape

IDirectMusicComposer::ComposeSegmentFromTemplate

The **IDirectMusicComposer::ComposeSegmentFromTemplate** method creates an original segment from a style, a chord map, and a template.

```
HRESULT ComposeSegmentFromTemplate(  
    IDirectMusicStyle* pStyle,  
    IDirectMusicSegment* pTemplate,  
    WORD wActivity,  
    IDirectMusicChordMap* pChordMap,  
    IDirectMusicSegment** ppSegment  
);
```

Parameters

pStyle

Style from which to create the segment.

pTemplate

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

Chord map from which to create the segment.

ppSegment

Address of a variable to receive a pointer to the created segment.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG

E_POINTER

Remarks

If *pStyle* is non-NULL, it is used in composing the segment; if it is NULL, a style is retrieved from the template specified in *pTempSeg*. Similarly, if *pChordMap* is non-NULL, it is used in composing the segment; if it is NULL, a chord map 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 chord-map 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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicComposer::ComposeSegmentFromShape,
IDirectMusicComposer::ComposeTemplateFromShape, Using Templates

IDirectMusicComposer::ComposeTemplateFromShape

The **IDirectMusicComposer::ComposeTemplateFromShape** method creates a new template segment, based on a predefined shape.

```
HRESULT ComposeTemplateFromShape(
    WORD wNumMeasures,
    WORD wShape,
    BOOL fIntro,
    BOOL fEnd,
    WORD wEndLength,
    IDirectMusicSegment** ppTemplate
);
```

Parameters

wNumMeasures

Length, in measures, of the segment to be composed. This value must be greater than 0.

wShape

Shape of the segment to be composed. Possible values are of the **DMUS_SHAPET_TYPES** enumerated type.

fIntro

TRUE if an introduction is to be composed for the segment.

fEnd

TRUE if an ending is to be composed for the segment.

wEndLength

Length in measures of the ending, if one is to be composed. If *fEnd* is TRUE, this value must be greater than 0 and equal to or less than the number of measures available (that is, not used in the introduction). See also Remarks.

ppTemp;ate

Address of a variable to receive a pointer to the created template segment.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_OUTOFMEMORY
E_POINTER

Remarks

The value of *wEndLength* should not be greater than the length of the longest ending available in any style likely to be associated with this template through the **IDirectMusicComposer::ComposeSegmentFromTemplate** method. The ending starts playing at *wEndLength* measures before the end of the segment. If the ending is less than *wEndLength* measures long, the music then reverts to the basic groove level.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicComposer::ComposeSegmentFromShape,
IDirectMusicComposer::ComposeSegmentFromTemplate, Using Templates

IDirectMusicComposer::ComposeTransition

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

```
HRESULT ComposeTransition(
    IDirectMusicSegment* pFromSeg,
    IDirectMusicSegment* pToSeg,
    MUSIC_TIME mtTime,
    WORD wCommand,
    DWORD dwFlags,
    IDirectMusicChordMap* pChordMap,
    IDirectMusicSegment** ppTransSeg
);
```

Parameters

pFromSeg

Segment from which to compose the transition.

pToSeg

Segment to which the transition should smoothly flow. Can be NULL if *dwFlags* does not include **DMUS_COMPOSEF_MODULATE**.

mtTime

Time in *pFromSeg* from which to compose the transition.

wCommand

Embellishment to use when composing the transition. See **DMUS_COMMANDT_TYPES**. If this value is **DMUS_COMMANDT_ENDANDINTRO**, the method composes a segment containing both an ending to *pFromSeg* and an introduction to *pToSeg*.

dwFlags

Composition options. This parameter can contain one or more of the **DMUS_COMPOSEF_FLAGS** enumerated type values.

pChordMap

Chord map to be used when composing the transition. See Remarks.

ppTransSeg

Address of a variable to receive a pointer to the created segment.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

E_INVALIDARG

E_OUTOFMEMORY

E_POINTER

Remarks

The value in *pChordMap* can be NULL. If it is, an attempt is made to obtain a chord map from a chord-map track, first from *pToSeg*, and then from *pFromSeg*. If neither of these segments contains a chord-map track, the chord occurring at *mtTime* in *pFromSeg* is used as the chord in the transition.

The composer looks for a tempo, first in *pFromSeg*, and then in *pToSeg*. If neither of those segments contains a tempo track, the tempo for the transition segment is taken from the style.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in *dmusici.h*.

See Also

IDirectMusicComposer::AutoTransition, Using Transitions

IDirectMusicDownload

The **IDirectMusicDownload** interface represents a contiguous memory chunk used for downloading to a DLS synthesizer port.

The **IDirectMusicDownload** interface and its contained memory chunk are created by the **IDirectMusicPortDownload::AllocateBuffer** method. The memory can then be accessed by using the single method of this interface.

The interface has the following method:

IDirectMusicDownload **GetBuffer**

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusic.h.

IDirectMusicDownload::GetBuffer

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

```
HRESULT GetBuffer(  
    void** ppvBuffer,  
    DWORD* pdwSize  
);
```

Parameters

ppvBuffer

Address of a variable to receive a pointer to the data buffer.

pdwSize

Address of a variable to receive the size of the returned buffer, in bytes.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_BUFFERNOTAVAILABLE
E_POINTER

Remarks

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

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusic.h.

IDirectMusicDownloadedInstrument

The **IDirectMusicDownloadedInstrument** interface is used to identify an instrument that has been downloaded to the synthesizer by using the **IDirectMusicPort::DownloadInstrument** or the **IDirectMusicPerformance::DownloadInstrument** method. The interface is then used to unload the instrument through a call to **IDirectMusicPort::UnloadInstrument**. Once the instrument has been unloaded, the interface pointer must be released by the application.

For an example, see Working with Instruments.

The **IDirectMusicDownloadedInstrument** interface has no methods of its own. Like all COM interfaces, it inherits the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusic.h`.

IDirectMusicGetLoader

The **IDirectMusicGetLoader** 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 **IDirectMusicGetLoader** interface.

For an example of how to obtain the **IDirectMusicGetLoader** interface from the stream, see **IDirectMusicGetLoader::GetLoader**.

The **IDirectMusicGetLoader** interface has the following method:

IDirectMusicGetLoader	GetLoader
------------------------------	------------------

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader, DirectMusic Loader, Custom Loading

IDirectMusicGetLoader::GetLoader

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

```
HRESULT GetLoader(  
    IDirectMusicLoader ** ppLoader  
);
```

Parameters

ppLoader

Address of a variable to receive the **IDirectMusicLoader** interface pointer. The reference count of the interface is incremented.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_NOINTERFACE.

Remarks

The following code example is from a file parser that finds a reference to an object that needs to be accessed by the loader.

```
HRESULT myGetReferencedObject(  
    DMUS_OBJECTDESC *pDesc,           // Descriptor already prepared  
    IStream *pIStream,                 // Stream being parsed  
    IDirectMusicObject **ppIObject) // Object to be accessed  
{  
    IDirectMusicGetLoader *pIGetLoader;  
    IDirectMusicLoader *pILoader;  
    ppIObject = NULL;  
    HRESULT hr = pIStream->QueryInterface(  
        IID_IDirectMusicGetLoader,  
        (void **) &pIGetLoader );
```

```

if (SUCCEEDED(hr))
{
    hr = pLoader->GetLoader(&pLoader);
    if (SUCCEEDED(hr))
    {
        hr = pLoader->GetObject(pDesc, IID_DirectMusicLoader,
            (void**) ppIObject);
        pLoader->Release();
    }
    pLoader->Release();
}
return hr;
}

```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

Custom Loading

IDirectMusicGraph

The **IDirectMusicGraph** 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.

The **IDirectMusicGraph** interface has the following methods:

Routing	StampPMsg
Tools	GetTool
	InsertTool
	RemoveTool

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicGraph::GetTool

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

```
HRESULT GetTool(  
    DWORD dwIndex,  
    IDirectMusicTool** ppTool  
);
```

Parameters

dwIndex

Zero-based index of the requested tool in the graph.

ppTool

Address of a variable to receive a pointer to the tool.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following values:

DMUS_E_NOT_FOUND

E_POINTER

Remarks

The application is responsible for releasing the retrieved tool.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicGraph::InsertTool

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

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

Parameters

pTool

Tool to insert.

pdwPChannels

Address of an array of PChannels on which the tool accepts messages. If the tool accepts messages on all PChannels, pass NULL.

cPChannels

Count of how many PChannels 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 either the start or (if it is negative) the end of the current tool list. If *lIndex* is out of range, the tool is placed at the beginning or end of the list. To place a tool at the end of the list, use a number for *lIndex* that is larger than the number of tools in the current tool list.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_ALREADY_EXISTS
E_OUTOFMEMORY
E_POINTER

Remarks

The reference count of the tool is incremented.

This method calls **IDirectMusicTool::Init**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicGraph::RemoveTool

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

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

Parameters

pTool

Address of a variable that contains the tool to be removed.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

```
DMUS_E_NOT_FOUND  
E_POINTER
```

Remarks

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

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicGraph::StampPMsg

The **IDirectMusicGraph::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.

```
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**. For an overview of message types, see Messages.

Return Values

If the method succeeds, the return value is **S_OK** or **DMUS_S_LAST_TOOL**. See Remarks.

If it fails, the method can return **E_POINTER**.

Remarks

On entry, *pPMSG->pTool* (see **DMUS_PMSG**) points to the current tool. **StampPMsg** uses this member to determine the current tool to find the next tool in the graph. A value of **NULL** represents the first tool in the graph.

The object pointed to by *pPMSG->pGraph* 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 tool graphs.

The value of *pPMSG->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 **IDirectMusicTool::GetMediaTypes** method.

The value of *pPMSG->dwPChannel* is used to determine which track the tool must be capable of processing. Tracks are identified by unique numbers when a segment is authored.

This method calls **Release** on the current **IDirectMusicTool** pointed to by *pPMSG->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 **IDirectMusicTool::GetMsgDeliveryType** method. This flag determines when the message is delivered to the next tool.

The implementations of this method in the **DirectMusicSegmentState** and **DirectMusicPerformance** objects always return **S_OK** on success. The implementation in **DirectMusicGraph** returns **DMUS_S_LAST_TOOL** if there is no tool other than the output tool waiting to receive the message.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in **dmusici.h**.

See Also

DirectMusic Tools

IDirectMusicInstrument

The **IDirectMusicInstrument** interface represents an individual instrument from a DLS collection.

The only way to create a DirectMusicInstrument object to download an instrument is to first create a DirectMusicCollection object, and then call the **IDirectMusicCollection::GetInstrument** method. **GetInstrument** creates a DirectMusicInstrument object and returns its **IDirectMusicInstrument** interface pointer.

To download the instrument, pass its interface pointer to the **IDirectMusicPort::DownloadInstrument** or the **IDirectMusicPerformance::DownloadInstrument** method. If the method succeeds, it returns a pointer to an **IDirectMusicDownloadedInstrument** interface, which is used only to unload the instrument.

The methods of **IDirectMusicInstrument** 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 **IDirectMusicInstrument::GetPatch** and **IDirectMusicInstrument::SetPatch** methods.

The interface has the following methods:

IDirectMusicInstrument	GetPatch
	SetPatch

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicInstrument::GetPatch

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

```
HRESULT GetPatch(  
    DWORD* pdwPatch  
);
```

Parameters

pdwPatch

Address of a variable to receive the patch number.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

The patch number returned at *pdwPatch* describes the full patch address, including the MIDI parameters for MSB and LSB bank select. In addition, the high bit is set if the instrument is specifically a drum kit, intended to be played on MIDI channel 10. This is a special tag for DLS Level 1, since DLS Level 1 always plays drums on MIDI channel 10. For more information, see MIDI Channel Messages.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicInstrument::SetPatch

The **IDirectMusicInstrument::SetPatch** method sets the patch number for the instrument. Although each instrument in a DLS collection has a predefined patch number, the patch number can be reassigned once the

IDirectMusicCollection::GetInstrument method has been used to retrieve the instrument from the collection. For more information on DirectMusic patch numbers, see **IDirectMusicInstrument::GetPatch**.

```
HRESULT SetPatch(  
    DWORD dwPatch
```

);

Parameters

dwPatch

New patch number to assign to instrument.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DMUS_E_INVALIDPATCH.

Remarks

The following code example gets an instrument from a collection, remaps its MSB bank select to a different bank, then downloads the instrument.

```
HRESULT myRemappedDownload(
    IDirectMusicCollection *pCollection,
    IDirectMusicPort *pPort,
    IDirectMusicDownloadedInstrument **ppDLInstrument,
    BYTE bMSB,      // Requested MIDI MSB for patch bank select
    DWORD dwPatch) // Requested patch
{
    HRESULT hr;
    IDirectMusicInstrument* pInstrument;
    hr = pCollection->GetInstrument(dwPatch, &pInstrument);
    if (SUCCEEDED(hr))
    {
        dwPatch &= 0xFF00FFFF; // Clear MSB.
        dwPatch |= bMSB << 16; // Insert new MSB value.
        pInstrument->SetPatch(dwPatch);
        hr = pPort->DownloadInstrument(pInstrument,
            ppDLInstrument,
            NULL, 0); // Download all regions.
        pInstrument->Release();
    }
    return hr;
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicLoader

The **IDirectMusicLoader** interface is used for finding, enumerating, caching, and loading objects. For an overview, see DirectMusic Loader.

The methods of the **IDirectMusicLoader** interface can be organized into the following groups:

Searching	EnumObject
	ScanDirectory
	SetSearchDirectory
Caching	CacheObject
	ClearCache
	EnableCache
Object management	GetObject
	ReleaseObject
	SetObject

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

The **LPDMUS_LOADER** type is defined as a pointer to the **IDirectMusicLoader** interface.

```
typedef IDirectMusicLoader __RPC_FAR *LPDMUS_LOADER;
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicLoader::CacheObject

The **IDirectMusicLoader::CacheObject** method tells the loader to keep a reference to the object. This guarantees that the object is not loaded twice.

HRESULT CacheObject(

```
    IDirectMusicObject * pObject  
);
```

Parameters

pObject

Address of the **IDirectMusicObject** interface of the object to cache.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if the object is already cached.

If it fails, the method can return one of the following error values:

```
    E_POINTER  
    DMUS_E_LOADER_OBJECTNOTFOUND
```

Remarks

If you have an object that is accessed in multiple places throughout the life of your application, letting the loader cache the object can significantly speed up performance. For an overview, see [Caching Objects](#).

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader::EnableCache, **IDirectMusicLoader::ClearCache**,
IDirectMusicLoader::ReleaseObject

IDirectMusicLoader::ClearCache

The **IDirectMusicLoader::ClearCache** method tells the loader to release all references to a particular type of object.

```
HRESULT ClearCache(  
    REFGUID rguidClass  
);
```

Parameters

rguidClass

Reference to (C++) or address of (C) the identifier of the class of objects to clear. You can specify all types with GUID_DirectMusicAllTypes.

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 **IDirectMusicLoader::EnableCache** method to turn off automatic caching.

To clear a single object from the cache, call the **IDirectMusicLoader::ReleaseObject** method.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader::CacheObject, Caching Objects

IDirectMusicLoader::EnableCache

The **IDirectMusicLoader::EnableCache** method tells the loader to enable or disable automatic caching of all objects it loads. By default, caching is enabled for all classes.

```
HRESULT EnableCache(
    REFGUID rguidClass,
    BOOL fEnable
);
```

Parameters

rguidClass

Reference to (C++) or address of (C) the identifier of the class of objects to cache. You can specify all types with GUID_DirectMusicAllTypes.

fEnable

TRUE to enable caching; FALSE to clear and disable.

Return Values

The method returns **S_OK** if the cache state is changed, or **S_FALSE** if the cache is already in the desired state.

Remarks

To clear the cache without disabling caching, call the **IDirectMusicLoader::ClearCache** method.

The following code example disables caching only for segment objects so that they do not stay in memory after the application releases them. Other objects that should be shared, such as styles, chord maps and DLS collections, continue to be cached. The first call to **EnableCache** would normally be unnecessary because caching is enabled for all objects by default.

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

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in **dmusici.h**.

See Also

IDirectMusicLoader::CacheObject, **IDirectMusicLoader::ClearCache**, Caching Objects

IDirectMusicLoader::EnumObject

The **IDirectMusicLoader::EnumObject** method enumerates all available objects of the requested type. Objects are available if they have been loaded or if **IDirectMusicLoader::ScanDirectory** has been called on the search directory.

```
HRESULT EnumObject(
    REFGUID rguidClass,
    DWORD dwIndex,
    LPDMUS_OBJECTDESC pDesc
);
```

Parameters

rguidClass

Reference to (C++) or address of (C) the identifier for the class of objects to view.

dwIndex

Index into the list. Typically, starts with 0 and increments.

pDesc

Address of a **DMUS_OBJECTDESC** structure to be filled with data about the object.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if *dwIndex* is past the end of the list.

Remarks

For an example of the use of this method, see Enumerating Objects.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader::ScanDirectory

IDirectMusicLoader::GetObject

The **IDirectMusicLoader::GetObject** method retrieves the specified object from a file or resource and returns the desired interface.

```
HRESULT GetObject(  
    LPDMUS_OBJECTDESC pDESC,  
    REFIID riid,  
    LPVOID FAR *ppv  
);
```

Parameters

pDESC

Address of a **DMUS_OBJECTDESC** structure describing the object.

riid

Unique identifier of the interface. See the IID defines in Dmusici.h. All the standard interfaces have a defined identifier consisting of "IID_" plus the name of the interface. For example, the identifier of **IDirectMusicTrack** is IID_IDirectMusicTrack.

ppv

Address of a variable to receive a pointer to the desired interface of the object.

Return Values

If the method succeeds, the return value is S_OK or DMUS_S_PARTIALLOAD.

If it fails, the method can return one of the following error values:

E_FAIL
E_INVALIDARG
E_OUTOFMEMORY
E_POINTER
DMUS_E_LOADER_NOCLASSID
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED
REGDB_E_CLASSNOTREG

Remarks

A return value of DMUS_S_PARTIALLOAD can mean that the default instrument collection file, Gm.dls, is not available.

The current version of DirectMusic does not support loading from URLs. If the **dwValidData** member of the **DMUS_OBJECTDESC** structure contains DMUS_OBJ_URL, the method returns DMUS_E_LOADER_FORMATNOTSUPPORTED.

The method does not require that all valid members of the **DMUS_OBJECTDESC** structure match before retrieving an object. It searches in the following order:

1. DMUS_OBJ_OBJECT
2. DMUS_OBJ_MEMORY
3. DMUS_OBJ_FILENAME and DMUS_OBJ_FULLPATH
4. DMUS_OBJ_NAME and DMUS_OBJ_CATEGORY
5. DMUS_OBJ_NAME
6. DMUS_OBJ_FILENAME

In other words, the highest priority goes to a unique GUID, followed by a resource, followed by the full file path name, followed by an internal name plus category, followed by an internal name, followed by a local file name.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader::ReleaseObject, **IDirectMusicLoader::ScanDirectory**

IDirectMusicLoader::ReleaseObject

The **IDirectMusicLoader::ReleaseObject** method releases the loader's reference to the object.

```
HRESULT ReleaseObject(  
    IDirectMusicObject * pObject  
);
```

Parameters

pObject

Address of a variable that contains the object to release.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
DMUS_E_LOADER_OBJECTNOTFOUND

Remarks

ReleaseObject is the reciprocal of **IDirectMusicLoader::CacheObject**.

Objects can be cached explicitly by using the **CacheObject** method, or automatically by using the **IDirectMusicLoader::EnableCache** method.

To tell the loader to flush all objects of a particular type, call the **IDirectMusicLoader::ClearCache** method.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader::GetObject, Caching Objects

IDirectMusicLoader::ScanDirectory

The **IDirectMusicLoader::ScanDirectory** method searches a directory on disk for all files of a requested class type and file extension. For each file found, it calls the **IDirectMusicObject::ParseDescriptor** method to extract the GUID and name of the object. This information is stored in an internal database. Once a directory has been scanned, all files of the requested type become available for enumeration through the **IDirectMusicLoader::EnumObject** method; in addition, an object can be retrieved by using **IDirectMusicLoader::GetObject**, even without a file name.

```
HRESULT ScanDirectory(
    REFGUID rguidClass,
    WCHAR* pwzFileExtension,
    WCHAR* pwzScanFileName
);
```

Parameters

rguidClass

Reference to (C++) or address of (C) the identifier of the class of objects.

pwzFileExtension

File extension for the type of file to look for—for example, L"sty" for style files. Use L"*" to look in files with any or no extension.

pwzScanFileName

Name of an optional storage file to store and retrieve cached file information. This file is created by the first call to **ScanDirectory** and used by subsequent calls. Pass NULL if a cache file is not wanted.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if no files were found.

If it fails, the method can return one of the following error values:

```
DMUS_E_NOT_FOUND
E_FAIL
```

E_OUTOFMEMORY
E_POINTER
REGDB_E_CLASSNOTREG

Remarks

The **IDirectMusicLoader::SetSearchDirectory** method must be called first to set the location to search.

The scanned information can be stored in a cache file defined by *pwzScanFileName*. Once it has been so stored, subsequent calls to **ScanDirectory** are much quicker because only files that have changed are scanned (the cache file stores the file size and date for each object, so it can tell if a file has changed).

If the file type has more than one extension, call **ScanDirectory** once for each file extension.

GUID_DirectMusicAllTypes is not a valid value for *rguidClass*.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

Scanning a Directory for Objects

IDirectMusicLoader::SetObject

The **IDirectMusicLoader::SetObject** method tells the loader where to find an object when it is later referenced by another object being loaded, and adds attributes to an object so that it can be referred to by those attributes. For an overview, see Setting Objects.

```
HRESULT SetObject(  
    LPDMUS_OBJECTDESC pDESC  
);
```

Parameters

pDESC

Address of a **DMUS_OBJECTDESC** structure describing the object.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

- E_FAIL
- E_INVALIDARG
- E_OUTOFMEMORY
- E_POINTER
- DMUS_E_LOADER_NOCLASSID
- DMUS_E_LOADER_FAILEDOPEN
- DMUS_E_LOADER_FAILEDCREATE
- DMUS_E_LOADER_FORMATNOTSUPPORTED
- REGDB_E_CLASSNOTREG

Remarks

This method can be used to set attributes that are not currently valid for an object. For example, you can supply a value in the **wszName** member of the **DMUS_OBJECTDESC** structure to assign an internal name to an unnamed object, such as a segment based on a MIDI file. However, it cannot be used to change existing attributes. Most authored segments, for example, already have names, and these cannot be changed by the application.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dmusici.h.

See Also

IDirectMusicLoader::GetObject

IDirectMusicLoader::SetSearchDirectory

The **IDirectMusicLoader::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.

```
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.

pwszPath

File path for directory. Must be a valid directory and must be less than MAX_PATH in length.

fClear

If TRUE, clears all information about objects before setting the directory. This avoids accessing objects from the previous directory that might have the same name. However, objects are not removed from the cache.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if the search directory is already set to *pwszPath*.

If it fails, the method can return one of the following error values:

E_OUTOFMEMORY

E_POINTER

DMUS_E_LOADER_BADPATH

Remarks

Once a search path is set, the loader does not need a full path every time it is given an object to load by file name. This enables objects that refer to other objects to find them by file name without knowing the full path.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicLoader::ScanDirectory, Setting the Loader's Search Directory

IDirectMusicObject

All DirectMusic objects that can be loaded from a file support the **IDirectMusicObject** interface so that they can work with the DirectMusic loader.

New types of objects need to implement this interface. For more information, see Custom Loading.

Most applications do not use the methods of this interface directly. However, **IDirectMusicObject::GetDescriptor** can be used to query an object for information, including its name, GUID, file path, and version.

The **IDirectMusicObject** interface must be obtained by calling another interface's **QueryInterface** method. It cannot be obtained by using **CoCreateInstance**.

The **IDirectMusicObject** interface has the following methods:

Descriptor	GetDescriptor
	ParseDescriptor
	SetDescriptor

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

The **LPDMUS_OBJECT** type is defined as a pointer to the **IDirectMusicObject** interface.

```
typedef IDirectMusicObject __RPC_FAR *LPDMUS_OBJECT;
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

DirectMusic Loader, Custom Loading

IDirectMusicObject::GetDescriptor

The **IDirectMusicObject::GetDescriptor** method retrieves the object's internal description.

The method takes a **DMUS_OBJECTDESC** structure and fills in everything the object knows about itself.

```
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 know which other members are valid.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return **E_POINTER**.

Remarks

For an example, see [Getting Object Descriptors](#).

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusici.h`.

See Also

IDirectMusicObject::SetDescriptor

IDirectMusicObject::ParseDescriptor

Given a file stream, the **IDirectMusicObject::ParseDescriptor** method scans the file for data that it can store in the **DMUS_OBJECTDESC** structure. All members that are supplied are marked with the appropriate flags in **dwValidData**.

This method is primarily used by the loader when scanning a directory for objects, and is not normally used directly by an application. However, if an application implements an object type in `DirectMusic`, it should support this method.

```
HRESULT ParseDescriptor(  
    LPSTREAM pStream,  
    LPDMUS_OBJECTDESC pDesc  
);
```

Parameters

pStream

Stream source for the file.

pDesc

Address of a **DMUS_OBJECTDESC** structure to receive data about the file.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_CHUNKNOTFOUND

DMUS_E_INVALID_BAND

DMUS_E_INVALIDFILE

DMUS_E_NOTADLSCOL

E_POINTER

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicObject::SetDescriptor

IDirectMusicObject::SetDescriptor

The **IDirectMusicObject::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.

```
HRESULT SetDescriptor(  
    LPDMUS_OBJECTDESC pDesc  
);
```

Parameters

pDesc

Address of a **DMUS_OBJECTDESC** structure to receive data about the object.
Data is copied to all members that are enabled in the **dwValidData** member

Return Values

If the method succeeds, the return value is **S_OK**, or **S_FALSE** (see Remarks).

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_POINTER

Remarks

Applications do not normally call this method on standard objects. Although it is possible to change the object descriptor returned by **IDirectMusicObject::GetDescriptor**, the new description cannot successfully be passed to the **IDirectMusicLoader::GetObject** method. For example, you could change the name of an object, but **GetObject** still find the object only under its original name, since 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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in **dmusici.h**.

See Also

IDirectMusicObject::ParseDescriptor, **IDirectMusicObject::GetDescriptor**

IDirectMusicPerformance

The **IDirectMusicPerformance** interface is the overall manager of music playback. It is used for adding and removing ports, mapping performance channels to ports,

playing segments, dispatching messages and routing them through tools, requesting and receiving event notification, and setting and retrieving music parameters. It also has several methods for getting information about timing and for converting time and music values from one system to another.

If an application needs two complete sets of music playing at the same time, it can do so by creating more than one performance. Separate performances obey separate tempo maps, and so play completely asynchronously, whereas all segments within one performance play in lock step.

The methods of the **IDirectMusicPerformance** interface can be organized into the following groups:

Channels	AssignPChannel
	AssignPChannelBlock
	PChannelInfo
Instruments	DownloadInstrument
Messages	AllocPMsg
	FreePMsg
	SendPMsg
MIDI conversion	MIDIToMusic
	MusicToMIDI
Notification	AddNotificationType
	GetNotificationPMsg
	RemoveNotificationType
	SetNotificationHandle
Parameters	GetGlobalParam
	GetParam
	SetGlobalParam
	SetParam
Ports	AddPort
	RemovePort
Segments	GetSegmentState
	IsPlaying
	PlaySegment
	Stop
Timing	AdjustTime
	GetBumperLength
	GetLatencyTime
	GetPrepareTime
	GetQueueTime

	GetResolvedTime
	GetTime
	MusicToReferenceTime
	ReferenceToMusicTime
	RhythmToTime
	SetBumperLength
	SetPrepareTime
	TimeToRhythm
Tools	GetGraph
	SetGraph
Miscellaneous	CloseDown
	Init
	Invalidate

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicPerformance::AddNotificationType

The **IDirectMusicPerformance::AddNotificationType** method adds a notification type to the performance. All segments and tracks are automatically updated with the new notification by calling their **AddNotificationType** methods.

```
HRESULT AddNotificationType(
    REFGUID rguidNotificationType
);
```

Parameters

rguidNotificationType

Reference to (C++) or address of (C) the identifier of the notification type to add. For the defined types, see **DMUS_NOTIFICATION_PMSG**. Applications can also define their own types for custom tracks.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

E_OUTOFMEMORY

E_POINTER

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in **dmusici.h**.

See Also

IDirectMusicPerformance::RemoveNotificationType,

IDirectMusicSegment::AddNotificationType,

IDirectMusicTrack::AddNotificationType, Notification and Event Handling

IDirectMusicPerformance::AddPort

The **IDirectMusicPerformance::AddPort** method assigns a port to the performance.

```
HRESULT AddPort(
    IDirectMusicPort* pPort
);
```

Parameters

pPort

Address of a variable that contains the port to add. If **NULL**, the default port is added. See Remarks.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT

DMUS_E_CANNOT_OPEN_PORT
 E_OUTOFMEMORY
 E_POINTER

Remarks

If you want to pass NULL to this method, you must first pass NULL to **IDirectMusicPerformance::Init**.

When the default port is specified by passing NULL in *pPort*, it is assigned one channel group. If no PChannels have been set up for any other port, PChannels from 0 through 15 are assigned to MIDI channels from 0 through 15.

If *pPort* is not NULL, the port must be activated by a call to **IDirectMusicPort::Activate**, and a block of channels must be assigned by a call to **IDirectMusicPerformance::AssignPChannelBlock**.

This method creates a reference to **IDirectMusicPort** that is released by **IDirectMusicPerformance::RemovePort** or **IDirectMusicPerformance::CloseDown**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::RemovePort, Default Port

IDirectMusicPerformance::AdjustTime

The **IDirectMusicPerformance::AdjustTime** method adjusts the internal performance time forward or backward. This is mostly used to compensate for drift when synchronizing to another source.

```
HRESULT AdjustTime(
    REFERENCE_TIME rtAmount
);
```

Parameters

rtAmount

Amount of time to add or subtract. This can be a number from -10,000,000 through 10,000,000 (-1 second through +1 second).

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_INVALIDARG.

Remarks

The adjusted time is used internally by DirectMusic. It is not reflected in the time retrieved by the **IDirectMusicPerformance::GetTime** method.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::GetTime, Timing

IDirectMusicPerformance::AllocPMsg

The **IDirectMusicPerformance::AllocPMsg** method allocates a performance message.

```
HRESULT AllocPMsg(  
    ULONG cb,  
    DMUS_PMSG** ppPMSG  
);
```

Parameters

cb

Size of the message structure. For the various types, see Messages.

ppPMSG

Address of a variable to receive the pointer to the allocated message structure.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

- E_INVALIDARG
- E_OUTOFMEMORY
- E_POINTER

Remarks

The memory returned is not initialized to any particular state, with the exception of the **pTool** member of the **DMUS_PMSG** structure, which is initialized to 0, and the **dwSize** member, which is set to the value of *cb*.

Once the message is sent by **IDirectMusicPerformance::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 **IDirectMusicTool::Flush** or its **IDirectMusicTool::ProcessPMsg** method. Applications are also responsible for freeing notification messages.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in *dmusici.h*.

See Also

IDirectMusicPerformance::FreePMsg, **IDirectMusicPerformance::SendPMsg**, **DirectMusic Messages**

IDirectMusicPerformance::AssignPChannel

The **IDirectMusicPerformance::AssignPChannel** method assigns a single performance channel (PChannel) to the performance and maps it to a port, group, and MIDI channel.

```
HRESULT AssignPChannel(
    DWORD dwPChannel,
    IDirectMusicPort* pPort,
    DWORD dwGroup,
    DWORD dwMChannel
);
```

Parameters

dwPChannel
PChannel to assign.

pPort
Address of a variable that contains the port to which the PChannel is assigned.

dwGroup

Channel group on the port.

dwMChannel

Channel in the group. Must be in the range from 0 through 15.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE (see Remarks).

If it fails, the method can return one of the following error values:

E_INVALIDARG

E_POINTER

Remarks

The method returns S_FALSE if *dwGroup* is out of the range of the port. The channel has been assigned, but the port cannot play this group.

The method returns E_INVALIDARG if *dwMChannel* is out of range or the port has not been added to the performance through a call to the **IDirectMusicPerformance::AddPort** method.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::AssignPChannelBlock,
IDirectMusicPerformance::PChannelInfo, Channels

IDirectMusicPerformance::AssignPChannelBlock

The **IDirectMusicPerformance::AssignPChannelBlock** method assigns a block of 16 performance channels (PChannels) to the performance and maps them to a port and a channel group. 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 **IDirectMusicPerformance::AddPort**.

```
HRESULT AssignPChannelBlock(
    DWORD dwBlockNum,
    IDirectMusicPort* pPort,
    DWORD dwGroup
```


);

Parameters

dwBlockNum

Block number, in which 0 represents channels 0 through 15, 1 represents channels 16 through 31, and so on.

pPort

Address of a variable that contains the port to which the channels are assigned.

dwGroup

Channel group on the port. Must be 1 or greater.

Return Values

If the method succeeds, the return value is S_OK or S_FALSE (see Remarks).

If it fails, the method can return one of the following error values:

E_INVALIDARG

E_POINTER

Remarks

The method returns S_FALSE if *dwGroup* is out of the range of the port. The channels have been assigned, but the port cannot play this group.

The method returns E_INVALIDARG if the port has not been added to the performance through a call to the **IDirectMusicPerformance::AddPort** method.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::AssignPChannel,
IDirectMusicPerformance::PChannelInfo, Channels

IDirectMusicPerformance::CloseDown

The **IDirectMusicPerformance::CloseDown** method closes down the performance object. An application that created the performance object and called **IDirectMusicPerformance::Init** on it must call **CloseDown** before the performance is released.

HRESULT CloseDown();

Parameters

None.

Return Values

The method returns S_OK.

Remarks

Failure to call **CloseDown** can cause memory leaks or program failures.

CloseDown handles the release of the **IDirectMusic** interface if the application passed NULL, or a pointer to NULL, to **IDirectMusicPerformance::Init**, causing a DirectMusic object to be created. In cases in which the application explicitly created the DirectMusic object and passed the pointer to **Init**, the application is responsible for releasing the **IDirectMusic** interface.

CloseDown also releases any downloaded instruments that have not been unloaded.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::Init

IDirectMusicPerformance::DownloadInstrument

The **IDirectMusicPerformance::DownloadInstrument** method downloads DLS data for an instrument to a port.

```
HRESULT DownloadInstrument(
    IDirectMusicInstrument* pInst,
    DWORD dwPChannel,
    IDirectMusicDownloadedInstrument** ppDownInst,
    DMUS_NOTERANGE* pNoteRanges,
    DWORD dwNumNoteRanges,
    IDirectMusicPort** ppPort,
    DWORD* pdwGroup,
```

```
DWORD* pdwMChannel
);
```

Parameters

pInst

Address of a variable that contains the instrument to download.

dwPChannel

PChannel to which the instrument is assigned.

ppDownInst

Address of a variable to receive a pointer to the downloaded instrument.

pNoteRanges

Address of an array of **DMUS_NOTERANGE** structures. Each entry in the array specifies a contiguous range of MIDI note messages to which the instrument must respond. An instrument region is downloaded only if at least one note in that region is specified in the **DMUS_NOTERANGE** structures.

dwNumNoteRanges

Number of **DMUS_NOTERANGE** structures in the array pointed to by *pNoteRanges*. If this value is set to 0, the *pNoteRanges* parameter is ignored, and all regions and wave data for the instrument are downloaded.

ppPort

Address of a variable to receive a pointer to the port to which the instrument was downloaded.

pdwGroup

Address of a variable to receive the group to which the instrument is assigned.

pdwMChannel

Address of a variable to receive the MIDI channel to which the instrument is assigned.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG

E_POINTER

Remarks

Most applications do not need to use this method because instrument downloading is normally handled by bands. See Downloading and Unloading Bands.

The method returns E_INVALIDARG if the PChannel is not assigned to a port.

To prevent loss of resources, unload the instrument by using the **IDirectMusicPort::UnloadInstrument** method when the instrument is no longer needed.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPort::DownloadInstrument, **IDirectMusicPort::UnloadInstrument**,
Working with Instruments

IDirectMusicPerformance::FreePMsg

The **IDirectMusicPerformance::FreePMsg** method frees a message.

```
HRESULT FreePMsg(  
    DMUS_PMSG* pPMSG  
);
```

Parameters

pPMSG

Address of a variable that contains a message to free. This message must have been allocated using the **IDirectMusicPerformance::AllocPMsg** method.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_CANNOT_FREE
E_POINTER

Remarks

Most messages are released automatically by the performance once they have been processed, and **IDirectMusicPerformance::FreePMsg** must not be called on a message that has been sent by using **IDirectMusicPerformance::SendPMsg**. However, **IDirectMusicPerformance::FreePMsg** can be used within **IDirectMusicTool::ProcessPMsg** or **IDirectMusicTool::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 currently in the performance queue because **IDirectMusicPerformance::SendPMsg** was called on it.
- 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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusici.h`.

See Also

IDirectMusicPerformance::AllocPMsg, DirectMusic Messages

IDirectMusicPerformance::GetBumperLength

The **IDirectMusicPerformance::GetBumperLength** method retrieves the amount of time between the time at which messages are placed in the port buffer and the time at which they begin to be processed by the port. For an overview of this topic, see [Timing](#).

```
HRESULT GetBumperLength(
    DWORD* pdwMilliseconds
);
```

Parameters

pdwMilliseconds

Address of a variable that contains the amount of preplay time.

Return Values

If the method succeeds, the return value is `S_OK`.

If it fails, the method can return `E_POINTER`.

Remarks

The default value is 50 milliseconds.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::SetBumperLength

IDirectMusicPerformance::GetGlobalParam

The **IDirectMusicPerformance::GetGlobalParam** method retrieves global values from the performance.

```
HRESULT GetGlobalParam(
    REFGUID rguidType,
    void* pParam,
    DWORD dwSize
);
```

Parameters

rguidType

Reference to (C++) or address of (C) the identifier of the type of data.

pParam

Pointer to the allocated memory to receive 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

IDirectMusicPerformance::SetGlobalParam method.

dwSize

Size of the data. This is constant for each *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG

E_POINTER

Remarks

If **SetGlobalParam** has never been called for *rguidType*, the parameter might not be in the list of global data being handled by this performance, and the method might return `E_INVALIDARG`. In other words, do not assume that any parameter has a default value that can be retrieved by using **GetGlobalParam**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusici.h`.

See Also

IDirectMusicPerformance::SetGlobalParam,
IDirectMusicPerformance::GetParam, Music Parameters

IDirectMusicPerformance::GetGraph

The **IDirectMusicPerformance::GetGraph** method retrieves the tool graph of a performance.

```
HRESULT GetGraph(
    IDirectMusicGraph** ppGraph
);
```

Parameters

ppGraph

Address of a variable to receive a pointer to the tool graph.

Return Values

If the method succeeds, the return value is `S_OK`.

If it fails, the method can return one of the following error values:

```
DMUS_E_NOT_FOUND
E_POINTER
```

Remarks

The reference count of the graph is incremented.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::SetGraph, **IDirectMusicSegment::GetGraph**,
IDirectMusicPerformance::SendPMsg

IDirectMusicPerformance::GetLatencyTime

The **IDirectMusicPerformance::GetLatencyTime** method retrieves the current latency time. Latency time is the time being heard from the speakers plus the time required to queue and render messages into the **IDirectMusicPort**. For an overview of this topic, see [Timing](#).

```
HRESULT GetLatencyTime(  
    REFERENCE_TIME * prtTime  
);
```

Parameters

prtTime

Address of a variable to receive the current latency time.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
DMUS_E_NO_MASTER_CLOCK

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicPerformance::GetNotificationPMsg

The **IDirectMusicPerformance::GetNotificationPMsg** method retrieves a pending notification message.

```
HRESULT GetNotificationPMsg(  
    DMUS_NOTIFICATION_PMSG** ppNotificationPMsg  
);
```

Parameters

ppNotificationPMsg

Address of a variable to receive a pointer to a **DMUS_NOTIFICATION_PMSG** structure. The application retrieving this message is responsible for calling **IDirectMusicPerformance::FreePMsg** on it.

Return Values

If the method succeeds, the return value is **S_OK** or, **S_FALSE** if there are no more notification events to return.

If it fails, the method can return **E_POINTER**.

Remarks

For an example, see Handle Notifications in the tutorial on using compositions.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in *dmusici.h*.

See Also

Notification and Event Handling

IDirectMusicPerformance::GetParam

The **IDirectMusicPerformance::GetParam** method retrieves data from a track inside the control segment.

```
HRESULT GetParam(  
    REFGUID rguidType,  
    DWORD dwGroupBits,
```

```

DWORD dwIndex,
MUSIC_TIME mtTime,
MUSIC_TIME* pmtNext,
void* pParam
);

```

Parameters

rguidType

Reference to (C++) or address of (C) the identifier of the type of data to obtain.
See Track Parameter Types.

dwGroupBits

Group that the desired track is in (see Remarks). Set this value to 0xFFFFFFFF for all groups.

dwIndex

Index of the track in the group from which to obtain the data.

mtTime

Time from which to obtain the data, in performance time.

pmtNext

Address of a variable to receive the time (relative to *mtTime*) until which the data is valid. If this returns a value of 0, either the data is always valid, or it is not known when it might become invalid. If this information is not needed, *pmtNext* can be set to NULL. See Remarks.

pParam

Address of an allocated structure in which the data is to be returned. The structure must be of the appropriate kind and size for the data type identified by *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

```

DMUS_E_GET_UNSUPPORTED
DMUS_E_NO_MASTER_CLOCK
DMUS_E_NOT_FOUND
DMUS_E_TRACK_NOT_FOUND
E_POINTER

```

Remarks

Normally, the primary segment is the control segment. However, a secondary segment can be designated as a control segment when it is played. For more information on control segments, see Segments and **DMUS_SEGF_FLAGS**.

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 **IDirectMusicSegment::InsertTrack** and Identifying the Track.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in *dmusici.h*.

See Also

IDirectMusicPerformance::SetParam, **IDirectMusicSegment::GetParam**, **IDirectMusicTrack::GetParam**, **IDirectMusicPerformance::SetGlobalParam**, **IDirectMusicPerformance::GetTime**, Music Parameters

IDirectMusicPerformance::GetPrepareTime

The **IDirectMusicPerformance::GetPrepareTime** method retrieves the amount of time ahead that **IDirectMusicTrack::Play** is called before the messages is heard through the speakers. This interval allows sufficient time for the message to be processed by tools.

```
HRESULT GetPrepareTime(
    DWORD* pdwMilliseconds
);
```

Parameters

pdwMilliseconds

Address of a variable to receive the amount of prepare time.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return **E_POINTER**.

Remarks

The default value is 1000 milliseconds.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::SetPrepareTime, Timing

IDirectMusicPerformance::GetQueueTime

The **IDirectMusicPerformance::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.

```
HRESULT GetQueueTime(
    REFERENCE_TIME * priTime
);
```

Parameters

priTime

Address of a variable to receive the current flush time.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

```
E_POINTER
DMUS_E_NO_MASTER_CLOCK
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

Latency and Bumper Time

IDirectMusicPerformance::GetResolvedTime

The **IDirectMusicPerformance::GetResolvedTime** method resolves a given time to a given boundary.

```
HRESULT GetResolvedTime(  
    REFERENCE_TIME rtTime,  
    REFERENCE_TIME* prtResolved,  
    DWORD dwTimeResolveFlags  
);
```

Parameters

rtTime

Time to resolve. If this is less than the current time, the current time is used.

prtResolved

Address of a variable to receive the resolved time.

dwTimeResolveFlags

One or more **DMUS_TIME_RESOLVE_FLAGS** describing the resolution desired.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return **E_POINTER**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in **dmusici.h**.

See Also

Timing

IDirectMusicPerformance::GetSegmentState

The **IDirectMusicPerformance::GetSegmentState** method retrieves the currently playing primary segment state or the primary segment state that is playing at a given time.

```
HRESULT GetSegmentState(  
    IDirectMusicSegmentState ** ppSegmentState,  
    MUSIC_TIME mtTime  
);
```

Parameters

ppSegmentState

Address of a variable to receive a pointer to the segment state. The caller is responsible for calling **Release** on this pointer.

mtTime

Time for which the segment state is to be retrieved.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_FOUND
E_POINTER

Remarks

To get the currently playing segment state, pass the time returned by the **IDirectMusicPerformance::GetTime** method. The currently playing segment state represents the segment currently generating messages. Because of latency, the currently playing segment state is not necessarily the one being heard.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicPerformance::GetTime

The **IDirectMusicPerformance::GetTime** method retrieves the current time of the performance.

```
HRESULT GetTime(  
    REFERENCE_TIME* prtNow,  
    MUSIC_TIME* pmtNow  
);
```

Parameters

priNow

Address of a variable to receive the current time in **REFERENCE_TIME** format. Can be NULL.

pmtNow

Address of a variable to receive the current time in **MUSIC_TIME** format. Can be NULL.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

DMUS_E_NO_MASTER_CLOCK
E_POINTER

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in **dmusic.h**.

See Also

Timing

IDirectMusicPerformance::Init

The **IDirectMusicPerformance::Init** method associates the performance with a **DirectMusic** object and a **DirectSound** object.

```
HRESULT Init(  
    IDirectMusic** ppDirectMusic,  
    LPGUID pDirectSound,  
    HWND hWnd  
);
```

Parameters

ppDirectMusic

Address of a variable containing the **IDirectMusic** interface pointer to be assigned to the performance, if one already exists. The reference count of the interface is incremented. Ports passed to the

IDirectMusicPerformance::AddPort method must be created from this DirectMusic object.

If the variable contains NULL, a DirectMusic object is created, and the interface pointer is returned.

If *ppDirectMusic* is NULL, a DirectMusic object is created and used internally by the performance.

See Remarks.

pDirectSound

Address of a **IDirectSound** interface to use by default for wave output. If this value is NULL, DirectMusic creates a DirectSound object. There should, however, only be one DirectSound object per process. If your application uses DirectSound separately, it should pass in that interface here, or to **IDirectMusic::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 **IDirectSound::SetCooperativeLevel**.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_ALREADY_INITED
E_OUTOFMEMORY
E_POINTER

Remarks

This method should be called only once; it must be called before the performance can play.

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 **IDirectMusic** pointer in *ppDirectMusic*. In this case, the *pDirectSound* and *hWnd* parameters are ignored because the application is responsible for calling **IDirectMusic::SetDirectSound**.

- The application allows the performance to create the DirectMusic object and wants 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 **IDirectMusicPerformance::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. In general, the top-level application window handle is passed to DirectMusic, DirectSound, and DirectDraw. See the Remarks for **IDirectSound::SetCooperativeLevel**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in *dmusici.h*.

See Also

Creating the Performance, Integrating DirectMusic and DirectSound

IDirectMusicPerformance::Invalidate

The **IDirectMusicPerformance::Invalidate** method flushes all queued messages from the supplied time forward and causes all tracks of all segments to resend their data from the given time forward.

```
HRESULT Invalidate(
    MUSIC_TIME mtTime,
    DWORD dwFlags
);
```

Parameters

mtTime

Time from which to invalidate, adjusted by *dwFlags*. Setting this value to 0 causes immediate invalidation.

dwFlags

Flags that adjust *mtTime* to align to measures, beats, or grids. This value can be 0 or one of the following members of the **DMUS_SEGF_FLAGS** enumeration:

DMUS_SEGF_MEASURE
 DMUS_SEGF_BEAT
 DMUS_SEGF_GRID

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DMUS_E_NO_MASTER_CLOCK.

Remarks

If *mtTime* is so long ago that it is impossible to invalidate that time, the earliest possible time is used.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

Prepare Time, Segment Timing

IDirectMusicPerformance::IsPlaying

The **IDirectMusicPerformance::IsPlaying** method determines whether a particular segment or segment state is currently being heard from the speakers.

```
HRESULT IsPlaying(
    IDirectMusicSegment* pSegment,
    IDirectMusicSegmentState* pSegState
);
```

Parameters

pSegment

Segment to check. If NULL, check only *pSegState*.

pSegState

Segment state to check. If NULL, check only *pSegment*.

Return Values

If the method succeeds and the requested segment or segment state is playing, the return value is `S_OK`. If neither is playing or only one was requested and it is not playing, the return value is `S_FALSE`.

If it fails, the method can return one of the following error values:

`E_POINTER`
`DMUS_E_NO_MASTER_CLOCK`

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusici.h`.

IDirectMusicPerformance::MIDIToMusic

The **IDirectMusicPerformance::MIDIToMusic** method converts a MIDI note value to a DirectMusic music value, using a supplied chord, subchord level, and play mode.

```
HRESULT MIDIToMusic(
    BYTE bMIDIValue,
    DMUS_CHORD_KEY* pChord,
    BYTE bPlayMode,
    BYTE bChordLevel,
    WORD *pwMusicValue
);
```

Parameters

bMIDIValue

MIDI note value to convert, in the range from 0 through 127.

pChord

Address of a **DMUS_CHORD_KEY** structure containing information about the chord and key structure to be used in translating the note. This includes the underlying scale. For example, if the chord is a CM7, the note is interpreted against the chord positions for root note C, chord intervals of a major seventh. The structure carries up to eight 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 to receive the music value. For information on this value, see **DMUS_NOTE_PMSG.**

Return Values

If the method succeeds, the return value is one of the following. See Remarks.

S_OK

DMUS_S_DOWN_OCTAVE

DMUS_S_UP_OCTAVE

If it fails, the method can return one of the following error values:

DMUS_E_CANNOT_CONVERT

E_INVALIDARG

Remarks

If the method fails, **pwMusicValue* is not changed.

If the return value is DMUS_S_UP_OCTAVE or DMUS_DOWN_OCTAVE, the note conversion generated a note value that is less than 0 or greater than 127, so it has been bumped up or down one or more octaves to be in the proper MIDI range of from 0 through 127. This can occur when using play modes

DMUS_PLAYMODE_FIXEDTOCHORD and

DMUS_PLAYMODE_FIXEDTOKEY, both of which return MIDI values in **pwMusicValue*.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::MusicToMIDI, Music Values and MIDI Notes

IDirectMusicPerformance::MusicToMIDI

The **IDirectMusicPerformance::MusicToMIDI** method converts a DirectMusic music value to a MIDI note value.

HRESULT MusicToMIDI(

```
WORD wMusicValue,  
DMUS_CHORD_KEY* pChord,  
BYTE bPlayMode,  
BYTE bChordLevel,  
BYTE *pbMIDIValue  
);
```

Parameters

wMusicValue

Music value to convert. For information on music values, see **DMUS_NOTE_PMSG**.

pChord

Address of a **DMUS_CHORD_KEY** structure containing information about the chord and key structure to be used in translating the note. This includes the underlying scale. For example, if the chord is a CM7, the note is interpreted against the chord positions for root note C, chord intervals of a major seventh. The structure carries up to eight 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 to receive the MIDI value, in the range from 0 through 127.

Return Values

If the method succeeds, the return value is one of the following. See Remarks.

S_OK
DMUS_S_OVER_CHORD
DMUS_S_DOWN_OCTAVE
DMUS_S_UP_OCTAVE

If it fails, the method can return one of the following error values:

DMUS_E_CANNOT_CONVERT
E_INVALIDARG

Remarks

If the method fails or returns `DMUS_S_OVER_CHORD`, **pwMIDIValue* is not changed.

The method returns `DMUS_S_OVER_CHORD` if no note has been calculated because the music value has the note at a position higher than the top note of the chord. This applies only to `DMUS_PLAYMODE_NORMALCHORD` play mode. The caller should not do anything with the note, which is not meant to be played against this chord.

If the return value is `DMUS_S_UP_OCTAVE` or `DMUS_DOWN_OCTAVE`, the note conversion generated a note value that is less than 0 or greater than 127, so it has been bumped up or down one or more octaves to be in the proper MIDI range of 0 through 127. This can occur when using any play mode except `DMUS_PLAYMODE_FIXED`.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusici.h`.

See Also

`IDirectMusicPerformance::MIDIToMusic`, Music Values and MIDI Notes

IDirectMusicPerformance::MusicToReferenceTime

The `IDirectMusicPerformance::MusicToReferenceTime` method converts time in `MUSIC_TIME` format to time in `REFERENCE_TIME` format.

```
HRESULT MusicToReferenceTime(
    MUSIC_TIME mtTime,
    REFERENCE_TIME* prtTime
);
```

Parameters

mtTime

Time in `MUSIC_TIME` format to convert.

prtTime

Address of a variable to receive the converted time in `REFERENCE_TIME` format.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
DMUS_E_NO_MASTER_CLOCK

Remarks

Because reference time has a greater precision than music time, a time that has been converted from reference time to music time, and then back again, probably does not have its original value.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::ReferenceToMusicTime, Clock Time vs. Music Time

IDirectMusicPerformance::PChannelInfo

The **IDirectMusicPerformance::PChannelInfo** method retrieves the port, group, and MIDI channel for a given performance channel.

```
HRESULT PChannelInfo(
    DWORD dwPChannel,
    IDirectMusicPort** ppPort,
    DWORD* pdwGroup,
    DWORD* pdwMChannel
);
```

Parameters

dwPChannel

PChannel for which information is desired.

ppPort

Address of a variable to receive an **IDirectMusicPort** pointer. This value can be NULL if the pointer is not wanted. If a non-NULL pointer is returned, the reference count is incremented, and it is the responsibility of the application to call **Release** on the pointer. See also Remarks.

pdwGroup

Address of a variable to receive the group on the port. Can be NULL if this value is not wanted.

pdwMChannel

Address of a variable to receive the MIDI channel on the group. Can be NULL if this value is not wanted.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG

E_POINTER

Remarks

A NULL pointer is returned in **ppPort* if the port has been removed by a call to **IDirectMusicPerformance::RemovePort**, but the method succeeds.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::AssignPChannel,
IDirectMusicPerformance::AssignPChannelBlock

IDirectMusicPerformance::PlaySegment

The **IDirectMusicPerformance::PlaySegment** method begins playback of a segment.

```
HRESULT PlaySegment(
    IDirectMusicSegment* pSegment,
    DWORD dwFlags,
    __int64 i64StartTime,
    IDirectMusicSegmentState** ppSegmentState
);
```


Parameters

pSegment

Segment to play.

dwFlags

Flags that modify the method's behavior. See **DMUS_SEGF_FLAGS**.

i64StartTime

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 to receive a pointer to the segment state for this instance of the playing segment. This field can be NULL. If it is non-NULL, the segment state pointer is returned, and the application must call **Release** on it.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

E_OUTOFMEMORY

E_POINTER

DMUS_E_NO_MASTER_CLOCK

DMUS_E_SEGMENT_INIT_FAILED

DMUS_E_TIME_PAST

Remarks

Segments should be greater than 250 milliseconds in length.

The boundary resolutions in *dwFlags* are relative to the currently playing primary segment.

If a primary segment is scheduled to play while another primary segment is playing, the first one stops unless you set the **DMUS_SEGF_QUEUE** flag for the second segment, in which case it plays as soon as the first one finishes.

For more information on the exact start time of segments, see Segment Timing. For information on how the start time of segments can be affected by tempo changes, see Clock Time vs. Music Time.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Header: Declared in dmusici.h.

See Also

Segments

IDirectMusicPerformance::ReferenceToMusicTime

The **IDirectMusicPerformance::ReferenceToMusicTime** method converts time in **REFERENCE_TIME** format to time in **MUSIC_TIME** format.

```
HRESULT ReferenceToMusicTime(  
    REFERENCE_TIME rtTime,  
    MUSIC_TIME* pmtTime  
);
```

Parameters

rtTime

Time in **REFERENCE_TIME** format.

pmtTime

Address of a variable to receive the converted time in **MUSIC_TIME** format.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

E_POINTER

DMUS_E_NO_MASTER_CLOCK

Remarks

If a master tempo has been set for the performance, it is taken into account when converting to music time. See [Setting and Retrieving Global Parameters](#).

Because music time is less precise than reference time, rounding off occurs.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::MusicToReferenceTime, Clock Time vs. Music Time

IDirectMusicPerformance::RemoveNotificationType

The **IDirectMusicPerformance::RemoveNotificationType** method removes a previously added notification type from the performance. All segments and tracks are updated by a call to their **RemoveNotificationType** methods.

```
HRESULT RemoveNotificationType(
    REFGUID rguidNotificationType
);
```

Parameters

rguidNotificationType

Reference to (C++) or address of (C) the identifier of the notification type to remove. (For the defined types, see **DMUS_NOTIFICATION_PMSG**.) If this value is GUID_NULL, all notifications are to be removed.

Return Values

If the method succeeds, the return value is S_OK or S_FALSE (see Remarks).

If it fails, the method can return E_POINTER.

Remarks

S_FALSE is returned when *rguidNotificationType* is not an active notification.

If a notification was added to a segment that has stopped playing, the performance cannot remove the notification type from that segment because it no longer has a reference to the segment.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::AddNotificationType,
IDirectMusicSegment::RemoveNotificationType,
IDirectMusicTrack::RemoveNotificationType, Notification and Event Handling

IDirectMusicPerformance::RemovePort

The **IDirectMusicPerformance::RemovePort** method removes a port from the performance. Any PChannels that map to this port are invalidated, and messages stamped with them do not play.

```
HRESULT RemovePort(  
    IDirectMusicPort* pPort  
);
```

Parameters

pPort
Port to remove.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_POINTER

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::AddPort

IDirectMusicPerformance::RhythmToTime

The **IDirectMusicPerformance::RhythmToTime** method converts rhythm time to music time.

```

HRESULT RhythmToTime(
    WORD wMeasure,
    BYTE bBeat,
    BYTE bGrid,
    short nOffset,
    DMUS_TIMESIGNATURE *pTimeSig,
    MUSIC_TIME *pmtTime)
);

```

Parameters

wMeasure

Measure of the time to convert.

bBeat

Beat of the time to convert.

bGrid

Grid of the time to convert.

nOffset

Offset from the grid, in music-time ticks, of the time to convert.

pTimeSig

Address of a **DMUS_TIMESIGNATURE** structure containing information about the time signature.

pmtTime

Address of a variable to receive the music time.

Return Values

If the method succeeds, the return value is S_OK.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::TimeToRhythm

IDirectMusicPerformance::SendPMsg

The **IDirectMusicPerformance::SendPMsg** method sends a performance message. This method is called by tracks when they are played. It might also be called by a tool to inject new data into a performance.

```

HRESULT SendPMsg(
    DMUS_PMSG* pPMSG
);

```

Parameters

pPMSG

Message allocated by **IDirectMusicPerformance::AllocPMsg**.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

```

    DMUS_E_NO_MASTER_CLOCK
    DMUS_E_ALREADY_SENT
    E_INVALIDARG
    E_POINTER

```

Remarks

The **dwFlags** member (see **DMUS_PMSG**) must contain either **DMUS_PMSGF_MUSICTIME** or **DMUS_PMSGF_REFTIME**, depending on the time stamp in either **rtTime** or **mtTime**. The **dwFlags** member should also contain the appropriate delivery type—**DMUS_PMSGF_TOOL_QUEUE**, **DMUS_PMSGF_TOOL_ATTIME**, or **DMUS_PMSGF_TOOL_IMMEDIATE**—depending on the type of message. If none is selected, **DMUS_PMSGF_TOOL_IMMEDIATE** is used by default.

If the time of the message is set to 0 and the **dwFlags** member contains **DMUS_PMSGF_REFTIME**, it is assumed that this message is cued to go out immediately.

In most cases, the **IDirectMusicGraph::StampPMsg** method should be called on the message before **SendPMsg** is called. However, when sending a message directly to the main output tool, this step can be skipped. If you want the message to pass only through the performance graph, obtain the **IDirectMusicGraph** interface by calling **IDirectMusicPerformance::QueryInterface**. Otherwise, obtain it by calling **IDirectMusicSegment::QueryInterface**. Do not attempt to obtain the interface by calling **IDirectMusicPerformance::GetGraph** or **IDirectMusicSegment::GetGraph**; these methods return a pointer to the graph object, rather than to the implementation of the **IDirectMusicGraph** interface on the performance or segment.

Normally, the performance frees the message after it has been processed. For more information, see the Remarks for **IDirectMusicPerformance::FreePMsg**.

The follow code example shows how to allocate and send a system-exclusive message and a tempo message:

```

/* Assume that pPerformance is a valid IDirectMusicPerformance
   pointer and that mtTime is an initialized MUSIC_TIME
   variable. */

IDirectMusicGraph* pGraph;

/* Get the graph pointer from the performance. If you wanted the
   message to go through a segment graph, you would
   QueryInterface a segment object instead. */

if ( SUCCEEDED( pPerformance->QueryInterface(
    IID_IDirectMusicGraph, (void**)&pGraph )))
{
    // Allocate a DMUS_SYSEX_PMSG of the appropriate size,
    // and read the system-exclusive data into it.

    DMUS_SYSEX_PMSG* pSysEx;

    if ( SUCCEEDED( pPerformance->AllocPMsg(
        sizeof(DMUS_SYSEX_PMSG) + m_dwSysExLength,
        (DMUS_PMSG*)&pSysEx )))
    {
        // All fields are initialized to 0 from the
        // AllocPMsg method.
        // Assume that m_pbSysExData is a pointer to an array
        // containing data of length m_dwSysExLength.

        memcpy( pSysEx->abData, m_pbSysExData, m_dwSysExLength );
        pSysEx->dwSize = sizeof(DMUS_SYSEX_PMSG);
        pSysEx->dwLen = dwSysExLength;
        pSysEx->mtTime = mtTime;
        pSysEx->dwFlags = DMUS_PMSGF_MUSICTIME;
        pSysEx->dwType = DMUS_PMSGT_SYSEX;

        pGraph->StampPMsg( (DMUS_PMSG*)pSysEx );
        if ( FAILED(pPerformance->SendPMsg( (DMUS_PMSG*)pSysEx )))
        {
            pPerformance->FreePMsg( (DMUS_PMSG*)pSysEx );
        }
    }

    // Change the tempo at time mtTime to 120 bpm.

```

```

DMUS_TEMPO_PMSG* pTempo;

if( SUCCEEDED( pPerformance->AllocPMsg(
    sizeof(DMUS_TEMPO_PMSG),
    (DMUS_PMSG**)&pTempo )))
{
    pTempo->dwSize = sizeof(DMUS_TEMPO_PMSG);
    pTempo->dblTempo = 120;
    pTempo->mtTime = mtTime;
    pTempo->dwFlags = DMUS_PMSGF_MUSICTIME;
    pTempo->dwType = DMUS_PMSGT_TEMPO;
    pGraph->StampPMsg( (DMUS_PMSG*)pTempo );
    if (FAILED(pPerformance->SendPMsg( (DMUS_PMSG*)pTempo )))
    {
        pPerformance->FreePMsg( (DMUS_PMSG*)pTempo );
    }
}

pGraph->Release();
}

```

The next code example shows a function that sends a note message associated with the track identified by *dwTrackID*. The virtual track ID should be 0 if the message is not being generated from a *DirectMusicTrack* object.

```

HRESULT CreateNotePMsg(IDirectMusicPerformance* pPerformance,
    MUSIC_TIME mtTime, DWORD dwTrackID)
{
    // Allocate a Note PMessage.
    DMUS_NOTE_PMSG* pNote = NULL;
    HRESULT hr = pPerformance->AllocPMsg( sizeof(DMUS_NOTE_PMSG),
        (DMUS_PMSG**) &pNote);
    if (FAILED(hr)) return hr;

    pNote->dwSize = sizeof(DMUS_NOTE_PMSG); // Size of a Note PMessage
    pNote->rtTime = 0; // Ignored
    pNote->mtTime = mtTime; // When to play the note
    pNote->dwFlags = DMUS_PMSGF_MUSICTIME; // Use the mtTime field.
    pNote->dwPChannel = 5; // Play on PChannel 5.
    pNote->dwVirtualTrackID = dwTrackID; // Track ID from parameter.

    // The following two fields should be set to NULL when a
    // message is initially sent. They will be updated in
    // IDirectMusicGraph::StampPMsg.
    pNote->pTool = NULL;
}

```

```
pNote->pGraph = NULL;
pNote->dwType = DMUS_PMSGT_NOTE;
pNote->dwVoiceID = 0;          // For DirectX 6.1, always 0
pNote->dwGroupID = 0xFFFFFFFF; // All track groups
pNote->punkUser = NULL;        // Always NULL

// Get the current time signature from the performance
// to compute measure and beat information.
DMUS_TIMESIGNATURE TimeSig;
MUSIC_TIME mtNext;
hr = pPerformance->GetParam(GUID_TimeSignature, 0xFFFFFFFF,
    0, mtTime, &mtNext, &TimeSig);
if (FAILED(hr)) return hr;

// Recompute TimeSig.mtTime to have the value expected
// by pPerformance->TimeToRhythm.
TimeSig.mtTime += mtTime;

// Get the current chord from the performance
// to create a note value.
DMUS_CHORD_KEY Chord;
hr = pPerformance->GetParam(GUID_ChordParam, 0xFFFFFFFF, 0,
    mtTime, &mtNext, &Chord);
if (FAILED(hr)) return hr;

// Create a note with octave 5, chord tone 2 (fifth), scale
// offset 1 (=> sixth), and no accidentals.
WORD wMusicValue = 0x5210;

// Use DMUS_PLAYMODE_PEDALPOINT as your play mode
// in pPerformance->MusicToMIDI.
BYTE bPlayModeFlags = DMUS_PLAYMODE_PEDALPOINT;

// Fill in the fields specific to DMUS_NOTE_PMSG.
pNote->wMusicValue = wMusicValue;
hr = pPerformance->MusicToMIDI(
    wMusicValue,
    &Chord,
    bPlayModeFlags,
    0,
    &(pNote->bMidiValue));
if (FAILED(hr)) return hr;

hr = pPerformance->TimeToRhythm(
    TimeSig.mtTime,
    &TimeSig,
```

```

        &(pNote->wMeasure),
        &(pNote->bBeat),
        &(pNote->bGrid),
        &(pNote->nOffset));
    if (FAILED(hr)) return hr;

    pNote->mtDuration = DMUS_PPQ;    // Quarter note duration
    pNote->bVelocity = 120;           // MIDI velocity (0 to 127)
    pNote->bFlags = DMUS_NOTEF_NOTEON; // Always set to this value.
    pNote->bTimeRange = 250;          // Randomize start time a lot.
    pNote->bDurRange = 5;              // Randomize duration a little.
    pNote->bVelRange = 0;              // Don't randomize velocity.
    pNote->bPlayModeFlags = bPlayModeFlags;
    pNote->bSubChordLevel = 0;         // Note uses subchord level 0.
    pNote->cTranspose = 0;             // No transposition

    // Stamp the message with the performance graph.
    IDirectMusicGraph* pGraph;
    hr = pPerformance->QueryInterface( IID_IDirectMusicGraph,
        (void**)&pGraph );
    if (FAILED(hr)) return hr;

    pGraph->StampPMsg( (DMUS_PMSG*)pNote );
    pGraph->Release();

    // Finally, send the message.
    hr = pPerformance->SendPMsg( (DMUS_PMSG*)pNote);
    if (FAILED(hr))
    {
        pPerformance->FreePMsg( (DMUS_PMSG*)pNote);
        return hr;
    }

    return S_OK;
}

```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicTool::ProcessPMsg, Messages, DirectMusic Messages, DirectMusic Tools

IDirectMusicPerformance::SetBumperLength

The **IDirectMusicPerformance::SetBumperLength** method sets the amount of time to buffer ahead of the port's latency for messages to be sent to the port for rendering. For an overview of this topic, see Timing.

```
HRESULT SetBumperLength(
    DWORD dwMilliseconds
);
```

Parameters

dwMilliseconds

Amount of preplay time, in milliseconds. The default value is 50.

Return Values

The method returns S_OK.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::GetBumperLength,
IDirectMusicPerformance::SetPrepareTime

IDirectMusicPerformance::SetGlobalParam

The **IDirectMusicPerformance::SetGlobalParam** method sets global values for the performance.

```
HRESULT SetGlobalParam(
    REFGUID rguidType,
    void* pParam,
```

DWORD *dwSize*
);

Parameters

rguidType

Reference to (C++) or address of (C) the identifier of the type of data.

pParam

Pointer to the data to be copied and stored by the performance.

dwSize

Size of the data. This is constant for each *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_FAIL

E_POINTER

E_OUTOFMEMORY

Remarks

The *dwSize* parameter is needed because the performance does not know about all types of data. New types can be created as needed.

For the parameters defined by DirectMusic and their associated data types, see Setting and Retrieving Global Parameters.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::GetGlobalParam,

IDirectMusicPerformance::SetParam, Music Parameters

IDirectMusicPerformance::SetGraph

The **IDirectMusicPerformance::SetGraph** method replaces the performance's tool graph.

```
HRESULT SetGraph(  
    IDirectMusicGraph* pGraph  
);
```

Parameters

pGraph

Tool graph to be set. Can be set to NULL to clear the graph from the performance.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

Any messages flowing through tools in the current tool graph are deleted.

The graph's reference count is incremented by this method, so it is safe to release the original reference.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::SetGraph, IDirectMusicPerformance::GetGraph,
IDirectMusicPerformance::SendPMsg

IDirectMusicPerformance::SetNotificationHandle

The **IDirectMusicPerformance::SetNotificationHandle** method sets the event handle (created by the Microsoft® Win32® **CreateEvent** function) for notifications. The application should use the Win32 **WaitForSingleObject** function on this handle. When signaled, the application should call the **IDirectMusicPerformance::GetNotificationPMsg** method to retrieve the notification event.

```
HRESULT SetNotificationHandle(  
    HANDLE hNotification,
```

```
REFERENCE_TIME rtMinimum
);
```

Parameters

hNotification

Event handle created by **CreateEvent**, or 0 to clear out an existing handle.

rtMinimum

Minimum time that the performance should hold onto old notify events before discarding them. The value 0 means to use the default minimum time of 20,000,000 reference time units, which is 2 seconds, or the previous value if this method has been called previously. If the application has not called **GetNotificationPMsg** by this time, the event is discarded to free the memory.

Return Values

The method returns S_OK.

Remarks

It is the application's responsibility to call the Win32 **CloseHandle** function on the notification handle when it is no longer needed.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

Notification and Event Handling

IDirectMusicPerformance::SetParam

The **IDirectMusicPerformance::SetParam** method sets data on a track inside the control segment.

```
HRESULT SetParam(
    REFGUID rguidType,
    DWORD dwGroupBits,
    DWORD dwIndex,
    MUSIC_TIME mtTime,
    void* pParam
);
```

Parameters

rguidType

Reference to (C++) or address of (C) the identifier of the type of data to set. See Track Parameter Types.

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.

mtTime

Time at which to set the data. Unlike **IDirectMusicSegment::SetParam**, this time is in performance time. The start time of the segment is subtracted from this time, and the result is passed to **IDirectMusicSegment::SetParam**.

pParam

Address of a structure containing the data. This structure must be of the appropriate kind and size for the data type identified by *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_SET_UNSUPPORTED
DMUS_E_TRACK_NOT_FOUND
E_POINTER

Remarks

Normally the primary segment is the control segment. However, a secondary segment can be designated as the control segment when it is played. See

DMUS_SEGF_FLAGS.

For an explanation of *dwGroupBits* and *dwIndex*, see Identifying the Track.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::GetParam,
IDirectMusicPerformance::SetGlobalParam, **IDirectMusicSegment::SetParam**,
IDirectMusicTrack::SetParam, **IDirectMusicPerformance::GetTime**, Music
Parameters

IDirectMusicPerformance::SetPrepareTime

The **IDirectMusicPerformance::SetPrepareTime** method sets the interval between the time that the **IDirectMusicTrack::Play** method is called and the time at which the messages should be heard through the speakers.

```
HRESULT SetPrepareTime(  
    DWORD dwMilliseconds  
);
```

Parameters

dwMilliseconds

Amount of prepare time, in milliseconds. The default value is 1000.

Return Values

The method returns S_OK.

Remarks

For an overview, see Timing.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::GetPrepareTime,
IDirectMusicPerformance::SetBumperLength

IDirectMusicPerformance::Stop

The **IDirectMusicPerformance::Stop** method stops playback of one or more segments.

```
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 0 is passed in this parameter, the requested segments and segment states stop playing immediately.

dwFlags

Flag that indicates when the stop should occur. Boundaries are in relation to the current primary segment. Must be one of the following values:

0

Stop immediately.

DMUS_SEGF_GRID

Stop on the next grid boundary at or after *mtTime*.

DMUS_SEGF_MEASURE

Stop on the next measure boundary at or after *mtTime*.

DMUS_SEGF_BEAT

Stop on the next beat boundary at or after *mtTime*.

DMUS_SEGF_DEFAULT

Stop on the default boundary, as set by the **IDirectMusicSegment::SetDefaultResolution** method.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

If *pSegment* and *pSegmentState* are both NULL, all music stops, and all currently cued segments are released. If either *pSegment* or *pSegmentState* is not NULL, only the requested segment states are removed from the performance. If both are non-NULL and DMUS_SEGF_DEFAULT is used, the default resolution from the *pSegment* is used.

If you set all parameters to NULL or 0, everything stops immediately, and controller reset messages and note-off messages are sent to all mapped PChannels.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::PlaySegment, DMUS_SEGF_FLAGS

IDirectMusicPerformance::TimeToRhythm

The **IDirectMusicPerformance::TimeToRhythm** method converts music time to rhythm time.

```
HRESULT TimeToRhythm(
    MUSIC_TIME mtTime,
    DMUS_TIMESIGNATURE *pTimeSig,
    WORD *pwMeasure,
    BYTE *pbBeat,
    BYTE *pbGrid,
    short *pnOffset
);
```

Parameters

mtTime

Time to convert.

pTimeSig

Address of a **DMUS_TIMESIGNATURE** structure that contains information about the time signature.

pwMeasure

Address of a variable to receive the measure in which the time falls.

pbBeat

Address of a variable to receive the beat at which the time falls.

pbGrid

Address of a variable to receive the grid at which the time falls.

pnOffset

Address of a variable to receive 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.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::RhythmToTime

IDirectMusicPort

The **IDirectMusicPort** interface provides access to a DirectMusicPort object, which represents a device that sends or receives music data. The input port of an MPU-401, the output port of an MPU-401, the Microsoft Software Synthesizer, and an IHV-provided filter are all ports. A physical device such as an MPU-401 might provide multiple ports. A single port, however, cannot both capture and render data.

For an overview, see Using Ports.

The methods of the **IDirectMusicPort** interface can be organized into the following groups:

Buffers	PlayBuffer
	Read
	SetReadNotificationHandle
Channels	GetChannelPriority
	GetNumChannelGroups
	SetChannelPriority
	SetNumChannelGroups
Device management	Activate
	DeviceIoControl
	SetDirectSound

Information	GetCaps
	GetFormat
	GetLatencyClock
	GetRunningStats
Downloadable sounds	Compact
	DownloadInstrument
	UnloadInstrument

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

The **LPDIRECTMUSICPORT** type is defined as a pointer to the **IDirectMusicPort** interface:

```
typedef IDirectMusicPort *LPDIRECTMUSICPORT;
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicPort::Activate

The **IDirectMusicPort::Activate** method activates or deactivates the port.

```
HRESULT Activate(  
    BOOL fActive  
);
```

Parameters

fActive

Switch to activate (TRUE) or deactivate (FALSE) the port.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return **DSERR_NODRIVER**, indicating that no sound driver is present.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusic::Activate, Using Ports

IDirectMusicPort::Compact

The **IDirectMusicPort::Compact** method is used to instruct the port to compact DLS or wave-table memory, thus making the largest possible contiguous chunk of memory available for new instruments to be downloaded.

HRESULT Compact();

Parameters

None.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_FAIL
E_INVALIDARG
E_NOTIMPL
E_OUTOFMEMORY

Remarks

This method only affects 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 utilize host system memory), the method will return E_NOTIMPL.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicPort::DeviceIoControl

The **IDirectMusicPort::DeviceIoControl** method calls the Win32 **DeviceIoControl** function on the underlying file handle implementing the port.

```
HRESULT DeviceIoControl(  
    DWORD dwIoControlCode,  
    LPVOID lpInBuffer,  
    DWORD nInBufferSize,  
    LPVOID lpOutBuffer,  
    DWORD nOutBufferSize,  
    LPDWORD lpBytesReturned,  
    LPOVERLAPPED lpOverlapped  
);
```

Parameters

dwIoControlCode

Control code of the operation to perform.

lpInBuffer

Pointer to the buffer that contains input data.

nInBufferSize

Size of input buffer.

lpOutBuffer

Pointer to the buffer to receive output data.

nOutBufferSize

Size of the output buffer.

lpBytesReturned

Address of a variable to receive the output byte count.

lpOverlapped

Address of an overlapped structure for asynchronous operation.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

E_POINTER

E_NOTIMPL

Remarks

This method is supported only on ports implemented by a Windows Driver Model (WDM) filter graph. In the case of a WDM filter graph, the file handle used is the topmost pin in the graph.

DirectMusic can refuse to perform defined kernel streaming operations on a pin that might collide with operations that it is performing on the filter graph. User-defined operations, however, are never blocked.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusicc.h`.

IDirectMusicPort::DownloadInstrument

The **IDirectMusicPort::DownloadInstrument** method is used to download an instrument to the DLS device. Downloading an instrument means handing the data that makes up the instrument to the DLS device. This includes articulation data and all waves needed by the instrument. To save wave space, only waves and articulation required for a range are downloaded. The method returns an

IDirectMusicDownloadedInstrument interface pointer, which is later used to unload the instrument.

```
HRESULT DownloadInstrument(
    IDirectMusicInstrument *pInstrument,
    IDirectMusicDownloadedInstrument **ppDownloadedInstrument,
    DMUS_NOTERANGE *pNoteRanges,
    DWORD dwNumNoteRanges;
);
```

Parameters

pInstrument

Instrument from which the method extracts the data to be downloaded.

ppDownloadedInstrument

Address of a variable to receive a pointer to the **IDirectMusicDownloadedInstrument** interface.

pNoteRanges

Address of an array of **DMUS_NOTERANGE** structures. Each entry in the array specifies a contiguous range of MIDI note messages to which the instrument must respond. An instrument region is downloaded only if at least one note in that region is specified in the **DMUS_NOTERANGE** structures.

dwNumNoteRanges

Number of **DMUS_NOTERANGE** structures in the array pointed to by *pNoteRanges*. If this value is set to 0, the *pNoteRanges* parameter is ignored, and all regions and wave data for the instrument are downloaded.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

E_POINTER
E_OUTOFMEMORY
E_NOTIMPL

Remarks

To prevent memory loss, the instrument must be unloaded by calling both **IDirectMusicPort::UnloadInstrument** and **IDirectMusicDownloadedInstrument::Release** when it is no longer needed.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in *dmusicc.h*.

See Also

IDirectMusicPort::Compact, Working with Instruments

IDirectMusicPort::GetCaps

The **IDirectMusicPort::GetCaps** method retrieves the port's capabilities.

```
HRESULT GetCaps(  
    LPDMUS_PORTCAPS pPortCaps  
);
```

Parameters

pPortCaps

Address of a **DMUS_PORTCAPS** structure to receive the capabilities of the port. The **dwSize** member of this structure must be properly initialized before the method is called.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
E_INVALIDARG

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicPort::GetChannelPriority

The **IDirectMusicPort::GetChannelPriority** method is used to retrieve the priority of a MIDI channel. For an overview, see Channels.

```
HRESULT GetChannelPriority(  
    DWORD dwChannelGroup,  
    DWORD dwChannel,  
    LPDWORD pdwPriority  
);
```

Parameters

dwChannelGroup
Group that the channel is in.

dwChannel
Index of the channel on the group.

pdwPriority
Address of a variable to receive the priority ranking. See Remarks.

Return Values

If the method succeeds, the return value is S_OK.

Remarks

The following values, defined in Dmusicc.h, each represent a range of priorities. They are listed here in descending order of priority.

DAUD_CRITICAL_VOICE_PRIORITY
DAUD_HIGH_VOICE_PRIORITY

DAUD_STANDARD_VOICE_PRIORITY
DAUD_LOW_VOICE_PRIORITY

The following values express the default ranking of the channels within a range, according to the DLS Level 1 standard. They are listed here in descending order. Channel 10, the percussion channel, has the highest priority.

DAUD_CHAN10_DEF_VOICE_PRIORITY_OFFSET
DAUD_CHAN1_DEF_VOICE_PRIORITY_OFFSET
DAUD_CHAN2_DEF_VOICE_PRIORITY_OFFSET
DAUD_CHAN3_DEF_VOICE_PRIORITY_OFFSET
DAUD_CHAN4_DEF_VOICE_PRIORITY_OFFSET
DAUD_CHAN5_DEF_VOICE_PRIORITY_OFFSET
DAUD_CHAN6_DEF_VOICE_PRIORITY_OFFSET
DAUD_CHAN7_DEF_VOICE_PRIORITY_OFFSET
DAUD_CHAN8_DEF_VOICE_PRIORITY_OFFSET
DAUD_CHAN9_DEF_VOICE_PRIORITY_OFFSET
DAUD_CHAN11_DEF_VOICE_PRIORITY_OFFSET
DAUD_CHAN12_DEF_VOICE_PRIORITY_OFFSET
DAUD_CHAN13_DEF_VOICE_PRIORITY_OFFSET
DAUD_CHAN14_DEF_VOICE_PRIORITY_OFFSET
DAUD_CHAN15_DEF_VOICE_PRIORITY_OFFSET
DAUD_CHAN16_DEF_VOICE_PRIORITY_OFFSET

The priority of a channel is represented by a range plus an offset. For example, DAUD_STANDARD_VOICE_PRIORITY plus DAUD_CHAN10_DEF_VOICE_PRIORITY represents the highest priority within the standard range.

Channels that have the same priority value have equal priority, regardless of which channel group they belong to.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicPort::SetChannelPriority

IDirectMusicPort::GetFormat

The **IDirectMusicPort::GetFormat** method retrieves information about the wave format specified in the **DMUS_PORTPARAMS** structure passed to **IDirectMusic::CreatePort**, and the recommended size of the buffer to use for wave output. The information can be used to create a compatible **IDirectSoundBuffer** for the port.

```
HRESULT GetFormat(  
    LPWAVEFORMATEX pWaveFormatEx,  
    LPDWORD pdwWaveFormatExSize  
    LPDWORD pdwBufferSize  
);
```

Parameters

pWaveFormatEx

Address of the **WAVEFORMATEX** structure to receive information about the format. This value can be NULL. See Remarks.

pdwWaveFormatExSize

Address of a variable that contains, or is to receive, the size of the structure. See Remarks.

pdwBufferSize

Address of a variable to receive the recommended size of the DirectSound buffer.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns **S_OK**.

If it fails, the method can return **E_POINTER**.

Remarks

The **WAVEFORMATEX** structure can have a variable length that depends on the details of the format. Before retrieving the format description, the application should query the synthesizer object for the size of the format by calling this method and specifying NULL for the *pWaveFormatEx* parameter. The size of the structure is returned in the variable pointed to by *pdwWaveFormatExSize*. The application can then allocate sufficient memory and call **GetFormat** again to retrieve the format description.

If *pWaveFormatEx* is not NULL, DirectMusic writes, at most, *pdwWaveFormatExSize* bytes to the structure.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicPort::SetDirectSound

IDirectMusicPort::GetLatencyClock

The **IDirectMusicPort::GetLatencyClock** method is used to get 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 **IDirectMusic::SetMasterClock** method.

```
HRESULT GetLatencyClock(  
    IReferenceClock** ppClock  
);
```

Parameters

ppClock

Address of a variable to receive the latency clock's **IReferenceClock** interface pointer.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

In accordance with COM rules, **GetLatencyClock** increments the reference count of the returned interface. Therefore, the application must call **Release** on the **IReferenceClock** interface at some point.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

Latency and Bumper Time

IDirectMusicPort::GetNumChannelGroup S

The **IDirectMusicPort::GetNumChannelGroups** method retrieves the number of channel groups on the port.

```
HRESULT GetNumChannelGroups  
    LPDWORD pdwChannelGroups  
);
```

Parameters

pdwChannelGroups

Address of a variable to receive the number of channel groups.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

- E_FAIL**
- E_INVALIDARG**
- E_NOTIMPL**
- E_OUTOFMEMORY**

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in *dmusicc.h*.

See Also

IDirectMusicPort::SetNumChannelGroups, Channels

IDirectMusicPort::GetRunningStats

The **IDirectMusicPort::GetRunningStats** method retrieves information about the state of the port's synthesizer.

```
HRESULT GetRunningStats(  
    LPDMUS_SYNTHSTATS pStats  
);
```

Parameters

pStats

Address of a **DMUS_SYNTHSTATS** structure to receive running statistics of the synthesizer. The **dwSize** member of this structure must be properly initialized before the method is called.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
E_INVALIDARG
E_NOTIMPL

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicPort::PlayBuffer

The **IDirectMusicPort::PlayBuffer** method is used to cue a buffer for playback by the port.

```
HRESULT PlayBuffer(  
    IDirectMusicBuffer* pBuffer  
);
```

Parameters

pBuffer

Address of a DirectMusicBuffer object to be added to the port's playback queue.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

- E_FAIL
- E_INVALIDARG
- E_NOTIMPL
- E_OUTOFMEMORY

Remarks

The buffer is in use by the system only for the duration of this method and can be reused after the method returns.

If no start time has been set by using the **IDirectMusicBuffer::SetStartTime** method, the start time is the time of the earliest event in the buffer, as set by the **IDirectMusicBuffer::PackStructured** or the **IDirectMusicBuffer::PackUnstructured** method.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicBuffer, **IDirectMusic::CreateMusicBuffer**

IDirectMusicPort::Read

The **IDirectMusicPort::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.

```
HRESULT Read(  
    IDirectMusicBuffer *pBuffer  
);
```

Parameters

pBuffer

Address of a **DirectMusicBuffer** object to be filled with the incoming MIDI data.

Return Values

If the method succeeds, the return value is S_OK or S_FALSE.

If it fails, the method can return one of the following error values:

E_POINTER
E_NOTIMPL

Remarks

When there is no more data to read, the method returns S_FALSE.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

Capturing Music

IDirectMusicPort::SetChannelPriority

The **IDirectMusicPort::SetChannelPriority** method is used to set the priority of a MIDI channel. For an overview, see Channels.

```
HRESULT SetChannelPriority(  
    DWORD dwChannelGroup,  
    DWORD dwChannel,  
    DWORD dwPriority  
);
```

Parameters

dwChannelGroup

Group that the channel is in. This value must be 1 or greater.

dwChannel

Index of the channel on the group.

dwPriority

The priority ranking. See Remarks for **IDirectMusicPort::GetChannelPriority**.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_FAIL

E_INVALIDARG
E_OUTOFMEMORY
E_NOTIMPL

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicPort::GetChannelPriority,
DMUS_CHANNEL_PRIORITY_PMSG.

IDirectMusicPort::SetDirectSound

The **IDirectMusicPort::SetDirectSound** method is used to override the default DirectSound object or buffer, or both, to which a port's wave data is streamed. It is also used to disconnect the port from DirectSound.

```
HRESULT SetDirectSound(  
    LPDIRECTSOUND pDirectSound,  
    LPDIRECTSOUNDBUFFER pDirectSoundBuffer  
);
```

Parameters

pDirectSound

Address of the **IDirectSound** interface of the DirectSound object to which the port is to be connected, or NULL to disconnect and release the existing DirectSound object.

pDirectSoundBuffer

Address of the **IDirectSoundBuffer** interface to connect the port to. This value can be NULL.

Return Values

If the method succeeds, the return value is S_OK or DMUS_S_NOBUFFERCONTROL. See Remarks.

If it fails, the method can return one of the following error values:

DMUS_E_ALREADY_ACTIVATED
E_INVALIDARG

Remarks

If a valid pointer is passed in *pDirectSoundBuffer*, the method returns **DMUS_S_NOBUFFERCONTROL** if control changes in the buffer such as pan and volume do not affect DirectMusic playback. This affects only 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_PORTPARAMS** structure passed to **IDirectMusic::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 once the port has been activated.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in *dmusicc.h*.

See Also

IDirectMusicPort::Activate, **IDirectMusicPort::GetFormat**, Integrating DirectMusic and DirectSound

IDirectMusicPort::SetNumChannelGroups

The **IDirectMusicPort::SetNumChannelGroups** method changes the number of channel groups that the application needs on the port.

```
HRESULT SetNumChannelGroups(  
    DWORD dwChannelGroups  
);
```

Parameters

dwChannelGroups

Number of channel groups on this port that the application wants to allocate.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

- E_FAIL
- E_INVALIDARG
- E_NOTIMPL
- E_OUTOFMEMORY

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicPort::GetNumChannelGroups, Channels

IDirectMusicPort::SetReadNotificationHandle

The **IDirectMusicPort::SetReadNotificationHandle** method specifies an event that is to be set when MIDI messages are available to be read with the

IDirectMusicPort::Read method. The event is signaled whenever new data is available. To turn off event notification, call **SetReadNotificationHandle** with a NULL value for the *hEvent* parameter.

```
HRESULT SetReadNotificationHandle(  
    HANDLE hEvent  
);
```

Parameters

hEvent

Event handle obtained from a call to the the Win32 **CreateEvent** function. It is the application's responsibility to close this handle once the port has been released.

Return Values

The method returns S_OK.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusic.h.

See Also

Capturing Music

IDirectMusicPort::UnloadInstrument

The **IDirectMusicPort::UnloadInstrument** method is used to unload a previously downloaded DLS instrument.

```
HRESULT UnloadInstrument(  
    IDirectMusicDownloadedInstrument *pDownloadedInstrument  
);
```

Parameters

pDownloadedInstrument

Address of an **IDirectMusicDownloadedInstrument** interface, obtained when the instrument was downloaded by calling the **IDirectMusicPort::DownloadInstrument** method.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_DOWNLOADED_TO_PORT
E_POINTER
E_NOTIMPL

Remarks

This method must be called to free memory allocated by **IDirectMusicPort::DownloadInstrument**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Header: Declared in dmusicc.h.

See Also

Working with Instruments

IDirectMusicPortDownload

The **IDirectMusicPortDownload** interface allows an application to communicate directly with a port that supports DLS downloading and to download memory chunks directly to the port. The interface is used primarily by authoring applications that edit DLS instruments directly. For an overview, see Low-Level DLS.

To obtain the **IDirectMusicPortDownload** interface, call the **IDirectMusicPort::QueryInterface** method, passing in IID_IDirectMusicPortDownload as the interface GUID. If the port does not support DLS downloading, this call might fail.

The methods of the **IDirectMusicPortDownload** interface can be grouped as follows:

Buffer management	AllocateBuffer
	GetAppend
	GetBuffer
	GetDLId
Loading	Download
	Unload

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicPortDownload::AllocateBuffer

The **IDirectMusicPortDownload::AllocateBuffer** method allocates a chunk of memory for downloading DLS data to the port and returns an **IDirectMusicDownload** interface pointer that allows access to this buffer.

```
HRESULT AllocateBuffer(  
    DWORD dwSize,  
    IDirectMusicDownload** ppIDMDownload  
);
```

Parameters

dwSize

Requested size of buffer.

ppIDMDownload

Address of a variable to receive the **IDirectMusicDownload** interface pointer.

Return Values

If the method succeeds, it returns S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
E_INVALIDARG
E_OUTOFMEMORY

Remarks

Once a buffer has been allocated, its size cannot change.

The buffer is freed when the **IDirectMusicDownload** interface is released.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicPortDownload::GetBuffer, Low-Level DLS

IDirectMusicPortDownload::Download

The **IDirectMusicPortDownload::Download** method downloads a wave or instrument definition to the port. The memory must first be allocated by using the **IDirectMusicPortDownload::AllocateBuffer** method.

```
HRESULT Download(  
    IDirectMusicDownload* pIDMDownload  
);
```

Parameters

pIDMDownload

Address of the **IDirectMusicDownload** interface for the buffer.

Return Values

Return values are determined by the implementation of the port.

If the method succeeds, it returns S_OK.

If the method fails, it can return one of the following values:

- E_POINTER
- E_FAIL
- DMUS_E_ALREADY_DOWNLOADED
- DMUS_E_BADARTICULATION
- DMUS_E_BADINSTRUMENT
- DMUS_E_BADOFFSETTABLE
- DMUS_E_BADWAVE
- DMUS_E_BADWAVELINK
- DMUS_E_BUFFERNOTSET
- DMUS_E_NOARTICULATION
- DMUS_E_NOTMONO
- DMUS_E_NOTPCM
- DMUS_E_UNKNOWNDOWNLOAD

Remarks

For more information on how to prepare the data to be downloaded, see Low-Level DLS.

Once 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 **IDirectMusicPortDownload::GetDLId** method, then send it down.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusicc.h`.

See Also

IDirectMusicPortDownload::Unload, **DMUS_DOWNLOADINFO**, **DMUS_OFFSETTABLE**

IDirectMusicPortDownload::GetAppend

The **IDirectMusicPortDownload::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.

```
HRESULT GetAppend(  
    DWORD* pdwAppend  
);
```

Parameters

pdwAppend

Address of a variable to receive the number of appended samples for which memory is required. The amount of memory can be calculated from the wave format.

Return Values

Return values are determined by the port implementation.

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

E_POINTER
E_NOTIMPL

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicPortDownload::GetBuffer

The **IDirectMusicPortDownload::GetBuffer** method retrieves the **IDirectMusicDownload** interface pointer of a buffer whose unique identifier is known.

```
HRESULT GetBuffer(  
    DWORD dwDLId,  
    IDirectMusicDownload** ppIDMDownload  
);
```

Parameters

dwDLId

Download identifier of the buffer. See **DMUS_DOWNLOADINFO**.

ppIDMDownload

Address of a variable to receive the **IDirectMusicDownload** interface pointer for the buffer.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

E_POINTER
DMUS_E_INVALID_DOWNLOADID
DMUS_E_NOT_DOWNLOADED_TO_PORT

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicPortDownload::GetDLId, **IDirectMusicDownload::GetBuffer**, Low-Level DLS.

IDirectMusicPortDownload::GetDLId

The **IDirectMusicPortDownload::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.

```
HRESULT GetDLId(  
    DWORD* pdwStartDLId,  
    DWORD dwCount  
);
```

Parameters

pdwStartDLId

Address of a variable to receive the first identifier.

dwCount

Number of identifiers to reserve. You might plan to download a whole series of chunks at once. Instead of calling **GetDLId** for each chunk, set *dwCount* to the number of chunks. **GetDLId** returns the first ID of the set, and the additional identifiers are automatically reserved up through **pdwStartDLId* plus *dwCount*. A subsequent call to **GetDLId** would skip past the reserved values.

Return Values

If the method succeeds, it returns S_OK.

If it fails, the method can return one of the following error values:

E_POINTER
E_INVALIDARG

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusicPortDownload::GetBuffer, Low-Level DLS

IDirectMusicPortDownload::Unload

The **IDirectMusicPortDownload::Unload** method unloads a buffer that was previously downloaded by using **IDirectMusicPortDownload::Download**.

```
HRESULT Unload(
    IDirectMusicDownload* pIDMDownload
);
```

Parameters

pIDMDownload

Address of the **IDirectMusicDownload** interface for the buffer.

Return Values

Return values are determined by the port implementation.

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following values:

```
E_NOINTERFACE
DMUS_E_SYNTHNOTCONFIGURED
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IDirectMusicSegment

The **IDirectMusicSegment** interface represents a segment, a chunk of music made up of multiple tracks. Because almost all the information that defines a segment is stored in the tracks, and because tracks can be nearly anything, the segment itself is a relatively simple object. There are methods to access data in tracks, even though the segment object has no knowledge of the nature of the data.

The **IDirectMusicSegment** object also supports the **IDirectMusicObject** and **IPersistStream** interfaces for loading its data.

The methods of the **IDirectMusicSegment** interface can be grouped as follows:

Timing and looping

GetDefaultResolution

GetLength

GetLoopPoints

	GetRepeats
	GetStartPoint
	SetDefaultResolution
	SetLength
	SetLoopPoints
	SetRepeats
	SetStartPoint
Tracks	GetTrack
	GetTrackGroup
	InsertTrack
	RemoveTrack
Notification	AddNotificationType
	RemoveNotificationType
Parameters	GetParam
	SetParam
Tools	GetGraph
	SetGraph
Miscellaneous	Clone
	InitPlay
	SetPChannelsUsed

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

Segments

IDirectMusicSegment::AddNotificationType

The **IDirectMusicSegment::AddNotificationType** method is similar to and called from the **IDirectMusicPerformance::AddNotificationType** method, allowing the segment to respond to notifications. The segment calls each track's **IDirectMusicTrack::AddNotificationType** method.

```
HRESULT AddNotificationType(  
    REFGUID rguidNotificationType  
);
```

Parameters

rguidNotificationType

Reference to (C++) or address of (C) the identifier of the notification type to add. For the defined types, see **DMUS_NOTIFICATION_PMSG**. Applications can also define their own types for custom tracks.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

E_POINTER
E_OUTOFMEMORY

Remarks

Segments cannot generate notifications of type **GUID_NOTIFICATION_PERFORMANCE**. To get notifications of this type, you must call **IDirectMusicPerformance::AddNotificationType**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in **dmusici.h**.

See Also

Notification and Event Handling

IDirectMusicSegment::Clone

The **IDirectMusicSegment::Clone** method copies all or part of the segment and the tracks that it contains.

```
HRESULT Clone(
    MUSIC_TIME mtStart,
    MUSIC_TIME mtEnd,
    IDirectMusicSegment** ppSegment
);
```

Parameters

mtStart

Start of the part to copy. If less than 0 or greater than the length of the segment, 0 is used.

mtEnd

End of the part to copy. If this value is past the end of the segment, the segment is copied to the end. A value of 0 or anything less than *mtStart* also copies to the end.

ppSegment

Address of a variable to receive a pointer to the created segment, if successful. It is the caller's responsibility to call **Release** when finished with the segment.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if some tracks failed to copy.

If it fails, the method can return one of the following error values:

E_OUTOFMEMORY
E_POINTER

Remarks

If an **IDirectMusicGraph** interface exists in the segment, a copy of the pointer is included in the clone, and the reference count is incremented. The start point and loop points set by the **IDirectMusicSegment::SetStartPoint** and

IDirectMusicSegment::SetLoopPoints methods are set to their default values (0, and 0 to the end of the segment, respectively) inside the clone. The number of repeats is also reset to 0. The resolution set by the

IDirectMusicSegment::SetDefaultResolution method is copied into the clone.

For style-based segments, if *mtStart* is greater than 0, it should be on a measure boundary.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicSegment::GetDefaultResolution

The **IDirectMusicSegment::GetDefaultResolution** method retrieves the default resolution for synchronization.

```
HRESULT GetDefaultResolution(
    DWORD* pdwResolution
);
```

Parameters

pdwResolution

Address of a variable to receive the default resolution. See **DMUS_SEGF_FLAGS**.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::SetDefaultResolution, Segment Timing

IDirectMusicSegment::GetGraph

The **IDirectMusicSegment::GetGraph** method retrieves the segment's tool graph.

```
HRESULT GetGraph(
    IDirectMusicGraph** ppGraph
);
```

Parameters

ppGraph

Address of a variable to receive a pointer to the tool graph.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_FAIL

E_POINTER

Remarks

If there is no graph in the segment, the method returns E_FAIL.

The reference count of the tool graph is incremented.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::SetGraph

IDirectMusicSegment::GetLength

The **IDirectMusicSegment::GetLength** method retrieves the length of the segment.

```
HRESULT GetLength(  
    MUSIC_TIME* pmtLength  
);
```

Parameters

pmtLength

Address of a variable to receive the segment's length in music time.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

If for some reason the segment's length was never set, **pmtLength* is set to 0.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::SetLength

IDirectMusicSegment::GetLoopPoints

The **IDirectMusicSegment::GetLoopPoints** method retrieves the start and end loop points inside the segment that repeat the number of times set by the **IDirectMusicSegment::SetRepeats** method.

```
HRESULT GetLoopPoints(  
    MUSIC_TIME* pmtStart,  
    MUSIC_TIME* pmtEnd  
);
```

Parameters

pmtStart

Address of a variable to receive the start point of the loop.

pmtEnd

Address of a variable to receive the end point of the loop. A value of 0 indicates that the entire segment loops.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::SetLoopPoints, Segment Timing

IDirectMusicSegment::GetParam

The **IDirectMusicSegment::GetParam** method retrieves data from a track inside this segment.

```
HRESULT GetParam(
    REFGUID rguidType,
    DWORD dwGroupBits,
    DWORD dwIndex,
    MUSIC_TIME mtTime,
    MUSIC_TIME* pmtNext,
    void* pParam
);
```

Parameters

rguidType

Reference to (C++) or address of (C) the identifier of the type of data to obtain. See Track Parameter Types.

dwGroupBits

Group that the desired track is in. Use 0xFFFFFFFF for all groups. For more information, see Identifying the Track.

dwIndex

Index of the track in the group identified by *dwGroupBits* from which to obtain the data.

mtTime

Time from which to obtain the data.

pmtNext

Address of a variable to receive the segment time (relative to *mtTime*) until which the data is valid. If this returns a value of 0, it means either that the data is always valid, or that it is unknown when it might become invalid. If this information is not needed, *pmtNext* can be set to NULL. See Remarks.

pParam

Address of an allocated structure in which the data is to be returned. The structure must be of the appropriate kind and size for the data type identified by *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_GET_UNSUPPORTED
DMUS_E_TRACK_NOT_FOUND
E_POINTER

Remarks

The data can become invalid before the time returned in **pmtNext* if another control segment is cued. For more information on control segments, see Segments.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::GetParam, **IDirectMusicSegment::SetParam**, **IDirectMusicTrack::GetParam**, Music Parameters

IDirectMusicSegment::GetRepeats

The **IDirectMusicSegment::GetRepeats** method retrieves the number of times the looping portion of the segment is set to repeat.

```
HRESULT GetRepeats(
    DWORD* pdwRepeats
);
```

Parameters

pdwRepeats

Address of a variable to receive the number of times that the looping portion of the segment is set to repeat.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::SetRepeats

IDirectMusicSegment::GetStartPoint

The **IDirectMusicSegment::GetStartPoint** method retrieves the point at which the segment starts playing in response to the **IDirectMusicPerformance::PlaySegment** method.

```
HRESULT GetStartPoint(  
    MUSIC_TIME* pmtStart  
);
```

Parameters

pmtStart

Address of a variable to receive the time within the segment at which it starts playing.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::SetStartPoint

IDirectMusicSegment::GetTrack

The **IDirectMusicSegment::GetTrack** method searches the list of tracks for the one with the supplied type, group, and index, and retrieves a pointer to the **DirectMusicTrack** object.

```
HRESULT GetTrack(
    REFGUID rguidType,
    DWORD dwGroupBits,
    DWORD dwIndex,
    IDirectMusicTrack** ppTrack
);
```

Parameters

rguidType

Reference to (C++) or address of (C) the identifier of the track to find (for example, **CLSID_DirectMusicChordTrack**). A value of **GUID_NULL** retrieves any track. For the track identifiers, see **IDirectMusicTrack**.

dwGroupBits

Track groups in which to scan for the track. A value of 0 is invalid. Each bit in *dwGroupBits* corresponds to a track group. To scan all tracks, regardless of groups, set this parameter to 0xFFFFFFFF.

dwIndex

Zero-based index into the list of tracks of type *rguidType* and in group *dwGroupBits* to return. If multiple groups are selected in *dwGroupBits*, this index indicates the *n*th track of type *rguidType* encountered in the union of the groups selected.

ppTrack

Address of a variable to receive 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 **E_FAIL**.

Remarks

To enumerate all tracks, use **GUID_NULL** for the *rguidType* and 0xFFFFFFFF for *dwGroupBits*. Call **GetTrack** starting with 0 for *dwIndex*, incrementing *dwIndex* until the method no longer returns a success code.

For more information on track groups, see Identifying the Track.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::InsertTrack

IDirectMusicSegment::GetTrackGroup

The **IDirectMusicSegment::GetTrackGroup** method retrieves the group bits set on a track inside the segment.

```
HRESULT GetTrackGroup(  
    IDirectMusicTrack* pTrack,  
    DWORD* pdwGroupBits  
);
```

Parameters

pTrack

Track for which to find the group bits.

pdwGroupBits

Address of a variable to receive the groups. Each bit in **pdwGroupBits* corresponds to a track group.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_FAIL

E_INVALIDARG

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::InsertTrack, Identifying the Track

IDirectMusicSegment::InitPlay

The **IDirectMusicSegment::InitPlay** method is called by the performance when the segment is about to be played.

```
HRESULT InitPlay(
    IDirectMusicSegmentState** ppSegState,
    IDirectMusicPerformance* pPerformance,
    DWORD dwFlags
);
```

Parameters

ppSegState

Address of a variable to receive a pointer to the **IDirectMusicSegmentState** interface, which 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 **IDirectMusicPerformance** 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. See Remarks.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

E_POINTER
E_OUTOFMEMORY

Remarks

This method is for internal use by DirectMusic and should not be called by applications. It will not be supported in future versions.

The **InitPlay** method is called by the performance engine when the segment is about to be played. The segment, in turn, collects state objects for each of the tracks by calling their **IDirectMusicTrack::InitPlay** methods and stores the result in a segment state object, accessed by using the **IDirectMusicSegmentState** interface.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicSegment::InsertTrack

The **IDirectMusicSegment::InsertTrack** method inserts the supplied track into the segment's list of tracks.

```
HRESULT InsertTrack(  
    IDirectMusicTrack* pTrack,  
    DWORD dwGroupBits  
);
```

Parameters

pTrack

Track to add to the segment.

dwGroupBits

Group or groups into which to insert the track. This value cannot be 0.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

- E_FAIL
- E_INVALIDARG
- E_OUTOFMEMORY
- E_POINTER

Remarks

Tracks are put in groups to link them correctly. For example, a segment might contain two style tracks and two mute tracks. Each style track would be put in a different group, along with its associated mute track. For more information on track groups, see Identifying the Track.

If the segment is currently playing, the new track is not included in playback because the segment state was not initialized with the new track.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::RemoveTrack, **IDirectMusicSegment::GetTrackGroup**

IDirectMusicSegment::RemoveNotificationType

The **IDirectMusicSegment::RemoveNotificationType** method is similar to and is called from the **IDirectMusicPerformance::RemoveNotificationType** method, allowing the segment to remove notifications. The segment calls each track's **IDirectMusicTrack::RemoveNotificationType** method.

```
HRESULT RemoveNotificationType(
    REFGUID rguidNotificationType
);
```

Parameters

rguidNotificationType

Reference to (C++) or address of (C) the identifier of the notification type to remove. (For the defined types, see **DMUS_NOTIFICATION_PMSG**.) Setting this value to **GUID_NULL** causes all notifications to be removed.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return **E_POINTER**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

Notification and Event Handling

IDirectMusicSegment::RemoveTrack

The **IDirectMusicSegment::RemoveTrack** method removes a track from the segment's track list.

```
HRESULT RemoveTrack(  
    IDirectMusicTrack* pTrack  
);
```

Parameters

pTrack

Track to remove from the segment's track list.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if the specified track is not in the track list.

If the method fails, the return value can be E_POINTER.

Remarks

The track is released when removed.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::InsertTrack

IDirectMusicSegment::SetDefaultResolution

The **IDirectMusicSegment::SetDefaultResolution** method sets the default resolution for synchronization.

```
HRESULT SetDefaultResolution(  
    DWORD dwResolution  
);
```

Parameters

dwResolution

Desired default resolution. This value can be 0 or one of the following members of the **DMUS_SEGF_FLAGS** enumeration:

DMUS_SEGF_MEASURE
DMUS_SEGF_BEAT
DMUS_SEGF_GRID

Return Values

The method returns S_OK.

Remarks

This method is used primarily by secondary segments (motifs) to define whether they are synchronized to the measure, beat, or grid resolutions.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::GetDefaultResolution, Segment Timing

IDirectMusicSegment::SetGraph

The **IDirectMusicSegment::SetGraph** method assigns a tool graph to the segment.

```
HRESULT SetGraph(  
    IDirectMusicGraph* pGraph  
);
```

Parameters

pGraph

Tool graph to be set. Can be set to NULL to clear the segment graph.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

Any messages flowing through tools in the current tool graph are deleted.

The graph's reference count is incremented, so it is safe to release the original reference.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusici.h`.

See Also

IDirectMusicPerformance::SetGraph, DirectMusic Tools

IDirectMusicSegment::SetLength

The **IDirectMusicSegment::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.

```
HRESULT SetLength(
    MUSIC_TIME mtLength
);
```

Parameters

mtLength

Desired length. Must be greater than 0.

Return Values

If the method succeeds, the return value is `S_OK`.

If it fails, the method can return one of the following error values:

```
E_INVALIDARG
DMUS_E_OUT_OF_RANGE
```

Remarks

Neglecting to set a primary segment length can cause problems when cuing other primary segments with the `DMUS_SEGF_QUEUE` flag.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::GetLength

IDirectMusicSegment::SetLoopPoints

The **IDirectMusicSegment::SetLoopPoints** method sets the start and end points of the part of the segment that repeats. It repeats the number of times set by the **IDirectMusicSegment::SetRepeats** method.

```
HRESULT SetLoopPoints(  
    MUSIC_TIME mtStart,  
    MUSIC_TIME mtEnd  
);
```

Parameters

mtStart

Point at which to begin the loop.

mtEnd

Point at which to end the loop. A value of 0 loops the entire segment.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DMUS_E_OUT_OF_RANGE.

Remarks

When the segment is played, it plays from the segment start time until *mtEnd*, then loops to *mtStart*, plays the looped portion the number of times set by **IDirectMusicSegment::SetRepeats**, 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 **IDirectMusicLoader::ReleaseObject** on it before releasing it.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::GetLoopPoints, Segment Timing

IDirectMusicSegment::SetParam

The **IDirectMusicSegment::SetParam** method sets data on a track inside this segment.

```
HRESULT SetParam(
    REFGUID rguidType,
    DWORD dwGroupBits,
    DWORD dwIndex,
    MUSIC_TIME mtTime,
    void* pParam
);
```

Parameters

rguidType

Reference to (C++) or address of (C) the type of data to set. See Track Parameter Types.

dwGroupBits

Group that the desired track is in. Use 0xFFFFFFFF for all groups. For more information, see the Remarks for **IDirectMusicPerformance::SetParam**.

dwIndex

Index of the track in the group identified by *dwGroupBits* in which to set the data.

mtTime

Time at which to set the data.

pParam

Address of a structure containing the data, or NULL if no data is required. The structure must be of the appropriate kind and size for the data type identified by *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_SET_UNSUPPORTED
DMUS_E_TRACK_NOT_FOUND
E_POINTER

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::SetParam, **IDirectMusicSegment::GetParam**,
IDirectMusicTrack::SetParam, Music Parameters

IDirectMusicSegment::SetPChannelsUsed

The **IDirectMusicSegment::SetPChannelsUsed** method sets the performance channels (PChannels) that this segment uses. This method is usually called by a track in the **IDirectMusicTrack::Init** method to inform the segment of which PChannels the track uses.

```
HRESULT SetPChannelsUsed(  
    DWORD dwNumPChannels,  
    DWORD* paPChannels  
);
```

Parameters

dwNumPChannels

Number of PChannels to set. This must be equal to the number of members in the array pointed to by *paPChannels*.

paPChannels

Address of an array of PChannels.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

E_INVALIDARG
E_OUTOFMEMORY
E_POINTER

Remarks

This method allows the performance to know which ports are being used by the segment so that it can determine the actual latency, rather than providing for the worst case.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

Latency and Bumper Time, Channels

IDirectMusicSegment::SetRepeats

The **IDirectMusicSegment::SetRepeats** method sets the number of times the looping portion of the segment is to repeat. By default, the entire segment is looped.

```
HRESULT SetRepeats(  
    DWORD dwRepeats  
);
```

Parameters

dwRepeats

Number of times that the looping portion of the segment is to repeat. A value of 0 indicates a single play with no repeats.

Return Values

The method returns S_OK.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::GetRepeats, **IDirectMusicSegment::SetLoopPoints**,
Segment Timing

IDirectMusicSegment::SetStartPoint

The **IDirectMusicSegment::SetStartPoint** method sets the point at which the segment starts playing in response to a call to the **IDirectMusicPerformance::PlaySegment** method.

```
HRESULT SetStartPoint(
    MUSIC_TIME mtStart
);
```

Parameters

mtStart

Point within the segment at which it is to start playing. If this value is less than 0 or greater than the length of the segment, the start point is set to 0.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return DMUS_E_OUT_OF_RANGE.

Remarks

By default, the start point is 0, meaning that the segment starts from the beginning.

The method fails if *mtStart* is greater than or equal to the length of the segment. If the segment does not already have a length, **IDirectMusicSegment::SetLength** must be called before this method.

The method does not affect any currently playing segment states created from this segment.

The start point of a cached segment persists even if the segment is released, and then reloaded. To ensure that a segment is not subsequently reloaded from the cache, call **IDirectMusicLoader::ReleaseObject** on it before releasing it.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::GetStartPoint,
IDirectMusicSegmentState::GetStartPoint, **IDirectMusicSegment::SetLength**,
IDirectMusicSegment::SetLoopPoints, Segment Timing

IDirectMusicSegmentState

When the **IDirectMusicPerformance::PlaySegment** method is called, the performance engine generates a segment-state object that tracks the state of the playing segment. It also provides the application with a handle to the segment, in the form of the **IDirectMusicSegmentState** interface, which can be used to track the playback status of the segment. This method can also be used directly to stop playback or remove the segment from the performance, using methods of **IDirectMusicPerformance**.

The interface has the following methods:

Information	GetRepeats
	GetSeek
	GetSegment
	GetStartPoint
	GetStartTime

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicSegmentState::GetRepeats

The **IDirectMusicSegmentState::GetRepeats** method returns the number of times that the looping portion of the segment is set to repeat.

HRESULT GetRepeats(

```
DWORD* pdwRepeats
);
```

Parameters

pdwRepeats

Address of a variable to receive the repeat count. A value of 0 indicates that the segment is to play through only once, with no portion repeated.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::SetRepeats

IDirectMusicSegmentState::GetSeek

The **IDirectMusicSegmentState::GetSeek** method retrieves the current seek pointer in the segment state. This is the value that is passed in the *mtStart* parameter of **IDirectMusicTrack::Play** the next time that method is called. It does not take into account looping and repeating; if the entire segment state repeats to the beginning, the seek pointer is reset to 0.

```
HRESULT GetSeek(
    MUSIC_TIME* pmtSeek
);
```

Parameters

pmtSeek

Address of a variable to receive the current seek pointer.

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value can be E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicSegmentState::GetSegment

The **IDirectMusicSegmentState::GetSegment** method returns a pointer to the segment that owns this segment state.

```
HRESULT GetSegment(  
    IDirectMusicSegment** ppSegment  
);
```

Parameters

ppSegment

Address of a variable to receive a pointer to the **IDirectMusicSegment** interface.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

The pointer returned in *ppSegment* must be released by the application.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicSegmentState::GetStartPoint

The **IDirectMusicSegmentState::GetStartPoint** method returns the offset into the segment at which play begins or began.

```
HRESULT GetStartPoint(  

```

```
MUSIC_TIME * pmtStart  
);
```

Parameters

pmtStart

Address of a variable to receive the music-time offset from the start of the segment at which the segment state initially plays.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::SetStartPoint, IDirectMusicSegmentState::GetStartTime

IDirectMusicSegmentState::GetStartTime

The **IDirectMusicSegmentState::GetStartTime** method gets the performance time at which the segment started or will start playing.

```
HRESULT GetStartTime(  
    MUSIC_TIME* pmtStart  
);
```

Parameters

pmtStart

Address of a variable to receive the music-time offset stored in this segment state.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Remarks

If the segment was started from some point other than the beginning, you can retrieve the time at which the beginning of the segment would have fallen by subtracting the time returned by **IDirectMusicSegmentState::GetStartPoint** from the value returned by this method.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicSegment::SetStartPoint, **IDirectMusicSegment::GetStartPoint**, **IDirectMusicSegmentState::GetStartPoint**

IDirectMusicStyle

The **IDirectMusicStyle** interface provides access to a style object. The style object provides the performance with the information that it needs to play musical patterns. For an overview, see Using Styles.

Since styles usually include bands and motifs, the **IDirectMusicStyle** interface provides methods for accessing these objects.

The **DirectMusicStyle** object also supports the **IDirectMusicObject** and **IPersistStream** interfaces for loading its data.

The methods of the **IDirectMusicStyle** interface can be organized in the following groups:

Enumeration	EnumBand
	EnumChordMap
	EnumMotif
Information	GetBand
	GetChordMap
	GetDefaultBand
	GetDefaultChordMap
	GetEmbellishmentLength
	GetMotif
	GetTempo
	GetTimeSignature

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicStyle::EnumBand

The **IDirectMusicStyle::EnumBand** method retrieves the name of the band with a given index value.

```
HRESULT EnumBand(
    DWORD dwIndex,
    WCHAR * pwszName
);
```

Parameters

dwIndex

Zero-based index into the style's band list.

pwszName

Address of a buffer to receive the band name. This should be of size MAX_PATH.

Return Values

If the method succeeds, it returns S_OK, S_FALSE if there is no band with the given index value, or DMUS_S_STRING_TRUNCATED if the length of the name is greater than MAX_PATH.

If it fails, the method can return one of the following error values:

```
DMUS_E_TYPE_UNSUPPORTED
E_POINTER
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicStyle::EnumChordMap

The **IDirectMusicStyle::EnumChordMap** method retrieves the name of the chord map with the given index value.

```
HRESULT EnumChordMap(
    DWORD dwIndex,
    WCHAR * pwszName
);
```

Parameters

dwIndex

Zero-based index of the chord map in the style's chord-map list.

pwszName

Address of a buffer to receive the chord-map 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 chord map with the given index value, or DMUS_S_STRING_TRUNCATED if the length of the name is greater than MAX_PATH.

If it fails, the method can return one of the following error values:

```
DMUS_E_TYPE_UNSUPPORTED
E_POINTER
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicStyle::EnumMotif

The **IDirectMusicStyle::EnumMotif** method retrieves the name of a motif with a given index value.


```
HRESULT EnumMotif(  
    DWORD dwIndex,  
    WCHAR * pwszName  
);
```

Parameters

dwIndex

Zero-based index into the style's motif list.

pwszName

Address of a buffer to receive the motif name. This should be of size MAX_PATH.

Return Values

If the method succeeds, the return value is S_OK, S_FALSE if there is no motif with the given index value, or DMUS_S_STRING_TRUNCATED if the length of the motif name is greater than MAX_PATH.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicStyle::GetMotif, Using Motifs

IDirectMusicStyle::GetBand

The **IDirectMusicStyle::GetBand** method retrieves the named band.

```
HRESULT GetBand(  
    WCHAR* pwszName,  
    IDirectMusicBand** ppBand  
);
```

Parameters

pwszName

Name of the band to be retrieved. This name is assigned by the author of the style.

ppBand

Address of a variable to receive a pointer to the band.

Return Values

If the method succeeds, the return value is S_OK if a band is returned, or S_FALSE if there is no band with that name.

If the method fails, the return value can be E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicStyle::GetDefaultBand, Using Bands

IDirectMusicStyle::GetChordMap

The **IDirectMusicStyle::GetChordMap** method retrieves a named chord map.

```
HRESULT GetChordMap(
    WCHAR* pwszName,
    IDirectMusicChordMap** ppChordMap
);
```

Parameters

pwszName

Name of the chord map to be retrieved.

ppChordMap

Address of a variable to receive a pointer to the **IDirectMusicChordMap** interface.

Return Values

If the method succeeds, the return value is S_OK if a chord map is returned, or S_FALSE if there is no chord map by that name.

If *ppChordMap* is not a valid pointer, the method returns E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicStyle::GetDefaultChordMap, Using Chord Maps

IDirectMusicStyle::GetDefaultBand

The **IDirectMusicStyle::GetDefaultBand** method retrieves the style's default band.

```
HRESULT GetDefaultBand(  
    IDirectMusicBand ** ppBand  
);
```

Parameters

ppBand

Address of a variable to receive a pointer to the default band.

Return Values

If the method succeeds, the return value is S_OK if a band is returned, or S_FALSE if the style does not have a default band.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicStyle::GetBand, Using Bands

IDirectMusicStyle::GetDefaultChordMap

The **IDirectMusicStyle::GetDefaultChordMap** method retrieves the style's default chord map.

```
HRESULT GetDefaultChordMap(  

```

```
IDirectMusicChordMap** ppChordMap
);
```

Parameters

ppChordMap

Address of a variable to receive a pointer to the **IDirectMusicChordMap** interface.

Return Values

If the method succeeds, the return value is S_OK if a chord map is returned, or S_FALSE if the style does not have a default chord map.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicStyle::GetChordMap, Using Chord Maps

IDirectMusicStyle::GetEmbellishmentLength

The **IDirectMusicStyle::GetEmbellishmentLength** method finds the shortest and longest lengths for patterns of the specified embellishment type and groove level.

```
HRESULT GetEmbellishmentLength(
    DWORD dwType,
    DWORD dwLevel,
    DWORD* pdwMin,
    DWORD* pdwMax
);
```

Parameters

dwType

Embellishment type. See **DMUS_COMMANDT_TYPES**.

dwLevel

Groove level in the range from 1 through 100. Ignored for nongroove embellishments.

pdwMin

Address of a variable to receive the length of the shortest pattern of the specified type and groove level.

pdwMax

Address of a variable to receive the length of the longest pattern of the specified type and groove level.

Return Values

If the method succeeds, the return value is S_OK or S_FALSE.

If it fails, the method can return E_POINTER.

Remarks

If there are no patterns of the specified type and groove level, the method returns S_FALSE.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicStyle::GetMotif

The **IDirectMusicStyle::GetMotif** method creates a segment containing the named motif.

```
HRESULT GetMotif(  
    WCHAR* pwszName,  
    IDirectMusicSegment** ppSegment  
);
```

Parameters

pwszName

Name of the motif to be retrieved.

ppSegment

Address of a variable to receive a pointer to a segment containing the named motif.

Return Values

If the method succeeds, the return value is S_OK or S_FALSE.

If it fails, the method can return E_POINTER.

Remarks

The method searches the style's list of motifs for one whose name matches *pwszName*. If one is found, a segment is created containing a motif track. The track references the style as its associated style and the motif as its pattern.

If there is no motif with the name, the method returns S_FALSE.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

Using Motifs

IDirectMusicStyle::GetTempo

The **IDirectMusicStyle::GetTempo** method retrieves the recommended tempo of the style.

```
HRESULT GetTempo(  
    double* pTempo  
);
```

Parameters

pTempo

Address of a variable to receive the recommended tempo of the style.

Return Values

If the method succeeds, the return value is S_OK.

If *pTempo* is not a valid pointer, the method returns E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Header: Declared in dmusici.h.

IDirectMusicStyle::GetTimeSignature

The **IDirectMusicStyle::GetTimeSignature** method retrieves the style's time signature.

```
HRESULT GetTimeSignature(  
    DMUS_TIMESIGNATURE* pTimeSig  
);
```

Parameters

pTimeSig

Address of a **DMUS_TIMESIGNATURE** structure to receive data.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicThru

The **IDirectMusicThru** interface supports thruing of music messages from a capture port to another port. It is obtained by calling **QueryInterface** on the **IDirectMusicPort** interface for the capture port. For an example, see the Remarks for **IDirectMusicThru::ThruChannel**.

The interface has the following method:

IDirectMusicThru **ThruChannel**

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusicc.h`.

See Also

Capturing Music

IDirectMusicThru::ThruChannel

The **IDirectMusicThru::ThruChannel** method establishes or breaks a thruing connection between a channel on a capture port and a channel on another port.

```
HRESULT ThruChannel(  
    DWORD dwSourceChannelGroup,  
    DWORD dwSourceChannel,  
    DWORD dwDestinationChannelGroup,  
    DWORD dwDestinationChannel,  
    LPDIRECTMUSICPORT pDestinationPort  
);
```

Parameters

dwSourceChannelGroup

Channel group on the capture port. In the current version of DirectMusic, this value is always 1.

dwSourceChannel

Source channel.

dwDestinationChannelGroup

Channel group on the destination port.

dwDestinationChannel

Destination channel.

pDestinationPort

Address of the **IDirectMusicPort** interface for the destination channel. Set this value to NULL to break an existing thruing connection.

Return Values

If the method succeeds, the return value is `S_OK`.

If it fails, the method can return one of the following error values:

`E_NOTIMPL`

`E_INVALIDARG`

DMUS_E_PORT_NOT_RENDER

Remarks

System-exclusive messages are not transmitted to the destination port.

Thruing to the Microsoft Software Synthesizer or other synthesizers that do not have a constant latency is not recommended. Thruing is done as soon as possible upon reception of the incoming MIDI events. Because of the comparatively high latency of the software synthesizer (compared with a hardware port) and the fact that it renders blocks of audio data at the same time, each event is delayed by a small, essentially random amount of time before it plays. This random offset shows up as jitter in the playback of the data. Latency of other devices (such as an MPU-401 port) is small enough that jitter does not occur.

If an application needs to thru to the software synthesizer, it should add a small offset to the incoming note event time stamps to compensate for the rendering latency of the synthesizer.

The following code example obtains the **IDirectMusicThru** 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(
    IDirectMusicPort *pCapturePort,
    IDirectMusicPort *pRenderPort)
{
    HRESULT hr;
    IDirectMusicThru *pThru;

    hr = pCapturePort->QueryInterface(IID_IDirectMusicThru,
        (void**)&pThru);
    if (FAILED(hr))
        return hr;

    for (DWORD dwChannel = 0; dwChannel < 16; dwChannel++)
    {
        hr = pThru->ThruChannel(1, dwChannel,
            1, dwChannel, pRenderPort);
        if (FAILED(hr))
            break;
    }

    pThru->Release();
    return hr;
}
```

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusic.h.

IDirectMusicTool

Tools are objects that implement the **IDirectMusicTool** interface. They are used inside graphs (see **IDirectMusicGraph**). When a message is sent using **IDirectMusicPerformance::SendPMsg**, the message flows through tools inside graphs. The tools can modify the message, make additional messages, remove messages, and so on.

This interface is of most interest to developers who want to create their own tools.

The methods of the **IDirectMusicTool** interface can be organized in the following groups:

Initialization	Init
Message management	Flush
	GetMediaTypeArraySize
	GetMediaTypes
	GetMsgDeliveryType
	ProcessPMsg

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

Overview of DirectMusic Data Flow, Message Creation and Delivery, DirectMusic Tools

IDirectMusicTool::Flush

The **IDirectMusicTool::Flush** method is called for each message in the queue when the performance stops. The tool can use the method to do whatever is necessary to flush the message. For instance, the output tool uses this method to ensure that any pending note-off messages are processed immediately.

```
HRESULT Flush(  
    IDirectMusicPerformance* pPerf,  
    DMUS_PMSG* pPMSG,  
    REFERENCE_TIME rtTime  
);
```

Parameters

pPerf
Address of the **IDirectMusicPerformance** interface.

pPMSG
Message to flush.

rtTime
Time at which to flush.

Return Values

Return values are determined by the implementation. If the method succeeds, the return value can be one of the following:

DMUS_S_REQUEUE
DMUS_S_FREE
S_OK

If it fails, the method can return E_POINTER.

Remarks

The message will have DMUS_PMSGF_TOOL_FLUSH set in its **dwFlags** member. See **DMUS_PMSG**.

If the method returns DMUS_S_REQUEUE, the tool wants the message to be requeued. This allows the tool to put a new time stamp and parameters on the message and requeue it, or to requeue the message with a different delivery type.

If the return value is DMUS_S_FREE, the tool wants the message freed automatically and does not want to requeue the message.

If S_OK is returned, the tool does not want the message to be freed automatically. Perhaps the tool is holding onto the message for some reason, or has freed it itself.

Be sure not to create a circular reference to the performance represented by *pPerf*. For more information, see DirectMusic Tools.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicTool::GetMediaTypeArraySize

The **IDirectMusicTool::GetMediaTypeArraySize** method retrieves the size of the array that must be passed in to the **IDirectMusicTool::GetMediaTypes** method. A return value of 0 indicates that the tool handles all types, and it is unnecessary to call **GetMediaTypes**.

```
HRESULT GetMediaTypeArraySize(  
    DWORD* pdwNumElements  
);
```

Parameters

pdwNumElements

Address of a variable to receive the number of media types. If 0 is returned in this field, all types are supported.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns S_OK. If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicTool::GetMediaTypes

The **IDirectMusicTool::GetMediaTypes** method retrieves a list of the types of messages that this tool supports.

```
HRESULT GetMediaTypes(  
    ...  
);
```

```
DWORD** padwMediaTypes,  
DWORD dwNumElements  
);
```

Parameters

padwMediaTypes

Address of an array of **DWORD**s. The method fills this array with the media types supported by this tool. For media types, see **DMUS_PMSGT_TYPES**.

dwNumElements

Number of elements in the *padwMediaTypes* array. This value is equal to the number returned by the **IDirectMusicTool::GetMediaTypeArraySize** method. If *dwNumElements* is less than this number, the method cannot return all the message types that are supported. If it is greater than this number, the extra elements in the array should be set to 0.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns **S_OK**, or **S_FALSE** if the method could not fill in all values because *dwNumElements* was too small. If it fails, the method can return one of the following error values:

E_POINTER
E_INVALIDARG
E_NOTIMPL

Remarks

If the method returns **E_NOTIMPL**, the tool processes all media types.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in *dmusic1.h*.

IDirectMusicTool::GetMsgDeliveryType

The **IDirectMusicTool::GetMsgDeliveryType** method retrieves the tool's delivery type, which determines when messages are to be delivered to the tool.

```
HRESULT GetMsgDeliveryType(  
    DWORD* pdwDeliveryType  
);
```

Parameters

pdwDeliveryType

Address of a variable to receive the delivery type. The returned value must be DMUS_PMSGF_TOOL_IMMEDIATE, DMUS_PMSGF_TOOL_QUEUE, or DMUS_PMSGF_TOOL_ATTIME. An unrecognized value in **pdwDeliveryType* is treated as DMUS_PMSGF_TOOL_IMMEDIATE by the graph.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns S_OK. If it fails, the method can return E_POINTER.

Remarks

For an overview of the delivery mechanism, see Message Creation and Delivery.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicTool::Init

The **IDirectMusicTool::Init** method is called when the tool is inserted into the graph, giving the tool an opportunity to perform any necessary initialization.

```
HRESULT Init(  
    IDirectMusicGraph* pGraph  
);
```

Parameters

pGraph

Calling graph.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns S_OK. If it fails, the method can return one of the following error values:

E_FAIL

E_NOTIMPL

Remarks

Because a tool can be inserted into more than one graph, this method must be able to deal gracefully with multiple calls.

Be sure not to create a circular reference to the graph represented by *pGraph*. For more information, see DirectMusic Tools.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicGraph::InsertTool

IDirectMusicTool::ProcessPMsg

The **IDirectMusicTool::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 **IDirectMusicTool::GetMediaTypes**.

```
HRESULT ProcessPMsg(  
    IDirectMusicPerformance* pPerf,  
    DMUS_PMSG* pPMSG  
);
```

Parameters

pPerf

Performance that is generating messages.

pPMSG

Message to process.

Return Values

Return values are determined by the implementation. If the method succeeds, the return value can be one of the following:

```
DMUS_S_QUEUE  
DMUS_S_FREE  
S_OK
```

If it fails, the method can return `E_POINTER`.

Remarks

If the method returns `DMUS_S_REQUEUE`, the tool wants the message to be requeued. This allows the tool to put a new time stamp and parameters on the message and requeue it, or to requeue the message with a different delivery type.

If the return value is `DMUS_S_FREE`, the tool wants the message freed automatically, and does not want to requeue the message.

If `S_OK` is returned, the tool does not want the message to be freed automatically. Perhaps the tool is holding onto the message for some reason, or has freed it itself.

Tools should not perform time-consuming activities because doing so can severely affect overall performance. Also be sure not to create a circular reference to the performance represented by *pPerf*. For more information, see DirectMusic Tools.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusic.h`.

See Also

IDirectMusicPerformance::SendPMsg, Messages, Message Creation and Delivery

IDirectMusicTrack

The **IDirectMusicTrack** interface represents a track object. Almost everything that has to do with the definition of a segment is stored in its tracks. The track mechanism allows segments to be infinitely extensible, and the segment does not need any knowledge of any of the music and audio technologies that it employs.

If you plan to install your own music playback mechanism in DirectMusic, you need to create a **IDirectMusicTrack** object to represent it. Otherwise, the methods of this interface are typically not called directly from the application.

Note

When implementing methods of the **IDirectMusicTrack** interface, be sure not to hold onto references to objects passed in. For example, if

IDirectMusicTrack::Init adds a reference to the **IDirectMusicSegment** interface that it receives as a parameter, ensure that this reference is released.

The **IDirectMusicTrack** interface has the following methods:

Initialization	Init
Playback	EndPlay

	InitPlay
	Play
Notification	AddNotificationType
	RemoveNotificationType
Parameters	GetParam
	IsParamSupported
	SetParam
Miscellaneous	Clone

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

The **DirectMusicTrack** object also supports the **IDirectMusicObject** and **IPersistStream** interfaces for loading its data.

The following table shows which methods are supported by the standard track types. For a general description of the standard types, see **Tracks**.

Track	IDirectMusicTrack methods		IPersistStream methods	
Band (CLSID_DirectMusicBandTrack)				
	AddNotificationType	No	IsDirty	Yes
	Clone	Yes	GetSizeMax	No
	EndPlay	Yes	Load	Yes
	GetParam	No	Save	Yes
	Init	Yes		
	InitPlay	Yes		
	IsParamSupported	Yes		
	Play	Yes		
	SetParam	Yes		
	RemoveNotificationType	No		
Chord (CLSID_DirectMusicChordTrack)				
	AddNotificationType	Yes	IsDirty	Yes
	Clone	Yes	GetSizeMax	No
	EndPlay	Yes	Load	Yes
	GetParam	Yes	Save	Yes
	Init	Yes		
	InitPlay	Yes		

	IsParamSupported	Yes		
	Play	Yes		
	SetParam	Yes		
	RemoveNotificationType	Yes		
Chord map (CLSID_DirectMusicChordMapTrack)				
	AddNotificationType	No	IsDirty	No
	Clone	Yes	GetSizeMax	No
	EndPlay	Yes*	Load	Yes
	GetParam	Yes	Save	No
	Init	Yes*		
	InitPlay	Yes*		
	IsParamSupported	Yes		
	Play	Yes*		
	SetParam	Yes		
	RemoveNotificationType	No		
Command (CLSID_DirectMusicCommandTrack)				
	AddNotificationType	Yes	IsDirty	Yes
	Clone	Yes	GetSizeMax	No
	EndPlay	Yes	Load	Yes
	GetParam	Yes	Save	Yes
	Init	Yes		
	InitPlay	Yes		
	IsParamSupported	Yes		
	Play	Yes		
	SetParam	Yes		
	RemoveNotificationType	Yes		
Motif (CLSID_DirectMusicMotifTrack)				
	AddNotificationType	Yes	IsDirty	No
	Clone	Yes	GetSizeMax	No
	EndPlay	Yes	Load	No
	GetParam	No	Save	No
	Init	Yes		
	InitPlay	Yes		
	IsParamSupported	Yes		
	Play	Yes		
	SetParam	Yes		
	RemoveNotificationType	Yes		
Mute (CLSID_DirectMusicMuteTrack)				

AddNotificationType	No	IsDirty	Yes
Clone	Yes	GetSizeMax	No
EndPlay	Yes*	Load	Yes
GetParam	Yes	Save	Yes
Init	Yes*		
InitPlay	Yes*		
IsParamSupported	Yes		
Play	Yes*		
SetParam	Yes		
RemoveNotificationType	No		
Sequence (CLSID_DirectMusicSeqTrack)			
AddNotificationType	No	IsDirty	No
Clone	Yes	GetSizeMax	No
EndPlay	Yes	Load	Yes
GetParam	No	Save	No
Init	Yes		
InitPlay	Yes		
IsParamSupported	No		
Play	Yes		
SetParam	No		
RemoveNotificationType	No		
Signpost (CLSID_DirectMusicSignPostTrack)			
AddNotificationType	No	IsDirty	Yes
Clone	Yes	GetSizeMax	No
EndPlay	Yes*	Load	Yes
GetParam	No	Save	Yes
Init	Yes*		
InitPlay	Yes*		
IsParamSupported	No		
Play	Yes*		
SetParam	No		
RemoveNotificationType	No		
Style (CLSID_DirectMusicStyleTrack)			
AddNotificationType	Yes	IsDirty	No
Clone	Yes	GetSizeMax	No
EndPlay	Yes	Load	Yes
GetParam	Yes	Save	No
Init	Yes		

	InitPlay	Yes		
	IsParamSupported	Yes		
	Play	Yes		
	SetParam	Yes		
	RemoveNotificationType	Yes		
SysEx (CLSID_DirectMusicSysExTrack)				
	AddNotificationType	No	IsDirty	No
	Clone	Yes	GetSizeMax	No
	EndPlay	Yes	Load	Yes
	GetParam	No	Save	No
	Init	Yes		
	InitPlay	Yes		
	IsParamSupported	No		
	Play	Yes		
	SetParam	No		
	RemoveNotificationType	No		
Tempo (CLSID_DirectMusicTempoTrack)				
	AddNotificationType	No	IsDirty	No
	Clone	Yes	GetSizeMax	No
	EndPlay	Yes	Load	Yes
	GetParam	Yes	Save	No
	Init	Yes		
	InitPlay	Yes		
	IsParamSupported	Yes		
	Play	Yes		
	SetParam	Yes		
	RemoveNotificationType	No		
Time signature** (CLSID_DirectMusicTimeSigTrack)				
	AddNotificationType	Yes	IsDirty	No
	Clone	Yes	GetSizeMax	No
	EndPlay	Yes	Load	Yes
	GetParam	Yes	Save	No
	Init	Yes		
	InitPlay	Yes		
	IsParamSupported	Yes		
	Play	Yes		
	SetParam	Yes		
	RemoveNotificationType	Yes		

Notes

- * The method returns a value other than E_NOTIMPL but does not do anything else.
- ** The time-signature track exists in imported MIDI files and DirectMusic Producer segments specifically created with one. In most cases, the style track implements the time-signature track's functionality, so it is not necessary for a segment that contains a style track to contain a time-signature track, as well.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

Tracks, Setting and Retrieving Track Parameters

IDirectMusicTrack::AddNotificationType

The **IDirectMusicTrack::AddNotificationType** method enables event notification for a track. It is similar to and called from the **IDirectMusicSegment::AddNotificationType** method.

```
HRESULT AddNotificationType(  
    REFGUID rguidNotificationType  
);
```

Parameters

rguidNotificationType

Reference to (C++) or address of (C) the identifier of the notification type to add. For the defined types, see **DMUS_NOTIFICATION_PMSG**. Applications can also define their own types for custom tracks.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if the track does not support the notification type.

If the track does not support notifications, the method returns E_NOTIMPL.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicTrack::RemoveNotificationType, Notification and Event Handling

IDirectMusicTrack::Clone

The **IDirectMusicTrack::Clone** method makes a copy of a track.

```
HRESULT Clone(  
    MUSIC_TIME mtStart,  
    MUSIC_TIME mtEnd,  
    IDirectMusicTrack** ppTrack  
);
```

Parameters

mtStart

Start of the part to clone. It should be 0 or greater and less than the length of the track.

mtEnd

End of the part to clone. It should be greater than *mtStart* and less than the length of the track.

ppTrack

Address of a variable to receive a pointer to the created track, if successful.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

```
E_FAIL  
E_INVALIDARG  
E_OUTOFMEMORY  
E_POINTER
```

Remarks

It is the caller's responsibility to call **Release** when finished with the track.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicTrack::EndPlay

The **IDirectMusicTrack::EndPlay** method is called when the object that originally called **IDirectMusicTrack::InitPlay** is destroyed.

```
HRESULT EndPlay(  
    void * pStateData  
);
```

Parameters

pStateData

State data returned from **IDirectMusicTrack::InitPlay**. This data should be freed in the **EndPlay** method.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return E_POINTER.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicTrack::GetParam

The **IDirectMusicTrack::GetParam** method retrieves data from a track.

```
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 Track Parameter Types.

mtTime

Time, in track time, from which to obtain the data.

pmtNext

Address of a variable to receive the track time (relative to the current time) until which the data is valid. If this returns a value of 0, either the data is always valid, or it is unknown when it might become invalid. If this information is not needed, *pmtNext* can be set to NULL.

pParam

Address of an allocated structure in which the data is to be returned. The structure must be of the appropriate kind and size for the data type identified by *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_FOUND
DMUS_E_NOT_INIT
DMUS_E_TYPE_DISABLED
DMUS_E_GET_UNSUPPORTED
E_POINTER

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicTrack::SetParam, **IDirectMusicTrack::IsParamSupported**, **IDirectMusicPerformance::GetParam**, **IDirectMusicSegment::GetParam**, Music Parameters

IDirectMusicTrack::Init

The **IDirectMusicTrack::Init** method is called by a segment when a track is added and performs any necessary initialization.


```

HRESULT Init(
    IDirectMusicSegment* pSegment
);

```

Parameters

pSegment

Segment to which this track belongs.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

```

    DMUS_E_NOT_INIT
    E_OUTOFMEMORY
    E_POINTER

```

Remarks

If the track plays messages, it should call

IDirectMusicSegment::SetPChannelsUsed in the **Init** method.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

IDirectMusicTrack::InitPlay

The **IDirectMusicTrack::InitPlay** method is called when a track is ready to start playing. It returns a pointer to state data, which is sent in to

IDirectMusicTrack::Play and **IDirectMusicTrack::EndPlay**.

```

HRESULT InitPlay(
    IDirectMusicSegmentState* pSegmentState,
    IDirectMusicPerformance* pPerformance,
    void** ppStateData,
    DWORD dwVirtualTrackID,
    DWORD dwFlags
);

```

Parameters

pSegmentState

Address of the calling **IDirectMusicSegmentState** interface.

pPerformance

Address of the calling **IDirectMusicPerformance** interface.

ppStateData

Address of a variable to receive 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 **IDirectMusicTrack::EndPlay** method. The pointer is passed to the **IDirectMusicTrack::Play** method.

dwVirtualTrackID

Virtual track ID assigned to this track instance.

dwFlags

DMUS_SEGF_FLAGS that control the track's behavior. See Remarks.

Return Values

If the method succeeds, the return value is **S_OK**.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT
E_OUTOFMEMORY
E_POINTER

Remarks

The track must store the *pSegmentState*, *pPerformance*, or *dwTrackID* parameters because they are also sent in to **IDirectMusicTrack::Play**.

The *dwFlags* parameter passes the flags that were handed to the performance in the call to **IDirectMusicPerformance::PlaySegment**. 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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in **dmusici.h**.

IDirectMusicTrack::IsParamSupported

The **IDirectMusicTrack::IsParamSupported** method determines whether the track supports a given data type in the **IDirectMusicTrack::GetParam** and **IDirectMusicTrack::SetParam** methods.

```
HRESULT IsParamSupported(
    REFGUID rguidType
);
```

Parameters

rguidType

Reference to (C++) or address of (C) the identifier of the type of data. See Track Parameter Types.

Return Values

If the method succeeds and the type is supported, the return value is S_OK.

If it fails, the method can return one of the following error values:

```
DMUS_E_TYPE_DISABLED
DMUS_E_TYPE_UNSUPPORTED
E_POINTER
E_NOTIMPL
```

Remarks

If a message type has been disabled by using one of the **SetParam** methods (see Disabling and Enabling Messages), the **IDirectMusicTrack::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 message type has already been disabled, or if passed GUID_EnableTempo when that message type 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 message type referred to by a GUID_EnableTempo, GUID_EnableTimeSig, GUID_DisableTempo, or GUID_DisableTimeSig parameter type.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicTrack::GetParam, **IDirectMusicTrack::SetParam**, Music Parameters

IDirectMusicTrack::Play

The **IDirectMusicTrack::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.

```
HRESULT Play(
    void* pStateData,
    MUSIC_TIME mtStart,
    MUSIC_TIME mtEnd,
    MUSIC_TIME mtOffset
    DWORD dwFlags,
    IDirectMusicPerformance* pPerf,
    IDirectMusicSegmentState* pSegSt,
    DWORD dwVirtualID
);
```

Parameters

pStateData

State data from the **IDirectMusicTrack::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 **IDirectMusicPerformance::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 the message.

pSegSt

Segment state that this track belongs to. The

IDirectMusicSegmentState::QueryInterface method can be called to obtain an

IDirectMusicGraph interface—to call **IDirectMusicGraph::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 **IDirectMusicPerformance::SendPMsg**.

Return Values

If the method succeeds, the return value can be **S_OK** or **DMUS_S_END**.

If it fails, the method can return one of the following error values:

DMUS_E_NOT_INIT
E_POINTER

Remarks

If the track is empty, the method returns **DMUS_S_END**.

Tracks generate messages in a medium-priority thread. You can call time-consuming functions, such as code to stream data from a file, from within a track's **Play** method. However, be sure to follow the guidelines for safe multithreading.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in **dmusici.h**.

IDirectMusicTrack::RemoveNotificationType

The **IDirectMusicTrack::RemoveNotificationType** method removes an event notification from a track. It is similar to and called from the **IDirectMusicSegment::RemoveNotificationType** method.

```
HRESULT RemoveNotificationType(
    REFGUID rguidNotificationType
);
```

Parameters

rguidNotificationType

Reference to (C++) or address of (C) the identifier of the notification type to remove. For the defined types, see **DMUS_NOTIFICATION_PMSG**.

Return Values

If the method succeeds, the return value is S_OK, or S_FALSE if the track does not support the notification type.

If the track does not support notifications, the method returns E_NOTIMPL.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicTrack::AddNotificationType, Notification and Event Handling

IDirectMusicTrack::SetParam

The **IDirectMusicTrack::SetParam** method sets data on a track.

```
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 Track Parameter Types.

mtTime

Time, in track time, at which to set the data.

pParam

Address of a structure containing the data, or NULL if no data is required. The structure must be of the appropriate kind and size for the data type identified by *rguidType*.

Return Values

If the method succeeds, the return value is S_OK.

If it fails, the method can return one of the following error values:

DMUS_E_SET_UNSUPPORTED

DMUS_E_TYPE_DISABLED
 E_OUTOFMEMORY
 E_POINTER

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicTrack::GetParam, **IDirectMusicTrack::IsParamSupported**, **IDirectMusicPerformance::SetParam**, **IDirectMusicSegment::SetParam**, Setting and Retrieving Track Parameters

IKsControl

The **IKsControl** interface is used to get, set, or query the support of properties, events, and methods. This interface is part of the Windows Driver Model kernel streaming architecture, but is also used by DirectMusic to expose properties of DirectMusic ports. To retrieve this interface, call the

IDirectMusicPort::QueryInterface method with IID_IKsControl in the *riid* parameter.

Routing of the property item request to the port varies, depending on the port implementation. No properties are supported by ports that represent DirectMusic emulation on top of the Win32 handle-based multimedia calls (**midiOut** and **midiIn** functions).

Property item requests to a port that represents a pluggable software synthesizer are answered totally in user mode. The topology of this type of port is a synthesizer (represented by an **IDirectMusicSynth** interface) connected to a sink node (an **IDirectMusicSynthSink** interface). The property request is given first to the synthesizer node, and then to the sink node if it is not recognized by the synthesizer.

The interface has the following methods. At present, only **KsProperty** is supported by DirectMusic.

IKsControl	KsProperty
	KsEvent
	KsMethod

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmksctrl.h.

See Also

Port Property Sets

IKsControl::KsProperty

The **IKsControl::KsProperty** method gets or sets the value of a property. For an overview, see Port Property Sets.

```
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 representing 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_BASICSSUPPORT` call, address of a variable to receive the number of bytes returned in *pvPropertyData* by the port.

Return Values

If the method succeeds, it returns `S_OK`.

If it fails, the method can return one of the following error values:

`E_FAIL`
`E_INVALIDARG`
`E_NOTIMPL`
`E_OUTOFMEMORY`
`E_POINTER`
`DMUS_E_UNKNOWN_PROPERTY`

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmksctrl.h`.

See Also

Port Property Sets

IReferenceClock

The **IReferenceClock** interface represents a system reference clock. The DirectMusic master clock and a port's latency clock implement this interface.

The interface has the following methods:

IReferenceClock	GetTime
	AdviseTime
	AdvisePeriodic
	Unadvise

All COM interfaces inherit the **IUnknown** interface methods. This interface supports the following three methods:

IUnknown	AddRef
	QueryInterface
	Release

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

See Also

IDirectMusic::GetMasterClock, **IDirectMusicPort::GetLatencyClock**, Timing

IReferenceClock::AdvisePeriodic

The **IReferenceClock::AdvisePeriodic** method requests an asynchronous, periodic notification that a duration has elapsed.

```
HRESULT AdvisePeriodic(
    REFERENCE_TIME rtStartTime,
    REFERENCE_TIME rtPeriodTime,
    HSEMAPHORE hSemaphore,
    DWORD * pdwAdviseCookie
);
```

Parameters

rtStartTime

Time that the notification should begin.

rtPeriodTime

Period of time between notifications.

hSemaphore

Handle of a semaphore through which to advise.

pdwAdviseCookie

Address of a variable to receive the identifier of the request. This is used to identify this call to **AdvisePeriodic** in the future—for example, to cancel it.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns **S_OK**.

If it fails, the method can return one of the following error values:

E_FAIL

E_POINTER

E_INVALIDARG

E_NOTIMPL

Remarks

When the time indicated by *rtStartTime* is reached, the semaphore whose handle is set as *hSemaphore* is released. Thereafter, the semaphore is released repetitively with a period of *rtPeriodTime*.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in *dmusicc.h*.

See Also

IReferenceClock::Unadvise

IReferenceClock::AdviseTime

The **IReferenceClock::AdviseTime** method requests an asynchronous notification that a time has elapsed.

```
HRESULT AdviseTime(  
    REFERENCE_TIME rtBaseTime,  
    REFERENCE_TIME rtStreamTime,  
    HEVENT hEvent,  
    DWORD * pdwAdviseCookie  
);
```

Parameters

rtBaseTime

Base reference time.

rtStreamTime

Stream offset time.

hEvent

Handle to an event through which to advise.

pdwAdviseCookie

Address of a variable to receive the identifier of the request. This is used to identify this call to **AdviseTime** in the future—for example, to cancel it.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns S_OK.

If it fails, the method can return one of the following error values:

E_FAIL
E_POINTER
E_INVALIDARG
E_NOTIMPL

Remarks

When the time *rtBaseTime* plus *rtStreamTime* is reached, the event whose handle is *hEvent* is set. If the time has already passed, the event is set immediately.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dmusicc.h.

See Also

IReferenceClock::Unadvise

IReferenceClock::GetTime

The **IReferenceClock::GetTime** method retrieves the current time.

```
HRESULT GetTime(  
    REFERENCE_TIME * pTime  
);
```

Parameters

pTime
Address of a variable to receive the current time.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns S_OK.

If it fails, the method can return one of the following error values:

E_FAIL

E_POINTER
E_INVALIDARG
E_NOTIMPL

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

IReferenceClock::Unadvise

The **IReferenceClock::Unadvise** method cancels a request for notification.

```
HRESULT Unadvise(  
    DWORD dwAdviseCookie  
);
```

Parameters

dwAdviseCookie

Identifier of the request that is to be canceled, as set in the **IReferenceClock::AdviseTime** or the **IReferenceClock::AdvisePeriodic** method.

Return Values

Return values are determined by the implementation. If the method succeeds, it returns S_OK.

If it fails, the method can return one of the following error values:

E_FAIL
E_POINTER
E_INVALIDARG
E_NOTIMPL

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

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 DirectMusic Messages.

This section contains information about the following structures used to contain message information:

- **DMUS_PMSG**
- **DMUS_CHANNEL_PRIORITY_PMSG**
- **DMUS_CURVE_PMSG**
- **DMUS_MIDI_PMSG**
- **DMUS_NOTE_PMSG**
- **DMUS_NOTIFICATION_PMSG**
- **DMUS_PATCH_PMSG**
- **DMUS_SYSEX_PMSG**
- **DMUS_TEMPO_PMSG**
- **DMUS_TIMESIG_PMSG**
- **DMUS_TRANSPOSE_PMSG**

See Also

IDirectMusicPerformance::AllocPMsg, **IDirectMusicPerformance::SendPMsg**, **IDirectMusicPerformance::FreePMsg**, **IDirectMusicTool::ProcessPMsg**

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 included in all message types (including **DMUS_PMSG** itself) by the inclusion of the **DMUS_PMSG_PART** macro, which expands to the syntax shown here.

```
typedef struct DMUS_PMSG {
    DWORD          dwSize;
    REFERENCE_TIME rtTime;
    MUSIC_TIME      mtTime;
    DWORD          dwFlags;
    DWORD          dwPChannel;
    DWORD          dwVirtualTrackID;
    IDirectMusicTool* pTool;
    IDirectMusicGraph* pGraph;
    DWORD          dwType;
```

```

        DWORD        dwVoiceID;
        DWORD        dwGroupID;
        IUnknown*     punkUser;
    } DMUS_PMSG;

```

dwSize

Size of the structure, in bytes. This member is initialized by **IDirectMusicPerformance::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

Various bits (see **DMUS_PMSGF_FLAGS** and **DMUS_TIME_RESOLVE_FLAGS**). Must contain **DMUS_PMSGF_REFTIME** or **DMUS_PMSGF_MUSICTIME**.

dwPChannel

Performance channel (PChannel). The port, channel group, and MIDI channel can be derived from this value by using the **IDirectMusicPerformance::PChannelInfo** method. Set this value to 0 for messages that are not channel-specific, such as tempo messages.

dwVirtualTrackID

Identifier of the track. Set to 0 if the message is not being sent by a track.

pTool

Address of the tool interface. Can be set by using **IDirectMusicGraph::StampPMsg**, or can be NULL if the message is not to go to tools other than the output tool.

pGraph

Address of the tool graph interface. Can be set by using **IDirectMusicGraph::StampPMsg**, or can be NULL if the message is not to go to tools other than the output tool.

dwType

Message type (see **DMUS_PMSGT_TYPES**).

dwVoiceID

Reserved for future use. Should be set to 0.

dwGroupID

Identifier of the track group or groups that the message belongs to if the message is being generated by a track. (Tracks are assigned to groups in the **IDirectMusicSegment::InsertTrack** method.) For most purposes, this value can be 0xFFFFFFFF.

punkUser

Address of an **IUnknown** interface supplied by the application. This pointer is always released when the message is freed. If the application wants to retain the object, it should call **AddRef** before the message is freed. If the message does not need a COM pointer, this value should be NULL.

Remarks

The **DMUS_PMSG** structure is used by itself for messages containing the following values in the **dwType** member:

DMUS_PMSGT_STOP

Sending a message of this type stops the performance at the specified time.

DMUS_PMSGT_DIRTY

When a control segment starts or ends, all tools in the segment and performance graphs receive a message of this type, indicating that if they cache data from **GetParam** calls, they must call **GetParam** again to refresh their data. Tools that want to receive this message type must indicate this through a call to **IDirectMusicTool::GetMediaTypes**. Tools in the performance graph receive one copy of the message for each segment in the performance. Such tools can safely ignore the extra messages with the same time stamp.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::SendPMsg

DMUS_CHANNEL_PRIORITY_PMSG

The **DMUS_CHANNEL_PRIORITY_PMSG** message structure contains data about a channel priority change.

```
typedef struct _DMUS_CHANNEL_PRIORITY_PMSG {
    DMUS_PMSG_PART
    DWORD dwChannelPriority;
} DMUS_CHANNEL_PRIORITY_PMSG;
```

DMUS_PMSG_PART

Macro for common message members. See **DMUS_PMSG**.

dwChannelPriority

Priority of the channel. For a list of defined values, see the Remarks for **IDirectMusicPort::GetChannelPriority**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPort::SetChannelPriority, **IDirectMusicPerformance::SendPMsg**

DMUS_CURVE_PMSG

DMUS_CURVE_PMSG is a message structure that represents a curve (for example, a sequence of continuous controller events).

```
typedef struct DMUS_CURVE_PMSG {
    DMUS_PMSG_PART
    MUSIC_TIME mtDuration;
    MUSIC_TIME mtOriginalStart;
    MUSIC_TIME mtResetDuration;
    short      nStartValue;
    short      nEndValue;
    short      nResetValue;
    WORD       wMeasure;
    short      nOffset;
    BYTE       bBeat;
    BYTE       bGrid;
    BYTE       bType;
    BYTE       bCurveShape;
    BYTE       bCCData;
    BYTE       bFlags;
} DMUS_CURVE_PMSG;
```

DMUS_PMSG_PART

Macro for common message members. See **DMUS_PMSG**.

mtDuration

Duration of the curve.

mtOriginalStart

Original start time. Must be set to either 0 when this message is created, or to the original start time of the curve.

mtResetDuration

How long after the curve is finished until the reset value is set. Ignored if DMUS_CURVE_RESET is not in **bFlags**.

nStartValue

Start value of the curve.

nEndValue

End value of the curve.

nResetValue

Reset value of the curve, set after **mtResetDuration** or upon a flush or invalidation. Ignored if DMUS_CURVE_RESET is not in **bFlags**.

wMeasure

Measure in which this curve occurs.

nOffset

Offset from the grid at which this curve occurs, in music time.

bBeat

Beat count (within a measure) at which this curve occurs.

bGrid

Grid offset from the beat at which this curve occurs.

bType

Type of curve. This can be one of the following values:

DMUS_CURVET_CCCURVE

Continuous controller curve (MIDI Control Change channel voice message; status byte &HB*n*, where *n* is the channel number).

DMUS_CURVET_MATCURVE

Monophonic aftertouch curve (MIDI Channel Pressure channel voice message; status byte &HD*n*).

DMUS_CURVET_PATCURVE

Polyphonic aftertouch curve (MIDI Poly Key Pressure channel voice message, status byte &HD*n*).

DMUS_CURVET_PBCURVE

Pitch-bend curve (MIDI Pitch Bend channel voice message; status byte &HE*n*).

bCurveShape

Shape of curve. This can be one of the following values:

DMUS_CURVES_EXP

Exponential curve shape.

DMUS_CURVES_INSTANT

Instant curve shape (beginning and end of curve happen at essentially the same time).

DMUS_CURVES_LINEAR

Linear curve shape.

DMUS_CURVES_LOG

Logarithmic curve shape.

DMUS_CURVES_SINE

Sine curve shape.

bCCData

CC number if this is a control change type.

bFlags

Set to **DMUS_CURVE_RESET** if **nResetValue** must be set when the time is reached or an invalidation occurs because of a transition. If 0, the curve stays permanently at the new value. All other bits are reserved.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::SendPMsg

DMUS_MIDI_PMSG

The **DMUS_MIDI_PMSG** structure contains data for a standard MIDI message not represented by another message type, such as a control change or pitch bend.

```
typedef struct DMUS_MIDI_PMSGG {
    DMUS_PMSG_PART
    BYTE bStatus;
    BYTE bByte1;
    BYTE bByte2;
    BYTE bPad[1];
} DMUS_MIDI_PMSG;
```

DMUS_PMSG_PART

Macro for common message members. See **DMUS_PMSG**.

bStatus

Standard MIDI status byte.

bByte1

First byte of the MIDI message. Ignored for MIDI messages that do not require it.

bByte2

Second byte of the MIDI message. Ignored for MIDI messages that do not require it.

bPad

Padding to a **WORD** boundary.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

MIDI Messages, **IDirectMusicPerformance::SendPMsg**

DMUS_NOTE_PMSG

The **DMUS_NOTE_PMSG** structure contains data for a music-note event.

```
typedef struct DMUS_NOTE_PMSG {
    DMUS_PMSG_PART
    MUSIC_TIME mtDuration;
    WORD      wMusicValue;
    WORD      wMeasure;
    short     nOffset;
    BYTE      bBeat;
    BYTE      bGrid;
    BYTE      bVelocity;
    BYTE      bFlags;
    BYTE      bTimeRange;
    BYTE      bDurRange;
    BYTE      bVelRange;
    BYTE      bPlayModeFlags;
    BYTE      bSubChordLevel;
    BYTE      bMidiValue;
} DMUS_NOTE_PMSG;
```

DMUS_PMSG_PART

Macro for common message members. See **DMUS_PMSG**.

mtDuration

Duration of the note.

wMusicValue

Description of the note. In most play modes, this is a packed array of 4-bit values, as follows:

Octave

In the range from –2 through 14. The note is transposed up or down by the octave times 12.

Chord position

In the range from 0 through 15, although it should never be above 3. The first position in the chord is 0.

Scale position

In the range from 0 through 15. Typically it is only from 0 through 2, but it is possible to have a one-note chord and have everything above the chord be interpreted as a scale position.

Accidental

In the range from –8 through 7, but typically in the range from –2 through 2. This represents an offset that takes the note out of the scale.

In the fixed-play modes, the music value is a MIDI note value in the range from 0 through 127.

wMeasure

Measure in which this note occurs.

nOffset

Offset from the grid at which this note occurs, in music time.

bBeat

Beat (in measure) at which this note occurs.

bGrid

Grid offset from the beat at which this note occurs.

bVelocity

Note velocity.

bFlags

See **DMUS_NOTEF_FLAGS**.

bTimeRange

Range by which to randomize time.

bDurRange

Range by which to randomize duration.

bVelRange

Range by which to randomize velocity.

bPlayModeFlags

Play mode determining how the music value is related to the chord and subchord. For a list of values, see **DMUS_PLAYMODE_FLAGS**.

bSubChordLevel

Subchord level that the note uses. See **DMUS_SUBCHORD**.

bMidiValue

MIDI note value, converted from **wMusicValue**.

Remarks

When a note is to be played, the **DMUS_NOTE_PMSG** flows through the graph and any tools in the graph until it reaches the final MIDI output tool. When the tool recognizes that **DMUS_NOTEF_NOTEON** is set in the **bFlags** member, it sends a MIDI note-on message to the correct port (according to the **dwPChannel** member of the **DMUS_PMSG** part). It then clears the flag, adds **mtDuration** to the time stamp, and queues the message so that the note is turned off at the appropriate time.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::SendPMsg, Music Values and MIDI Notes

DMUS_NOTIFICATION_PMSG

The **DMUS_NOTIFICATION_PMSG** structure is a **DMUS_PMSG** that represents a notification.

```
typedef struct DMUS_NOTIFICATION_PMSG {
    DMUS_PMSG_PART
    GUID  guidNotificationType;
    DWORD dwNotificationOption;
    DWORD dwField1;
    DWORD dwField2;
} DMUS_NOTIFICATION_PMSG;
```

DMUS_PMSG_PART

Macro for common message members. See **DMUS_PMSG**.

guidNotificationType

Identifier of the notification type. The following types are defined:

GUID_NOTIFICATION_CHORD

Chord change.

GUID_NOTIFICATION_COMMAND

Command event.

GUID_NOTIFICATION_MEASUREANDBEAT

Measure and beat event.

GUID_NOTIFICATION_PERFORMANCE

Performance event, further defined in **dwNotificationOption**.

GUID_NOTIFICATION_SEGMENT

Segment event, further defined in **dwNotificationOption**.

dwNotificationOption

Identifier of the notification subtype.

If the notification type is **GUID_NOTIFICATION_SEGMENT**, this member can contain one of the following values:

DMUS_NOTIFICATION_SEGABORT

The segment was stopped by **IDirectMusicPerformance::Stop**.

DMUS_NOTIFICATION_SEGALMOSTEND

The segment has reached the end minus the prepare time.

DMUS_NOTIFICATION_SEGEND

The segment has ended.

DMUS_NOTIFICATION_SEGLOOP

The segment has looped.

DMUS_NOTIFICATION_SEGSTART

The segment has started.

If the notification type is GUID_NOTIFICATION_COMMAND, this member can contain one of the following values:

DMUS_NOTIFICATION_GROOVE

Groove change.

DMUS_NOTIFICATION_EMBELLISHMENT

Embellishment command (intro, fill, break, or end).

If the notification type is GUID_NOTIFICATION_PERFORMANCE, this member can contain one of the following values:

DMUS_NOTIFICATION_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.

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 **IDirectMusicSegmentState** interface, call **IDirectMusicSegmentState::GetSegment** to get the **IDirectMusicSegment** pointer, and then call the **IDirectMusicSegment::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 `IDirectMusicTrack::AddNotificationType`.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusici.h`.

See Also

Notification and Event Handling, `IDirectMusicPerformance::SendPMsg`

DMUS_PATCH_PMSG

The `DMUS_PATCH_PMSG` structure contains data for a program-change message.

```
typedef struct DMUS_PATCH_PMSG {
    DMUS_PMSG_PART
    BYTE  byInstrument;
    BYTE  byMSB;
    BYTE  byLSB;
    BYTE  byPad[1];
} DMUS_PATCH_PMSG;
```

DMUS_PMSG_PART

Macro for common message members. See `DMUS_PMSG`.

byInstrument

Patch number of the instrument.

byMSB

Most significant byte of bank select.

byLSB

Least significant byte of bank select.

byPad

Padding to a **WORD** boundary. This value is ignored.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

DMUS_MIDI_PMSG, IDirectMusicPerformance::SendPMsg, MIDI Messages

DMUS_SYSEX_PMSG

The **DMUS_SYSEX_PMSG** structure is a **DMUS_PMSG** that represents a MIDI system-exclusive message.

```
typedef struct DMUS_SYSEX_PMSG {
    DMUS_PMSG_PART
    DWORD dwLen;
    BYTE  abData[1];
} DMUS_SYSEX_PMSG;
```

DMUS_PMSG_PART

Macro for common message members. See **DMUS_PMSG**.

dwLen

Length of the data, in bytes.

abData

Array of data. For an example of how to allocate memory and copy data to this member, see the Remarks for **IDirectMusicPerformance::SendPMsg**.

Remarks

The data part of a system-exclusive message must begin with the System Exclusive identifier (0xF0) and end with EOX (0xF7).

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

DMUS_MIDI_PMSG, DMUS_PATCH_PMSG, MIDI Messages, IDirectMusicPerformance::SendPMsg

DMUS_TEMPO_PMSG

The **DMUS_TEMPO_PMSG** structure contains data for a message that controls the performance's tempo.

```
typedef struct DMUS_TEMPO_PMSG {
    DMUS_PMSG_PART
    double dblTempo;
} DMUS_TEMPO_PMSG;
```

DMUS_PMSG_PART

Macro for common message members. See **DMUS_PMSG**.

dblTempo

Tempo, in the range from **DMUS_TEMPO_MIN** through **DMUS_TEMPO_MAX**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::SendPMsg

DMUS_TIMESIG_PMSG

The **DMUS_TIMESIG_PMSG** structure contains data for a message that controls the time signature of the performance.

```
typedef struct _DMUS_TIMESIG_PMSG {
    DMUS_PMSG_PART
    BYTE bBeatsPerMeasure;
    BYTE bBeat;
    WORD wGridsPerBeat;
} DMUS_TIMESIG_PMSG;
```

DMUS_PMSG_PART

Macro for common message members. See **DMUS_PMSG**.

bBeatsPerMeasure

Beats per measure (top of the time signature).

bBeat

Note that receives the beat (bottom of the time signature), where 1 is a whole note, 2 is a half note, 4 is a quarter note, and so on. Zero is a 256th note.

wGridsPerBeat

Grids (subdivisions) per beat. This value determines the timing resolution for certain music events—for example, segments cued with the DMUS_SEGF_GRID flag (see **DMUS_SEGF_FLAGS**).

Remarks

Time-signature messages are generated by the time-signature track and the style track. In general, a segment contains one or the other, but not both. A segment representing a MIDI file has a time-signature track, but most segments authored with a tool such as DirectMusic Producer contain time-signature information in the style track.

By default, only the primary segment sends time-signature messages. For information on how to change this behavior, see *Disabling and Enabling Messages*.

The time signature is used by the performance to resolve time to measure, beat, and grid boundaries in all methods in which the time can be adjusted by **DMUS_SEGF_FLAGS** or **DMUS_TIME_RESOLVE_FLAGS**. The time-signature and style tracks also use the time signature to generate notifications on measure and beat boundaries. See **DMUS_NOTIFICATION_PMSG**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::SendPMsg, **DMUS_TIMESIGNATURE**

DMUS_TRANSPOSE_PMSG

The **DMUS_TRANSPOSE_PMSG** structure contains data for a message that controls a transposition.

```
typedef struct _DMUS_TRANSPOSE_PMSG {
    DMUS_PMSG_PART
    short nTranspose;
} DMUS_TRANSPOSE_PMSG;
```

DMUS_PMSG_PART

Macro for common message members. See **DMUS_PMSG**.

nTranspose

Number of semitones by which to transpose. This can be a negative value.

Remarks

If the transposition of a note puts it outside the standard MIDI range from 0 through 127, it does not play.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::SendPMsg

Structures

This section contains reference information for the following run-time structures used in DirectMusic:

- **DMUS_BUFFERDESC**
- **DMUS_CHORD_KEY**
- **DMUS_CHORD_PARAM**
- **DMUS_CLOCKINFO**
- **DMUS_COMMAND_PARAM**
- **DMUS_COMMAND_PARAM2**
- **DMUS_EVENTHEADER**
- **DMUS_MUTE_PARAM**
- **DMUS_NOTERANGE**
- **DMUS_OBJECTDESC**
- **DMUS_PORTCAPS**
- **DMUS_PORTPARAMS**
- **DMUS_RHYTHM_PARAM**
- **DMUS_SUBCHORD**
- **DMUS_SYNTHSTATS**
- **DMUS_TEMPO_PARAM**
- **DMUS_TIMESIGNATURE**
- **DMUS_VERSION**
- **DMUS_WAVES_REVERB_PARAMS**
- **KSPROPERTY**

Special categories of structures are contained in the following sections:

- Messages
- File Structures
- DLS Structures

Note

The memory for all DirectX structures must be initialized to 0 before use. In addition, all structures that contain a **dwSize** member must set the member to the size of the structure, in bytes, before use. The following DirectDraw example performs these tasks on a common structure, **DDCAPS**:

```
DDCAPS ddcaps; // Can't use this yet.

ZeroMemory(&ddcaps, sizeof(ddcaps));
ddcaps.dwSize = sizeof(ddcaps);

// Now the structure can be used.
.
```

DMUS_BUFFERDESC

The **DMUS_BUFFERDESC** structure is used to describe a buffer for the **IDirectMusic::CreateMusicBuffer** method.

```
typedef struct _DMUS_BUFFERDESC {
    DWORD dwSize;
    DWORD dwFlags;
    GUID guidBufferFormat;
    DWORD cbBuffer;
} DMUS_BUFFERDESC, *LPDMUS_BUFFERDESC;
```

Members

dwSize

Size of this structure, in bytes. This member must be initialized before the structure is used.

dwFlags

No flags are defined.

guidBufferFormat

Identifier of the KS format of the buffer. The value **GUID_NULL** represents **KSDATAFORMAT_SUBTYPE_DIRECTMUSIC**.

If **guidBufferFormat** represents a KS format other than **KSDATAFORMAT_SUBTYPE_DIRECTMUSIC**, the application must verify that the port playing back the data understands the specified format; if not, the

buffer is ignored. To find out if 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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

DMUS_EVENTHEADER, **IDirectMusicBuffer::PackStructured**, **IDirectMusicBuffer::PackUnstructured**

DMUS_CHORD_KEY

The **DMUS_CHORD_KEY** structure is used to describe a chord in the **IDirectMusicPerformance::MIDIToMusic** and **IDirectMusicPerformance::MusicToMIDI** methods.

```
typedef struct _DMUS_CHORD_KEY {
    WCHAR      wszName[16];
    WORD       wMeasure;
    BYTE       bBeat;
    BYTE       bSubChordCount;
    DMUS_SUBCHORD SubChordList[DMUS_MAXSUBCHORD];
    DWORD      dwScale;
    BYTE       bKey;
} DMUS_CHORD_KEY;
```

Members

wszName

Name of the chord.

wMeasure

Measure that the chord falls on.

bBeat

Beat that the chord falls on.

bSubChordCount

Number of chords in the chord's list of subchords.

SubChordList

Array of **DMUS_SUBCHORD** structures, describing the components that make up the chord.

dwScale

Scale underlying the entire chord.

bKey

Key underlying the entire chord.

Remarks

This structure is also defined as a **DMUS_CHORD_PARAM** structure for use in setting and retrieving the GUID_ChordParam track parameter.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::GetParam, **IDirectMusicPerformance::SetParam**, **IDirectMusicSegment::GetParam**, **IDirectMusicSegment::SetParam**, **IDirectMusicTrack::GetParam**, **IDirectMusicTrack::SetParam**, Music Parameters

DMUS_CHORD_PARAM

The **DMUS_CHORD_PARAM** structure is used as the *pParam* parameter in calls to the various **GetParam** and **SetParam** methods when the track is a chord track and *rguidType* is GUID_ChordParam.

```
typedef DMUS_CHORD_KEY DMUS_CHORD_PARAM;
```

See **DMUS_CHORD_KEY**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

IDirectMusicPerformance::GetParam, **IDirectMusicPerformance::SetParam**,
IDirectMusicSegment::GetParam, **IDirectMusicSegment::SetParam**,
IDirectMusicTrack::GetParam, **IDirectMusicTrack::SetParam**, Music
Parameters

DMUS_CLOCKINFO

The **DMUS_CLOCKINFO** structure reports information about a clock enumerated by using the **IDirectMusic::EnumMasterClock** method.

```
typedef struct _DMUS_CLOCKINFO{
    DWORD        dwSize;
    DMUS_CLOCKTYPE ctType;
    GUID         guidClock;
    WCHAR        wszDescription[DMUS_MAX_DESCRIPTION];
} DMUS_CLOCKINFO, *LPDMUS_CLOCKINFO;
```

Members

dwSize

Size of the structure, in bytes This member must be initialized to **sizeof(DMUS_CLOCKINFO)** 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 **IDirectMusic::SetMasterClock** method to set the master clock for DirectMusic.

wszDescription

Description of the clock.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

DMUS_COMMAND_PARAM

The **DMUS_COMMAND_PARAM** structure is used as the *pParam* parameter in calls to various **GetParam** and **SetParam** methods when the track is a command track and the *rguidType* parameter is **GUID_CommandParam**.


```
typedef struct {
    BYTE bCommand;
    BYTE bGrooveLevel;
    BYTE bGrooveRange;
} DMUS_COMMAND_PARAM;
```

Members

bCommand

Command type. See **DMUS_COMMANDT_TYPES**.

bGrooveLevel

Groove level of the command. The groove level is a value in the range from 1 through 100.

bGrooveRange

Amount by which the groove level can be randomized. For instance, if the groove level is 35 and the range is 4, the actual groove level could be anywhere from 33 through 37. Not currently implemented.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmuscf.h.

See Also

IDirectMusicPerformance::GetParam, **IDirectMusicPerformance::SetParam**,
IDirectMusicSegment::GetParam, **IDirectMusicSegment::SetParam**,
IDirectMusicTrack::GetParam, **IDirectMusicTrack::SetParam**

DMUS_COMMAND_PARAM2

The **DMUS_COMMAND_PARAM2** structure is used as the *pParam* parameter in calls to various **GetParam** and **SetParam** methods when the track is a command track and the *rguidType* parameter is GUID_CommandParam2.

```
typedef struct {
    MUSIC_TIME mtTime;
    BYTE bCommand;
    BYTE bGrooveLevel;
    BYTE bGrooveRange;
} DMUS_COMMAND_PARAM2;
```

Members

mtTime

Time of the command.

bCommand

Command type. See **DMUS_COMMANDT_TYPES**.

bGrooveLevel

Groove level of the command. The groove level is a value in the range from 1 through 100.

bGrooveRange

Amount by which the groove level can be randomized. For instance, if the groove level is 35 and the range is 4, the groove level could be anywhere from 33 through 37. Not currently implemented.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

IDirectMusicPerformance::GetParam, **IDirectMusicPerformance::SetParam**,
IDirectMusicSegment::GetParam, **IDirectMusicSegment::SetParam**,
IDirectMusicTrack::GetParam, **IDirectMusicTrack::SetParam**

DMUS_EVENTHEADER

The **DMUS_EVENTHEADER** structure precedes and describes an event in a port buffer.

```
typedef struct _DMUS_EVENTHEADER {
    DWORD      cbEvent;
    DWORD      dwChannelGroup;
    REFERENCE_TIME  rtDelta;
    DWORD      dwFlags;
} DMUS_EVENTHEADER, *LPDMUS_EVENTHEADER;
```

Members

cbEvent

Number of bytes in the event.

dwChannelGroup

Group to which the event belongs.

rtDelta

Offset from the start time of the buffer.

dwFlags

Set to `DMUS_EVENT_STRUCTURED` if the event is parsable MIDI data.

Remarks

The `Pshpack4.h` header file is included before the declaration of this structure to turn off automatic alignment of structures so that the data immediately follows the header. (For more information, see the comments in `Pshpack4.h`.) `Poppack.h` is then included to turn alignment back on, and the entire structure (header plus event) is padded to an 8-byte boundary.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusbuff.h`.

See Also

IDirectMusicBuffer::GetNextEvent, **IDirectMusicBuffer::PackStructured**, **IDirectMusicBuffer::PackUnstructured**

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 **IDirectMusicPerformance::DownloadInstrument** and **IDirectMusicPort::DownloadInstrument** methods to specify what notes the instrument should respond to and, therefore, what instrument regions need to be downloaded.

```
typedef struct _DMUS_NOTERANGE {
    DWORD dwLowNote;
    DWORD dwHighNote;
} DMUS_NOTERANGE, *LPDMUS_NOTERANGE;
```

Members**dwLowNote**

Low note for this range of MIDI notes to which the instrument must respond.

dwHighNote

High note for this range of MIDI notes to which the instrument must respond.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

DMUS_MUTE_PARAM

The **DMUS_MUTE_PARAM** structure is used as the *pParam* parameter in calls to the various **GetParam** and **SetParam** methods when the track is a mute track and *rguidType* is GUID_MuteParam.

```
typedef struct _DMUS_MUTE_PARAM {  
    DWORD  dwPChannel;  
    DWORD  dwPChannelMap;  
    BOOL   fMute;  
} DMUS_MUTE_PARAM;
```

Members

dwPChannel

Performance channel to mute or remap. If the structure is being passed to **GetParam**, this member must be initialized.

dwPChannelMap

Channel to which **dwPChannel** is being mapped. This member is ignored if **fMute** is TRUE.

fMute

TRUE if **dwPChannel** is being muted.

Remarks

If you wanted all the notes on PChannel 3 to play on PChannel 9 instead, you would set **dwPChannel** to 3 and **dwPChannelMap** to 9 before passing the structure to one of the **SetParam** methods. If you wanted to mute the notes on PChannel 8, you would set **dwPChannel** to 8 and **dwPChannelMap** to 0xFFFFFFFF.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

IDirectMusicPerformance::GetParam, **IDirectMusicPerformance::SetParam**,
IDirectMusicSegment::GetParam, **IDirectMusicSegment::SetParam**,
IDirectMusicTrack::GetParam, **IDirectMusicTrack::SetParam**

DMUS_OBJECTDESC

The **DMUS_OBJECTDESC** structure is used to describe a DirectMusic object. It is passed to the **IDirectMusicLoader::GetObject** method to identify the object that the loader should retrieve from storage. Information about an object is retrieved in this structure by the **IDirectMusicLoader::EnumObject** and **IDirectMusicObject::GetDescriptor** methods.

```
typedef struct _DMUS_OBJECTDESC {
    DWORD      dwSize;
    DWORD      dwValidData;
    GUID       guidObject;
    GUID       guidClass;
    FILETIME   ftDate;
    DMUS_VERSION vVersion;
    WCHAR      wszName[DMUS_MAX_NAME];
    WCHAR      wszCategory[DMUS_MAX_CATEGORY];
    WCHAR      wszFileName[DMUS_MAX_FILENAME];
    LONGLONG   lMemLength;
    PBYTE      pbMemData;
} DMUS_OBJECTDESC, *LPDMUS_OBJECTDESC;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized to **sizeof(DMUS_OBJECTDESC)** before the structure is passed to any method.

dwValidData

Flags describing which members are valid and giving further information about some members. The following values are defined:

DMUS_OBJ_OBJECT

The **guidObject** member is valid.

DMUS_OBJ_CLASS

The **guidClass** member is valid.

DMUS_OBJ_NAME

The **wszName** member is valid.

DMUS_OBJ_CATEGORY

The **wszCategory** member is valid.

DMUS_OBJ_FILENAME

The **wszFileName** member is valid.

DMUS_OBJ_FULLPATH

The **wszFileName** member contains the full path. If this flag is set, the loader assumes that **wszFileName** is valid even if **DMUS_OBJ_FILENAME** has not been set.

DMUS_OBJ_MEMORY

The object is in memory, and **llMemLength** and **pbMemData** are valid.

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.

DMUS_OBJ_DATE

The **ftDate** member is valid.

DMUS_OBJ_LOADED

The object is currently loaded in memory.

guidObject

Unique identifier for this object.

guidClass

Unique identifier for the class of object. All the standard objects have defined identifiers consisting of "CLSID_" plus the name of the object. For example, a segment object is identified as **CLSID_DirectMusicSegment**. See the defines in the **Dmusic.h** header file.

ftDate

Date that the object was last edited.

vVersion

DMUS_VERSION structure containing version information.

wszName

Name of the object.

wszCategory

Category for the object.

wszFileName

File path. If **DMUS_OBJ_FULLPATH** is set, this is the full path; otherwise, it is the file name.

llMemLength

Size of data in memory.

pbMemData

Data in memory.

Remarks

At least one of **wszName**, **guidObject**, and **wszFileName** must be filled with valid data to retrieve the object by using the **IDirectMusicLoader::GetObject** method.

The name and category strings use 16-bit characters in the **WCHAR** format, not 8-bit ANSI **chars**. 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.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

DirectMusic Loader

DMUS_PORTCAPS

The **DMUS_PORTCAPS** structure receives information about a port enumerated by a call to the **IDirectMusic::EnumPort** method. The structure is also used to return information through the **IDirectMusicPort::GetCaps** method.

```
typedef struct _DMUS_PORTCAPS {
    DWORD dwSize;
    DWORD dwFlags;
    GUID guidPort;
    DWORD dwClass;
    DWORD dwType;
    DWORD dwMemorySize;
    DWORD dwMaxChannelGroups;
    DWORD dwMaxVoices;
    DWORD dwMaxAudioChannels;
    DWORD dwEffectFlags;
    WCHAR wszDescription[DMUS_MAX_DESCRIPTION];
} DMUS_PORTCAPS, *LPDMUS_PORTCAPS;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized to **sizeof(DMUS_PORTCAPS)** before the structure is passed to any method.

dwFlags

Flags describing various capabilities of the port. This field can contain one or more of the following values:

DMUS_PC_DIRECTSOUND

The port supports streaming wave data to DirectSound.

DMUS_PC_DLS

The port supports DLS Level 1 sample collections.

DMUS_PC_DLS2

The port supports DLS Level 2 sample collections.

DMUS_PC_EXTERNAL

The port connects to devices outside the host—for example, devices connected over an external MIDI port such as the MPU-401.

DMUS_PC_GMINHARDWARE

The synthesizer has its own GM instrument set, so GM instruments do not need to be downloaded.

DMUS_PC_GSINHARDWARE

This port contains the Roland GS sound set in hardware.

DMUS_PC_MEMORYSIZEFIXED

Memory available for DLS instruments cannot be adjusted.

DMUS_PC_SHAREABLE

More than one port can be created that uses the same range of channel groups on the device. Unless this bit is set, the port can be opened only in exclusive mode. In exclusive mode, an attempt to create a port fails unless free channel groups are available to assign to the create request.

DMUS_PC_SOFTWARESYNTH

The port is a software synthesizer.

DMUS_PC_XGINHARDWARE

The port contains the Yamaha XG extensions in hardware.

guidPort

Identifier of the port. This value can be passed to the **IDirectMusic::CreatePort** method to get an **IDirectMusicPort** interface for the port.

dwClass

Class of this port. The following classes are defined:

DMUS_PC_INPUTCLASS

Input port.

DMUS_PC_OUTPUTCLASS

Output port.

dwType

Type of this port. The following types are defined:

DMUS_PORT_WINMM_DRIVER

Windows multimedia driver.

DMUS_PORT_USER_MODE_SYNTH

User-mode synthesizer.

DMUS_PORT_KERNEL_MODE

WDM driver.

dwMemorySize

Amount of memory available to store DLS instruments. If the port is using system memory and the amount is therefore limited only by the available system memory, this field contains `DMUS_PC_SYSTEMMEMORY`.

dwMaxChannelGroups

Maximum number of channel groups supported by this port. A channel group is a set of 16 MIDI channels.

dwMaxVoices

Maximum number of voices that can be allocated when this port is opened. The value can be -1 if the driver does not support returning this parameter.

dwMaxAudioChannels

Maximum number of audio channels that can be rendered by the port. The value can be -1 if the driver does not support returning this parameter.

dwEffectFlags

Flags indicating what audio effects are available on the port.

The following flags are defined:

`DMUS_EFFECT_NONE`

No effects are supported.

`DMUS_EFFECT_REVERB`

The port supports reverb.

`DMUS_EFFECT_CHORUS`

The port supports chorus.

wszDescription

Description of the port. This can be a system-generated name, such as `L"MPU-401 Output Port [330]"`, or a user-specified friendly name, such as `L"Port w/ External SC-55"`.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusicc.h`.

DMUS_PORTPARAMS

The **DMUS_PORTPARAMS** structure contains parameters for the opening of a DirectMusic port. These parameters are passed in when the **IDirectMusic::CreatePort** method is called.

```
typedef struct _DMUS_PORTPARAMS {
    DWORD dwSize;
    DWORD dwValidParams;
    DWORD dwVoices;
    DWORD dwChannelGroups;
    DWORD dwAudioChannels;
```

```
    DWORD dwSampleRate;  
    DWORD dwEffectFlags;  
    DWORD fShare;  
} DMUS_PORTPARAMS, *LPDMUS_PORTPARAMS;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized to **sizeof(DMUS_PORTPARAMS)** before the structure is passed to a method.

dwValidParams

Specifies which port parameters in this structure have been filled in. Setting the flag for a particular 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
```

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 **IDirectMusic::EnumPort** and **IDirectMusicPort::GetCaps** methods.

dwAudioChannels

Desired number of output channels.

dwSampleRate

Desired sample rate, in hertz.

dwEffectFlags

Flags indicating which special effects are desired. The following flags are defined:

```
DMUS_EFFECT_NONE  
DMUS_EFFECT_REVERB  
DMUS_EFFECT_CHORUS
```

fShare

If TRUE, all ports use the channel groups assigned to this port. If FALSE, the port is opened in exclusive mode, and the use of the same channel groups by other ports is forbidden.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusic.h.

See Also

DMUS_PORTCAPS

DMUS_RHYTHM_PARAM

The **DMUS_RHYTHM_PARAM** structure is used as the *pParam* parameter in calls to the various **GetParam** methods when the track is a chord track and *rguidType* is GUID_RhythmParam.

```
typedef struct {
    DMUS_TIMESIGNATURE TimeSig;
    DWORD               dwRhythmPattern;
} DMUS_RHYTHM_PARAM;
```

Members

TimeSig

DMUS_TIMESIGNATURE structure containing the time signature of the rhythm parameter. This structure must be initialized before the **DMUS_RHYTHM_PARAM** structure is passed to **GetParam**.

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

IDirectMusicPerformance::GetParam, **IDirectMusicPerformance::SetParam**, **IDirectMusicSegment::GetParam**, **IDirectMusicSegment::SetParam**, **IDirectMusicTrack::GetParam**, **IDirectMusicTrack::SetParam**

DMUS_SUBCHORD

The **DMUS_SUBCHORD** structure is used in the **SubChordList** member of a **DMUS_CHORD_PARAM** structure.

```
typedef struct {
    DWORD dwChordPattern;
    DWORD dwScalePattern;
    DWORD dwInversionPoints;
    DWORD dwLevels;
    BYTE  bChordRoot;
    BYTE  bScaleRoot;
} DMUS_SUBCHORD;
```

Members

dwChordPattern

Notes in the subchord. Each of the lower 24 bits represents a semitone, starting with the root at the least significant bit, and the bit is set if the note is in the chord.

dwScalePattern

Notes in the scale. Each of the lower 24 bits represents a semitone, starting with the root at the least significant bit, and the bit is set if the note is in the scale.

dwInversionPoints

Points in the scale at which inversions can occur. Bits that are off signify that the notes in the interval cannot be inverted. Thus, the pattern 100001111111 indicates that inversions are allowed anywhere except between the fifth and seventh degrees of a major scale.

dwLevels

Bit field showing which levels are supported by this subchord. Each part in a style is assigned a level, and this chord is used only for parts whose levels are contained in this member.

bChordRoot

Root of the subchord, in which 0 is the bottom C in the range and 23 is the top B.

bScaleRoot

Root of the scale, in which 0 is the bottom C in the range and 23 is the top B.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

DMUS_SYNTHSTATS

The **DMUS_SYNTHSTATS** structure is used by the **IDirectMusicPort::GetRunningStats** method to return the current running status of a synthesizer.

```
typedef struct DMUS_SYNTHSTATS {
    DWORD dwSize;
    DWORD dwValidStats;
    DWORD dwVoices;
    DWORD dwTotalCPU;
    DWORD dwCPUPerVoice;
    DWORD dwLostNotes;
    DWORD dwFreeMemory;
    long lPeakVolume;
} DMUS_SYNTHSTATS, *LPDMUS_SYNTHSTATS;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized to **sizeof(DMUS_SYNTHSTATS)** before the structure is passed to a method.

dwValidStats

Flags that specify which fields in this structure have been filled in by the synthesizer. The following flags have been defined:

```
DMUS_SYNTHSTATS_VOICES
DMUS_SYNTHSTATS_TOTAL_CPU
DMUS_SYNTHSTATS_CPU_PER_VOICE
DMUS_SYNTHSTATS_FREE_MEMORY
DMUS_SYNTHSTATS_LOST_NOTES
DMUS_SYNTHSTATS_PEAK_VOLUME
```

dwVoices

Average number of voices playing.

dwTotalCPU

Total percentage of the CPU being consumed, multiplied by 100.

dwCPUPerVoice

Percentage of the CPU being consumed per voice, multiplied by 100.

dwLostNotes

Number of notes lost. Notes can be dropped because of voice-stealing or because too much of the CPU is being consumed.

dwFreeMemory

Amount of memory currently available to store DLS instruments. If the synthesizer is using system memory and the amount is therefore limited only by the available system memory, this value is set to **DMUS_SYNTHSTATS_SYSTEMMEMORY**.

lPeakVolume

Peak volume, measured in hundredths of decibels.

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

DMUS_TEMPO_PARAM

The **DMUS_TEMPO_PARAM** structure is used as the *pParam* parameter in calls to the various **GetParam** and **SetParam** methods when the track is a tempo track and *rguidType* is GUID_TempoParam.

```
typedef struct _DMUS_TEMPO_PARAM {
    MUSIC_TIME  mtTime;
    double      dblTempo;
} DMUS_TEMPO_PARAM;
```

Members**mtTime**

Time for which the tempo was retrieved. (This member is not used in **SetParam** methods, which use their *mtTime* parameter instead.)

dblTempo

The tempo, in the range from DMUS_TEMPO_MIN through DMUS_TEMPO_MAX.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

IDirectMusicPerformance::GetParam, **IDirectMusicPerformance::SetParam**,
IDirectMusicSegment::GetParam, **IDirectMusicSegment::SetParam**,
IDirectMusicTrack::GetParam, **IDirectMusicTrack::SetParam**

DMUS_TIMESIGNATURE

The **DMUS_TIMESIGNATURE** structure is passed to the **IDirectMusicStyle::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 **GetParam** methods when the *rguidType* parameter is **GUID_TimeSignature** and the track is a time signature or style track.

```
typedef struct _DMUS_TIMESIGNATURE {
    MUSIC_TIME mtTime;
    BYTE bBeatsPerMeasure;
    BYTE bBeat;
    WORD wGridsPerBeat;
} DMUS_TIMESIGNATURE;
```

Members

mtTime

Music time at which this time signature occurs.

bBeatsPerMeasure

Top of time signature.

bBeat

Bottom of time signature.

wGridsPerBeat

Grids (subdivisions) per beat. This value determines the timing resolution for certain music events—for example, segments cued with the **DMUS_SEGF_GRID** flag (see **DMUS_SEGF_FLAGS**).

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in **dmusici.h**.

See Also

IDirectMusicPerformance::GetParam, **IDirectMusicPerformance::SetParam**,
IDirectMusicSegment::GetParam, **IDirectMusicSegment::SetParam**,

**IDirectMusicTrack::GetParam, IDirectMusicTrack::SetParam,
DMUS_TIMESIG_PMSG**

DMUS_VERSION

The **DMUS_VERSION** structure contains version information for an object described in the **DMUS_OBJECTDESC** structure.

```
typedef struct _DMUS_VERSION {  
    DWORD dwVersionMS;  
    DWORD dwVersionLS;  
} DMUS_VERSION, FAR *LPDMUS_VERSION;
```

Members

dwVersionMS

Most significant **DWORD** of the version number.

dwVersionLS

Least significant **DWORD** of the version number.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

DMUS_WAVES_REVERB_PARAMS

The **DMUS_WAVES_REVERB_PARAMS** structure contains information about reverberation effects.

```
typedef struct _DMUS_WAVES_REVERB_PARAMS {  
    float flnGain;  
    float fReverbMix;  
    float fReverbTime;  
    float fHighFreqRTRatio;  
} DMUS_WAVES_REVERB_PARAMS;
```

Members

flnGain

Input gain, in decibels (to avoid output overflows). The default value is 0.

fReverbMix

Reverb mix, in decibels. A value of 0 means 100 percent wet reverb (no direct signal). Negative values gives less wet signal. The coefficients are calculated so that the overall output level stays approximately constant, regardless of the amount of reverb mix. The default value is -10.0 .

fReverbTime

Reverb decay time, in milliseconds. The default value is 1000.

fHighFreqRTRatio

Ratio of the high frequencies to the global reverb time. Unless very bright reverbs are wanted, this should be set to a value less than 1. For example, if **fReverbTime** is 1000 ms and **dHighFreqRTRatio** is 0.1, the decay time for high frequencies is 100 ms. The default value is 0.001.

Remarks

The TrueVerb reverberation technology from Waves is licensed to Microsoft as the SimpleVerb implementation for use in the Microsoft Software Synthesizer.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicc.h.

KSPROPERTY

The **KSPROPERTY** structure is passed to the **IKsControl::KsProperty** method to identify a property and operation.

KSPROPERTY is defined as a **KSIDENTIFIER** structure, which is declared as follows:

```
typedef struct {
    union {
        struct {
            GUID Set;
            ULONG Id;
            ULONG Flags;
        };
        LONGLONG Alignment;
    };
} KSIDENTIFIER, *PKSIDENTIFIER;
```

Members

Set

Identifier of the property set. The following property-set GUIDs are predefined by DirectMusic:

GUID_DMUS_PROP_DLS1

Item 0 is a Boolean indicating whether or not this port supports downloading DLS samples.

GUID_DMUS_PROP_Effects

Item 0 contains **DMUS_EFFECT_NONE** or one or more effects flags (see the **dwEffectFlags** member of **DMUS_PORTCAPS**). This property is used to set or retrieve the current state of the effects.

GUID_DMUS_PROP_GM_Hardware

Item 0 is a Boolean indicating whether or not this port supports GM in hardware.

GUID_DMUS_PROP_GS_Capable

Item 0 is a Boolean indicating whether or not this port supports the minimum requirements for Roland GS extensions.

GUID_DMUS_PROP_GS_Hardware

Item 0 is a Boolean indicating whether or not this port supports Roland GS extensions in hardware.

GUID_DMUS_PROP_LegacyCaps

Item 0 is the **MIDIINCAPS** or **MIDIOUTCAPS** structure that describes the underlying Windows multimedia device implementing this port. A **MIDIINCAPS** structure is returned if **dwClass** is **DMUS_PC_INPUTCLASS** in this port's capabilities structure. Otherwise, a **MIDIOUTCAPS** structure is returned.

GUID_DMUS_PROP_MemorySize

Item 0 is a Boolean indicating how many bytes of sample RAM are available on this device.

GUID_DMUS_PROP_SynthSink_DSOUND

Item 0 is a Boolean indicating whether or not this port supports DirectSound.

GUID_DMUS_PROP_SynthSink_WAVE

Item 0 is a Boolean indicating whether or not this port supports wave output using the **waveOut** functions.

GUID_DMUS_PROP_Volume

Item 1 (**DMUS_ITEM_Volume**) is a **LONG** in the range from **DMUS_VOLUME_MAX** through **DMUS_VOLUME_MIN**. This is the signed value, in hundredths of a decibel, which is added to the gain of all voices after all DLS articulation has been performed. By default, when a port is added to the performance, this property is set to the master volume. For master volume, see Setting and Retrieving Global Parameters.

GUID_DMUS_PROP_WavesReverb

Item 0 is a **DMUS_WAVES_REVERB_PARAMS** structure containing reverb parameters.

GUID_DMUS_PROP_WriteLatency

Item 0 is the write latency of the user-mode synthesizer (the **dwType** member of the **DMUS_PORTCAPS** structure is **DMUS_PORT_USER_MODE_SYNTH**) that streams its output to DirectSound. The write latency is the delay between when the synthesizer creates a buffer of sound and when it is heard. By adjusting this value, an application can fine-tune the synthesizer for minimum latency without sound breakup. On some computers, in particular ones without hardware support for DirectSound, the initial latency is much larger than on others, so the value should always be read first, and then adjusted with a relative value. The write latency can have different values for each port instance. This property must be set each time the port is activated.

GUID_DMUS_PROP_WritePeriod

Item 0 is the write period, in milliseconds, of the user-mode synthesizer (the **dwType** member of the **DMUS_PORTCAPS** structure is **DMUS_PORT_USER_MODE_SYNTH**) that streams its output to DirectSound. The write period controls how frequently the synthesizer sink allows the synthesizer to mix. By reducing this value, the application can reduce the overall latency of the synthesizer. However, values under 10 milliseconds increase the CPU load. The write period has the same value for all port instances that use the standard DirectSound sink. This property must be set each time the port is activated.

GUID_DMUS_PROP_XG_Capable

Item 0 is a Boolean indicating whether or not this port supports the minimum requirements for Yamaha XG extensions.

GUID_DMUS_PROP_XG_Hardware

Item 0 is a Boolean indicating whether or not this port supports Yamaha XG extensions in hardware.

Id

Item within the property set.

Flags

One of the following flags to specify the operation:

KSPROPERTY_TYPE_GET

To retrieve the given property item's value.

KSPROPERTY_TYPE_SET

To set the given property item's value.

KSPROPERTY_TYPE_BASIC SUPPORT

To determine the type of support available for the property set. The data returned by **IKsControl::KsProperty** in **pvPropertyData* is a **DWORD** containing one or both of **KSPROPERTY_TYPE_GET** and **KSPROPERTY_TYPE_SET**, indicating which operations are possible.

Alignment

Not used in DirectMusic.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmksctrl.h.

See Also

Port Property Sets

File Structures

This section contains reference information for data structures used in DirectMusic files. Most applications do not need to know about these structures because each standard DirectMusic object handles the loading of its own data through its **IPersistStream** interface. The structures are chiefly of interest for music-authoring applications that need to save data in a format compatible with DirectMusic.

The following structures are used in DirectMusic files:

- **DMUS_IO_BAND_ITEM_HEADER**
- **DMUS_IO_BAND_TRACK_HEADER**
- **DMUS_IO_CHORD**
- **DMUS_IO_CHORDENTRY**
- **DMUS_IO_CHORDMAP**
- **DMUS_IO_CHORDMAP_SIGNPOST**
- **DMUS_IO_CHORDMAP_SUBCHORD**
- **DMUS_IO_COMMAND**
- **DMUS_IO_CURVE_ITEM**
- **DMUS_IO_INSTRUMENT**
- **DMUS_IO_MOTIFSETTINGS**
- **DMUS_IO_MUTE**
- **DMUS_IO_NEXTCHORD**
- **DMUS_IO_PARTREF**
- **DMUS_IO_PATTERN**
- **DMUS_IO_REFERENCE**
- **DMUS_IO_SEGMENT_HEADER**
- **DMUS_IO_SEQ_ITEM**
- **DMUS_IO_SIGNPOST**
- **DMUS_IO_STYLE**
- **DMUS_IO_STYLECURVE**

- **DMUS_IO_STYLENOTE**
- **DMUS_IO_STYLEPART**
- **DMUS_IO_SUBCHORD**
- **DMUS_IO_SYSEX_ITEM**
- **DMUS_IO_TEMPO_ITEM**
- **DMUS_IO_TIMESIG**
- **DMUS_IO_TIMESIGNATURE_ITEM**
- **DMUS_IO_TOOL_HEADER**
- **DMUS_IO_TRACK_HEADER**
- **DMUS_IO_VERSION**

See Also

DirectMusic File Format

DMUS_IO_BAND_ITEM_HEADER

The **DMUS_IO_BAND_ITEM_HEADER** structure contains information about a band change. It is used in the Band Track Form.

```
typedef struct _DMUS_IO_BAND_ITEM_HEADER {  
    MUSIC_TIME IBandTime;  
} DMUS_IO_BAND_ITEM_HEADER;
```

IBandTime

Time of the band change.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

DMUS_IO_BAND_TRACK_HEADER

The **DMUS_IO_BAND_TRACK_HEADER** structure contains information about the default behavior of a band track. It is used in the Band Track Form.

```
typedef struct _DMUS_IO_BAND_TRACK_HEADER {  
    BOOL bAutoDownload;  
} DMUS_IO_BAND_TRACK_HEADER;
```

bAutoDownload

Flag for automatic downloading of instruments when a segment is played.

Remarks

For more information on automatic downloading, see Using Bands.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmuscf.h.

DMUS_IO_CHORD

The **DMUS_IO_CHORD** structure contains information about a chord change. It is used in the Chord Track List.

```
typedef struct _DMUS_IO_CHORD {  
    WCHAR    wszName[16];  
    MUSIC_TIME mtTime;  
    WORD     wMeasure;  
    BYTE     bBeat;  
} DMUS_IO_CHORD;
```

wszName

Name of the chord.

mtTime

Time of the chord.

wMeasure

Measure that the chord falls on.

bBeat

Beat that the chord falls on.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmuscf.h.

DMUS_IO_CHORDENTRY

The **DMUS_IO_CHORDENTRY** structure contains information about a chord entry. It is used in the Chord-map Form.

```
typedef struct _DMUS_IO_CHORDENTRY {
    DWORD  dwFlags;
    WORD   wConnectionID;
} DMUS_IO_CHORDENTRY;
```

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

DMUS_IO_CHORDMAP

The **DMUS_IO_CHORDMAP** structure contains information about a chord map. It is used in the Chord-map Form.

```
typedef struct _DMUS_IO_CHORDMAP {
    WCHAR  wszLoadName[20];
    DWORD  dwScalePattern;
    DWORD  dwFlags;
} DMUS_IO_CHORDMAP;
```

wszLoadName

Name of the chord map, used in the object description when the chord map is loaded.

dwScalePattern

Scale associated with the chord map. 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

Reserved for future use.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

DMUS_IO_CHORDMAP_SIGNPOST

The **DMUS_IO_CHORDMAP_SIGNPOST** structure contains information about a signpost chord in a chord map. It is used in the Chord-map Form.

```
typedef struct _DMUS_IO_CHORDMAP_SIGNPOST {
    DWORD   dwChords;
    DWORD   dwFlags;
} DMUS_IO_CHORDMAP_SIGNPOST;
```

dwChords

Types of signpost supported by this chord. The values are used to match against the same values as they appear in templates. Composing from template consists of (among other things) looking for these values in the template and finding actual chords in the chord map that match these values. The following flags are defined:

```
DMUS_SIGNPOSTF_A
DMUS_SIGNPOSTF_B
DMUS_SIGNPOSTF_C
DMUS_SIGNPOSTF_D
DMUS_SIGNPOSTF_E
DMUS_SIGNPOSTF_F
DMUS_SIGNPOSTF_LETTER
DMUS_SIGNPOSTF_1
DMUS_SIGNPOSTF_2
DMUS_SIGNPOSTF_3
DMUS_SIGNPOSTF_4
DMUS_SIGNPOSTF_5
DMUS_SIGNPOSTF_6
DMUS_SIGNPOSTF_7
DMUS_SIGNPOSTF_ROOT
DMUS_SIGNPOSTF_CADENCE
```

dwFlags

Flags defining whether this chord is to be preceded by cadence chords. Signpost chords can have up to two cadence chords. This value can be SPOST_CADENCE1 (first cadence), SPOST_CADENCE2 (second cadence), or a combination of these two flags.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusief.h.

See Also

DMUS_IO_SIGNPOST

DMUS_IO_CHORDMAP_SUBCHORD

The **DMUS_IO_CHORDMAP_SUBCHORD** structure contains information about a subchord. It is used in the Chord-map Form.

```
typedef struct _DMUS_IO_CHORDMAP_SUBCHORD {
    DWORD dwChordPattern;
    DWORD dwScalePattern;
    DWORD dwInvertPattern;
    BYTE bChordRoot;
    BYTE bScaleRoot;
    WORD wCFlags;
    DWORD dwLevels;
} DMUS_IO_CHORDMAP_SUBCHORD;
```

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 bottom C in the range and 23 is the top B.

bScaleRoot

Root of the scale, where 0 is the bottom C in the range and 23 is the top B.

wCFlags

Reserved for future use.

dwLevels

Bit field showing which levels are supported by this subchord. Each part in a style is assigned a level, and this chord is used only for parts whose levels are contained in this member.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

DMUS_SUBCHORD

DMUS_IO_COMMAND

The **DMUS_IO_COMMAND** structure contains information about a command event. It is used in the Command Track Chunk.

```
typedef struct _DMUS_IO_COMMAND {
    MUSIC_TIME  mtTime;
    WORD        wMeasure;
    BYTE        bBeat;
    BYTE        bCommand;
    BYTE        bGrooveLevel;
    BYTE        bGrooveRange;
} DMUS_IO_COMMAND;
```

mtTime

Time of the command.

wMeasure

Measure that the command falls on.

bBeat

Beat that the command falls on.

bCommand

Command type. See **DMUS_COMMANDT_TYPES**.

bGrooveLevel

Groove level, or 0 if the command is not a groove command.

bGrooveRange

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.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

DMUS_IO_CURVE_ITEM

The **DMUS_IO_CURVE_ITEM** structure contains information about a curve event in a track. It is used in the Sequence Track List.

```
typedef struct _DMUS_IO_CURVE_ITEM {
    MUSIC_TIME  mtStart;
    MUSIC_TIME  mtDuration;
    MUSIC_TIME  mtResetDuration;
    DWORD       dwPChannel;
    short        nOffset;
    short        nStartValue;
    short        nEndValue;
    short        nResetValue;
    BYTE         bType;
    BYTE         bCurveShape;
    BYTE         bCCData;
    BYTE         bFlags;
} DMUS_IO_CURVE_ITEM;
```

mtStart

Start time of the curve.

mtDuration

Duration of the curve.

mtResetDuration

Time after the curve is finished until the reset value is set.

dwPChannel

Performance channel for the event.

nOffset

Offset from the grid boundary at which the curve occurs, in music time. MIDI curves are associated with the closest grid when loaded, so this value can be positive or negative.

nStartValue

Start value.

nEndValue

End value.

nResetValue

Reset value, set after **mtResetDuration** or upon a flush or invalidation.

bType

Type of curve. The following types are defined:

DMUS_CURVET_CCCURVE

Continuous controller curve (MIDI Control Change channel voice message; status byte &HB*n*, where *n* is the channel number).

DMUS_CURVET_MATCURVE

Monophonic aftertouch curve (MIDI Channel Pressure channel voice message; status byte &HD*n*).

DMUS_CURVET_PATCURVE

Polyphonic aftertouch curve (MIDI Poly Key Pressure channel voice message, status byte &HD*n*).

DMUS_CURVET_PBCURVE

Pitch-bend curve (MIDI Pitch Bend channel voice message; status byte &HE*n*).

bCurveShape

Shape of curve. The following shapes are defined:

DMUS_CURVES_EXP

Exponential curve shape.

DMUS_CURVES_INSTANT

Instant curve shape (beginning and end of curve happen at essentially the same time).

DMUS_CURVES_LINEAR

Linear curve shape.

DMUS_CURVES_LOG

Logarithmic curve shape.

DMUS_CURVES_SINE

Sine curve shape.

bCCData

CC number if this is a control change type.

bFlags

Set to DMUS_CURVE_RESET if the **nResetValue** must be set when the time is reached or an invalidation occurs because of a transition. If 0, the curve stays permanently at the new value. All other bits are reserved.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

DMUS_IO_SEQ_ITEM

DMUS_IO_INSTRUMENT

The **DMUS_IO_INSTRUMENT** structure contains information about an instrument. It is used in the Band Form.

```
typedef struct _DMUS_IO_INSTRUMENT {
```

```

DWORD dwPatch;
DWORD dwAssignPatch;
DWORD dwNoteRanges[4];
DWORD dwPChannel;
DWORD dwFlags;
BYTE bPan;
BYTE bVolume;
short nTranspose;
DWORD dwChannelPriority;
} DMUS_IO_INSTRUMENT;

```

dwPatch

MSB, LSB, and program change to define instrument.

dwAssignPatch

MSB, LSB, and program change to assign to instrument when downloading.

dwNoteRanges

128 bits; one for each MIDI note that the instrument must be able to play.

dwPChannel

Performance channel that the instrument plays on.

dwFlags

Control flags. The following values are defined:

DMUS_IO_INST_ASSIGN_PATCH

The **dwAssignPatch** member is valid.

DMUS_IO_INST_BANKSELECT

The **dwPatch** member contains a valid bank select, both MSB and LSB.

DMUS_IO_INST_CHANNEL_PRIORITY

The **dwChannelPriority** member is valid.

DMUS_IO_INST_GM

Instrument is from the General MIDI collection.

DMUS_IO_INST_GS

Instrument is from the Roland GS collection.

DMUS_IO_INST_NOTERANGES

The **dwNoteRanges** member is valid.

DMUS_IO_INST_PAN

The **bPan** member is valid.

DMUS_IO_INST_PATCH

The **dwPatch** member is valid.

DMUS_IO_INST_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. If a MIDI file with the XG or GS reset system-exclusive message is parsed, the bank-select message is sent, whether

or not GUID_StandardMIDIFile was commanded on the band. In other words, GUID_StandardMIDIFile is effective only for pure GM files.

DMUS_IO_INST_VOLUME

The **bVolume** member is valid.

DMUS_IO_INST_XG

Instrument is from the Yamaha XG collection.

bPan

Pan for the instrument.

bVolume

Volume for the instrument.

nTranspose

Number of semitones to transpose notes.

dwChannelPriority

Channel priority. For a list of defined values, see

IDirectMusicPort::GetChannelPriority.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

DMUS_IO_MOTIFSETTINGS

The **DMUS_IO_MOTIFSETTINGS** structure contains information about a motif. It is used in the Style Form.

```
typedef struct _DMUS_IO_MOTIFSETTINGS {
    DWORD      dwRepeats;
    MUSIC_TIME mtPlayStart;
    MUSIC_TIME mtLoopStart;
    MUSIC_TIME mtLoopEnd;
    DWORD      dwResolution;
} DMUS_IO_MOTIFSETTINGS;
```

dwRepeats

Number of repetitions.

mtPlayStart

Start of playback, normally 0.

mtLoopStart

Start of looping portion, normally 0.

mtLoopEnd

End of looping portion.

dwResolution

Default resolution. See **DMUS_TIME_RESOLVE_FLAGS**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

IDirectMusicSegment::SetLoopPoints

DMUS_IO_MUTE

The **DMUS_IO_MUTE** structure contains information about a mute event on a channel. It is used in the Mute Track Chunk.

```
typedef struct _DMUS_IO_MUTE {  
    MUSIC_TIME  mtTime;  
    DWORD       dwPChannel;  
    DWORD       dwPChannelMap;  
} DMUS_IO_MUTE;
```

mtTime

Time of the event.

dwPChannel

Performance channel to mute or remap.

dwPChannelMap

Channel to which **dwPChannel** is being mapped, or 0xFFFFFFFF if **dwPChannel** is to be muted.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

DMUS_MUTE_PARAM

DMUS_IO_NEXTCHORD

The **DMUS_IO_NEXTCHORD** structure contains information about the next chord in a chord graph. It is used in the Chord-map Form.

```
typedef struct _DMUS_IO_NEXTCHORD {
    DWORD   dwFlags;
    WORD    nWeight;
    WORD    wMinBeats;
    WORD    wMaxBeats;
    WORD    wConnectionID;
} DMUS_IO_NEXTCHORD;
```

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

DMUS_IO_PARTREF

The **DMUS_IO_PARTREF** structure contains information about a part reference. It is used in the Style Form.

```
typedef struct _DMUS_IO_PARTREF {
    GUID   guidPartID;
    WORD   wLogicalPartID;
    BYTE   bVariationLockID;
    BYTE   bSubChordLevel;
    BYTE   bPriority;
```



```

    BYTE bRandomVariation;
} DMUS_IO_PARTREF;

```

guidPartID

Identifier of the part.

wLogicalPartID

Identifier corresponding to a particular MIDI channel on a port.

bVariationLockID

Parts with the same value in this member always play the same variation.

bSubChordLevel

Subchord level that this part wants. See Remarks.

bPriority

Priority of the part. For information on priorities, see Channels.

bRandomVariation

When set, matching variations play in random order. When clear, matching variations play sequentially.

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusief.h.

DMUS_IO_PATTERN

The **DMUS_IO_PATTERN** structure contains information about a pattern. It is used in the Style Form.

```

typedef struct _DMUS_IO_PATTERN {
    DMUS_IO_TIMESIG timeSig;
    BYTE            bGrooveBottom;
    BYTE            bGrooveTop;
    WORD            wEmbellishment;
    WORD            wNbrMeasures;
} DMUS_IO_PATTERN;

```

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. See **DMUS_COMMANDT_TYPES**.

wNbrMeasures

Length of the pattern in measures.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusief.h.

DMUS_IO_REFERENCE

The **DMUS_IO_REFERENCE** structure contains information about a reference to another object that might be stored in another file. It is used in the reference list chunk. See Common Chunks.

```
typedef struct _DMUS_IO_REFERENCE {
    GUID    guidClassID;
    DWORD   dwValidData;
} DMUS_IO_REFERENCE;
```

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusief.h.

DMUS_IO_SEGMENT_HEADER

The **DMUS_IO_SEGMENT_HEADER** structure contains information about a segment. It is used in the Segment Form.

```
typedef struct _DMUS_IO_SEGMENT_HEADER {
    DWORD      dwRepeats;
    MUSIC_TIME mtLength;
    MUSIC_TIME mtPlayStart;
    MUSIC_TIME mtLoopStart;
    MUSIC_TIME mtLoopEnd;
    DWORD      dwResolution;
} DMUS_IO_SEGMENT_HEADER;
```

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.

dwResolution

Default resolution. See **DMUS_TIME_RESOLVE_FLAGS**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

DMUS_IO_MOTIFSETTINGS, **IDirectMusicSegment::SetLoopPoints**.

DMUS_IO_SEQ_ITEM

The **DMUS_IO_SEQ_ITEM** structure contains information about an item of data in a sequence track. It is used in the Sequence Track List.

```
typedef struct _DMUS_IO_SEQ_ITEM {
    MUSIC_TIME mtTime;
```

```

    MUSIC_TIME  mtDuration;
    DWORD       dwPChannel;
    short       nOffset;
    BYTE        bStatus;
    BYTE        bByte1;
    BYTE        bByte2;
} DMUS_IO_SEQ_ITEM;

```

mtTime

Time of the event.

mtDuration

Duration for which the event is valid.

dwPChannel

Performance channel for the event.

nOffset

Offset from the grid boundary at which the note is played, in music time. MIDI notes are associated with the closest grid when loaded, so this value can be positive or negative.

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

DMUS_IO_CURVE_ITEM, MIDI Messages

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 chord map. It is used in the Signpost Track Chunk.

```

typedef struct _DMUS_IO_SIGNPOST {
    MUSIC_TIME  mtTime;

```

```

    DWORD    dwChords;
    WORD     wMeasure;
} DMUS_IO_SIGNPOST;

```

mtTime

Time of the signpost.

dwChords

Types of signpost chords allowed to be associated with this signpost. The values are used to match against the same values as they appear in templates.

Composing from a template consists of (among other things) looking for these values in the template and finding actual chords in the chord map that match these values. The following flags are defined:

```

DMUS_SIGNPOSTF_A
DMUS_SIGNPOSTF_B
DMUS_SIGNPOSTF_C
DMUS_SIGNPOSTF_D
DMUS_SIGNPOSTF_E
DMUS_SIGNPOSTF_F
DMUS_SIGNPOSTF_LETTER
DMUS_SIGNPOSTF_1
DMUS_SIGNPOSTF_2
DMUS_SIGNPOSTF_3
DMUS_SIGNPOSTF_4
DMUS_SIGNPOSTF_5
DMUS_SIGNPOSTF_6
DMUS_SIGNPOSTF_7
DMUS_SIGNPOSTF_ROOT
DMUS_SIGNPOSTF_CADENCE

```

wMeasure

Measure on which the signpost falls.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

DMUS_IO_CHORDMAP_SIGNPOST

DMUS_IO_STYLE

The **DMUS_IO_STYLE** structure contains information about the time signature and tempo of a style. It is used in the Style Form.

```
typedef struct _DMUS_IO_STYLE {
    DMUS_IO_TIMESIG timeSig;
    double          dblTempo;
} DMUS_IO_STYLE;
```

timeSig

DMUS_IO_TIMESIG structure containing the default time signature for the style.

dblTempo

Tempo of the style.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

DMUS_IO_STYLECURVE

The **DMUS_IO_STYLECURVE** structure contains information about a curve in a style. It is used in the Style Form.

```
typedef struct _DMUS_IO_STYLECURVE {
    MUSIC_TIME mtGridStart;
    DWORD      dwVariation;
    MUSIC_TIME mtDuration;
    MUSIC_TIME mtResetDuration;
    short      nTimeOffset;
    short      nStartValue;
    short      nEndValue;
    short      nResetValue;
    BYTE       bEventType;
    BYTE       bCurveShape;
    BYTE       bCCData;
    BYTE       bFlags;
} DMUS_IO_STYLECURVE;
```

mtGridStart

Start time of the grid in 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

How long after the curve is finished until the reset value is set.

nTimeOffset

Offset from **mtGridStart** at which the curve occurs.

nStartValue

Start value.

nEndValue

End value.

nResetValue

Reset value, set after **mtResetDuration** or upon a flush or invalidation.

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 the time is reached or an invalidation occurs because of a transition. If 0, the curve stays permanently at the new value. All other bits are reserved.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in **dmusicf.h**.

See Also

DMUS_CURVE_PMSG, **DMUS_IO_CURVE_ITEM**

DMUS_IO_STYLENOTE

The **DMUS_IO_STYLENOTE** structure contains information about a note in a style. It is used in the Style Form.

```
typedef struct _DMUS_IO_STYLENOTE {
    MUSIC_TIME  mtGridStart;
    DWORD       dwVariation;
    MUSIC_TIME  mtDuration;
    short       nTimeOffset;
    WORD        wMusicValue;
    BYTE        bVelocity;
    BYTE        bTimeRange;
    BYTE        bDurRange;
```

```

    BYTE    bVelRange;
    BYTE    bInversionID;
    BYTE    bPlayModeFlags;
} DMUS_IO_STYLENOTE;

```

mtGridStart

When the note occurs.

dwVariation

Variations that this note belongs to. Each bit corresponds to one of 32 variations.

mtDuration

Duration of the note.

nTimeOffset

Offset from **mtGridStart**.

wMusicValue

Position in the scale.

bVelocity

Note velocity.

bTimeRange

Range within which to randomize start time.

bDurRange

Range within which to randomize duration.

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**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

DMUS_IO_STYLEPART

The **DMUS_IO_STYLEPART** structure contains information about a musical part. It is used in the Style Form.

```

typedef struct _DMUS_IO_STYLEPART {
    DMUS_IO_TIMESIG timeSig;
    DWORD           dwVariationChoices[32];
    GUID            guidPartID;
}

```

```

WORD        wNbrMeasures;
BYTE        bPlayModeFlags;
BYTE        blInvertUpper;
BYTE        blInvertLower;
} DMUS_IO_STYLEPART;

```

timeSig

DMUS_IO_TIMESIG structure containing a time signature to override the style's default time signature.

dwVariationChoices

Each **WORD** corresponds to one of 32 possible variations. The flags set in each **WORD** indicate which types of chord are supported by that variation (see Remarks). The following flags are defined:

DMUS_VARIATIONF_MAJOR

Seven positions in the scale for major chords.

DMUS_VARIATIONF_MINOR

Seven positions in the scale for minor chords.

DMUS_VARIATIONF_OTHER

Seven positions in the scale for other chords.

DMUS_VARIATIONF_ROOT_SCALE

Handles chord roots in the scale.

DMUS_VARIATIONF_ROOT_FLAT

Handles flat chord roots (based on scale notes).

DMUS_VARIATIONF_ROOT_SHARP

Handles sharp chord roots (based on scale notes).

DMUS_VARIATIONF_TYPE_TRIAD

Handles simple chords for triads.

DMUS_VARIATIONF_TYPE_6AND7

Handles simple chords for 6 and 7.

DMUS_VARIATIONF_TYPE_COMPLEX

Handles complex chords.

DMUS_VARIATIONF_DEST_TO1

Handles transitions to the 1 chord.

DMUS_VARIATIONF_DEST_TO5

Handles transitions to the 5 chord.

DMUS_VARIATIONF_MODES

DMUS_VARIATIONF_IMA25_MODE

DMUS_VARIATIONF_DMUS_MODE

One of these flags is set to indicate the mode. For DirectMusic, this value should always be **DMUS_VARIATIONF_DMUS_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.

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, but 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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

DMUS_IO_SUBCHORD

The **DMUS_IO_SUBCHORD** structure contains information about a subchord. It is used in the Chord Track List.

```
typedef struct _DMUS_IO_SUBCHORD {
    DWORD   dwChordPattern;
    DWORD   dwScalePattern;
    DWORD   dwInversionPoints;
    DWORD   dwLevels;
    BYTE    bChordRoot;
    BYTE    bScaleRoot;
} DMUS_IO_SUBCHORD;
```

dwChordPattern

Notes in the subchord. Each of the lower 24 bits represents a semitone, starting with the root at the least significant bit, and the bit is set if the note is in the chord.

dwScalePattern

Notes in the scale. Each of the lower 24 bits represents a semitone, starting with the root at the least significant bit, and the bit is set if the note is in the scale.

dwInversionPoints

Points in the scale at which inversions can occur. Bits that are off signify that the notes in the interval cannot be inverted. Thus, the pattern 100001111111 indicates that inversions are allowed anywhere except between the fifth and seventh degrees of a major scale.

dwLevels

Which levels are supported by this subchord. Certain instruments can be assigned different levels (such as to play only the lower subchords of a chord), and this value is a way of mapping subchords to those levels.

bChordRoot

Root of the subchord, where 0 is the bottom C in the range and 23 is the top B.

bScaleRoot

Root of the scale, where 0 is the bottom C in the range and 23 is the top B.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also**DMUS_SUBCHORD**

DMUS_IO_SYSEX_ITEM

The **DMUS_IO_SYSEX_ITEM** structure contains information about a system-exclusive MIDI message. It is used in the Sysex Track Chunk.

```
typedef struct _DMUS_IO_SYSEX_ITEM {
    MUSIC_TIME  mtTime;
    DWORD       dwPChannel;
    DWORD       dwSysExLength;
} DMUS_IO_SYSEX_ITEM;
```

mtTime

Time of the message.

dwPChannel

Performance channel of the event.

dwSysExLength

Length of the data, in bytes.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

MIDI System Messages

DMUS_IO_TEMPO_ITEM

The **DMUS_IO_TEMPO_ITEM** structure contains information about a tempo change in a track. It is used in the Tempo Track Chunk.

```
typedef struct _DMUS_IO_TEMPO_ITEM {  
    MUSIC_TIME  mtTime;  
    double      dblTempo;  
} DMUS_IO_TEMPO_ITEM;
```

mtTime

Time of the tempo change.

dblTempo

Tempo, in beats per minute.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

DMUS_IO_TIMESIG

The **DMUS_IO_TIMESIG** structure contains information about the time signature of a segment. It is used in the **DMUS_IO_STYLE**, **DMUS_IO_VERSION**, and **DMUS_IO_PATTERN** structures.

```
typedef struct _DMUS_IO_TIMESIG {  
    BYTE  bBeatsPerMeasure;
```

```
    BYTE bBeat;  
    WORD wGridsPerBeat;  
} DMUS_IO_TIMESIG;
```

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmuscf.h.

See Also

DMUS_IO_TIMESIGNATURE_ITEM

DMUS_IO_TIMESIGNATURE_ITEM

The **DMUS_IO_TIMESIGNATURE_ITEM** structure contains information about a time signature change. It is used in the Time Signature Track Chunk.

```
typedef struct _DMUS_IO_TIMESIGNATURE_ITEM {  
    MUSIC_TIME mtTime;  
    BYTE      bBeatsPerMeasure;  
    BYTE      bBeat;  
    WORD      wGridsPerBeat;  
} DMUS_IO_TIMESIGNATURE_ITEM;
```

mtTime

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

DMUS_IO_TIMESIG, DMUS_TIMESIG_PMSG

DMUS_IO_TOOL_HEADER

The **DMUS_IO_TOOL_HEADER** structure contains information about a tool. It is used in the Tool Form.

```
typedef struct _DMUS_IO_TOOL_HEADER {
    GUID  guidClassID;
    long  lIndex;
    DWORD cPChannels;
    FOURCC ckid;
    FOURCC fccType;
    DWORD dwPChannels[1];
} DMUS_IO_TOOL_HEADER;
```

guidClassID

Class identifier of the tool.

lIndex

Position in the graph.

cPChannels

Number of items in the **dwPChannels** array.

ckid

Identifier of tool's data chunk. If this value is 0, it is assumed that the chunk is of type LIST, so **fccType** is valid and must be nonzero.

fccType

List type. If this value is 0, **ckid** is valid and must be nonzero.

dwPChannels

Array of performance channels for which the tool is valid.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

IDirectMusicGraph::InsertTool

DMUS_IO_TRACK_HEADER

The **DMUS_IO_TRACK_HEADER** structure contains information about a track. It is used in the Track Form.

```
typedef struct _DMUS_IO_TRACK_HEADER {
    GUID    guidClassID;
    DWORD   dwPosition;
    DWORD   dwGroup;
    FOURCC  ckid;
    FOURCC  fccType;
} DMUS_IO_TRACK_HEADER;
```

guidClassID

Class identifier of the track.

dwPosition

Position in the track list.

dwGroup

Group bits for the track.

ckid

Identifier of the track's data chunk. If this value is 0, it is assumed that the chunk is of type LIST, so **fccType** is valid and must be nonzero.

fccType

List type. If this value is 0, **ckid** is valid and must be nonzero.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

See Also

IDirectMusicSegment::GetTrackGroup, **IDirectMusicSegment::InsertTrack**, Track Form

DMUS_IO_VERSION

The **DMUS_IO_VERSION** structure contains the version number of the data. It is used in the version subchunk of various chunks. See Common Chunks.

```
typedef struct _DMUS_IO_VERSION {  
    DWORD dwVersionMS;  
    DWORD dwVersionLS;  
} DMUS_IO_VERSION;
```

dwVersionMS

High-order 32 bits of the version number.

dwVersionLS

Low-order 32 bits of the version number.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusicf.h.

DLS Structures

This section contains reference information for structures used with downloadable sounds. Most applications do not need to use these structures because DirectMusic handles the details of loading DLS collections and downloading instruments to the synthesizer. They are of interest chiefly for applications that edit DLS.

For an overview of using DLS data, see Low-Level DLS.

For more information on DLS data formats, see the specification from the MIDI Manufacturers Association.

The following structures are included in this section:

- **DMUS_ARTICPARAMS**
- **DMUS ARTICULATION**
- **DMUS ARTICULATION2**
- **DMUS_COPYRIGHT**
- **DMUS_DOWNLOADINFO**
- **DMUS_EXTENSIONCHUNK**
- **DMUS_INSTRUMENT**
- **DMUS_LFOPARAMS**
- **DMUS_MSCPARAMS**
- **DMUS_OFFSETTABLE**
- **DMUS_PEGPARAMS**
- **DMUS_REGION**
- **DMUS_VEGPARAMS**

- **DMUS_WAVE**
- **DMUS_WAVEDATA**

DMUS__ARTICPARAMS

The **DMUS__ARTICPARAMS** structure describes parameters for a DLS articulation chunk. All parameters for articulation are stored in one chunk, which is composed of a series of structures that define each functional area of the articulation. If an instrument or region uses articulation, it references this chunk by index from the **DMUS__ARTICULATION** chunk.

```
typedef struct {  
    DMUS__LFOPARAMS LFO;  
    DMUS__VEGPARAMS VoIEG;  
    DMUS__PEGPARAMS PitchEG;  
    DMUS__MSCPARAMS Misc;  
} DMUS__ARTICPARAMS;
```

LFO

DMUS__LFOPARAMS structure containing parameters for a low-frequency oscillator.

VoIEG

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.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

DMUS__ARTICULATION

The **DMUS__ARTICULATION** structure describes a DLS instrument articulation chunk. 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**, whereas all additional articulation chunks are referenced by the list that starts with **ulFirstExtCkIdx**.

```
typedef struct {  
    ULONG ulArt1Idx;  
    ULONG ulFirstExtCkIdx;  
} DMUS_ARTICULATION;
```

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.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

See Also

DMUS_ARTICULATION2

DMUS_ARTICULATION2

The **DMUS_ARTICULATION2** structure describes a DLS instrument-articulation chunk. It is used when the format is defined as **DMUS_DOWNLOADINFO_INSTRUMENT2** (declared in the Dmdls.h header file). The DLS level 1 chunk is referenced by **ulArt1Idx**, and all additional articulation chunks are referenced by the list that starts with **ulFirstExtCkIdx**. DLS level 2 articulation chunks also use **ulNextArtIdx**.

```
typedef struct {  
    ULONG ulArt1Idx;  
    ULONG ulFirstExtCkIdx;  
    ULONG ulNextArtIdx;  
  
} DMUS_ARTICULATION;
```

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

Additional articulation chunks to better support DLS level 2 articulations

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

See Also

DMUS_ARTICULATION

DMUS_COPYRIGHT

The **DMUS_COPYRIGHT** structure describes an optional copyright chunk in DLS data.

```
typedef struct {  
    ULONG cbSize;  
    BYTE  byCopyright[ ];  
} DMUS_COPYRIGHT;
```

cbSize

Size of data.

byCopyright[]

Copyright data.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

DMUS_DOWNLOADINFO

The **DMUS_DOWNLOADINFO** structure is used as a header for DLS data to be downloaded to a port. It defines the size and functionality of the download and is always followed by a **DMUS_OFFSETTABLE** chunk.

```
typedef struct _DMUS_DOWNLOADINFO {
```

```

    DWORD dwDLType;
    DWORD dwDLId;
    DWORD dwNumOffsetTableEntries;
    DWORD cbSize;
} DMUS_DOWNLOADINFO;

```

dwDLType

Type of data being downloaded. The following types are defined:

DMUS_DOWNLOADINFO_INSTRUMENT

Instrument definition, starting with the **DMUS_INSTRUMENT** structure

DMUS_DOWNLOADINFO_WAVE

PCM wave data, starting with the **DMUS_WAVE** structure.

dwDLId

Unique 32-bit identifier for the object. See Remarks.

dwNumOffsetTableEntries

Number of entries in the **DMUS_OFFSETTABLE** structure that follows.

cbSize

Total size of **DMUS_DOWNLOADINFO**, **DMUS_OFFSETTABLE**, and the actual data chunk

Remarks

The identifier in **dwDLId** is used to connect objects and is obtained by using the **IDirectMusicPortDownload::GetDLId** method. Primarily it connects the regions in an instrument to wave chunks. For example, if a wave download is given a **dwDLId** of 3, an instrument chunk downloads with the value 3 placed in the **WaveLink.ulTableIndex** member of one of its **DMUS_REGION** structures. This indicates that the region is connected to the wave chunk.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

See Also

Low-Level DLS

DMUS_EXTENSIONCHUNK

The **DMUS_EXTENSIONCHUNK** structure describes a DLS extension chunk. All extensions to the DLS file format that are unknown to DirectMusic are downloaded in this variable-size chunk.

```
typedef struct {
    ULONG cbSize;
    ULONG ulNextExtCkIdx;
    FOURCC ExtCkID;
    BYTE byExtCk[ ];
} DMUS_EXTENSIONCHUNK;
```

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

DMUS_INSTRUMENT

The **DMUS_INSTRUMENT** structure contains an instrument definition in a DLS download chunk.

```
typedef struct {
    ULONG ulPatch;
    ULONG ulFirstRegionIdx;
    ULONG ulGlobalArtIdx;
    ULONG ulFirstExtCkIdx;
    ULONG ulCopyrightIdx;
    ULONG ulFlags;
} DMUS_INSTRUMENT;
```

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**) 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 associated with the instrument.

ulCopyrightIdx

Index, in the **DMUS_OFFSETTABLE** structure, of an optional copyright chunk (see **DMUS_COPYRIGHT**). If 0, no copyright information is associated with the instrument.

ulFlags

Additional flags for the instrument. The following flag is defined:

DMUS_INSTRUMENT_GM_INSTRUMENT

The instrument is a standard General MIDI instrument. In the case of patch overlap, GM instruments always have lower priority than other DLS instruments. For example, if a GM instrument is downloaded with patch 0 and a non-GM instrument is also downloaded at patch 0, the non-GM instrument is always selected for playback.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

DMUS_LFOPARAMS

The **DMUS_LFOPARAMS** structure defines the low-frequency oscillator for a DLS articulation chunk. It is used in the **DMUS_ARTICPARAMS** structure.

```
typedef struct {
    PCENT pcFrequency;
    TCENT tcDelay;
    GCENT gcVolumeScale;
    PCENT pcPitchScale;
    GCENT gcMWTtoVolume;
    PCENT pcMWTtoPitch;
} DMUS_LFOPARAMS;
```

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

See Also

DMUS_ARTICPARAMS

DMUS_MSCPARAMS

The **DMUS_MSCPARAMS** structure defines the pan for a DLS articulation chunk. It is used in the **DMUS_ARTICPARAMS** structure.

```
typedef struct {
    PERCENT ptDefaultPan;
} DMUS_MSCPARAMS;
```

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

See Also

DMUS_ARTICPARAMS

DMUS_OFFSETTABLE

The **DMUS_OFFSETTABLE** structure is used in the header of DLS instrument data being downloaded to a port.

```
typedef struct _DMUS_OFFSETTABLE {
    ULONG ulOffsetTable[DMUS_DEFAULT_SIZE_OFFSETTABLE];
} DMUS_OFFSETTABLE;
```

ulOffsetTable

Array of byte offsets into the data.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

See Also

Low-Level DLS

DMUS_PEGPARAMS

The **DMUS_PEGPARAMS** structure defines the pitch envelope for a DLS articulation chunk. It is used in the **DMUS_ARTICPARAMS** structure.

```
typedef struct {
    TCENT tcAttack;
    TCENT tcDecay;
    PERCENT ptSustain;
    TCENT tcRelease;
    TCENT tcVel2Attack;
    TCENT tcKey2Decay;
    PCENT pcRange;
```

```
} DMUS_PEGPARAMS;
```

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

See Also

DMUS_ARTICPARAMS

DMUS_REGION

The **DMUS_REGION** structure defines a region for a DLS download. One or more regions can be embedded in an instrument buffer and referenced by the instrument header chunk, **DMUS_INSTRUMENT**.

```
typedef struct {
    RGNRANGE RangeKey;
    RGNRANGE RangeVelocity;
    USHORT   fusOptions;
    USHORT   usKeyGroup;
```

```

    ULONG    ulRegionArtIdx;
    ULONG    ulNextRegionIdx;
    ULONG    ulFirstExtCkIdx;
    WAVELINK WaveLink;
    WSMPL    WSMP;
    WLOOP    WLOOP[1];
} DMUS_REGION;

```

RangeKey

Key range for this region.

RangeVelocity

Velocity range for this region.

fusOptions

Options for the synthesis of this region. The following flag is defined:

F_RGN_OPTION_SELFNONEXCLUSIVE

If a second note-on for the same note is received by the synthesis engine, the second note is played, as well as the first. This option is off by default so that the synthesis engine forces a note-off of the first note.

usKeyGroup

Key group for a drum instrument. Key group values allow multiple regions within a drum instrument to belong to the same group. If a synthesis engine is instructed to play a note with a key group setting and any other notes are currently playing with this same key group, the synthesis engine turns off all notes with the same key group value as soon as possible. Currently, key groups from 1 through 15 are legal, and 0 indicates no key group.

ulRegionArtIdx

Index, in the **DMUS_OFFSETTABLE** structure, of the global articulation chunk for the region. If 0, the region does not have an articulation and relies on the instrument's global articulation.

ulNextRegionIdx

Index, in the **DMUS_OFFSETTABLE** structure, of the next region in the region list. If 0, there are no more regions.

ulFirstExtCkIdx

Index, in the **DMUS_OFFSETTABLE** structure, of the third-party extension chunk list. If 0, no extension chunks are associated with the region.

WaveLink

Standard DLS structure (declared in the Dls1.h header file) for managing a link from the region to a wave. The **ulTableIndex** member of the **WAVELINK** structure contains the download identifier of the associated wave buffer. (For more information, see **DMUS_DOWNLOADINFO** and Low-Level DLS.)

WSMP

Standard DLS structure (declared in Dls1.h) for managing the playback of the wave. If the **cSampleLoops** member is 1, the following **WLOOP** structure carries the loop start and end points.

WLOOP[]

Standard DLS structure (declared in Dls1.h) for describing a loop.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

DMUS_VEGPARAMS

The **DMUS_VEGPARAMS** structure defines a volume envelope for a DLS articulation chunk.

```
typedef struct {  
    TCENT    tcAttack;  
    TCENT    tcDecay;  
    PERCENT  ptSustain;  
    TCENT    tcRelease;  
    TCENT    tcVel2Attack;  
    TCENT    tcKey2Decay;  
} DMUS_VEGPARAMS;
```

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

See Also

DMUS_ARTICPARAMS

DMUS_WAVE

The **DMUS_WAVE** structure defines a wave chunk for a DLS download.

```
typedef struct {  
    ULONG ulFirstExtCkIdx;  
    ULONG ulCopyrightIdx;  
    ULONG ulWaveDataIdx;  
    WAVEFORMATEX WaveformatEx;  
} DMUS_WAVE;
```

ulFirstExtCkIdx

Index, in the **DMUS_OFFSETTABLE** structure, of third-party extension chunks. If 0, no extension chunks are associated with the wave.

ulCopyrightIdx

Index, in the **DMUS_OFFSETTABLE** structure, of copyright chunks. If 0, no copyright information is associated with the wave.

ulWaveDataIdx

Index, in the **DMUS_OFFSETTABLE** structure, of wave data. See **DMUS_WAVEDATA**.

WaveformatEx

Wave format of the chunk.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

DMUS_WAVEDATA

The **DMUS_WAVEDATA** structure comprises a data chunk for a DLS wave download. The nature of the wave data is defined by the **WAVEFORMATEX** chunk, embedded in the **DMUS_WAVE** structure.

```
typedef struct {  
    ULONG cbSize;  
    BYTE byData[ ];  
} DMUS_WAVEDATA;
```

cbSize

Size of data.

byData[]

PCM wave data.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmdls.h.

Enumerated Types

This section contains references for the following enumerated types:

- **DMUS_CLOCKTYPE**
- **DMUS_COMMANDT_TYPES**
- **DMUS_COMPOSEF_FLAGS**
- **DMUS_NOTEF_FLAGS**
- **DMUS_PLAYMODE_FLAGS**
- **DMUS_PMSGF_FLAGS**
- **DMUS_PMSGT_TYPES**
- **DMUS_SEGF_FLAGS**
- **DMUS_SHAPET_TYPES**
- **DMUS_TIME_RESOLVE_FLAGS**
- **DMUS_TRACKF_FLAGS**

DMUS_CLOCKTYPE

The members of the **DMUS_CLOCKTYPE** enumeration are used in the **ctType** member of the **DMUS_CLOCKINFO** structure.

```
typedef enum {  
    DMUS_CLOCK_SYSTEM = 0,  
    DMUS_CLOCK_WAVE = 1  
} DMUS_CLOCKTYPE;
```

DMUS_CLOCK_SYSTEM

Clock is the system clock.

DMUS_CLOCK_WAVE

Clock is on a wave-playback device.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusicc.h`.

DMUS_COMMANDT_TYPES

The members of the **DMUS_COMMANDT_TYPES** enumeration are used in the *wCommand* parameter of the **IDirectMusicComposer::AutoTransition** and **IDirectMusicComposer::ComposeTransition** methods and in the **bCommand** member of the **DMUS_COMMAND_PARAM** structure.

```
enum enumDMUS_COMMANDT_TYPES {
    DMUS_COMMANDT_GROOVE    = 0,
    DMUS_COMMANDT_FILL      = 1,
    DMUS_COMMANDT_INTRO     = 2,
    DMUS_COMMANDT_BREAK     = 3,
    DMUS_COMMANDT_END       = 4,
    DMUS_COMMANDT_ENDANDINTRO = 5
} DMUS_COMMANDT_TYPES;
```

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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Header: Declared in dmusici.h.

DMUS_COMPOSEF_FLAGS

The **DMUS_COMPOSEF_FLAGS** are used in the *dwFlags* parameter of the **IDirectMusicComposer::ComposeTransition** and **IDirectMusicComposer::AutoTransition** methods.

```
typedef enum enumDMUS_COMPOSEF_FLAGS {
    DMUS_COMPOSEF_NONE           = 0,
    DMUS_COMPOSEF_ALIGN          = 0x1,
    DMUS_COMPOSEF_OVERLAP        = 0x2,
    DMUS_COMPOSEF_IMMEDIATE      = 0x4,
    DMUS_COMPOSEF_GRID           = 0x8,
    DMUS_COMPOSEF_BEAT           = 0x10,
    DMUS_COMPOSEF_MEASURE        = 0x20,
    DMUS_COMPOSEF_AFTERPREPARETIME = 0x40,
    DMUS_COMPOSEF_MODULATE       = 0x1000,
    DMUS_COMPOSEF_LONG           = 0x2000
} DMUS_COMPOSEF_FLAGS;
```

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. This flag is not currently implemented.

DMUS_COMPOSEF_OVERLAP

Overlap the transition into *pToSeg*. This flag is not currently implemented.

DMUS_COMPOSEF_IMMEDIATE

AutoTransition only. Start transition on a music or a reference-time boundary.

DMUS_COMPOSEF_GRID

AutoTransition only. Start transition on a grid boundary.

DMUS_COMPOSEF_BEAT

AutoTransition only. Start transition on a beat boundary.

DMUS_COMPOSEF_MEASURE

AutoTransition only. Start transition on a measure boundary.

DMUS_COMPOSEF_AFTERPREPARETIME

AutoTransition only. Use the **DMUS_SEGF_AFTERPREPARETIME** flag when cuing the transition.

DMUS_COMPOSEF_MODULATE

Compose a transition that modulates smoothly from *pFromSeg* to *pToSeg*, using the chord of *pToSeg*.

DMUS_COMPOSEF_LONG

Composes a long transition. If this flag is not set, the length of the transition is at most one measure unless the *wCommand* parameter of **ComposeTransition** or **AutoTransition** specifies an ending and the style contains an ending of greater than one measure. If this flag is set, the length of the transition increases by one measure.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

DMUS_SEGF_FLAGS

DMUS_NOTEF_FLAGS

The **DMUS_NOTEF_FLAGS** are used in the **bFlags** member of the **DMUS_NOTE_PMSG** structure.

```
typedef enum enumDMUS_NOTEF_FLAGS {
    DMUS_NOTEF_NOTEON = 1,
} DMUS_NOTEF_FLAGS;
```

DMUS_NOTEF_NOTEON

Set if this is a MIDI note-on; clear if it is a MIDI note-off. When a

DMUS_NOTE_PMSG is first sent by the

IDirectMusicPerformance::SendPMsg method, this flag should be set.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

DMUS_PLAYMODE_FLAGS

The **DMUS_PLAYMODE_FLAGS** are used in various structures for the basic play modes. The play mode determines how a music value is transposed to a MIDI note.

```
typedef enum enumDMUS_PLAYMODE_FLAGS {
    DMUS_PLAYMODE_KEY_ROOT      = 1,
    DMUS_PLAYMODE_CHORD_ROOT    = 2,
```

```

DMUS_PLAYMODE_SCALE_INTERVALS = 4,
DMUS_PLAYMODE_CHORD_INTERVALS = 8,
DMUS_PLAYMODE_NONE           = 16,
} DMUS_PLAYMODE_FLAGS;

```

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 only track the intervals in the scale. The key root and chord intervals are ignored. This is useful for melodic lines that play relative to the chord root.

DMUS_PLAYMODE_NORMALCHORD

Combination of `DMUS_PLAYMODE_CHORD_ROOT` and `DMUS_PLAYMODE_CHORD_INTERVALS`. This is the prevalent playback mode. The notes track the intervals in the chord, which is based on the chord root. If there is a scale component to the music value, the additional intervals are pulled from the scale and added. If the chord does not have an interval to match the chord component of the music value, the note is silent.

`DMUS_PLAYMODE_PEDALPOINT`

Combination of `DMUS_PLAYMODE_KEY_ROOT` and `DMUS_PLAYMODE_SCALE_INTERVALS`. The key root is used, and the notes only track 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.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in `dmusici.h`.

See Also

`IDirectMusicPerformance::MIDIToMusic`,
`IDirectMusicPerformance::MusicToMIDI`, `DMUS_NOTE_PMSG`,
`DMUS_IO_STYLENOTE`, `DMUS_IO_STYLEPART`, Music Values and MIDI Notes

DMUS_PMSGF_FLAGS

The members of the `DMUS_PMSGF_FLAGS` enumeration are used in the `dwFlags` member of the `DMUS_PMSG` structure.

```
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  = 16,
    DMUS_PMSGF_TOOL_FLUSH   = 32
} DMUS_PMSGF_FLAGS;
```

`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_ETIME

Message should be processed at the time stamp.

DMUS_PMSGF_TOOL_FLUSH

Message is being flushed.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::SendPMsg, **IDirectMusicTool::GetMsgDeliveryType**

DMUS_PMSGT_TYPES

The members of the **DMUS_PMSGT_TYPES** enumeration are used in the **dwType** member of the **DMUS_PMSG** structure to identify the type of message.

```
typedef enum enumDMUS_PMSGT_TYPES {
    DMUS_PMSGT_MIDI          = 0,
    DMUS_PMSGT_NOTE          = 1,
    DMUS_PMSGT_SYSEX         = 2,
    DMUS_PMSGT_NOTIFICATION  = 3,
    DMUS_PMSGT_TEMPO         = 4,
    DMUS_PMSGT_CURVE         = 5,
    DMUS_PMSGT_TIMESIG       = 6,
    DMUS_PMSGT_PATCH         = 7,
    DMUS_PMSGT_TRANSPOSE     = 8,
    DMUS_PMSGT_CHANNEL_PRIORITY = 9,
    DMUS_PMSGT_STOP          = 10,
    DMUS_PMSGT_DIRTY         = 11,
    DMUS_PMSGT_USER          = 255
} DMUS_PMSGT_TYPES;
```

DMUS_PMSGT_MIDI

MIDI channel message. See **DMUS_MIDI_PMSG**.

DMUS_PMSGT_NOTE

Music note. See **DMUS_NOTE_PMSG**.

DMUS_PMSGT_SYSEX

MIDI system-exclusive message. See **DMUS_SYSEX_PMSG**.

DMUS_PMSGT_NOTIFICATION

Notification message. See **DMUS_NOTIFICATION_PMSG**.

DMUS_PMSGT_TEMPO

Tempo message. See **DMUS_TEMPO_PMSG**.

DMUS_PMSGT_CURVE

Control change and pitch-bend curve. See **DMUS_CURVE_PMSG**.

DMUS_PMSGT_TIMESIG

Time signature. See **DMUS_TIMESIG_PMSG**.

DMUS_PMSGT_PATCH

Patch change. See **DMUS_PATCH_PMSG**.

DMUS_PMSGT_TRANSPOSE

Transposition. See **DMUS_TRANSPOSE_PMSG**.

DMUS_PMSGT_CHANNEL_PRIORITY

Channel priority change. See **DMUS_CHANNEL_PRIORITY_PMSG**.

DMUS_PMSGT_STOP

Stop message. See **DMUS_PMSG**.

DMUS_PMSGT_DIRTY

A control segment has started or ended. See **DMUS_PMSG**.

DMUS_PMSGT_USER

User-defined message.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

DMUS_SEGF_FLAGS

The members of the **DMUS_SEGF_FLAGS** enumeration are passed to various methods of **IDirectMusicPerformance** to control the timing and other aspects of actions on a segment.

```
typedef enum enumDMUS_SEGF_FLAGS {
    DMUS_SEGF_REFTIME          = 64,
    DMUS_SEGF_SECONDARY        = 128,
    DMUS_SEGF_QUEUE             = 256,
    DMUS_SEGF_CONTROL           = 512
    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_FLAGS;

```

DMUS_SEGF_REFTIME

Time parameter is in reference time.

DMUS_SEGF_SECONDARY

Secondary segment.

DMUS_SEGF_QUEUE

Put at the end of the primary segment queue (primary segment only).

DMUS_SEGF_CONTROL

Play as a control segment (secondary segments only). See Remarks.

DMUS_SEGF_AFTERPREPARETIME

Play after the prepare time. See **IDirectMusicPerformance::GetPrepareTime**.

DMUS_SEGF_GRID

Play on a grid boundary.

DMUS_SEGF_BEAT

Play on a beat boundary.

DMUS_SEGF_MEASURE

Play on a measure boundary.

DMUS_SEGF_DEFAULT

Use the segment's default boundary.

DMUS_SEGF_NOINVALIDATE

Setting this flag in **IDirectMusicPerformance::PlaySegment** 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.

Remarks

Normally the primary segment is the control segment. The **DMUS_SEGF_CONTROL** flag can be used to make a secondary segment the control segment. There should be only one control segment at a time. (It is possible to create multiple control segments, but there is no guarantee of which one is actually used by DirectMusic as the control segment.) When a track calls **GetParam** on another track, it does so on the control segment. By default, only the control segment sends tempo messages.

If the **DMUS_SEGF_CONTROL** flag is set, **DMUS_SEGF_SECONDARY** is assumed.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

See Also

IDirectMusicPerformance::Invalidate, **IDirectMusicPerformance::PlaySegment**, **IDirectMusicPerformance::Stop**, **IDirectMusicSegment::GetDefaultResolution**, **IDirectMusicSegment::SetDefaultResolution**, **DMUS_TIME_RESOLVE_FLAGS**

DMUS_SHAPET_TYPES

The members of the **DMUS_SHAPET_TYPES** enumeration are used in the *wShape* parameter of the **IDirectMusicComposer::ComposeSegmentFromShape** and **IDirectMusicComposer::ComposeTemplateFromShape** methods to specify the desired pattern of the groove level.

```
typedef enum enumDMUS_SHAPET_TYPES {
    DMUS_SHAPET_FALLING = 0,
    DMUS_SHAPET_LEVEL = 1,
    DMUS_SHAPET_LOOPABLE = 2,
    DMUS_SHAPET_LOUD = 3,
    DMUS_SHAPET_QUIET = 4,
    DMUS_SHAPET_PEAKING = 5,
    DMUS_SHAPET_RANDOM = 6,
    DMUS_SHAPET_RISING = 7,
    DMUS_SHAPET_SONG = 8
} DMUS_SHAPET_TYPES;
```

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, 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

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dmusici.h.

DMUS_TIME_RESOLVE_FLAGS

The member of the **DMUS_TIME_RESOLVE_FLAGS** enumeration are used in the **dwFlags** member of the **DMUS_PMSG** structure and in the *dwTimeResolveFlags* parameter of the **IDirectMusicPerformance::GetResolvedTime** method.

```
typedef enum enumDMUS_TIME_RESOLVE_FLAGS {
    DMUS_TIME_RESOLVE_AFTERPREPARETIME = 1<<10,
    DMUS_TIME_RESOLVE_GRID             = 1<<11,
    DMUS_TIME_RESOLVE_BEAT             = 1<<12,
    DMUS_TIME_RESOLVE_MEASURE          = 1<<13
} DMUS_TIME_RESOLVE_FLAGS;
```

DMUS_TIME_RESOLVE_AFTERPREPARETIME

Resolve to a time after the prepare 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.

Remarks

These flags can be used interchangeably with the corresponding **DMUS_SEGF_FLAGS**.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Header: Declared in dmusici.h.

DMUS_TRACKF_FLAGS

The **DMUS_TRACKF_FLAGS** values are used in the *dwFlags* parameter of the **IDirectMusicTrack::Play** method.

```
typedef enum enumDMUS_TRACKF_FLAGS {
    DMUS_TRACKF_SEEK    = 1,
    DMUS_TRACKF_LOOP    = 2,
    DMUS_TRACKF_START    = 4,
    DMUS_TRACKF_FLUSH    = 8,
    DMUS_TRACKF_DIRTY    = 16
} DMUS_TRACKF_FLAGS;
```

DMUS_TRACKF_SEEK

IDirectMusicTrack::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 **IDirectMusicTrack::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 **IDirectMusicTrack::GetParam** 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**.

Remarks

When **Play** is called in response to a repeat, **DMUS_TRACKF_LOOP** and **DMUS_TRACKF_SEEK** are set.

Tracks must support seeking to support invalidation.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.

Header: Declared in dmusici.h.

Return Values

The following are the values typically returned by DirectMusic interface methods. For a list of the error codes that each method can return, see the individual method descriptions. These lists are not necessarily comprehensive.

DMUS_E_ALL_TOOLS_FAILED

The graph object was unable to load all tools from the **IStream** object data, perhaps because of errors in the stream or because the tools are incorrectly registered on the client.

DMUS_E_ALL_TRACKS_FAILED

The segment object was unable to load all tracks from the **IStream** object data, perhaps because of errors in the stream or because the tracks are incorrectly registered on the client.

DMUS_E_ALREADY_ACTIVATED

The port has been activated, and the parameter cannot be changed.

DMUS_E_ALREADY_DOWNLOADED

The buffer has already been downloaded.

DMUS_E_ALREADY_EXISTS

The tool is already contained in the graph. You must create a new instance.

DMUS_E_ALREADY_INITED

The object has already been initialized.

DMUS_E_ALREADY_LOADED

A DLS collection is already open.

DMUS_E_ALREADY_SENT

The message has already been sent.

DMUS_E_ALREADYCLOSED

The port is not open.

DMUS_E_ALREADYOPEN

The port was already opened.

DMUS_E_BADARTICULATION

Invalid articulation chunk in DLS collection.

DMUS_E_BADINSTRUMENT

Invalid instrument chunk in DLS collection.

DMUS_E_BADOFFSETTABLE

The offset table has errors.

DMUS_E_BADWAVE

Corrupt wave header.

DMUS_E_BADWAVELINK

The wave-link chunk in DLS collection points to invalid wave.

DMUS_E_BUFFER_EMPTY

There is no data in the buffer.

DMUS_E_BUFFER_FULL

The specified number of bytes exceeds the maximum buffer size.

DMUS_E_BUFFERNOTAVAILABLE

The buffer is not available for download.

DMUS_E_BUFFERNOTSET

No buffer was prepared for the data.

DMUS_E_CANNOT_CONVERT

The requested conversion between music and MIDI values could not be made.

This usually occurs when the provided **DMUS_CHORD_KEY** structure has an invalid chord or scale pattern.

DMUS_E_CANNOT_FREE

The message could not be freed, either because it was not allocated or because it has already been freed.

DMUS_E_CANNOT_OPEN_PORT

The default system port could not be opened.

DMUS_E_CANNOTREAD

An error occurred when trying to read from the **IStream** object.

DMUS_E_CANNOTSEEK

The **IStream** object does not support **Seek**.

DMUS_E_CANNOTWRITE

The **IStream** object does not support **Write**.

DMUS_E_CHUNKNOTFOUND

A chunk with the specified header could not be found.

DMUS_E_DESCEND_CHUNK_FAIL

An attempt to descend into a chunk failed.

DMUS_E_DEVICE_IN_USE

The device is already 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_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_TOOL_HDR

The **IStream** object's data contains an invalid tool header and, therefore, cannot be read by the graph object.

DMUS_E_INVALID_TRACK_HDR

The **IStream** object's data contains an invalid track header and, therefore, cannot be read by the segment object.

DMUS_E_INVALIDBUFFER

An invalid DirectSound buffer was handed to a port.

DMUS_E_INVALIDFILE

Not a valid file.

DMUS_E_INVALIDOFFSET

Wave chunks in the DLS collection file are at incorrect offsets.

DMUS_E_INVALIDPATCH

No instrument in the collection matches the patch number.

DMUS_E_INVALIDPOS

Error reading wave data from a DLS collection. Indicates 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 ID was supplied in **DMUS_OBJECTDESC**.

DMUS_E_LOADER_NOFILENAME

No file name was supplied in **DMUS_OBJECTDESC**.

DMUS_E_LOADER_OBJECTNOTFOUND

The object was not found.

DMUS_E_NO_MASTER_CLOCK

There is no master clock in the performance. Be sure to call the **IDirectMusicPerformance::Init** method.

DMUS_E_NOARTICULATION

Articulation missing from an instrument in the DLS collection.

DMUS_E_NOSYNTHSINK

No sink is connected to the synthesizer.

DMUS_E_NOT_DOWNLOADED_TO_PORT

The object cannot be unloaded because it is not present on the port.

DMUS_E_NOT_FOUND

The requested item is not contained by the object.

DMUS_E_NOT_INIT

A required object is not initialized or failed to initialize.

DMUS_E_NOTADLSCOL

The object being loaded is not a valid DLS collection.

DMUS_E_NOTMONO

The wave chunk has more than one interleaved channel. DLS format requires mono.

DMUS_E_NOTPCM

Wave data is not in PCM format.

DMUS_E_OUT_OF_RANGE

The requested time is outside the range of the segment.

DMUS_E_PORT_NOT_CAPTURE

Not a capture port.

DMUS_E_PORT_NOT_RENDER

Not an output port.

DMUS_E_PORTS_OPEN

The requested operation cannot be performed while there are instantiated ports in any process in the system.

DMUS_E_SEGMENT_INIT_FAILED

Segment initialization failed, probably because of a critical memory situation.

DMUS_E_SET_UNSUPPORTED

Setting the parameter is not supported.

DMUS_E_SYNTHACTIVE

The synthesizer has been activated, and the parameter cannot be changed.

DMUS_E_SYNTHINACTIVE

The synthesizer has not been activated and cannot process data.

DMUS_E_SYNTHNOTCONFIGURED

The synthesizer is not properly configured or opened.

DMUS_E_TIME_PAST

The time requested is in the past.

DMUS_E_TOOL_HDR_NOT_FIRST_CK

The **IStream** 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 **IStream** object's data does not have a track header as the first chunk and, therefore, cannot be read by the segment object.

DMUS_E_TRACK_NOT_FOUND

There is no track of the requested type.

DMUS_E_TYPE_DISABLED

A track parameter is unavailable because it has been disabled.

DMUS_E_TYPE_UNSUPPORTED

Parameter is unsupported on this track.

DMUS_E_UNKNOWNDOWNLOAD

The synthesizer does not support this type of download.

DMUS_E_UNKNOWN_PROPERTY

The property set or item is not implemented by this port.

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_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.

DMUS_S_REQUEUE

The message should be passed to the next tool.

DMUS_S_STRING_TRUNCATED

The method succeeded, but the returned string had to be 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_NOAGGREGATION

Aggregation is not supported. The **LPUNKNOWN** parameter should be set to **NULL**.

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.

DirectMusic Visual Basic Reference

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

- Classes
- Types
- Enumerations
- Error Codes

Classes

This section contains references for methods of the following DirectMusic classes:

- **DirectMusicBand**
- **DirectMusicChordMap**

- **DirectMusicCollection**
- **DirectMusicComposer**
- **DirectMusicLoader**
- **DirectMusicPerformance**
- **DirectMusicSegment**
- **DirectMusicSegmentState**
- **DirectMusicStyle**

DirectMusicBand

An **object** of the **DirectMusicBand** class represents a band, which is used to set the instrument choices and mixer settings for a set of performance channels.

Bands can be stored directly in their own files or embedded in a style. The **DirectMusicBand** object is obtained by using one of the following methods:

- **DirectMusicLoader.LoadBand**
- **DirectMusicLoader.LoadBandFromResource**
- **DirectMusicStyle.GetDefaultBand**
- **DirectMusicStyle.GetBand**

The **DirectMusicBand** class has the following methods:

Segment creation	CreateSegment
Instrument data	Download
	Unload

DirectMusicBand.CreateSegment

The **DirectMusicBand.CreateSegment** method creates a **DirectMusicSegment** object that can be used to perform the volume, pan, transposition, and patch change commands in the band dynamically, using the **DirectMusicPerformance.PlaySegment** method.

object.CreateSegment() As DirectMusicSegment

Parameters

object

Object expression that resolves to a **DirectMusicBand** object.

IDH_DirectMusicBand_dmusic_vb

IDH_DirectMusicBand.CreateSegment_dmusic_vb

Return Values

If the method succeeds, it returns a **DirectMusicSegment** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_FAIL
DMUS_E_OUTOFMEMORY

DirectMusicBand.Download

The **DirectMusicBandDirectDownload** method downloads the DLS data for instruments in the band to a performance object. Once a band has been downloaded, the instruments in the band can be selected, either individually with program change MIDI messages, or all at once by playing a band segment created through a call to the **DirectMusicBand.CreateSegment** method.

object.**Download**(*performance* As **DirectMusicPerformance**)

Parameters

object

Object expression that resolves to a **DirectMusicBand** object.

performance

Performance in which the band is to perform. The performance manages the mapping of channels to DirectMusic ports.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NOT_INIT
DMUS_E_OUTOFMEMORY

Remarks

Because a downloaded band uses synthesizer resources, it should be unloaded when no longer needed by using the **DirectMusicBand.Unload** method.

In the current version of DirectMusic, this method may succeed even though the port does not support DLS.

IDH_DirectMusicBand.Download_dmusic_vb

If `DMUS_E_NOT_INIT` is raised, it usually means that the performance was not properly connected up to an initialized port. Since this is a complete failure, there is no need to call **DirectMusicBand.Unload** later.

If the download only partially succeeds, no error is raised but some instruments might not play. The following are some common causes of a partial download:

- The band has instruments on PChannels 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.

DirectMusicBand.Unload

The **DirectMusicBand.Unload** method unloads the DLS data for instruments in the band previously downloaded by **DirectMusicBand.Download**.

object.Unload(*performance* As **DirectMusicPerformance**)

Parameters

object

Object expression that resolves to a **DirectMusicBand** object.

performance

Performance from which to unload instruments.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

DirectMusicChordMap

An object of the **DirectMusicChordMap** class represents a chord map. Chord maps provide the composer (represented by the **DirectMusicComposer** object) with the information that it needs to create chord progressions for segments that it composes. A chord map can also be applied to an existing segment to change the chords.

The class has no public methods. An instance of it is obtained by using the **DirectMusicPerformance.GetChordmap** method.

See Also

DirectMusicComposer.AutoTransition,
DirectMusicComposer.ChangeChordMap,
DirectMusicComposer.ComposeSegmentFromShape,

IDH_DirectMusicBand.Unload_dmusic_vb

IDH_DirectMusicChordMap_dmusic_vb

DirectMusicComposer.ComposeSegmentFromTemplate,
DirectMusicComposer.ComposeTransition,
DirectMusicPerformance.GetChordmap

DirectMusicCollection

An object of the **DirectMusicCollection** class manages an instance of a DLS file.

The class has no public methods. An instance of it is obtained by using the **DirectMusicLoader.LoadCollection** or the **DirectMusicLoader.LoadCollectionFromResource** method, and is associated with a segment by a call to **DirectMusicSegment.ConnectToCollection**.

DirectMusicComposer

Applications use the methods of the **DirectMusicComposer** class to compose segments and transitions from compositional elements, and to change the chord map of an existing segment.

A **DirectMusicComposer** object is obtained by using the **DirectX7.DirectMusicComposerCreate** method.

The methods of the **DirectMusicComposer** class can be organized into the following groups.

Changing chord maps	ChangeChordMap
Composing ordinary segments	ComposeSegmentFromShape
	ComposeSegmentFromTemplate
Composing template segments	ComposeTemplateFromShape
Composing transition segments	AutoTransition
	ComposeTransition

DirectMusicComposer.AutoTransition

The **DirectMusicComposer.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.

```
object.AutoTransition( _  
    performance As DirectMusicPerformance, _  
    toSeg As DirectMusicSegment, _  
    ICommand As Long, _  
    IFlags As Long, _
```

```
# IDH_DirectMusicCollection_dmusic_vb  
# IDH_DirectMusicComposer_dmusic_vb  
# IDH_DirectMusicComposer.AutoTransition_dmusic_vb
```

chordmap As **DirectMusicChordMap** _
As **DirectMusicSegment**

Parameters

object

Object expression that resolves to a **DirectMusicComposer** object.

performance

Performance in which to do the transition.

toSeg

Segment to which the transition should smoothly flow. See Remarks.

lCommand

Embellishment to use when composing the transition. See **CONST_DMUS_COMMANDT_TYPES**. If this value is **DMUS_COMMANDT_ENDANDINTRO**, the method composes a segment containing both an ending to the primary segment and an introduction to *toSeg*.

lFlags

Composition options. See **CONST_DMUS_COMPOSEF_FLAGS**.

chordmap

DirectMusicChordmap to be used when composing the transition.

Return Values

If the method succeeds, it returns a **DirectMusicSegment** object, unless no style is available for the composition of the transitional segment. See Remarks.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The value in *toSeg* can be **Nothing**, as long as *lFlags* does not include **DMUS_COMPOSEF_MODULATE**. If *toSeg* is **Nothing** or a segment that contain no style track, intro embellishments are not valid. If there is no currently playing segment or it contain no style track, then fill, break, end, and groove embellishments are not valid.

It is possible for both the currently playing segment and *toSeg* to be **Nothing** or segments that contain no style tracks (such as segments based on MIDI files). If so, all embellishments are invalid, and no transition occurs between the currently playing segment and *toSeg*. The method returns **Nothing**, but it succeeds and cues the segment represented by *toSeg*.

The value in *chordmap* can be **Nothing**. If so, an attempt is made to obtain a chord map from a chord-map track, first from *toSeg*, and then from the performance's primary segment. If neither of these segments contains a chord-map track, the chord

occurring at the current time in the primary segment is used as the chord in the transition.

DirectMusicComposer.ChangeChordMap

The **DirectMusicComposer.ChangeChordMap** method modifies the chords and scale pattern of an existing segment to reflect a new chord map.

```
object.ChangeChordMap(segment As DirectMusicSegment, _  
    trackScale As Boolean, _  
    chordmap As DirectMusicChordMap)
```

Parameters

object

Object expression that resolves to a **DirectMusicComposer** object.

segment

Segment in which to change the chord map.

trackScale

If True, the method transposes all the chords to be relative to the root of the new chord map's scale, rather than leaving their roots as they were.

chordmap

New chord map for the segment.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The method can be called while the segment is playing.

DirectMusicComposer.ComposeSegmentFromShape

The **DirectMusicComposer.ComposeSegmentFromShape** method creates an original segment from a style and a chord map, based on a predefined shape. The shape represents the way chords and embellishments occur over time across the segment.

```
object.ComposeSegmentFromShape(style As DirectMusicStyle, _  
    numberOfMeasures As Integer, _  
    shape As Integer, _
```

IDH_DirectMusicComposer.ChangeChordMap_dmusic_vb

IDH_DirectMusicComposer.ComposeSegmentFromShape_dmusic_vb

activity As Integer, _
blIntro As Boolean, _
bEnd As Boolean, _
chordmap As DirectMusicChordMap) As DirectMusicSegment

Parameters

object

Object expression that resolves to a DirectMusicComposer object.

style

Style from which to compose the segment.

numberOfMeasures

Length, in measures, of the segment to be composed.

shape

Shape of the segment to be composed, based on changes in the groove level.
Possible values are of the **CONST_DMUS_SHAPET_TYPES** enumeration.

activity

Rate of harmonic motion. Valid values are from 0 through 3. Lower values mean more chord changes.

blIntro

True if an introduction is to be composed for the segment, False otherwise.

bEnd

True if an ending is to be composed for the segment, False otherwise.

chordmap

DirectMusicChordmap from which to create the segment.

Return Values

If the method succeeds, it returns a **DirectMusicSegment** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicComposer.ComposeSegmentFromTemplate,
DirectMusicComposer.ComposeTemplateFromShape

DirectMusicComposer.ComposeSegmentFromTemplate

The **DirectMusicComposer.ComposeSegmentFromTemplate** method creates an original segment from a style, a chord map, and a template.

```
object.ComposeSegmentFromTemplate( _  
    style As DirectMusicStyle, _  
    templateSeg As DirectMusicSegment, _  
    activity As Integer, _  
    chordmap As DirectMusicChordMap)  
As DirectMusicSegment
```

Parameters

object

Object expression that resolves to a **DirectMusicComposer** object.

style

DirectMusicStyle object from which to create the segment.

templateSeg

DirectMusicSegment object representing the template from which to create the segment.

activity

Rate of harmonic motion. Valid values are from 0 through 3. Lower values mean more chord changes.

chordmap

DirectMusicChordmap object representing the chord map from which to create the segment.

Return Values

If the method succeeds, it returns a **DirectMusicSegment** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_INVALIDARG

Remarks

If *style* is not Nothing, it is used in composing the segment; if it is Nothing, a style is retrieved from the template specified in *templateSeg*. Similarly, if *chordmap* is not

IDH_DirectMusicComposer.ComposeSegmentFromTemplate_dmusic_vb

Nothing, it is used in composing the segment; if it is Nothing, a chord map is retrieved from the template.

If *style* is Nothing and there is no style track in the template, or *chordmap* is Nothing and there is no chord-map track, the method returns DMUS_E_INVALIDARG.

The length of the segment is equal to the length of the template passed in.

See Also

DirectMusicComposer.ComposeSegmentFromShape,
DirectMusicComposer.ComposeTemplateFromShape

DirectMusicComposer.ComposeTemplateFromShape

The **DirectMusicComposer.ComposeTemplateFromShape** method creates a new template segment, based on a predefined shape.

```
object.ComposeTemplateFromShape(numMeasures As Integer, _
    shape As Integer, _
    bIntro As Boolean, _
    bEnd As Boolean, _
    endLength As Integer) As DirectMusicSegment
```

Parameters

object

Object expression that resolves to a **DirectMusicComposer** object.

numMeasures

Length, in measures, of the segment to be composed. This value must be greater than 0.

shape

Shape of the segment to be composed, based on groove levels. Possible values are of the **CONST_DMUS_SHAPET_TYPES** enumeration.

bIntro

True if an introduction is to be composed for the segment, False otherwise.

bEnd

True if an ending is to be composed for the segment, False otherwise.

endLength

Length in measures of the ending, if one is to be composed. If *bEnd* is True, this value must be greater than 0 and equal to or less than the number of measures available (that is, not used in the introduction). See also Remarks.

IDH_DirectMusicComposer.ComposeTemplateFromShape_dmusic_vb

Return Values

If the method succeeds, it returns a **DirectMusicSegment** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY

Remarks

The value of *endLength* should not be greater than the length of the longest ending available in any style likely to be associated with this template through the **DirectMusicComposer.ComposeSegmentFromTemplate** method. The ending starts playing at *endLength* measures before the end of the segment. If the ending is less than *endLength* measures long, the music then reverts to the regular groove.

See Also

DirectMusicComposer.ComposeSegmentFromShape,
DirectMusicComposer.ComposeSegmentFromTemplate

DirectMusicComposer.ComposeTransition

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

```
object.ComposeTransition(fromSeg As DirectMusicSegment, _  
    toSeg As DirectMusicSegment, _  
    mtTime As Long, _  
    ICommand As Long, _  
    IFlags As Long, _  
    chordmap As DirectMusicChordMap) _  
    As DirectMusicSegment
```

Parameters

object

Object expression that resolves to a **DirectMusicComposer** object.

fromSeg

IDH_DirectMusicComposer.ComposeTransition_dmusic_vb

DirectMusicSegment object representing the segment from which to compose the transition.

toSeg

Segment to which the transition should smoothly flow. Can be Nothing if *lFlags* does not include DMUS_COMPOSEF_MODULATE.

mtTime

Time in *fromSeg* from which to compose the transition.

lCommand

Embellishment to use when composing the transition. See **CONST_DMUS_COMMANDT_TYPES**. If this value is DMUS_COMMANDT_ENDANDINTRO, the method composes a segment containing both an ending to *fromSeg* and an introduction to *toSeg*.

lFlags

Composition options. This parameter can contain one or more of the **CONST_DMUS_COMPOSEF_FLAGS** enumeration.

chordmap

DirectMusicChordmap object representing the chord map to be used when composing the transition. See Remarks.

Return Values

If the method succeeds, it returns a **DirectMusicSegment** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY

Remarks

The value in *chordmap* can be Nothing. If so, an attempt is made to obtain a chord map from a chord-map track, first from *toSeg*, and then from *fromSeg*. If neither of these segments contains a chord-map track, the chord occurring at *mtTime* in *fromSeg* is used as the chord in the transition.

The composer looks for a tempo, first in *fromSeg*, and then in *toSeg*. If neither of those segments contains a tempo track, the tempo for the transition segment is taken from the style.

See Also

DirectMusicComposer.AutoTransition

DirectMusicLoader

The **DirectMusicLoader** object is used for finding and loading objects that represent musical and instrument data.

The object is obtained by using the **DirectX7.DirectMusicLoaderCreate** method.

Objects referred to by other objects are loaded automatically. For example, a style may contain references to bands and collections in other files, and these are loaded, if possible, when the **DirectMusicLoader.LoadStyle** method is called.

The methods of **DirectMusicLoader** can be organized into the following groups:

Loading	LoadBand
	LoadBandFromResource
	LoadChordMap
	LoadChordMapFromResource
	LoadCollection
	LoadCollectionFromResource
	LoadSegment
	LoadSegmentFromResource
	LoadStyle
	LoadStyleFromResource
Searching	SetSearchDirectory

DirectMusicLoader.LoadBand

The **DirectMusicLoader.LoadBand** method loads a band from a file.

object.LoadBand(*filename* As String) As DirectMusicBand

Parameters

object

Object expression that resolves to a **DirectMusicLoader** object.

filename

Name of the file containing the band. If the file is not in the current directory or in the directory set by **DirectMusicLoader.SetSearchDirectory**, the full path must be given.

IDH_DirectMusicLoader_dmusic_vb

IDH_DirectMusicLoader.LoadBand_dmusic_vb

Return Values

If the method succeeds, it returns a **DirectMusicBand** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

- DMUS_E_FAIL
- DMUS_E_INVALIDARG
- DMUS_E_OUTOFMEMORY
- DMUS_E_INVALID_BAND
- DMUS_E_LOADER_FAILEDOPEN
- DMUS_E_LOADER_FAILEDCREATE
- DMUS_E_LOADER_FORMATNOTSUPPORTED

See Also

DirectMusicLoader.LoadBandFromResource

DirectMusicLoader.LoadBandFromResource

The **DirectMusicLoader.LoadBandFromResource** method loads a band from a resource.

```
object.LoadBandFromResource(moduleName As String, _  
                             resourceName As String) As DirectMusicBand
```

Parameters

object

Object expression that resolves to a **DirectMusicLoader** object.

moduleName

Name of the module containing the resource.

resourceName

Identifier of the resource containing the band. The resource must be of type "DMBAND".

Return Values

If the method succeeds, it returns a **DirectMusicBand** object.

IDH_DirectMusicLoader.LoadBandFromResource_dmusic_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

- DMUS_E_FAIL
- DMUS_E_INVALIDARG
- DMUS_E_OUTOFMEMORY
- DMUS_E_INVALID_BAND
- DMUS_E_LOADER_FAILEDCREATE
- DMUS_E_LOADER_FORMATNOTSUPPORTED

See Also

DirectMusicLoader.LoadBand

DirectMusicLoader.LoadChordMap

The **DirectMusicLoader.LoadChordMap** method loads a chord map from a file.

object.**LoadChordMap**(*filename* As String) _
As **DirectMusicChordMap**

Parameters

object

Object expression that resolves to a **DirectMusicLoader** object.

filename

Name of the file containing the chord map. If the file is not in the current directory or in the directory set by **DirectMusicLoader.SetSearchDirectory**, the full path must be given.

Return Values

If the method succeeds, it returns a **DirectMusicChordMap** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

- DMUS_E_FAIL
- DMUS_E_INVALIDARG
- DMUS_E_OUTOFMEMORY

IDH_DirectMusicLoader.LoadChordMap_dmusic_vb

DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED

See Also

DirectMusicLoader.LoadChordMapFromResource

DirectMusicLoader.LoadChordMapFromResource

The **DirectMusicLoader.LoadChordMapFromResource** method loads a chord map from a resource.

```
object.LoadChordMapFromResource( _  
    moduleName As String, _  
    resourceName As String) As DirectMusicChordMap
```

Parameters

object

Object expression that resolves to a **DirectMusicLoader** object.

moduleName

Name of the module containing the resource.

resourceName

Identifier of the resource containing the chord map. The resource must be of type "DMCHORD".

Return Values

If the method succeeds, it returns a **DirectMusicChordMap** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_FAIL
DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED

IDH_DirectMusicLoader.LoadChordMapFromResource_dmusic_vb

See Also

DirectMusicLoader.LoadChordMap

DirectMusicLoader.LoadCollection

The **DirectMusicLoader.LoadCollection** method loads a DLS collection from a file.

```
object.LoadCollection(filename As String) _  
    As DirectMusicCollection
```

Parameters

object

Object expression that resolves to a **DirectMusicLoader** object.

filename

Name of the file containing the DLS collection. If the file is not in the current directory or in the directory set by **DirectMusicLoader.SetSearchDirectory**, the full path must be given.

Return Values

If the method succeeds, it returns a **DirectMusicCollection** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

```
DMUS_E_FAIL  
DMUS_E_INVALIDARG  
DMUS_E_OUTOFMEMORY  
DMUS_E_LOADER_FAILEDOPEN  
DMUS_E_LOADER_FAILEDCREATE  
DMUS_E_LOADER_FORMATNOTSUPPORTED  
DMUS_E_NOTADLSCOL
```

See Also

DirectMusicLoader.LoadCollectionFromResource

IDH_DirectMusicLoader.LoadCollection_dmusic_vb

DirectMusicLoader.LoadCollectionFromResource

The **DirectMusicLoader.LoadCollectionFromResource** method loads a DLS collection from a resource.

```
object.LoadCollectionFromResource( _  
    moduleName As String, _  
    resourceName As String) As DirectMusicCollection
```

Parameters

object

Object expression that resolves to a **DirectMusicLoader** object.

moduleName

Name of the module containing the resource.

resourceName

Identifier of the resource containing the collection. The resource must be of type "DMCOLL".

Return Values

If the method succeeds, it returns a **DirectMusicCollection** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

```
DMUS_E_FAIL  
DMUS_E_INVALIDARG  
DMUS_E_OUTOFMEMORY  
DMUS_E_LOADER_FAILEDOPEN  
DMUS_E_LOADER_FAILEDCREATE  
DMUS_E_LOADER_FORMATNOTSUPPORTED  
DMUS_E_NOTADLSCOL
```

See Also

DirectMusicLoader.LoadCollection

DirectMusicLoader.LoadSegment

The **DirectMusicLoader.LoadSegment** method loads a segment from a file.

object.LoadSegment(*filename* As String) As DirectMusicSegment

Parameters

object

Object expression that resolves to a **DirectMusicLoader** object.

filename

Name of the file containing the segment. If the file is not in the current directory or in the directory set by **DirectMusicLoader.SetSearchDirectory**, the full path must be given.

Return Values

If the method succeeds, it returns a **DirectMusicSegment** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_FAIL
 DMUS_E_INVALIDARG
 DMUS_E_OUTOFMEMORY
 DMUS_E_LOADER_FAILEDOPEN
 DMUS_E_LOADER_FAILEDCREATE
 DMUS_E_LOADER_FORMATNOTSUPPORTED

See Also

DirectMusicLoader.LoadSegmentFromResource

DirectMusicLoader.LoadSegmentFromResource

The **DirectMusicLoader.LoadSegmentFromResource** method loads a segment from a resource.

object.LoadSegmentFromResource(_

IDH_DirectMusicLoader.LoadSegment_dmusic_vb

IDH_DirectMusicLoader.LoadSegmentFromResource_dmusic_vb

```
moduleName As String, _  
resourceName As String) As DirectMusicSegment
```

Parameters

object

Object expression that resolves to a **DirectMusicLoader** object.

moduleName

Name of the module containing the resource. The resource must be of type "DMSEG".

resourceName

Identifier of the resource containing the segment.

Return Values

If the method succeeds, it returns a **DirectMusicSegment** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

```
DMUS_E_FAIL  
DMUS_E_INVALIDARG  
DMUS_E_OUTOFMEMORY  
DMUS_E_LOADER_FAILEDOPEN  
DMUS_E_LOADER_FAILEDCREATE  
DMUS_E_LOADER_FORMATNOTSUPPORTED  
DMUS_E_UNSUPPORTED_STREAM
```

See Also

DirectMusicLoader.LoadSegment

DirectMusicLoader.LoadStyle

The **DirectMusicLoader.LoadStyle** method loads a style object from a file.

```
object.LoadStyle(filename As String) As DirectMusicStyle
```

Parameters

object

IDH_DirectMusicLoader.LoadStyle_dmusic_vb

Object expression that resolves to a **DirectMusicLoader** object.

filename

Name of the file containing the style object. If the file is not in the current directory or in the directory set by **DirectMusicLoader.SetSearchDirectory**, the full path must be given.

Return Values

If the method succeeds, it returns a **DirectMusicStyle** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_FAIL
DMUS_E_INVALIDARG
DMUS_E_OUTOFMEMORY
DMUS_E_LOADER_FAILEDOPEN
DMUS_E_LOADER_FAILEDCREATE
DMUS_E_LOADER_FORMATNOTSUPPORTED

See Also

DirectMusicLoader.LoadStyleFromResource

DirectMusicLoader.LoadStyleFromResource

The **DirectMusicLoader.LoadStyleFromResource** method loads a style object from a resource.

```
object.LoadStyleFromResource( _  
    moduleName As String, _  
    resourceName As String) As DirectMusicStyle
```

Parameters

object

Object expression that resolves to a **DirectMusicLoader** object.

moduleName

Name of the module containing the resource.

resourceName

IDH_DirectMusicLoader.LoadStyleFromResource_dmusic_vb

Identifier of the resource containing the style object. The resource must be of type "DMSTYLE".

Return Values

If the method succeeds, it returns a **DirectMusicStyle** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

- DMUS_E_FAIL
- DMUS_E_INVALIDARG
- DMUS_E_OUTOFMEMORY
- DMUS_E_LOADER_FAILEDOPEN
- DMUS_E_LOADER_FAILEDCREATE
- DMUS_E_LOADER_FORMATNOTSUPPORTED

See Also

DirectMusicLoader.LoadStyle

DirectMusicLoader.SetSearchDirectory

The **DirectMusicLoader.SetSearchDirectory** method sets the directory to be searched by the **DirectMusicLoader.LoadBand**, **DirectMusicLoader.LoadChordmap**, **DirectMusicLoader.LoadCollection**, **DirectMusicLoader.LoadSegment**, and **DirectMusicLoader.LoadStyle** methods when a fully qualified path is not supplied.

object.**SetSearchDirectory**(*dir* As String)

Parameters

object

Object expression that resolves to a **DirectMusicLoader** object.

dir

Directory to search.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

IDH_DirectMusicLoader.SetSearchDirectory_dmusic_vb

DMUS_E_OUTOFMEMORY
DMUS_E_LOADER_BADPATH

Remarks

Once a search path is set, the loader does not need a full path every time it is given an object to load by file name. This allows objects that refer to other objects to find them by file name without knowing the full path.

DirectMusicPerformance

An object of the **DirectMusicPerformance** class is the overall manager of music playback. It maps performance channels to a port, plays segments, dispatches messages, requests and receives event notifications, and sets and retrieves music parameters. It also has several methods for getting information about timing and for converting time and music values from one system to another.

If an application wants to have two or more complete sets of music playing at the same time, it can do so by creating more than one performance object. Separate performances obey separate tempo maps and play completely asynchronously, whereas all segments within one performance play in lock step.

The **DirectMusicPerformance** object is obtained by using the **DirectX7.DirectMusicPerformanceCreate** method.

The methods of the **DirectMusicPerformance** class can be organized into the following groups:

Messages	SendCurvePMSG
	SendMIDIPIMSG
	SendNotePMSG
	SendPatchPMSG
	SendTempoPMSG
	SendTimeSigPMSG
	SendTransposePMSG
Notification	AddNotificationType
	GetNotificationPMSG
	RemoveNotificationType
	SetNotificationHandle
Parameters	GetChordmap
	GetCommand
	GetGrooveLevel

IDH_DirectMusicPerformance_dmusic_vb

	GetMasterAutoDownload
	GetMasterGrooveLevel
	GetMasterTempo
	GetMasterVolume
	GetStyle
	GetTempo
	GetTimeSig
	Reset
	SetMasterAutoDownload
	SetMasterGrooveLevel
	SetMasterTempo
	SetMasterVolume
Ports	GetPortCaps
	GetPortCount
	GetPortName
	SetPort
Segments	GetSegmentState
	IsPlaying
	PlaySegment
	Stop
Timing	AdjustTime
	ClockToMusicTime
	GetBumperLength
	GetClockTime
	GetLatencyTime
	GetMusicTime
	GetPrepareTime
	GetQueueTime
	GetResolvedTime
	MusicToClockTime
	SetBumperLength
	SetPrepareTime
Miscellaneous	CloseDown
	Init
	Invalidate

DirectMusicPerformance.AddNotificationType

The **DirectMusicPerformance.AddNotificationType** method causes the performance to generate notification messages whenever events of the requested type occur.

```
object.AddNotificationType( _  
    type As CONST_DMUS_NOTIFICATION_TYPE)
```

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

type

Type of event for which notification messages are to be sent. For possible values, see **CONST_DMUS_NOTIFICATION_TYPE**.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

```
DMUS_E_INVALIDARG  
DMUS_E_OUTOFMEMORY
```

See Also

DirectMusicPerformance.RemoveNotificationType

DirectMusicPerformance.AdjustTime

The **DirectMusicPerformance.AdjustTime** method adjusts the internal performance time forward or backward. This is mostly used to compensate for drift when synchronizing to another source.

```
object.AdjustTime(rtAmount As Long)
```

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

rtAmount

IDH_DirectMusicPerformance.AddNotificationType_dmusic_vb

IDH_DirectMusicPerformance.AdjustTime_dmusic_vb

Amount of time, in clock time units, to add or subtract. This can be a number in the range from –1000 through 1000 (–1 second through +1 second).

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_INVALIDARG.

Remarks

The adjusted time is used internally by DirectMusic. It is not reflected in the time retrieved by the **DirectMusicPerformance.GetClockTime** method.

DirectMusicPerformance.ClockToMusicTime

The **DirectMusicPerformance.ClockToMusicTime** method converts clock time to music time. Clock time is an absolute time. Music time is relative to the tempo of the performance.

object.ClockToMusicTime(*ctTime As Long*) As Long

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

ctTime

Time to convert, in clock time units.

Return Values

If the method succeeds, it returns the equivalent music time.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_NO_MASTER_CLOCK.

Remarks

If a master tempo has been set for the performance, it is taken into account when converting to music time.

Because music time is less precise than clock time, rounding off occurs.

IDH_DirectMusicPerformance.ClockToMusicTime_dmusic_vb

See Also

DirectMusicPerformance.MusicToClockTime, Clock Time vs. Music Time

DirectMusicPerformance.CloseDown

The **DirectMusicPerformance.CloseDown** method closes down the performance. An application that created the performance object and called **DirectMusicPerformance.Init** on it must call **CloseDown** before the performance object is released.

object.CloseDown()

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

Failure to call **CloseDown** can cause memory leaks or application failures.

CloseDown releases any downloaded instruments that have not been unloaded.

DirectMusicPerformance.GetBumperLength

The **DirectMusicPerformance.GetBumperLength** method retrieves the amount of time between the time at which messages are placed in the port buffer and the time at which they begin to be processed by the port.

object.GetBumperLength() As Long

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

IDH_DirectMusicPerformance.CloseDown_dmusic_vb

IDH_DirectMusicPerformance.GetBumperLength_dmusic_vb

Return Values

If the method succeeds, it returns the bumper length, in milliseconds.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The default value is 50 milliseconds.

See Also

DirectMusicPerformance.SetBumperLength

DirectMusicPerformance.GetChordmap

The **DirectMusicPerformance.GetChordmap** method retrieves the chord map from the performance's control segment.

object.**GetChordMap**(*mtTime* As Long, _
 mtUntil As Long) As **DirectMusicChordMap**

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

mtTime

Time for which the chord map is to be retrieved, in music time.

mtUntil

Variable to receive the music time (relative to *mtTime*) until which the chordmap is valid. If this returns a value of 0, either the chord map is always valid, or it is unknown when it might become invalid. See Remarks.

Return Values

If the method succeeds, it returns a **DirectMusicChordmap** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK

IDH_DirectMusicPerformance.GetChordmap_dmusic_vb

DMUS_E_GET_UNSUPPORTED
DMUS_E_NOT_FOUND
DMUS_E_TRACK_NOT_FOUND

Remarks

Normally, the primary segment is the control segment. However, a secondary segment can be designated as a control segment when it is played. The object returned by the method can become invalid before the time returned in *mtUntil* if another control segment is cued.

See Also

CONST_DMUS_SEGF_FLAGS

DirectMusicPerformance.GetClockTime

The **DirectMusicPerformance.GetClockTime** method retrieves the current time.

object.GetClockTime() As Long

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

Return Values

If the method succeeds, it returns the time, in units of approximately 1 millisecond, relative to an arbitrary start time.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_NO_MASTER_CLOCK.

See Also

DirectMusicPerformance.GetMusicTime, Clock Time vs. Music Time

DirectMusicPerformance.GetCommand

IDH_DirectMusicPerformance.GetClockTime_dmusic_vb

The **DirectMusicPerformance.GetCommand** method retrieves a command from the performance's control segment. The command indicates what type of pattern is being played at the specified time.

object.**GetCommand**(*mtTime* As Long, *mtUntil* As Long) As Byte

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

mtTime

Time for which the command is to be retrieved, in music time.

mtUntil

Variable to receive the music time (relative to *mtTime*) until which the command is valid. If this returns a value of 0, either the command is always valid, or it is unknown when it might become invalid. See Remarks.

Return Values

If the method succeeds, it returns a command type. See **CONST_DMUS_COMMANDT_TYPES**.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following error codes:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_GET_UNSUPPORTED
DMUS_E_NOT_FOUND
DMUS_E_TRACK_NOT_FOUND

Remarks

Normally, the primary segment is the control segment. However, a secondary segment can be designated as a control segment when it is played. The object returned by the method can become invalid before the time returned in *mtUntil* if another control segment is cued.

See Also

CONST_DMUS_SEGF_FLAGS

DirectMusicPerformance.GetGrooveLevel

The **DirectMusicPerformance.GetGrooveLevel** method retrieves the groove level from the performance's control segment. The groove level determines which patterns can be played at the specified time.

object.**GetGrooveLevel**(*mtTime* As Long, _
 mtUntil As Long) As Byte

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

mtTime

Time for which the groove level is to be retrieved, in music time.

mtUntil

Variable to receive the music time (relative to *mtTime*) until which the groove level is valid. If this returns a value of 0, either the groove level is always valid, or it is unknown when it might become invalid. See Remarks.

Return Values

If the method succeeds, it returns a value in the range from 1 through 100.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following error codes:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_GET_UNSUPPORTED
DMUS_E_NOT_FOUND
DMUS_E_TRACK_NOT_FOUND

Remarks

Normally, the primary segment is the control segment. However, a secondary segment can be designated as a control segment when it is played. The object returned by the method can become invalid before the time returned in *mtUntil* if another control segment is cued.

See Also

CONST_DMUS_SEGF_FLAGS

DirectMusicPerformance.GetLatencyTime

The **DirectMusicPerformance.GetLatencyTime** method retrieves the current latency time. Latency time is the time at which messages are sent to the port to be rendered.

object.GetLatencyTime() As Long

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

Return Values

If the method succeeds, it returns the latency time, in clock time units.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_NO_MASTER_CLOCK

DirectMusicPerformance.GetMasterAutoDownload

The **DirectMusicPerformance.GetMasterAutoDownload** method retrieves the current setting for automatic downloading of instruments.

object.GetMasterAutoDownload() As Boolean

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

IDH_DirectMusicPerformance.GetLatencyTime_dmusic_vb

IDH_DirectMusicPerformance.GetMasterAutoDownload_dmusic_vb

Return Values

If the method succeeds, it returns **True** if autodownloading is on, and **False** otherwise. The default value is **False**.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicPerformance.SetMasterAutoDownload,
DirectMusicSegment.SetAutoDownloadEnable

DirectMusicPerformance.GetMasterGrooveLevel

The **DirectMusicPerformance.GetMasterGrooveLevel** method retrieves the current master groove level, which is a value added to all groove levels in the performance. The resulting value is adjusted, if necessary, to fall within the range from 1 through 100.

object.**GetMasterGrooveLevel()** As Integer

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

Return Values

If the method succeeds, it returns the master groove level.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicPerformance.SetMasterGrooveLevel

DirectMusicPerformance.GetMasterTempo

The **DirectMusicPerformance.GetMasterTempo** method retrieves the current master tempo.

object.GetMasterTempo() As Single

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

Return Values

If the method succeeds, it returns a value in the range from 0.25 through 2.0.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The master tempo is a scaling factor that is applied to the tempo by the final output tool. By default it is 1. A value of 0.5 would halve the tempo, and a value of 2.0 would double it.

See Also

DirectMusicPerformance.SetMasterTempo

DirectMusicPerformance.GetMasterVolume

The **DirectMusicPerformance.GetMasterVolume** method retrieves the current master volume.

object.GetMasterVolume() As Long

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

IDH_DirectMusicPerformance.GetMasterTempo_dmusic_vb

IDH_DirectMusicPerformance.GetMasterVolume_dmusic_vb

Return Values

If the method succeeds, it returns the current master volume, in hundredths of a decibel.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The master volume is an amplification or attenuation factor applied to the default volume of the entire performance. The range of permitted values is determined by the port.

See Also

DirectMusicPerformance.SetMasterVolume

DirectMusicPerformance.GetMusicTime

The **DirectMusicPerformance.GetMusicTime** method returns the current time of the performance, in music time.

object.**GetMusicTime()** As Long

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

Return Values

If the method succeeds, it returns the current time, in music time units.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_NO_MASTER_CLOCK.

See Also

DirectMusicPerformance.GetClockTime, Clock Time vs. Music Time

IDH_DirectMusicPerformance.GetMusicTime_dmusic_vb

DirectMusicPerformance.GetNotificationPMsg

The **DirectMusicPerformance.GetNotificationPMsg** method retrieves a pending notification message.

object.**GetNotificationPMSG**(
 message As DMUS_NOTIFICATION_PMSG) As Boolean

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

message

DMUS_NOTIFICATION_PMSG type to receive the message.

Return Values

If the method succeeds, it returns True if a message was received, and False if there was no message pending.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

DirectMusicPerformance.GetPortCaps

The **DirectMusicPerformance.GetPortCaps** method retrieves information about the capabilities of a port.

object.**GetPortCaps**(*index* As Long, *caps* As DMUS_PORTCAPS)

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

index

Index of the port, in the range from 1 through
DirectMusicPerformance.GetPortCount.

caps

DMUS_PORTCAPS type to receive information about the port.

IDH_DirectMusicPerformance.GetNotificationPMsg_dmusic_vb

IDH_DirectMusicPerformance.GetPortCaps_dmusic_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to `DMUS_E_INVALIDARG`.

DirectMusicPerformance.GetPortCount

The **DirectMusicPerformance.GetPortCount** method returns the number of DirectMusic ports available on the system.

object.**GetPortCount()** As Long

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

Return Values

If the method succeeds, it returns the number of available ports.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

DirectMusicPerformance.GetPortName

The **DirectMusicPerformance.GetPortName** returns the name of a port.

object.**GetPortName(index As Long)** As String

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

index

Index of the port. Must be in the range from 1 through the value returned by **DirectMusicPerformance.GetPortCount**.

Return Values

If the method succeeds, it returns the name of the port — for example, Microsoft Synthesizer.

IDH_DirectMusicPerformance.GetPortCount_dmusic_vb

IDH_DirectMusicPerformance.GetPortName_dmusic_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

DirectMusicPerformance.GetPrepareTime

The **DirectMusicPerformance.GetPrepareTime** method retrieves the approximate interval between the time at which messages are prepared and the time at which they are processed and heard.

object.**GetPrepareTime()** As Long

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

Return Values

If the method succeeds, it returns the prepare time, in milliseconds.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The default value is 1000 milliseconds.

See Also

DirectMusicPerformance.SetPrepareTime

DirectMusicPerformance.GetQueueTime

The **DirectMusicPerformance.GetQueueTime** method retrieves the current queue (or flush) time. Messages that have time stamps earlier than this time have already been queued to the port and cannot be invalidated.

object.**GetQueueTime()** As Long

IDH_DirectMusicPerformance.GetPrepareTime_dmusic_vb

IDH_DirectMusicPerformance.GetQueueTime_dmusic_vb

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

Return Values

If the method succeeds, it returns the queue time, in clock time units.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_NO_MASTER_CLOCK.

Remarks

Queue time is equal to the value returned by **DirectMusicPerformance.GetLatencyTime** plus the value returned by **DirectMusicPerformance.GetBumperLength**.

When a segment is stopped immediately, all messages that have been sent but not queued to the port buffer are flushed. If you want to resume playing the segment again at the last point heard, set the new start point to the offset of queue time within the segment when the segment was stopped.

See Also

DirectMusicPerformance.Invalidate

DirectMusicPerformance.GetResolvedTime

The **DirectMusicPerformance.GetResolvedTime** method adjusts a given time to a given boundary.

```
object.GetResolvedTime(ctTime As Long, _  
    flags As Long) As Long
```

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

ctTime

IDH_DirectMusicPerformance.GetResolvedTime_dmusic_vb

Time to resolve, in clock time units. If this is less than the current time, the current time is used.

flags

One or more of the following **CONST_DMUS_SEGF_FLAGS** describing the resolution desired:

DMUS_SEGF_AFTERPREPARETIME

Resolve to a time after the prepare time.

DMUS_SEGF_GRID

Resolve to a time on a grid boundary.

DMUS_SEGF_BEAT

Resolve to a time on a beat boundary.

DMUS_SEGF_MEASURE

Resolve to a time on a measure boundary.

Return Values

If the method succeeds, it returns the resolved time.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

DirectMusicPerformance.GetSegmentState

The **DirectMusicPerformance.GetSegmentState** method retrieves the **DirectMusicSegmentState** object representing the primary segment playing at a given time.

object.**GetSegmentState**(*mtTime* As Long) _
As DirectMusicSegmentState

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

mtTime

Time for which the segment state is to be retrieved, in music time.

Return Values

If the method succeeds, it returns a **DirectMusicSegmentState** object.

IDH_DirectMusicPerformance.GetSegmentState_dmusic_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to `DMUS_E_NOT_FOUND`.

Remarks

To get the currently playing segment state, pass the time retrieved by the **DirectMusicPerformance.GetMusicTime** method. Currently playing in this context means that it is being called into to perform messages. Because of latency, the currently playing segment state is not necessarily the one actually being heard.

Because of latency, it is also a good idea to add 150 to *mtTime* if retrieving a segment state immediately after calling **DirectMusicPerformance.PlaySegment**.

DirectMusicPerformance.GetStyle

The **DirectMusicPerformance.GetStyle** method retrieves the style underlying the control segment at a given time.

object.**GetStyle**(*mtTime* As Long, *mtUntil* As Long) _
As DirectMusicStyle

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

mtTime

Time for which the style is to be retrieved, in music time.

mtUntil

Variable to receive the music time (relative to *mtTime*) until which the style is valid. If this returns a value of 0, either the style is always valid, or it is unknown when it might become invalid. See Remarks.

Return Values

If the method succeeds, it returns a **DirectMusicStyle** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

`DMUS_E_NO_MASTER_CLOCK`
`DMUS_E_GET_UNSUPPORTED`

IDH_DirectMusicPerformance.GetStyle_dmusic_vb

DMUS_E_NOT_FOUND
DMUS_E_TRACK_NOT_FOUND

Remarks

Normally, the primary segment is the control segment. However, a secondary segment can be designated as a control segment when it is played. The object returned by the method can become invalid before the time returned in *mtUntil* if another control segment is cued.

DirectMusicPerformance.GetTempo

The **DirectMusicPerformance.GetTempo** method retrieves the tempo at a given time.

object.**GetTempo**(*mtTime* As Long, *mtUntil* As Long) As Double

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

mtTime

Time for which to retrieve the tempo, in music time. The last tempo change before or at this time is used to determine the tempo.

mtUntil

Variable to receive the music time (relative to *mtTime*) until which the tempo is valid. If this returns a value of 0, either the tempo is always valid, or it is unknown when it might become invalid.

Return Values

If the method succeeds, it returns the tempo, in beats per minute. This value is in the range from DMUS_TEMPO_MIN through DMUS_TEMPO_MAX (see **CONST_DMUS**).

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_GET_UNSUPPORTED
DMUS_E_NOT_FOUND
DMUS_E_TRACK_NOT_FOUND

IDH_DirectMusicPerformance.GetTempo_dmusic_vb

DirectMusicPerformance.GetTimeSig

The **DirectMusicPerformance.GetTimeSig** method retrieves the time signature at a given time.

```
object.GetTimeSig(mtTime As Long, _  
    mtUntil As Long, _  
    timeSig As DMUS_TIMESIGNATURE)
```

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

mtTime

Time for which to retrieve the time signature, in music time. The last time signature change before or at this time is used to determine the time signature.

mtUntil

Variable to receive the music time (relative to *mtTime*) until which the time signature is valid. If this returns a value of 0, either the time signature is always valid, or it is unknown when it might become invalid. See Remarks.

timeSig

DMUS_TIMESIGNATURE type to receive information about the time signature. The **mtTime** member receives the offset of the last time signature change from the requested time, and is always 0 or less.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

```
DMUS_E_NO_MASTER_CLOCK  
DMUS_E_GET_UNSUPPORTED  
DMUS_E_NOT_FOUND  
DMUS_E_TRACK_NOT_FOUND
```

DirectMusicPerformance.Init

The **DirectMusicPerformance.Init** method initializes the performance and associates it with a DirectSound object. This method should be called only once, before any other methods are called on the performance.

```
object.Init(DirectSound As DirectSound, hwnd As Long)
```

IDH_DirectMusicPerformance.GetTimeSig_dmusic_vb

IDH_DirectMusicPerformance.Init_dmusic_vb

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

DirectSound

Existing **DirectSound** object, or Nothing if you want DirectMusic to create the object.

hwnd

Window handle to be used for the creation of DirectSound. This parameter can be 0, in which case the foreground window is used. See Remarks.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_ALREADY_INITED

DMUS_E_OUTOFMEMORY

Remarks

There should be only one **DirectSound** object per process. If your application uses a **DirectSound** object for playing waves, it must pass in that object here.

The *hwnd* parameter is significant only if *DirectSound* is Nothing. If a **DirectSound** object is created separately by the application and passed to this method, the application is responsible for setting the window handle in a call to **DirectSound.SetCooperativeLevel**.

Do not 0 pass as *hwnd* because the application window might not be in the foreground when the method is called. In general, the top-level application window handle should be passed to **DirectMusicPerformance.Init**, **DirectSound.SetCooperativeLevel**, and **DirectDraw7.SetCooperativeLevel**.

The performance must be terminated by using the **DirectMusicPerformance.CloseDown** method before being released.

DirectMusicPerformance.Invalidate

The **DirectMusicPerformance.Invalidate** method flushes all queued messages whose time stamps are later than the supplied time and causes all tracks of all segments to resend their data from the given time forward.

object.Invalidate(mtTime As Long, flags As Long)

IDH_DirectMusicPerformance.Invalidate_dmusic_vb

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

mtTime

Time from which to invalidate, adjusted by *flags*. Setting this value to 0 causes immediate invalidation.

flags

Adjusts *mtTime* to align to measures, beats, or grids. This value can be 0 or one of the following members of **CONST_DMUS_SEGF_FLAGS**:

DMUS_SEGF_MEASURE

DMUS_SEGF_BEAT

DMUS_SEGF_GRID

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_NO_MASTER_CLOCK.

Remarks

If *mtTime* is so long ago that it is impossible to invalidate that time, the earliest possible time is used.

See Also

DirectMusicPerformance.QueueTime

DirectMusicPerformance.IsPlaying

The **DirectMusicPerformance.IsPlaying** method determines whether a particular segment or segment state is currently playing at the speakers.

*object.IsPlaying(segment As DirectMusicSegment, _
segmentState As DirectMusicSegmentState) As Boolean*

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

segment

DirectMusicSegment to check. If Nothing, check only *segmentState*.

segmentState

DirectMusicSegmentState state to check. If Nothing, check only *segment*.

IDH_DirectMusicPerformance.IsPlaying_dmusic_vb

Return Values

If the method succeeds and the requested segment or segment state is playing, the return value is True. If neither is playing, or only one was requested and it is not playing, the return value is False.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK

DirectMusicPerformance.MusicToClockTime

The **DirectMusicPerformance.MusicToClockTime** method converts time in music time format to time in clock time format.

object.**MusicToClockTime**(*mtTime* As Long) As Long

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

mtTime

Time in music time format to convert.

Return Values

If the method succeeds, it returns the time, in clock time units.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_NO_MASTER_CLOCK.

Remarks

Because clock time has a greater precision than music time, a time that has been converted from clock time to music time, and then back again, will probably not have its original value.

IDH_DirectMusicPerformance.MusicToClockTime_dmusic_vb

See Also

DirectMusicPerformance.ClockToMusicTime, Clock Time vs. Music Time

DirectMusicPerformance.PlaySegment

The **DirectMusicPerformance.PlaySegment** method begins playback of a segment.

```
object.PlaySegment(segment As DirectMusicSegment, _  
    lFlags As Long, _  
    startTime As Long) As DirectMusicSegmentState
```

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

segment

DirectMusicSegment to play.

lFlags

Flags that modify the method's behavior. See **CONST_DMUS_SEGF_FLAGS**.

startTime

Time at which to begin playing the segment, adjusted to any resolution boundary specified in *lFlags*. The time is in music time unless the **DMUS_SEGF_REFTIME** flag is set. A value of 0 causes the segment to start playing as soon as possible.

Return Values

If the method succeeds, it returns a **DirectMusicSegmentState** object representing the playing segment.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

```
DMUS_E_OUTOFMEMORY  
DMUS_E_NO_MASTER_CLOCK  
DMUS_E_SEGMENT_INIT_FAILED  
DMUS_E_TIME_PAST
```

IDH_DirectMusicPerformance.PlaySegment_dmusic_vb

Remarks

Segments should be greater than 250 milliseconds in length.

The boundary resolutions in *lFlags* are relative to the currently playing 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 this case, it plays as soon as the first one reaches its end.

See Also

DirectMusicPerformance.IsPlaying

DirectMusicPerformance.RemoveNotificationType

The **DirectMusicPerformance.RemoveNotificationType** method removes a previously added notification type from the performance so that notification messages of that type are no longer sent.

```
object.RemoveNotificationType( _  
    type As CONST_DMUS_NOTIFICATION_TYPE)
```

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

type

Type of event for which notification messages are no longer to be sent. For possible values, see `CONST_DMUS_NOTIFICATION_TYPE`.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to `DMUS_E_INVALIDARG`

See Also

DirectMusicPerformance.AddNotificationType

IDH_DirectMusicPerformance.RemoveNotificationType_dmusic_vb

DirectMusicPerformance.Reset

The **DirectMusicPerformance.Reset** method resets the port.

object.Reset(resetflags As Long)

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

resetflags

Flags. By default (*resetflags* = 0), the method performs a GMReset. If this value is 1, the port is reset by being closed and reopened.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

DirectMusicPerformance.SendCurvePMSG

The **DirectMusicPerformance.SendCurvePMSG** method sends a performance message containing information about a MIDI curve.

*object.SendCurvePMSG(lTime As Long, _
flags As Long, _
channel As Long, _
msg As DMUS_CURVE_PMSG)*

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

lTime

Time at which the message is to play. This is in music time unless **DMUS_PMSGF_REFTIME** is in *flags*.

flags

Flags modifying how and when the message is processed. See **CONST_DMUS_PMSGF_FLAGS**.

channel

Performance channel that is the destination for the message.

msg

DMUS_CURVE_PMSG type containing information about the curve.

IDH_DirectMusicPerformance.Reset_dmusic_vb

IDH_DirectMusicPerformance.SendCurvePMSG_dmusic_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_ALREADY_SENT
DMUS_E_INVALIDARG

Remarks

If the time of the message is set to 0 and the *flags* parameter contains DMUS_PMSGF_REFTIME, it is assumed that this message is cued to go out immediately.

DirectMusicPerformance.SendMIDIPIMSG

The **DirectMusicPerformance.SendMIDIPIMSG** method sends a performance message containing information about a MIDI channel message not covered by other methods.

object.SendMIDIPIMSG(*lTime* As Long, _
 flags As Long, _
 channel As Long, _
 status As Byte, _
 byte1 As Byte, _
 byte2 As Byte)

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

lTime

Time at which the message is to play. This is in music time unless DMUS_PMSGF_REFTIME is in *flags*.

flags

Flags modifying how and when the message is processed. See **CONST_DMUS_PMSGF_FLAGS**.

channel

Performance channel that is the destination for the message.

status

Standard MIDI status byte. See Remarks.

byte1

IDH_DirectMusicPerformance.SendMIDIPIMSG_dmusic_vb

First data byte. Ignored for MIDI messages that do not require it.

byte2

Second data byte. Ignored for MIDI messages that do not require it.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK

DMUS_E_ALREADY_SENT

DMUS_E_INVALIDARG

Remarks

If the time of the message is set to 0 and the *flags* parameter contains DMUS_PMSGF_REFTIME, it is assumed that this message is cued to go out immediately.

Because the channel is specified in another parameter, *status* does not contain the channel number in the 4 lower bits, as it would in a standard MIDI message. Thus *status* is &H80 for a note-off, &H90 for a note-on, and so on. See the MIDI specification for other status bytes.

DirectMusicPerformance.SendNotePMSG

The **DirectMusicPerformance.SendNotePMSG** method sends a performance message containing information about a note.

```
object.SendNotePMSG(lTime As Long, _
    flags As Long, _
    channel As Long, _
    msg As DMUS_NOTE_PMSG)
```

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

lTime

Time at which the message is to play. This is in music time unless DMUS_PMSGF_REFTIME is in *flags*.

flags

Flags modifying how and when the message is processed. See **CONST_DMUS_PMSGF_FLAGS**.

IDH_DirectMusicPerformance.SendNotePMSG_dmusic_vb

channel

Performance channel that is the destination for the message.

msg

DMUS_NOTE_PMSG type containing information about the note.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK

DMUS_E_ALREADY_SENT

DMUS_E_INVALIDARG

Remarks

If the time of the message is set to 0 and the *flags* parameter contains **DMUS_PMSGF_REFTIME**, it is assumed that this message is cued to go out immediately.

See Also

Music Values and MIDI Notes

DirectMusicPerformance.SendPatchPMSG

The **DirectMusicPerformance.SendPatchPMSG** method sends a performance message containing information about a MIDI patch change.

```
object.SendPatchPMSG(lTime As Long, _
    flags As Long, _
    channel As Long, _
    instrument As Byte, _
    byte1 As Byte, _
    byte2 As Byte)
```

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

lTime

Time at which the message is to play. This is in music time unless **DMUS_PMSGF_REFTIME** is in *flags*.

IDH_DirectMusicPerformance.SendPatchPMSG_dmusic_vb

flags

Flags modifying how and when the message is processed. See **CONST_DMUS_PMSGF_FLAGS**.

channel

Performance channel that is the destination for the message.

instrument

Patch number to assign to the channel.

byte1

Most significant byte of bank select.

byte2

Least significant byte of bank select.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK

DMUS_E_ALREADY_SENT

DMUS_E_INVALIDARG

Remarks

If the time of the message is set to 0 and the *flags* parameter contains DMUS_PMSGF_REFTIME, it is assumed that this message is cued to go out immediately.

DirectMusicPerformance.SendTempoMSG

The **DirectMusicPerformance.SendTempoMSG** method sends a performance message containing information about a tempo change.

object.**SendTempoMSG**(*lTime* As Long, _
 flags As Long, _
 tempo As Double)

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

lTime

IDH_DirectMusicPerformance.SendTempoMSG_dmusic_vb

Time at which the message is to play. This is in music time unless DMUS_PMSGF_REFTIME is in *flags*.

flags

Flags modifying how and when the message is processed. See **CONST_DMUS_PMSGF_FLAGS**.

tempo

New tempo, in beats per minute.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK

DMUS_E_ALREADY_SENT

DMUS_E_INVALIDARG

Remarks

If the time of the message is set to 0 and the *flags* parameter contains DMUS_PMSGF_REFTIME, it is assumed that this message is cued to go out immediately.

DirectMusicPerformance.SendTimeSigPMSG

The **DirectMusicPerformance.SendCurvePMSG** method sends a performance message containing information about a MIDI curve.

object.SendTimeSigPMSG(*lTime* As Long, _
flags As Long, _
timeSig As DMUS_TIMESIGNATURE)

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

lTime

Time at which the message is to play. This is in music time unless DMUS_PMSGF_REFTIME is in *flags*.

flags

Flags modifying how and when the message is processed. See **CONST_DMUS_PMSGF_FLAGS**.

timeSig

IDH_DirectMusicPerformance.SendTimeSigPMSG_dmusic_vb

DMUS_TIMESIGNATURE type containing information about the time signature.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK
DMUS_E_ALREADY_SENT
DMUS_E_INVALIDARG

Remarks

If the time of the message is set to 0 and the *flags* parameter contains DMUS_PMSGF_REFTIME, it is assumed that this message is cued to go out immediately.

DirectMusicPerformance.SendTransposePMSG

The **DirectMusicPerformance.SendTransposePMSG** method sends a performance message causing a transposition to begin.

object.SendTransposePMSG(*lTime* As Long, _
 flags As Long, _
 channel As Long, _
 transpose As Integer)

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

lTime

Time at which the message is to play. This is in music time unless DMUS_PMSGF_REFTIME is in *flags*.

flags

Flags modifying how and when the message is processed. See **CONST_DMUS_PMSGF_FLAGS**.

channel

Pchannel on which the transposition is to take place.

transpose

IDH_DirectMusicPerformance.SendTransposePMSG_dmusic_vb

Number of semitones by which to transpose notes. This can be a negative value. If the transposition of a note puts it outside the standard MIDI range from 0 through 127, it does not play.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_NO_MASTER_CLOCK

DMUS_E_ALREADY_SENT

DMUS_E_INVALIDARG

Remarks

If the time of the message is set to 0 and the *flags* parameter contains DMUS_PMSGF_REFTIME, it is assumed that this message is cued to go out immediately.

DirectMusicPerformance.SetBumperLength

The **DirectMusicPerformance.SetBumperLength** method sets the amount of time between the time at which messages are placed in the port buffer and the at which they begin to be processed by the port.

object.SetBumperLength(*IMilliSeconds As Long*)

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

IMilliSeconds

Desired bumper length, in milliseconds. The default value is 50.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicPerformance.GetBumperLength,
DirectMusicPerformance.SetPrepareTime

IDH_DirectMusicPerformance.SetBumperLength_dmusic_vb

DirectMusicPerformance.SetMasterAutoDownload

The **DirectMusicPerformance.SetMasterAutoDownload** method turns automatic downloading of instruments on or off.

object.SetMasterAutoDownload(*b* As Boolean)

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

b

True to turn autodownloading on, False to turn it off. The default value is False.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicPerformance.GetMasterAutoDownload,
DirectMusicSegment.SetAutoDownloadEnable

DirectMusicPerformance.SetMasterGrooveLevel

The **DirectMusicPerformance.SetMasterGrooveLevel** method sets a value to be added to all groove levels in the performance.

object.SetMasterGrooveLevel(*level* As Integer)

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

level

Value to add to the groove level, in the range from -99 through 99.

IDH_DirectMusicPerformance.SetMasterAutoDownload_dmusic_vb

IDH_DirectMusicPerformance.SetMasterGrooveLevel_dmusic_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_FAIL
DMUS_E_OUTOFMEMORY

See Also

DirectMusicPerformance.GetMasterGrooveLevel

Remarks

The adjusted groove level is always in the range from 1 through 100.

DirectMusicPerformance.SetMasterTempo

The **DirectMusicPerformance.SetMasterTempo** method sets a scaling factor that is applied to the tempo.

object.**SetMasterTempo**(*tempo* As Single)

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

tempo

Desired master tempo, in the range from 0.25 through 2.0.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

By default, the master tempo is 1. A value of 0.5 would halve the tempo, and a value of 2.0 would double it.

See Also

DirectMusicPerformance.GetMasterTempo

IDH_DirectMusicPerformance.SetMasterTempo_dmusic_vb

DirectMusicPerformance.SetMasterVolume

The **DirectMusicPerformance.SetMasterVolume** method adjusts the master volume of the performance.

object.**SetMasterVolume**(*vol* As Long)

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

vol

Master volume adjustment, in hundredths of a decibel.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The master volume is an amplification or attenuation factor applied to the default volume of the entire performance. The range of permitted values is determined by the port.

See Also

DirectMusicPerformance.GetMasterVolume

DirectMusicPerformance.SetNotificationHandle

The **DirectMusicPerformance.SetNotificationHandle** method sets the event handle for notifications. When signaled, the application should call the **DirectMusicPerformance.GetNotificationPMsg** method to retrieve the notification event.

object.**SetNotificationHandle**(*hnd* As Long)

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

IDH_DirectMusicPerformance.SetMasterVolume_dmusic_vb

IDH_DirectMusicPerformance.SetNotificationHandle_dmusic_vb

hnd

Event handle, or 0 to clear an existing handle.

See Also

DirectXEvent

DirectMusicPerformance.SetPort

The **DirectMusicPerformance.SetPort** method sets the active port for the performance. This method must be called after the performance is initialized and before any instruments are downloaded or any segment is played.

object.SetPort(index As Long, numGroups As Long)

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

index

Index of the port. Must be in the range from 1 through the value returned by **DirectMusicPerformance.GetPortCount**, or -1 for the default port.

numGroups

Number of channel groups on the port. Must be less than or equal to the number of channel groups reported in the **IMaxChannelGroups** member of the **DMUS_PORTCAPS** type returned by the **DirectMusicPerformance.GetPortCaps** method.

Remarks

Each channel group consists of 16 channels. Allocate enough channel groups to accommodate all performance channels in the segments that you intend to play. For MIDI files, one channel group is sufficient. The Microsoft Software Synthesizer supports up to 1000 channel groups.

DirectMusicPerformance.SetPrepareTime

The **DirectMusicPerformance.SetPrepareTime** method sets the approximate interval between the time at which messages are prepared and the time at which they are processed and heard.

object.SetPrepareTime(MilliSeconds As Long)

IDH_DirectMusicPerformance.SetPort_dmusic_vb

IDH_DirectMusicPerformance.SetPrepareTime_dmusic_vb

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

lMilliseconds

Prepare time, in milliseconds. The default value is 1000.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicPerformance.GetPrepareTime,
DirectMusicPerformance.SetBumperLength

DirectMusicPerformance.Stop

The **DirectMusicPerformance.Stop** method stops playback of one or more segments.

```
object.Stop(segment As DirectMusicSegment, _  
            segmentState As DirectMusicSegmentState, _  
            mtTime As Long, _  
            lFlags As Long)
```

Parameters

object

Object expression that resolves to a **DirectMusicPerformance** object.

segment

DirectMusicSegment to stop playing. All segment states based on this segment are stopped at *mtTime*. See Remarks.

segmentState

DirectMusicSegmentState object representing the instance of the segment to stop playing. See Remarks.

mtTime

Music time at which to stop the segment, segment state, or both. If the time is in the past or this value is 0, the requested segments and segment states stop playing immediately.

lFlags

Flag that indicates when the stop should occur. Boundaries are in relation to the current primary segment. Must be one of the following values:

0

IDH_DirectMusicPerformance.Stop_dmusic_vb

Stop immediately.

DMUS_SEGF_GRID

Stop on the next grid boundary at or after *mtTime*.

DMUS_SEGF_MEASURE

Stop on the next measure boundary at or after *mtTime*.

DMUS_SEGF_BEAT

Stop on the next beat boundary at or after *mtTime*.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

If *segment* and *segmentState* are both **Nothing**, all music stops, and all currently cued segments are released. If either *segment* or *segmentState* is not **Nothing**, only the requested segment states are removed from the performance.

If you set all parameters to **Nothing** or 0, everything stops immediately, and controller reset messages and note-off messages are sent to all mapped performance channels.

See Also

DirectMusicPerformance.PlaySegment, **CONST_DMUS_SEGF_FLAGS**

DirectMusicSegment

An object of the **DirectMusicSegment** class represents a single piece of music or a template.

Segments are usually loaded by calling **DirectMusicLoader.LoadSegment** and **DirectMusicLoader.LoadSegmentFromResource**. They can also be composed from musical elements by using methods of the **DirectMusicComposer** object, or created from existing segments by using the **DirectMusicSegment.Clone** method.

The methods of the **DirectMusicSegment** object can be grouped as follows:

Timing and looping

GetLength
GetLoopPointStart
GetLoopPointEnd
GetRepeats
GetStartPoint
SetLength
SetLoopPoints

IDH_DirectMusicSegment_dmusic_vb

	SetRepeats
	SetStartPoint
Duplication	Clone
Instruments	ConnectToCollection
	Download
	Unload
Parameters	SetAutoDownloadEnable
	SetStandardMidiFile
	SetTempoEnable
	SetTimeSigEnable

DirectMusicSegment.Clone

The **DirectMusicSegment.Clone** method creates a copy of all or part of the segment.

```
object.Clone(mtStart As Long, _
            mtEnd As Long) As DirectMusicSegment
```

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

mtStart

Start of the part to clone, in music time. If less than 0 or greater than the length of the segment, 0 is used.

mtEnd

End of the part to clone, in music time. If this value is past the end of the segment, the segment is cloned to the end. A value of 0 or anything less than *mtStart* also clones to the end.

Return Values

If the method succeeds, it returns a **DirectMusicSegment** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_OUTOFMEMORY.

IDH_DirectMusicSegment.Clone_dmusic_vb

Remarks

The start point and loop points set by the **DirectMusicSegment.SetStartPoint** and **DirectMusicSegment.SetLoopPoints** methods are set to their default values (0, and 0 to the end of the segment respectively) inside the clone. The number of repeats is also reset to 0.

For style-based segments, if *mtStart* is greater than 0, it should be on a measure boundary.

DirectMusicSegment.ConnectToCollection

The **DirectMusicSegment.ConnectToCollection** method associates a segment with a DLS instrument collection. This is the collection that is downloaded when the **DirectMusicSegment.Download** method is called.

object.**ConnectToCollection**(*c* As **DirectMusicCollection**)

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

c

DirectMusicCollection object representing instruments to be used in playing the segment.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_SET_UNSUPPORTED
DMUS_E_TRACK_NOT_FOUND

Remarks

By default, the General MIDI collection in the Gm.dls file is used. This method needs to be called only if the segment is to be played with custom instruments.

DirectMusicSegment.Download

The **DirectMusicSegment.Download** method downloads the collection associated with the segment so that the port can play the instruments.

IDH_DirectMusicSegment.ConnectToCollection_dmusic_vb

IDH_DirectMusicSegment.Download_dmusic_vb

object.Download(*performance* As DirectMusicPerformance)

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

performance

DirectMusicPerformance object to whose port the instruments are being downloaded.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicSegment.ConnectToCollection, **DirectMusicSegment.Unload**,
DirectMusicSegment.SetAutoDownloadEnable

DirectMusicSegment.GetLength

The **DirectMusicSegment.GetLength** method retrieves the length of the segment.

object.GetLength() As Long

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

Return Values

If the method succeeds, it returns the length of the segment, in music time.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

If for some reason the segment's length was never set, the method returns 0.

IDH_DirectMusicSegment.GetLength_dmusic_vb

See Also

DirectMusicSegment.SetLength

DirectMusicSegment.GetLoopPointEnd

The **DirectMusicSegment.GetLoopPointEnd** method retrieves the point in the segment at which a repeating section is to end.

object.**GetLoopPointEnd()** As Long

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

Return Values

If the method succeeds, it returns the end point of the loop, in music time. If this value is 0, the entire segment loops.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The section does not repeat unless the number of repetitions has been set to 1 or more by using the **DirectMusicSegment.SetRepeats** method. By default, the entire segment repeats.

See Also

DirectMusicSegment.GetLoopPointStart, **DirectMusicSegment.SetLoopPoints**, **DirectMusicSegment.GetRepeats**

DirectMusicSegment.GetLoopPointStart

The **DirectMusicSegment.GetLoopPointStart** method retrieves the point in the segment at which a repeating section is to start.

object.**GetLoopPointStart()** As Long

IDH_DirectMusicSegment.GetLoopPointEnd_dmusic_vb

IDH_DirectMusicSegment.GetLoopPointStart_dmusic_vb

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

Return Values

If the method succeeds, it returns the start point of the loop, in music time.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The section does not repeat unless the number of repetitions has been set to 1 or more by using the **DirectMusicSegment.SetRepeats** method. By default, the entire segment repeats.

See Also

DirectMusicSegment.GetLoopPointEnd, **DirectMusicSegment.SetLoopPoints**, **DirectMusicSegment.GetRepeats**

DirectMusicSegment.GetRepeats

The **DirectMusicSegment.GetRepeats** method retrieves the number of times that the looping portion of a segment is set to repeat.

object.**GetRepeats()** As Long

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

Return Values

If the method succeeds, it returns the number of times that the looping portion repeats.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

IDH_DirectMusicSegment.GetRepeats_dmusic_vb

See Also

DirectMusicSegment.SetRepeats, **DirectMusicSegment.SetLoopPoints**,
DirectMusicSegment.GetLoopPointStart,
DirectMusicSegment.GetLoopPointEnd, **DirectMusicSegmentState.GetRepeats**

DirectMusicSegment.GetStartPoint

The **DirectMusicSegment.GetStartPoint** method retrieves the point at which the segment starts playing in response to the **DirectMusicPerformance.PlaySegment** method.

object.**GetStartPoint()** As Long

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

Return Values

If the method succeeds, it returns the start point of the segment, in music time.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicSegment.SetStartPoint

DirectMusicSegment.SetAutoDownloadEnable

The **DirectMusicSegment.SetAutoDownloadEnable** method enables or disables automatic downloading and unloading of instruments in the collection associated with the segment.

object.**SetAutoDownloadEnable(b As Boolean)**

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

IDH_DirectMusicSegment.GetStartPoint_dmusic_vb

IDH_DirectMusicSegment.SetAutoDownloadEnable_dmusic_vb

b

True to enable autodownloading, or **False** to disable it.

Remarks

Automatic downloading is disabled by default. When it is enabled, instruments are automatically downloaded when the segment is played, and unloaded when it is stopped.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_SET_UNSUPPORTED
DMUS_E_TRACK_NOT_FOUND

See Also

DirectMusicSegment.Download

DirectMusicSegment.SetLength

The **DirectMusicSegment.SetLength** method sets the length of the segment.

object.SetLength(mtLength As Long)

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

mtLength

Desired length, in music time. Must be greater than 0.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_OUT_OF_RANGE.

Remarks

In most cases, applications do not need to set the length, which is automatically set when the segment is loaded. However, this method can be used to shorten a segment.

IDH_DirectMusicSegment.SetLength_dmusic_vb

See Also

DirectMusicSegment.GetLength

DirectMusicSegment.SetLoopPoints

The **DirectMusicSegment.SetLoopPoints** method sets the start and end points inside the segment to repeat the number of times set by the **DirectMusicSegment.SetRepeats** method.

object.SetLoopPoints(mtStart As Long, mtEnd As Long)

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

mtStart

Point at which to begin the loop, in music time.

mtEnd

Point at which to end the loop, in music time. A value of 0 loops the entire segment.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to **DMUS_E_OUT_OF_RANGE**.

Remarks

When the segment is played, it plays from the segment start time until *mtEnd*, then loops to *mtStart*, plays the looped portion the number of times set by **DirectMusicSegment.SetRepeats**, then plays to the end.

The default values are set to loop the entire segment from beginning to end.

The method fails if *mtStart* is greater than or equal to the length of the segment, or if *mtEnd* is greater than the length of the segment. If *mtEnd* is 0, *mtStart* must be 0 as well.

This method does not affect any currently playing segment states created from this segment.

A segment that is reused might be loaded from an internal cache, in which case it has the same loop points that it had the last time these values were set. It is a good idea to reset the loop points to 0 before releasing or resetting the object.

IDH_DirectMusicSegment.SetLoopPoints_dmusic_vb

See Also

DirectMusicSegment.GetLoopPointStart,
DirectMusicSegment.GetLoopPointEnd, **DirectMusicSegment.SetRepeats**

DirectMusicSegment.SetRepeats

The **DirectMusicSegment.SetRepeats** method sets the number of times that the looping portion of the segment is to repeat.

*object.SetRepeats(*lRepeats* As Long)*

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

lRepeats

Number of repetitions.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicSegment.GetRepeats, **DirectMusicSegment.SetLoopPoints**

DirectMusicSegment.SetStandardMidiFile

The **DirectMusicSegment.SetStandardMidiFile** method informs DirectMusic that the segment is based on a standard MIDI file, not one authored specifically for DirectMusic. Calling this method ensures that certain events are handled properly when the segment is played.

object.SetStandardMidiFile()

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

IDH_DirectMusicSegment.SetRepeats_dmusic_vb

IDH_DirectMusicSegment.SetStandardMidiFile_dmusic_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

The method should be called before instruments are downloaded.

DirectMusicSegment.SetStartPoint

The **DirectMusicSegment.SetStartPoint** method sets the point at which the segment starts playing in response to a call to the **DirectMusicPerformance.PlaySegment** method.

object.SetStartPoint(*mtStart* As Long)

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

mtStart

Point within the segment at which it is to start playing, in music time. If this value is less than 0 or greater than the length of the segment, the start point is set to 0.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to **DMUS_E_OUT_OF_RANGE**.

Remarks

By default, the start point is 0, meaning that the segment starts from the beginning.

The call fails if *mtStart* is greater than or equal to the length of the segment.

The method does not affect any currently playing segment states created from this segment.

A segment that is reused might be loaded from an internal cache, in which case it has the same start point that it had the last time that this value was set. It is a good idea to set the start point to 0 before the object is released or reset.

See Also

DirectMusicSegment.GetStartPoint, **DirectMusicSegmentState.GetStartPoint**, **DirectMusicSegment.SetLoopPoints**

IDH_DirectMusicSegment.SetStartPoint_dmusic_vb

DirectMusicSegment.SetTempoEnable

The **DirectMusicSegment.SetTempoEnable** method enables or disables tempo messages for the segment.

object.SetTempoEnable(*b* As Boolean)

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

b

True to enable tempo messages, or False to disable them.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_SET_UNSUPPORTED

DMUS_E_TRACK_NOT_FOUND

DirectMusicSegment.SetTimeSigEnable

The **DirectMusicSegment.SetTimeSigEnable** method enables or disables time signature messages for the segment.

object.SetTempoEnable(*b* As Boolean)

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

b

True to enable time signature messages, or False to disable them.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to one of the following values:

DMUS_E_SET_UNSUPPORTED

DMUS_E_TRACK_NOT_FOUND

IDH_DirectMusicSegment.SetTempoEnable_dmusc_vb

IDH_DirectMusicSegment.SetTimeSigEnable_dmusc_vb

DirectMusicSegment.Unload

The **DirectMusicSegment.Unload** method unloads instruments that were downloaded to the port by the **DirectMusicSegment.Download** method.

object.Unload(performance As DirectMusicPerformance)

Parameters

object

Object expression that resolves to a **DirectMusicSegment** object.

performance

DirectMusicPerformance object from whose port the instruments are being unloaded.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicSegment.SetAutoDownloadEnable

DirectMusicSegmentState

The **DirectMusicSegmentState** class is used to get information about a segment instance. When the **DirectMusicPerformance.PlaySegment** method is called, it creates a **DirectMusicSegmentState** object that represents that instance of the segment and enables the application to retrieve information about it. The object can also be passed to methods of **DirectMusicPerformance** to determine whether a segment instance is still playing, or to stop it.

The class has the following methods:

Information

GetRepeats
GetSeek
GetSegment
GetStartPoint
GetStartTime

IDH_DirectMusicSegment.Unload_dmusic_vb

IDH_DirectMusicSegmentState_dmusic_vb

See Also

DirectMusicPerformance.GetSegmentState, **DirectMusicPerformance.IsPlaying**, **DirectMusicPerformance.Stop**

DirectMusicSegmentState.GetRepeats

The **DirectMusicSegmentState.GetRepeats** method returns the number of times that the looping portion of the segment is set to repeat.

object.**GetRepeats()** As Long

Parameters

object

Object expression that resolves to a **DirectMusicSegmentState** object.

Return Values

If the method succeeds, it returns the repeat count. A value of 0 indicates that the segment is to play through only once, with no portion repeated.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicSegment.SetRepeats

DirectMusicSegmentState.GetSeek

The **DirectMusicSegmentState.GetSeek** method retrieves the current seek pointer in the segment state. This is immediately after the last point in the segment for which messages have been generated.

object.**GetSeek()** As Long

Parameters

object

Object expression that resolves to a **DirectMusicSegmentState** object.

IDH_DirectMusicSegmentState.GetRepeats_dmusic_vb

IDH_DirectMusicSegmentState.GetSeek_dmusic_vb

Return Values

If the method succeeds, it returns the current seek pointer, in music time.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Remarks

When a segment is stopped, messages that have been sent but not yet queued to the port buffer are flushed. Therefore, if you stop a segment and then restart it at the last seek pointer, some notes are lost.

DirectMusicSegmentState.GetSegment

The **DirectMusicSegmentState.GetSegment** method returns an object representing the segment that owns this segment state.

object.GetSegment() As DirectMusicSegment

Parameters

object

Object expression that resolves to a **DirectMusicSegmentState** object.

Return Values

If the method succeeds, it returns a **DirectMusicSegment** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

DirectMusicSegmentState.GetStartPoint

The **DirectMusicSegmentState.GetStartPoint** method returns the offset into the segment at which play began or will begin.

object.GetStartPoint() As Long

Parameters

object

IDH_DirectMusicSegmentState.GetSegment_dmusic_vb

IDH_DirectMusicSegmentState.GetStartPoint_dmusic_vb

Object expression that resolves to a **DirectMusicSegmentState** object.

Return Values

If the method succeeds, it returns the start point for this segment state, in music time. This might not be same value as is returned by **DirectMusicSegment.GetStartPoint** if the start point of the segment has been changed since this segment state was created. Different instances of a playing segment can have different start points.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicSegment.SetStartPoint, **DirectMusicSegmentState.GetStartTime**

DirectMusicSegmentState.GetStartTime

The **DirectMusicSegmentState.GetStartTime** method gets the performance time at which the segment started or will start playing.

object.GetStartTime() As Long

Parameters

object

Object expression that resolves to a **DirectMusicSegmentState** object.

Return Values

If the method succeeds, it returns the start time, in music time, of this instance of the segment.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to **DMUS_E_BUFFER_EMPTY**.

Remarks

If the segment was started from some point other than the beginning, you can retrieve the time at which the beginning of the segment would have fallen by subtracting the time returned by **DirectMusicSegmentState.GetStartPoint** from the value returned by this method.

IDH_DirectMusicSegmentState.GetStartTime_dmusic_vb

See Also

DirectMusicSegment.SetStartPoint, **DirectMusicSegment.GetStartPoint**,
DirectMusicSegmentState.GetStartPoint

DirectMusicStyle

An object of the **DirectMusicStyle** class provides the performance with the information that it needs to play musical patterns. Styles usually include bands and motifs, and can include chord maps, so the **DirectMusicStyle** object also provides methods for accessing these objects.

The object is generally obtained by using the **DirectMusicLoader.LoadStyle** or the **DirectMusicLoader.LoadStyleFromResource** method. It can also be obtained from the performance by using the **DirectMusicPerformance.GetStyle** method, provided the current control segment is based on a style.

The methods of the **DirectMusicStyle** class can be organized in the following groups:

Bands	GetBand
	GetBandCount
	GetBandName
	GetDefaultBand
Motifs	GetMotif
	GetMotifCount
	GetMotifName
Time	GetTempo
	GetTimeSignature

DirectMusicStyle.GetBand

The **DirectMusicStyle.GetBand** method gets a band object by name.

object.**GetBand**(*name* As String) As DirectMusicBand

Parameters

object

Object expression that resolves to a **DirectMusicStyle** object.

name

Name assigned to the band by the author of the style.

IDH_DirectMusicStyle_dmusic_vb

IDH_DirectMusicStyle.GetBand_dmusic_vb

Return Values

If the method succeeds, it returns a **DirectMusicBand** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** can be set to DMUS_E_FAIL.

See Also

DirectMusicStyle.GetBandName, **DirectMusicStyle.GetDefaultBand**

DirectMusicStyle.GetBandCount

The **DirectMusicStyle.GetBandCount** method gets the number of bands available in the style.

object.**GetBandCount()** As Long

Parameters

object

Object expression that resolves to a **DirectMusicStyle** object.

Return Values

If the method succeeds, it returns the number of bands in the style.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicStyle.GetBand, **DirectMusicStyle.GetBandName**

DirectMusicStyle.GetBandName

The **DirectMusicStyle.GetBandName** method gets the name of a band in the style.

object.**GetBandName(index As Long)** As String

IDH_DirectMusicStyle.GetBandCount_dmusic_vb

IDH_DirectMusicStyle.GetBandName_dmusic_vb

Parameters

object

Object expression that resolves to a **DirectMusicStyle** object.

index

Index of the band in the style, in the range from 1 through **DirectMusicStyle.GetBandCount**.

Return Values

If the method succeeds, it returns the name assigned to the band by the author of the style.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicStyle.GetBand

DirectMusicStyle.GetDefaultBand

The **DirectMusicStyle.GetDefaultBand** method gets the default band for the style .

object.GetDefaultBand() As **DirectMusicBand**

Parameters

object

Object expression that resolves to a **DirectMusicStyle** object.

name

Name assigned to the band by the author of the style.

Return Values

If the method succeeds, it returns a **DirectMusicBand** object.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicStyle.GetBand

IDH_DirectMusicStyle.GetDefaultBand_dmusic_vb

DirectMusicStyle.GetMotif

The **DirectMusicStyle.GetMotif** method creates a segment containing the named motif.

object.**GetMotif**(*name* As String) As DirectMusicSegment

Parameters

object

Object expression that resolves to a **DirectMusicStyle** object.

name

Name assigned to the motif by the author of the style

Return Values

If the method succeeds, it returns a **DirectMusicSegment** object representing the motif.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicStyle.GetMotifCount, **DirectMusicStyle.GetMotifName**

DirectMusicStyle.GetMotifCount

The **DirectMusicStyle.GetMotifCount** method gets the number of motifs available in the style.

object.**GetMotifCount**() As Long

Parameters

object

Object expression that resolves to a **DirectMusicStyle** object.

Return Values

If the method succeeds, it returns the number of motifs in the style.

IDH_DirectMusicStyle.GetMotif_dmusic_vb

IDH_DirectMusicStyle.GetMotifCount_dmusic_vb

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

DirectMusicStyle.GetMotifName

The **DirectMusicStyle.GetMotifName** method gets the name of a motif in the style.

object.**GetMotifName**(*index* As Long) As String

Parameters

object

Object expression that resolves to a **DirectMusicStyle** object.

index

Index of the motif in the style, in the range from 1 through **DirectMusicStyle.GetMotifCount**.

Return Values

If the method succeeds, it returns the name assigned to the motif by the author of the style.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

See Also

DirectMusicStyle.GetMotif

DirectMusicStyle.GetTempo

The **DirectMusicStyle.GetTempo** method retrieves the recommended tempo of the style.

object.**GetTempo**() As Double

Parameters

object

Object expression that resolves to a **DirectMusicStyle** object.

IDH_DirectMusicStyle.GetMotifName_dmusic_vb

IDH_DirectMusicStyle.GetTempo_dmusic_vb

Return Values

If the method succeeds, it returns the recommended tempo, in beats per minute.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

DirectMusicStyle.GetTimeSignature

The **DirectMusicStyle.GetTimeSignature** method retrieves the style's time signature.

object.GetTimeSignature(*pTimeSig* As **DMUS_TIMESIGNATURE**)

Parameters

object

Object expression that resolves to a **DirectMusicStyle** object.

pTimeSig

DMUS_TIMESIGNATURE type to receive information about the time signature.

Error Codes

If the method fails, it raises an error, and **Err.Number** is set.

Types

This section contains information on the following types used in DirectMusic for Visual Basic:

- **DMUS_CURVE_PMSG**
- **DMUS_NOTE_PMSG**
- **DMUS_NOTIFICATION_PMSG**
- **DMUS_PORTCAPS**
- **DMUS_TIMESIGNATURE**

DMUS_CURVE_PMSG

The **DMUS_CURVE_PMSG** type contains information about a MIDI curve message.

IDH_DirectMusicStyle.GetTimeSignature_dmusic_vb

IDH_DMUS_CURVE_PMSG_dmusic_vb

Type DMUS_CURVE_PMSG

beat As Byte

ccData As Byte

curveShape As Byte

endValue As Integer

flags As Byte

grid As Byte

measure As Integer

mtDuration As Long

mtOriginalStart As Long

mtResetDuration As Long

offset As Integer

resetValue As Integer

startValue As Integer

type As Byte

End Type

Members

beat

Beat count (within a measure) at which this curve occurs.

ccData

CC number if this is a control change type.

curveShape

Shape of the curve. This can be one of the values from the **CONST_DMUS_CURVES** enumeration.

endValue

End value of the curve.

flags

Set to DMUS_CURVE_RESET if the **resetValue** must be set when the time is reached or an invalidation occurs because of a transition. If 0, the curve stays permanently at the new value.

grid

Grid offset from a beat at which this curve occurs.

measure

Measure in which this curve occurs.

mtDuration

How long the curve lasts.

mtOriginalStart

Original start time, in music time. Must be set to either 0 when this message is created, or to the original time of the curve.

mtResetDuration

How long after the curve is finished until the reset value is set, in music time.

offset

Offset from a grid at which this curve occurs, in music time.

resetValue

Reset value of the curve, set after **mtResetDuration** or upon a flush or invalidation.

startValue

Start value of the curve.

type

Type of curve. This can be one of the values from the **CONST_DMUS_CURVET** enumeration.

See Also

DirectMusicPerformance.SendCurvePMSG

DMUS_NOTE_PMSG

The **DMUS_NOTE_PMSG** type contains data for a music note event.

Type DMUS_NOTE_PMSG

beat As Byte

durRange As Byte

flags As Byte

grid As Byte

measure As Integer

midiValue As Byte

mtDuration As Long

musicValue As Integer

offset As Integer

playModeFlags As Byte

subChordLevel As Byte

timeRange As Byte

velocity As Byte

velRange As Byte

End Type

Members

beat

Beat count (within a measure) at which this note occurs.

durRange

Range to randomize duration.

flags

Should be set to **DMUS_NOTEF_NOTEON**. See Remarks.

IDH_DMUS_NOTE_PMSG_dmusic_vb

grid

Grid offset from a beat at which this note occurs.

measure

Measure in which this note occurs.

midiValue

MIDI note value, converted from **musicValue**.

mtDuration

Duration of the note.

musicValue

Description of the note. In most play modes, this is a packed array of four-bit values, as follows:

Octave, in the range from –2 through 14. The note is transposed up or down by the octave times 12.

Chord position, in the range from 0 through 15, though it should never be above 3. The first position in the chord is 0.

Scale position, in the range from 0 through 15. Typically, it is from 0 through 2, but it is possible to have a one-note chord and have everything above the chord be interpreted as a scale position.

Accidental, in the range from –8 through 7, but typically in the range from –2 through 2. This represents an offset that takes the note out of the scale.

In the fixed play modes, the music value is a MIDI note value in the range from 0 through 127.

offset

Offset from a grid at which this note occurs, in music time.

playModeFlags

Play mode determining how the music value is related to the chord and subchord. For a list of values, see **CONST_DMUS_PLAYMODE_FLAGS**.

subChordLevel

Subchord level that the note uses.

timeRange

Range by which to randomize time.

velocity

Note velocity.

velRange

Range by which to randomize velocity.

Remarks

Applications sending note messages must provide values only in the **flags**, **midiValue**, **mtDuration**, and **velocity** members. The other members are of interest only to tools, which are not supported in DirectX for Visual Basic.

Normally, the application sets **flags** to **DMUS_NOTEF_NOTEON**. The application is not responsible for sending note-off messages. When the DirectMusic output tool

receives a **DMUS_NOTE_PMSG** and sees that **DMUS_NOTEF_NOTEON** is set, it clears the flag, adds **mtDuration** to the time stamp, and requeues the message so that the note is turned off at the appropriate time.

It is possible, however, for the application to stop a note prematurely by sending the same note on the same channel with **flags** set to 0.

See Also

DirectMusicPerformance.SendNotePMSG, Music Values and MIDI Notes

DMUS_NOTIFICATION_PMSG

The **DMUS_NOTIFICATION_PMSG** type contains information about a notification message sent by the performance.

Type **DMUS_NOTIFICATION_PMSG**

ctTime As Long

IField1 As Long

IField2 As Long

INotificationOption As Long

INotificationType As Long

mtTime As Long

End Type

Members

ctTime

Time stamp of the message, in clock time.

IField1

Extra data specific to the type of notification. For **DMUS_NOTIFY_ON_MEASUREANDBEAT** notifications, this member returns the beat number within the measure.

IField2

Extra data specific to the type of notification. Reserved for future or application-defined use.

INotificationOption

Identifier of the notification subtype, from the **CONST_DMUS_NOTIFICATION_SUBTYPE** enumeration.

If the notification type is **DMUS_NOTIFY_ON_SEGMENT**, this member can contain one of the following values:

DMUS_NOTIFICATION_SEGABORT

The segment was stopped by **DirectMusicPerformance.Stop**.

DMUS_NOTIFICATION_SEGALMOSTEND

IDH_DMUS_NOTIFICATION_PMSG_dmusic_vb

The segment has reached the end minus the prepare time.

DMUS_NOTIFICATION_SEGEND

The segment has ended.

DMUS_NOTIFICATION_SEGLOOP

The segment has looped.

DMUS_NOTIFICATION_SEGSTART

The segment has started.

If the notification type is DMUS_NOTIFY_ON_COMMAND, this member can contain one of the following values:

DMUS_NOTIFICATION_GROOVE

Groove change.

DMUS_NOTIFICATION_EMBELLISHMENT

Embellishment command (intro, fill, break, or end).

If the notification type is DMUS_NOTIFY_ON_PERFORMANCE, this member can contain one of the following values:

DMUS_NOTIFICATION_MUSICSTARTED

Playback has started.

DMUS_NOTIFICATION_MUSICSTOPPED

Playback has stopped.

If the notification type is DMUS_NOTIFY_ON_MEASUREANDBEAT, this member contains DMUS_NOTIFICATION_MEASUREBEAT. No other subtypes are defined.

If the notification type is DMUS_NOTIFY_ON_CHORD, this member contains DMUS_NOTIFICATION_CHORD. No other subtypes are defined.

INotificationType

Identifier of the notification type, from the

CONST_DMUS_NOTIFICATION_TYPE enumeration.

mtTime

Time stamp of the message, in music time.

See Also

DirectMusicPerformance.AddNotificationType,
DirectMusicPerformance.GetNotificationPMSG

DMUS_PORTCAPS

The **DMUS_PORTCAPS** type returns information about the capabilities of a port. It is passed to the **DirectMusicPerformance.GetPortCaps** method.

Type DMUS_PORTCAPS

IClass as Long

IDH_DMUS_PORTCAPS_dmusic_vb

```

IEffectFlags As Long
IFlags As Long
IMaxAudioChannels As Long
IMaxChannelGroups As Long
IMaxVoices As Long
IMemorySize As Long
IType As Long
End Type

```

Members

IClass

Class of this port. One of the members of the **CONST_DMUS_PC_CLASS** enumeration.

IEffectFlags

Flags from the **CONST_DMUS_EFFECT_FLAGS** enumeration indicating what audio effects are available on the port.

IFlags

Flags describing various capabilities of the port. See **CONST_DMUS_PC_FLAGS**.

IMaxAudioChannels

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.

IMaxChannelGroups

Maximum number of channel groups supported by this port. A channel group is a set of 16 MIDI channels.

IMaxVoices

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.

IMemorySize

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 field contains **DMUS_PC_SYSTEMMEMORY**.

IType

Type of this port. See **CONST_DMUS_PORT_TYPE**.

DMUS_TIMESIGNATURE

The **DMUS_TIMESIGNATURE** type contains information about a time signature. It is passed to the **DirectMusicPerformance.GetTimeSig** and **DirectMusicStyle.GetTimeSignature** methods, and is also used in messages sent by the **DirectMusicPerformance.SendTimeSigPMSG** method.

```
# IDH_DMUS_TIMESIGNATURE_dmusic_vb
```

Type DMUS_TIMESIGNATURE
 beat As Byte
 beatsPerMeasure As Byte
 gridsPerBeat As Integer
 mtTime As Long
End Type

Members

beat

Bottom of time signature.

beatsPerMeasure

Top of time signature.

gridsPerBeat

Grids (subdivisions) per beat. This value determines the timing resolution for certain music events—for example, segments cued with the DMUS_SEGF_GRID flag (see **CONST_DMUS_SEGF_FLAGS**).

mtTime

Music time at which this time signature occurs.

Enumerations

DirectMusic for Visual Basic uses enumerations to group constants to take advantage of the statement completion feature of the Microsoft® Visual Studio® development environment.

This section contains reference information for the following enumerations:

- **CONST_DMUS**
- **CONST_DMUS_COMMANDT_TYPES**
- **CONST_DMUS_COMPOSEF_FLAGS**
- **CONST_DMUS_CURVE_FLAGS**
- **CONST_DMUS_CURVES**
- **CONST_DMUS_CURVET**
- **CONST_DMUS_EFFECT_FLAGS**
- **CONST_DMUS_NOTEF_FLAGS**
- **CONST_DMUS_NOTIFICATION_SUBTYPE**
- **CONST_DMUS_NOTIFICATION_TYPE**
- **CONST_DMUS_PC_CLASS**
- **CONST_DMUS_PC_FLAGS**
- **CONST_DMUS_PLAYMODE_FLAGS**
- **CONST_DMUS_PMSGF_FLAGS**

- **CONST_DMUS_PORT_TYPE**
- **CONST_DMUS_SEGF_FLAGS**
- **CONST_DMUS_SHAPET_TYPES**
- **CONST_DMUSERR**

CONST_DMUS

The **CONST_DMUS** enumeration contains miscellaneous constants used in **DirectMusic**.

```
Enum CONST_DMUS
    DMUS_MAXSUBCHORD = 8
    DMUS_TEMPO_MAX = 350 (&H15E)
    DMUS_TEMPO_MIN = 10
End Enum
```

DMUS_MAXSUBCHORD
Maximum number of subchords allowed in a chord.

DMUS_TEMPO_MAX
Maximum tempo, in beats per minute.

DMUS_TEMPO_MIN
Minimum tempo, in beats per minute.

CONST_DMUS_COMMANDT_TYPES

The members of the **CONST_DMUS_COMMANDT_TYPES** enumeration represent commands that establish musical patterns. They are used in the *lCommand* parameter of the **DirectMusicComposer.AutoTransition** and **DirectMusicComposer.ComposeTransition** methods and are returned by the **DirectMusicPerformance.GetCommand** method.

```
Type CONST_DMUS_COMMANDT_TYPES
    DMUS_COMMANDT_BREAK      = 3
    DMUS_COMMANDT_END        = 4
    DMUS_COMMANDT_ENDANDINTRO = 5
    DMUS_COMMANDT_FILL       = 1
    DMUS_COMMANDT_GROOVE     = 0
    DMUS_COMMANDT_INTRO      = 2
End Enum
```

IDH_CONST_DMUS_dmusic_vb

IDH_CONST_DMUS_COMMANDT_TYPES_dmusic_vb

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 intro.

DMUS_COMMANDT_FILL

The command is a fill.

DMUS_COMMANDT_GROOVE

The command is a groove command.

DMUS_COMMANDT_INTRO

The command is an intro.

CONST_DMUS_COMPOSEF_FLAGS

The members of the **CONST_DMUS_COMPOSEF_FLAGS** enumeration are used in the **lFlags** parameter of the **DirectMusicComposer.AutoTransition** and **DirectMusicComposer.ComposeTransition** methods.

Enum CONST_DMUS_COMPOSEF_FLAGS

DMUS_COMPOSEF_AFTERPREPARETIME	= 64 (&H40)
DMUS_COMPOSEF_ALIGN	= 1
DMUS_COMPOSEF_BEAT	= 16 (&H10)
DMUS_COMPOSEF_GRID	= 8
DMUS_COMPOSEF_IMMEDIATE	= 4
DMUS_COMPOSEF_LONG	= 8192 (&H2000)
DMUS_COMPOSEF_MEASURE	= 32 (&H20)
DMUS_COMPOSEF_MODULATE	= 4096 (&H1000)
DMUS_COMPOSEF_NONE	= 0
DMUS_COMPOSEF_OVERLAP	= 2

End Enum

DMUS_COMPOSEF_AFTERPREPARETIME

AutoTransition only. Use the DMUS_SEGF_AFTERPREPARETIME flag (see **CONST_DMUS_SEGF_FLAGS**) when cuing the transition.

DMUS_COMPOSEF_ALIGN

Align transition to the time signature of the currently playing segment. Not currently implemented.

DMUS_COMPOSEF_BEAT

AutoTransition only. Start transition on a beat boundary.

DMUS_COMPOSEF_GRID

IDH_CONST_DMUS_COMPOSEF_FLAGS_dmusic_vb

AutoTransition only. Start transition on a grid boundary.

DMUS_COMPOSEF_IMMEDIATE

AutoTransition only. Start transition on a music or clock time boundary.

DMUS_COMPOSEF_MEASURE

AutoTransition only. Start transition on a measure boundary.

DMUS_COMPOSEF_LONG

Composes a long transition. If this flag is not set, the length of the transition is at most one measure unless the *lCommand* parameter of **ComposeTransition** or **AutoTransition** specifies an ending and the style contains an ending of greater than one measure. If this flag is set, the length of the transition increases by one measure.

DMUS_COMPOSEF_MODULATE

Compose a transition that modulates smoothly from *fromSeg* to *toSeg*, using the chord of *toSeg*.

DMUS_COMPOSEF_NONE

No flags. By default, the transition starts on a measure boundary.

DMUS_COMPOSEF_OVERLAP

Overlap the transition into *toSeg*. Not currently implemented.

CONST_DMUS_CURVE_FLAGS

The **CONST_DMUS_CURVE_FLAGS** enumeration contains a single constant used in the **DMUS_CURVE_PMSG** type.

```
Enum CONST_DMUS_CURVE_FLAGS
    DMUS_CURVE_RESET = 1
End Enum
```

DMUS_CURVE_RESET

The reset value must be set when the time is reached or an invalidation occurs because of a transition.

CONST_DMUS_CURVES

Members of the **CONST_DMUS_CURVES** enumeration are used to define the shape of a curve in the **DMUS_CURVE_PMSG** type.

```
Enum CONST_DMUS_CURVES
    DMUS_CURVES_EXP = 2
    DMUS_CURVES_INSTANT = 1
    DMUS_CURVES_LINEAR = 0
    DMUS_CURVES_LOG = 3
```

IDH_CONST_DMUS_CURVE_FLAGS_dmusic_vb

IDH_CONST_DMUS_CURVES_dmusic_vb

```

    DMUS_CURVES_SINE = 4
End Enum

```

```

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.

```

CONST_DMUS_CURVET

Members of the **CONST_DMUS_CURVET** enumeration are used to describe the type of curve in the **DMUS_CURVE_PMSG** type.

```

Enum CONST_DMUS_CURVET
    DMUS_CURVET_CCCURVE = 4
    DMUS_CURVET_MATCURVE = 5
    DMUS_CURVET_PATCURVE = 6
    DMUS_CURVET_PBCURVE = 3
End Enum

```

```

DMUS_CURVET_CCCURVE
    Continuous controller curve (MIDI Control Change channel voice message;
    status byte &HBn, where n 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 &HEn).

```

```

# IDH_CONST_DMUS_CURVET_dmusic_vb

```

CONST_DMUS_EFFECT_FLAGS

Members of the **CONST_DMUS_EFFECT_FLAGS** enumeration are used in the **DMUS_PORTCAPS** type to specify which effects are supported by a port.

```
Enum CONST_DMUS_EFFECT_FLAGS
```

```
    DMUS_EFFECT_CHORUS = 2
```

```
    DMUS_EFFECT_NONE  = 0
```

```
    DMUS_EFFECT_REVERB = 1
```

```
End Enum
```

```
DMUS_EFFECT_CHORUS
```

The port supports chorus.

```
DMUS_EFFECT_NONE
```

The port does not support any effects.

```
DMUS_EFFECT_REVERB
```

The port supports reverb.

CONST_DMUS_NOTEF_FLAGS

The **CONST_DMUS_NOTEF_FLAGS** enumeration contains a single constant used in note messages.

```
Enum CONST_DMUS_NOTEF_FLAGS
```

```
    DMUS_NOTEF_NOTEON = 1
```

```
End Enum
```

```
DMUS_NOTEF_NOTEON
```

See the Remarks for **DMUS_NOTE_PMSG**.

CONST_DMUS_NOTIFICATION_SUBTYPE

The members of the **CONST_DMUS_NOTIFICATION_SUBTYPE** enumeration provide information about the musical events reported in notification messages.

```
Enum CONST_DMUS_NOTIFICATION_SUBTYPE
```

```
    DMUS_NOTIFICATION_CHORD = 0
```

```
    DMUS_NOTIFICATION_EMBELLISHMENT = 1
```

```
    DMUS_NOTIFICATION_GROOVE = 0
```

```
    DMUS_NOTIFICATION_MEASUREBEAT = 0
```

```
    DMUS_NOTIFICATION_MUSICSTARTED = 0
```

```
    DMUS_NOTIFICATION_MUSICSTOPPED = 1
```

```
# IDH_CONST_DMUS_EFFECT_FLAGS_dmusic_vb
```

```
# IDH_CONST_DMUS_NOTEF_FLAGS_dmusic_vb
```

```
# IDH_CONST_DMUS_NOTIFICATION_SUBTYPE_dmusic_vb
```

```

DMUS_NOTIFICATION_SEGABORT = 4
DMUS_NOTIFICATION_SEGALMOSTEND = 2
DMUS_NOTIFICATION_SEGEND = 1
DMUS_NOTIFICATION_SEGLOOP = 3
DMUS_NOTIFICATION_SEGSTART = 0
End Enum

```

For an explanation of the values, see **DMUS_NOTIFICATION_PMSG**.

CONST_DMUS_NOTIFICATION_TYPE

The members of the **CONST_DMUS_NOTIFICATION_TYPE** enumeration identify a notification type. They are passed to the **DirectMusicPerformance.AddNotificationType** and **DirectMusicPerformance.RemoveNotificationType** methods and identify the notification type in messages retrieved by **DirectMusicPerformance.GetNotificationPMSG**.

```

Enum CONST_DMUS_NOTIFICATION_TYPE
    DMUS_NOTIFY_ON_CHORD = 1
    DMUS_NOTIFY_ON_COMMAND = 2
    DMUS_NOTIFY_ON_MEASUREANDBEAT = 3
    DMUS_NOTIFY_ON_PERFORMANCE = 4
    DMUS_NOTIFY_ON_SEGMENT = 5
End Enum

```

DMUS_NOTIFY_ON_CHORD

Chord change.

DMUS_NOTIFY_ON_COMMAND

Command event.

DMUS_NOTIFY_ON_MEASUREANDBEAT

Measure and beat event.

DMUS_NOTIFY_ON_PERFORMANCE

Performance event. When this value is found in the **INotificationType** member of a **DMUS_NOTIFICATION_PMSG** type, the event is further defined in the **INotificationOption** member.

DMUS_NOTIFY_ON_SEGMENT

Segment event. When this value is found in the **INotificationType** of a **DMUS_NOTIFICATION_PMSG** type, the event is further defined in the **INotificationOption** member.

IDH_CONST_DMUS_NOTIFICATION_TYPE_dmusic_vb

See Also**DMUS_NOTIFICATION_PMSG****CONST_DMUS_PC_CLASS**

Members of the **CONST_DMUS_PC_CLASS** enumeration are used in the **DMUS_PORTCAPS** type to specify the class of the port.

```
Enum CONST_DMUS_PC_CLASS
    DMUS_PC_INPUTCLASS = 0
    DMUS_PC_OUTPUTCLASS = 1
End Enum
```

DMUS_PC_INPUTCLASS

Input port.

DMUS_PC_OUTPUTCLASS

Output port.

CONST_DMUS_PC_FLAGS

Members of the **CONST_DMUS_PC_FLAGS** enumeration are used in the **DMUS_PORTCAPS** type to describe miscellaneous capabilities of the port.

```
Enum CONST_DMUS_PC_FLAGS
    DMUS_PC_DIRECTSOUND = 128 (&H80)
    DMUS_PC_DLS = 1
    DMUS_PC_EXTERNAL = 2
    DMUS_PC_GMINHARDWARE = 16 (&H10)
    DMUS_PC_GSINHARDWARE = 32 (&H20)
    DMUS_PC_MEMORYSIZEFIXED = 8
    DMUS_PC_SHAREABLE = 256 (&H100)
    DMUS_PC_SOFTWARESYNTH = 4
    DMUS_PC_SYSTEMMEMORY = 2147483647 (&H7FFFFFFF)
    DMUS_PC_XGINHARDWARE = 64 (&H40)
End Enum
```

DMUS_PC_DIRECTSOUND

The port supports streaming wave data to DirectSound.

DMUS_PC_DLS

The port supports DLS Level 1 sample collections.

DMUS_PC_EXTERNAL

IDH_CONST_DMUS_PC_CLASS_dmusc_vb

IDH_CONST_DMUS_PC_FLAGS_dmusc_vb

This port connects to devices outside of 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_SYSTEMMEMORY

The port is using system memory, and the amount is therefore limited only by the available system memory. This constant is used in

DMUS_PORTCAPS.IMemorysize, not **IFlags**.

DMUS_PC_XGINHARDWARE

The port contains the Yamaha XG extensions in hardware.

CONST_DMUS_PLAYMODE_FLAGS

The members of the **CONST_DMUS_PLAYMODE_FLAGS** enumeration are used to set the play mode in a **DMUS_NOTE_PMSG** message type. The play mode determines how the note is transposed to the current chord before it is converted to a MIDI note.

```
Enum CONST_DMUS_PLAYMODE_FLAGS
    DMUS_PLAYMODE_ALWAYSPLAY    = 14
    DMUS_PLAYMODE_CHORD_INTERVALS = 8
    DMUS_PLAYMODE_CHORD_ROOT    = 2
    DMUS_PLAYMODE_FIXED         = 0
    DMUS_PLAYMODE_FIXEDTOCHORD  = 2
    DMUS_PLAYMODE_FIXEDTOKEY    = 1
    DMUS_PLAYMODE_KEY_ROOT      = 1
    DMUS_PLAYMODE_MELODIC       = 6
    DMUS_PLAYMODE_NONE          = 16 (&H10)
    DMUS_PLAYMODE_NORMALCHORD   = 10
```

IDH_CONST_DMUS_PLAYMODE_FLAGS_dmusic_vb

```

DMUS_PLAYMODE_PEDALPOINT    = 5
DMUS_PLAYMODE_SCALE_INTERVALS = 4
End Enum

```

The following members are the basic flags:

```

DMUS_PLAYMODE_CHORD_INTERVALS
    Use chord intervals from chord pattern.
DMUS_PLAYMODE_CHORD_ROOT
    Transpose over the chord root.
DMUS_PLAYMODE_KEY_ROOT
    Transpose over the key root.
DMUS_PLAYMODE_NONE
    No mode. Indicates that the parent part's mode should be used.
DMUS_PLAYMODE_SCALE_INTERVALS
    Use scale intervals from scale pattern.

```

The following members represent combinations of the basic flags:

```

DMUS_PLAYMODE_ALWAYSPLAY
    Combination of DMUS_PLAYMODE_SCALE_INTERVALS,
    DMUS_PLAYMODE_CHORD_INTERVALS, and
    DMUS_PLAYMODE_CHORD_ROOT. If it is desirable to play a note that is
    above the top of the chord, this mode finds a position for the note by using
    intervals from the scale. Essentially, this mode is a combination of the normal
    and melodic playback modes, in which a failure in normal mode causes a second
    try in melodic mode.
DMUS_PLAYMODE_FIXED
    Interpret the music value as a MIDI value. This is defined as 0 and signifies the
    absence of other flags. This flag is used for drums, sound effects, and sequenced
    notes that should not be transposed by the chord or scale.
DMUS_PLAYMODE_FIXEDTOCHORD
    Same as DMUS_PLAYMODE_CHORD_ROOT. The music value is a fixed
    MIDI value, but it is transposed over the chord root.
DMUS_PLAYMODE_FIXEDTOKEY
    Same as DMUS_PLAYMODE_KEY_ROOT. The music value is a fixed MIDI
    value, but it is transposed over the key root.
DMUS_PLAYMODE_MELODIC
    Combination of DMUS_PLAYMODE_CHORD_ROOT and
    DMUS_PLAYMODE_SCALE_INTERVALS. The chord root is used, but the
    notes track only the intervals in the scale. The key root and chord intervals are
    ignored. This is useful for melodic lines that play relative to the chord root.
DMUS_PLAYMODE_NORMALCHORD
    Combination of DMUS_PLAYMODE_CHORD_ROOT and
    DMUS_PLAYMODE_CHORD_INTERVALS. This is the prevalent playback
    mode. The notes track the intervals in the chord, which is based on the chord
    root. If there is a scale component to the music value, the additional intervals are

```


pulled from the scale and added. If the chord does not have an interval to match the chord component of the music value, the note is silent.

DMUS_PLAYMODE_PEDALPOINT

Combination of DMUS_PLAYMODE_KEY_ROOT and DMUS_PLAYMODE_SCALE_INTERVALS. The key root is used, and the notes track only the intervals in the scale. The chord root and intervals are ignored. This is useful for melodic lines that play relative to the key root.

CONST_DMUS_PMSGF_FLAGS

The members of the **CONST_DMUS_PMSGF_FLAGS** enumeration are used in the various message-sending methods of **DirectMusicPerformance** to specify when a message should be delivered to tools.

Enum CONST_DMUS_PMSGF_FLAGS

```
DMUS_PMSGF_MUSICTIME    = 2
DMUS_PMSGF_REFTIME      = 1
DMUS_PMSGF_TOOL_ATTIME  = 16 (&H10)
DMUS_PMSGF_TOOL_FLUSH   = 32 (&H20)
DMUS_PMSGF_TOOL_IMMEDIATE = 4
DMUS_PMSGF_TOOL_QUEUE    = 8
```

End Enum

DMUS_PMSGF_REFTIME

The time stamp is in clock time.

DMUS_PMSGF_MUSICTIME

The time stamp is in music time.

DMUS_PMSGF_TOOL_IMMEDIATE

DMUS_PMSGF_TOOL_QUEUE

DMUS_PMSGF_TOOL_ATTIME

DMUS_PMSGF_TOOL_FLUSH

See Remarks.

Remarks

Because DirectX for Visual Basic does not support DirectMusic tools, only DMUS_PMSGF_REFTIME and DMUS_PMSGF_MUSICTIME are currently valid. The time of messages is in music time by default, so DMUS_PMSGF_REFTIME is the only flag that applications normally use.

CONST_DMUS_PORT_TYPE

The **CONST_DMUS_PORT_TYPE** enumeration is used in the **DMUS_PORTCAPS** type to specify the driver model of the port.

```
Enum CONST_DMUS_PORT_TYPE
    DMUS_PORT_KERNEL_MODE = 2
    DMUS_PORT_USER_MODE_SYNTH = 1
    DMUS_PORT_WINMM_DRIVER = 0
End Enum
```

DMUS_PORT_WINMM_DRIVER
Windows multimedia driver.

DMUS_PORT_USER_MODE_SYNTH
User-mode synthesizer.

DMUS_PORT_KERNEL_MODE
Windows Driver Model (WDM) driver.

CONST_DMUS_SEGF_FLAGS

The members of the **CONST_DMUS_SEGF_FLAGS** enumeration are used in various methods of the **DirectMusicPerformance** object to control the timing and other aspects of actions on a segment.

```
Enum CONST_DMUS_SEGF_FLAGS
    DMUS_SEGF_AFTERPREPARETIME = 1024 (&H400)
    DMUS_SEGF_BEAT = 4096 (&H1000)
    DMUS_SEGF_CONTROL = 512 (&H200)
    DMUS_SEGF_DEFAULT = 16384 (&H4000)
    DMUS_SEGF_GRID = 2048 (&H800)
    DMUS_SEGF_MEASURE = 8192 (&H2000)
    DMUS_SEGF_NOINVALIDATE = 32768 (&H8000)
    DMUS_SEGF_QUEUE = 256 (&H100)
    DMUS_SEGF_REFTIME = 64 (&H40)
    DMUS_SEGF_SECONDARY = 128 (&H80)
End Enum
```

DMUS_SEGF_AFTERPREPARETIME
Play after the prepare time. (See **DirectMusicPerformance.GetPrepareTime.**)

DMUS_SEGF_BEAT
Play on a beat boundary.

DMUS_SEGF_CONTROL
Play as a control segment (secondary segments only). See Remarks.

IDH_CONST_DMUS_PORT_TYPE_dmusic_vb

IDH_CONST_DMUS_SEGF_FLAGS_dmusic_vb

DMUS_SEGF_DEFAULT

Use segment's default boundary.

DMUS_SEGF_GRID

Play on a grid boundary.

DMUS_SEGF_MEASURE

Play on a measure boundary.

DMUS_SEGF_NOINVALIDATE

Setting this flag in **DirectMusicPerformance.PlaySegment** for a primary or control segment causes the new segment not to cause an invalidation. Without this flag, an invalidation occurs, cutting off and resetting any currently playing curve or note. This flag should be combined with **DMUS_SEGF_AFTERPREPARETIME** so that notes in the new segment do not play over notes played by the old segment.

DMUS_SEGF_QUEUE

Put at the end of the primary segment queue (primary segment only).

DMUS_SEGF_REFTIME

Time parameter is in clock time.

DMUS_SEGF_SECONDARY

Secondary segment.

Remarks

Normally, the primary segment is the control segment. The **DMUS_SEGF_CONTROL** flag can be used to make a secondary segment the control segment. There should be only one control segment at a time. (It is possible to create multiple control segments, but there is no guarantee of which one will actually be used by DirectMusic as the control segment.) By default, only the control segment sends tempo messages.

If the **DMUS_SEGF_CONTROL** flag is set, **DMUS_SEGF_SECONDARY** is assumed.

See Also

DirectMusicPerformance.GetResolvedTime,
DirectMusicPerformance.Invalidated, **DirectMusicPerformance.PlaySegment**,
DirectMusicPerformance.Stop

CONST_DMUS_SHAPET_TYPES

The members of the **CONST_DMUS_SHAPET_TYPES** enumeration are used in the *wShape* parameter of the **DirectMusicComposer.ComposeSegmentFromShape** and **DirectMusicComposer.ComposeTemplateFromShape** methods to specify the desired pattern of the groove level.

IDH_CONST_DMUS_SHAPET_TYPES_dmusic_vb

```
Enum CONST_DMUS_SHAPET_TYPES
```

```
    DMUS_SHAPET_FALLING = 0
    DMUS_SHAPET_LEVEL   = 1
    DMUS_SHAPET_LOOPABLE = 2
    DMUS_SHAPET_LOUD    = 3
    DMUS_SHAPET_PEAKING = 5
    DMUS_SHAPET_QUIET   = 4
    DMUS_SHAPET_RANDOM  = 6
    DMUS_SHAPET_RISING  = 7
    DMUS_SHAPET_SONG    = 8
```

```
End Enum
```

```
DMUS_SHAPET_FALLING
```

The groove level falls.

```
DMUS_SHAPET_LEVEL
```

The groove level remains even.

```
DMUS_SHAPET_LOOPABLE
```

The segment is arranged to loop back to the beginning.

```
DMUS_SHAPET_LOUD
```

The groove level is high.

```
DMUS_SHAPET_PEAKING
```

The groove level rises to a peak, then falls.

```
DMUS_SHAPET_QUIET
```

The groove level is low.

```
DMUS_SHAPET_RANDOM
```

The groove level is random.

```
DMUS_SHAPET_RISING
```

The groove level rises.

```
DMUS_SHAPET_SONG
```

The segment is in a song form. Several phrases of 6 to 8 bars are composed and put together to give a verse-chorus effect, with variations in groove level.

CONST_DMUSERR

The **CONST_DMUSERR** enumeration represents DirectMusic error codes.

Error Codes

This section provides a brief explanation of the various error codes that can be returned by DirectMusic methods. For a list of the specific codes that each method can return, see the individual method descriptions. These lists are not necessarily comprehensive.

IDH_CONST_DMUSERR_dmusic_vb

DMUS_E_ALL_TRACKS_FAILED

The segment object was unable to load all tracks from the **IStream** object data, perhaps because of errors in the stream or because the tracks are incorrectly registered on the client.

DMUS_E_ALREADY_ACTIVATED

The port has been activated, and the parameter cannot be changed.

DMUS_E_ALREADY_DOWNLOADED

Buffer has already been downloaded.

DMUS_E_ALREADY_EXISTS

The tool is already contained in the graph. You must create a new instance.

DMUS_E_ALREADY_INITED

The object has already been initialized.

DMUS_E_ALREADY_LOADED

The DLS collection is already open.

DMUS_E_ALREADY_SENT

The message has already been sent.

DMUS_E_ALREADYCLOSED

The port is not open.

DMUS_E_ALREADYOPEN

The port was already opened.

DMUS_E_BADARTICULATION

Invalid articulation chunk in DLS collection.

DMUS_E_BADINSTRUMENT

Invalid instrument chunk in DLS collection.

DMUS_E_BADOFFSETTABLE

Offset table has errors.

DMUS_E_BADWAVE

Corrupt wave header.

DMUS_E_BADWAVELINK

Wave-link chunk in DLS collection points to an invalid wave.

DMUS_E_BUFFER_EMPTY

There is no data in the buffer.

DMUS_E_BUFFER_FULL

The specified number of bytes exceeds the maximum buffer size.

DMUS_E_BUFFERNOTAVAILABLE

The buffer is not available for download.

DMUS_E_BUFFERNOTSET

No buffer was prepared for the data.

DMUS_E_CANNOT_OPEN_PORT

The default system port could not be opened.

DMUS_E_DEVICE_IN_USE

Device is already in use (possibly by a non-DirectMusic client) and cannot be opened again.

DMUS_E_DMUSIC_RELEASED

Operation cannot be performed because the final instance of the DirectMusic object was released. Ports cannot be used after final release of the DirectMusic object.

DMUS_E_DRIVER_FAILED

An unexpected error was returned from a device driver, indicating possible failure of the driver or hardware.

DMUS_E_DSOUND_ALREADY_SET

A DirectSound object has already been set.

DMUS_E_DSOUND_NOT_SET

Port could not be created because no DirectSound object has been specified.

DMUS_E_FAIL

The method did not succeed.

DMUS_E_GET_UNSUPPORTED

Getting the parameter is not supported.

DMUS_E_INSUFFICIENTBUFFER

Buffer is not large enough for the requested operation.

DMUS_E_INVALIDARG

Invalid argument.

DMUS_E_INVALID_BAND

File does not contain a valid band.

DMUS_E_INVALID_DOWNLOADID

Invalid download identifier was used in the process of creating a download buffer.

DMUS_E_INVALID_EVENT

The event either is not a valid MIDI message or makes use of running status, and cannot be packed into the buffer.

DMUS_E_INVALIDBUFFER

Invalid DirectSound buffer was handed to port.

DMUS_E_INVALIDFILE

Not a valid file.

DMUS_E_INVALIDPATCH

No instrument in the collection matches the patch number.

DMUS_E_INVALIDPOS

Error reading wave data from a DLS collection. Indicates a bad file.

DMUS_E_LOADER_BADPATH

The file path is invalid.

DMUS_E_LOADER_FAILEDCREATE

Object could not be found or created.

DMUS_E_LOADER_FAILEDOPEN

File open failed because the file does not exist or is locked.

DMUS_E_LOADER_FORMATNOTSUPPORTED

The object cannot be loaded because the data format is not supported.

DMUS_E_LOADER_OBJECTNOTFOUND

The object was not found.

DMUS_E_NO_MASTER_CLOCK

There is no master clock in the performance. Be sure to call the **DirectMusicPerformance.Init** method.

DMUS_E_NOINTERFACE

No object interface is available.

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_NOTADLSCOL

The object being loaded is not a valid DLS collection.

DMUS_E_NOTIMPL

The method is not implemented. This value can be returned if a driver does not support a feature necessary for the operation.

DMUS_E_OUT_OF_RANGE

The requested time is outside the range of the segment.

DMUS_E_OUTOFMEMORY

Insufficient memory to complete task.

DMUS_E_PORT_NOT_RENDER

Not an output port.

DMUS_E_PORTS_OPEN

The requested operation cannot be performed while there are instantiated ports in any process in the system.

DMUS_E_SEGMENT_INIT_FAILED

Segment initialization failed, probably because of a critical memory situation.

DMUS_E_SET_UNSUPPORTED

Setting the parameter is not supported.

DMUS_E_TIME_PAST

The time requested is in the past.

DMUS_E_TRACK_NOT_FOUND

There is no track of the requested type.

DMUS_E_TYPE_DISABLED

Parameter is unavailable because it has been disabled.

DMUS_E_TYPE_UNSUPPORTED

Parameter is unsupported on this track.

DMUS_E_UNKNOWN_PROPERTY

The property set or item is not implemented by this port.

DMUS_E_UNSUPPORTED_STREAM

The stream does not contain data supported by the loading object.

DirectMusic Tutorials

This section contains tutorials providing step-by-step instructions for implementing basic Microsoft® DirectMusic® functionality.

- DirectMusic C/C++ Tutorials
- DirectMusic Visual Basic Tutorials

DirectMusic C/C++ Tutorials

[\[Visual Basic\]](#)

This section pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

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

- Tutorial 1: Playing a MIDI File
 - Tutorial 2: Using Tools
 - Tutorial 3: Using Compositions
-

Tutorial 1: Playing a MIDI File

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

This tutorial is a guide to setting up the simplest possible DirectMusic application, one that just plays a primary segment. In this example the segment represents a MIDI file, but the process of loading and playing the data would be exactly the same if the source were a segment authored in a tool such as DirectMusic Producer.

The tutorial is divided into the following steps:

- Step 1: Initialize COM

- Step 2: Create the Performance
 - Step 3: Create the Loader
 - Step 4: Load the MIDI File
 - Step 5: Play the MIDI File
 - Step 6: Shut Down DirectMusic
-

Step 1: Initialize COM

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

Before making any calls to DirectMusic, you have to initialize COM as follows:

```
if (FAILED(CoInitialize(NULL)))
{
    // Terminate the application.
} // Else full speed ahead!
```

Next: Step 2: Create the Performance

Step 2: Create the Performance

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

The central object of any DirectMusic application is the performance, which manages the playback of segments. It is created by using the COM **CoCreateInstance** function, as in the following sample function:

```
IDirectMusicPerformance* CreatePerformance(void)
{
    IDirectMusicPerformance* pPerf;

    if (FAILED(CoCreateInstance(
        CLSID_DirectMusicPerformance,
        NULL,
```

```
        CLSCTX_INPROC,  
        IID_IDirectMusicPerformance2,  
        (void**)&pPerf  
    )))  
{  
    pPerf = NULL;  
}  
return pPerf;  
}
```

You can use this function to initialize a global performance pointer that will be used in later steps:

```
IDirectMusicPerformance* g_pPerf = CreatePerformance();  
if (g_pPerf == NULL)  
{  
    // Failure -- performance not created  
}
```

Once the performance has been created, you need to initialize it by calling the **IDirectMusicPerformance::Init** method. The method creates a DirectMusic object to manage the default port. Because you don't need to access the **IDirectMusic** methods directly, you don't need to retrieve a pointer to it, so you pass NULL as the first parameter to **Init**. You also pass NULL as the **IDirectSound** pointer and as the window handle, so that DirectMusic will create a DirectSound object and pass the current focus window to it when setting the cooperative level.

```
if (FAILED(g_pPerf->Init(NULL, NULL, NULL)))  
{  
    // Failure -- performance not initialized  
};
```

Now you need to add a port to the performance. Calling the **IDirectMusicPerformance::AddPort** method with a NULL parameter automatically adds the default port (normally the Microsoft Software Synthesizer) with one channel group, and assigns PChannels 0-15 to the group's MIDI channels.

```
if (FAILED(pPerf->AddPort(NULL)))  
{  
    // Failure -- port not initialized  
}
```

Next: Step 3: Create the Loader

Step 3: Create the Loader

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

In order to load any object from disk, you first need to create the DirectMusicLoader object. This is done just as for any other COM object, as shown in the following sample function:

```
IDirectMusicLoader* CreateLoader(void)
{
    IDirectMusicLoader* pLoader;

    if (FAILED(CoCreateInstance(
        CLSID_DirectMusicLoader,
        NULL,
        CLSCTX_INPROC,
        IID_IDirectMusicLoader,
        (void**)&pLoader
    )))
    {
        pLoader = NULL;
    }
    return pLoader;
}
```

You'll use this function to initialize a global variable:

```
IDirectMusicLoader* g_pLoader = CreateLoader();
if (g_pLoader == NULL)
{
    // Failure -- loader not created
}
```

Next: Step 4: Load the MIDI File

Step 4: Load the MIDI File

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[C++]

In this step you will implement a function, LoadMIDISegment, that takes a pointer to the **IDirectMusicLoader** created in the last step and uses it to create a segment object encapsulating the data from a MIDI file.

```
IDirectMusicSegment* LoadMIDISegment(IDirectMusicLoader* pLoader,
    WCHAR wszMidiFileName )
{
    DMUS_OBJECTDESC ObjDesc;
    IDirectMusicSegment* pSegment = NULL;
```

Let's assume that all the MIDI files you want to play are in the current working directory. You need to let the loader know this, by setting the search directory. (If the search directory is not being changed elsewhere, in order to load objects from other directories, this actually has to be done only once, not each time you load a file.)

```
    char szDir[_MAX_PATH];
    WCHAR wszDir[_MAX_PATH];

    if(_getcwd( szDir, _MAX_PATH ) == NULL)
    {
        return NULL;
    }

    /*
    Convert from multibyte format to Unicode using the following macro:
    #define MULTI_TO_WIDE( x,y ) MultiByteToWideChar( CP_ACP, \
        MB_PRECOMPOSED, y, -1, x, _MAX_PATH );
    */

    MULTI_TO_WIDE(wszDir, szDir);
    HRESULT hr = pLoader->SetSearchDirectory(GUID_DirectMusicAllTypes,
        wszDir, FALSE);
    if (FAILED(hr))
    {
        return NULL;
    }
```

You then describe the object to be loaded, in a **DMUS_OBJECTDESC** structure:

```
    ObjDesc.guidClass = CLSID_DirectMusicSegment;
    ObjDesc.dwSize = sizeof(DMUS_OBJECTDESC);
    wcsncpy( ObjDesc.wszFileName, wszMidiFileName );
    ObjDesc.dwValidData = DMUS_OBJ_CLASS | DMUS_OBJ_FILENAME;
```

Now load the object and query it for the **IDirectMusicSegment** interface. This is done in a single call to **IDirectMusicLoader::GetObject**. Note that loading the object also initializes the tracks and does everything else necessary to get the MIDI data ready for playback.

```
pLoader->GetObject(&ObjDesc,
    IID_IDirectMusicSegment2, (void**) &pSegment);
```

To ensure that the segment plays as a standard MIDI file, you now need to set a parameter on the band track. Use the **IDirectMusicSegment::SetParam** method and let DirectMusic find the track, by passing -1 (or 0xFFFFFFFF) in the *dwGroupBits* method parameter.

```
g_pMIDISeg->SetParam(GUID_StandardMIDIFile,
    -1, 0, 0, (void*)g_pPerf);
```

This step is necessary because DirectMusic handles program changes and bank selects differently for standard MIDI files than it does for MIDI content authored specifically for DirectMusic. The GUID_StandardMIDIFile parameter must be set before the instruments are downloaded.

The next step is to download the instruments. This is necessary even for playing a simple MIDI file, because the default software synthesizer needs the DLS data for the General MIDI instrument set. If you skip this step, the MIDI file will play silently. Again, you call **SetParam** on the segment, this time specifying the GUID_Download parameter:

```
g_pMIDISeg->SetParam(GUID_Download, -1, 0, 0, (void*)g_pPerf);
```

Note that there's no harm in requesting the download even though this might already have been done in a previous call to the LoadMIDIsegment function. A redundant request is simply ignored. Eventually you have to unload the instruments, but that can wait until you're ready to shut down DirectMusic.

The function now returns a pointer to the segment, which is ready to be played.

```
return pSegment;

} // End of LoadSegment()
```

Before loading a new segment, clean up any existing one. Then pass a file name to the LoadMIDIsegment function.

```
if (g_pMIDISeg)
{
    g_pMIDISeg->Release();
    g_pMIDISeg = NULL;
}

if (g_pLoader)
```

```
{  
    IDirectMusicSegment* g_pMIDISeg = LoadMIDIsegment(g_pLoader,  
        L"tune.mid");  
}
```

Next: Step 5: Play the MIDI File

Step 5: Play the MIDI File

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

Now that all of the preparatory work has been done, playing the music is simplicity itself.

```
IDirectMusicSegmentState* g_pSegState;  
if (g_pMidiSEG)  
{  
    g_pPerf->PlaySegment(g_pMIDISeg, 0, 0, &g_pSegState);  
}
```

The call to **IDirectMusicPerformance::PlaySegment** takes the following parameters:

- The segment you created in the previous step.
- A set of timing flags and a start time (not needed here because you simply want the segment to play as soon as possible).
- The address of a pointer to a segment state object. You can use the returned pointer in order to retrieve information about the segment. This parameter can be NULL if you won't be needing the segment state.

If you want the file to be played more than once, before calling **PlaySegment** you must call the **IDirectMusicSegment::SetRepeats** method.

If you want to stop the music before it has played to the end or finished its repeats, you do so by using the **IDirectMusicPerformance::Stop** method. The simplest way to use this method is simply to request that all music currently playing or cued to play be stopped immediately:

```
g_pPerf->Stop(NULL, NULL, 0, 0)
```

Alternatively, you can supply a pointer to the current segment or segment state in order to stop playback of just one segment, or one instance of that segment.

Next: Step 6: Shut Down DirectMusic

Step 6: Shut Down DirectMusic

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

Before exiting, the program must unload the instruments, release all the objects that have been created, and dereference COM (remember, every call to **CoInitialize** must have a matching call to **CoUninitialize**).

The following function performs the necessary cleanup:

```
HRESULT FreeDirectMusic()
{
    // If there is any music playing, stop it. This is
    // not really necessary, because the music will stop when
    // the instruments are unloaded or the performance is
    // closed down.
    g_pPerf->Stop( NULL, NULL, 0, 0 );

    // Unload instruments – this will cause silence.
    // CloseDown unloads all instruments, so this call is also not
    // strictly necessary.
    g_pMIDISeg->SetParam(GUID_Unload, -1, 0, 0, (void*)g_pPerf);

    // Release the segment.
    g_pMIDISeg->Release();

    // CloseDown and Release the performance object.
    g_pPerf->CloseDown();
    g_pPerf->Release();

    // Release the loader object.
    g_pLoader->Release();

    // Release COM.
    CoUninitialize();

    return S_OK;
}
```

Tutorial 2: Using Tools

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

This tutorial shows how a tool might be implemented as a C++ class and used in a DirectMusic application. The CEchoTool class enables an application to add one or more echoes to every music note in a performance.

The sample code is based on the EchoTool sample application included with the DirectX SDK.

The tutorial is broken down into the following steps:

- Step 1: Declare the Tool Class
 - Step 2: Define the IUnknown Methods
 - Step 3: Specify Message Types
 - Step 4: Define the ProcessPMsg Method
 - Step 5: Define the Class Methods
 - Step 6: Add the Tool to the Performance
-

Step 1: Declare the Tool Class

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

The first step in creating a tool for DirectMusic in C++ is to declare a class derived from **IDirectMusicTool**.

Here is the declaration for the sample CEchoTool class:

```
class CEchoTool : public IDirectMusicTool
{
public:
    CEchoTool();
    ~CEchoTool();
```



```
public:
// IUnknown
virtual STDMETHODCALLTYPE QueryInterface(const IID &iid, void **ppv);
virtual STDMETHODCALLTYPE AddRef();
virtual STDMETHODCALLTYPE Release();

// IDirectMusicTool
HRESULT STDMETHODCALLTYPE Init(
    IDirectMusicGraph* pGraph);
HRESULT STDMETHODCALLTYPE GetMsgDeliveryType(
    DWORD* pdwDeliveryType);
HRESULT STDMETHODCALLTYPE GetMediaTypeArraySize(
    DWORD* pdwNumElements);
HRESULT STDMETHODCALLTYPE GetMediaTypes(
    DWORD** padwMediaTypes,
    DWORD dwNumElements);
HRESULT STDMETHODCALLTYPE ProcessPMsg(
    IDirectMusicPerformance* pPerf,
    DMUS_PMSG* pDMUS_PMSG);
HRESULT STDMETHODCALLTYPE Flush(
    IDirectMusicPerformance* pPerf,
    DMUS_PMSG* pDMUS_PMSG,
    REFERENCE_TIME rtTime);

private:
    long m_cRef;           // Reference counter
    DWORD m_dwEchoNum;     // Number of echoes to generate
    MUSIC_TIME m_mtDelay;  // Delay time between echoes
    CRITICAL_SECTION m_CrSec; // To make SetEchoNum()
                             // and SetDelay() thread-safe

public:
// Public class methods
    void SetEchoNum(DWORD);
    void SetDelay(MUSIC_TIME);
};
```

Next: Step 2: Define the IUnknown Methods

Step 2: Define the IUnknown Methods

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[C++]

The inherited **IUnknown** methods of the sample CEchoTool class are implemented as follows:

```
STDMETHODIMP CEchoTool::QueryInterface(const IID &iid, void **ppv)
{
    if (iid == IID_IUnknown || iid == IID_IDirectMusicTool)
    {
        *ppv = static_cast<IDirectMusicTool*>(this);
    } else
    {
        *ppv = NULL;
        return E_NOINTERFACE;
    }

    reinterpret_cast<IUnknown*>(this)->AddRef();
    return S_OK;
}

STDMETHODIMP_(ULONG) CEchoTool::AddRef()
{
    return InterlockedIncrement(&m_cRef);
}

STDMETHODIMP_(ULONG) CEchoTool::Release()
{
    if (0 == InterlockedDecrement(&m_cRef) )
    {
        delete this;
        return 0;
    }
    return m_cRef;
}
```

Next: Step 3: Specify Message Types

Step 3: Specify Message Types

[Visual Basic]

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[C++]

The CEchoTool class needs to define all the **IDirectMusicTool** methods. But because it does not need to perform any work in **Init** or **Flush**, it simply returns E_NOTIMPL from those methods.

The following methods are used to specify what messages will get passed to the tool for processing:

- **GetMsgDeliveryType** specifies when messages should be passed to the tool—as soon as they are available, at the exact time they are stamped for, or just in time to be processed and put in the port buffer. These three "delivery types" are represented respectively by the DMUS_PMSGF_TOOL_IMMEDIATE, DMUS_PMSGF_TOOL_ATTIME, and DMUS_PMSGF_TOOL_QUEUE flags in the **dwFlags** member of the **DMUS_PMSG** structure.
- **GetMediaTypes** establishes what type of messages should be passed to the tool, based on the content of the message. CEchoTool processes only music notes and patch changes, so it will accept MIDI short messages, music note messages, and DirectMusic patch messages. These messages are identified by flags of the **DMUS_PMSGT_TYPES** enumerated type in the **DMUS_PMSG** message structure.
- **GetMediaTypeArraySize** returns the number of elements in the media type array.

CEchoTool implements these methods in the following sample code:

```
HRESULT STDMETHODCALLTYPE CEchoTool::GetMsgDeliveryType(
    DWORD* pdwDeliveryType )
{
    // This tool wants messages immediately.
    // This is the default, so returning E_NOTIMPL
    // would work. The other method is to specifically
    // set *pdwDeliveryType to the delivery type,
    // DMUS_PMSGF_TOOL_IMMEDIATE, DMUS_PMSGF_TOOL_QUEUE,
    // or DMUS_PMSGF_TOOL_ATTIME.

    *pdwDeliveryType = DMUS_PMSGF_TOOL_IMMEDIATE;
    return S_OK;
}

HRESULT STDMETHODCALLTYPE CEchoTool::GetMediaTypeArraySize(
    DWORD* pdwNumElements)
{
    *pdwNumElements = 3;
    return S_OK;
}

HRESULT STDMETHODCALLTYPE CEchoTool::GetMediaTypes(
    DWORD** padwMediaTypes,
```

```

        DWORD dwNumElements)
{
    if (dwNumElements == 3)
    {
        (*padwMediaTypes)[0] = DMUS_PMSGT_NOTE;
        (*padwMediaTypes)[1] = DMUS_PMSGT_MIDI;
        (*padwMediaTypes)[2] = DMUS_PMSGT_PATCH;
        return S_OK;
    }
    else
    {
        // This should never happen.
        return E_FAIL;
    }
}

```

Next: Step 4: Define the ProcessPMsg Method

Step 4: Define the ProcessPMsg Method

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

The actual work of the tool is performed in the method derived from **IDirectMusicTool::ProcessPMsg**. This method will be called for every message of the type listed in the array returned by **IDirectMusicTool::GetMediaTypes**.

The CEchoTool::ProcessPMsg method first performs some initialization of local variables:

```

HRESULT STDMETHODCALLTYPE CEchoTool::ProcessPMsg(
    IDirectMusicPerformance* pPerf,
    DMUS_PMSG* pMsg)
{
    DMUS_NOTE_PMSG* pNote;
    DWORD dwCount;
    DWORD dwEchoNum;
    MUSIC_TIME mtDelay;

    // SetEchoNum() and SetDelay() use these member variables,
    // so use a critical section to make them thread-safe.
    EnterCriticalSection(&m_CrSec);

```

```

dwEchoNum = m_dwEchoNum;
mtDelay = m_mtDelay;
LeaveCriticalSection(&m_CrSec);

```

Next the method calls the **IDirectMusicGraph::StampPMsg** method on the message. If there is another tool to which this message must be directed after we're finished with it here, **StampPMsg** succeeds. (Note that DirectMusic provides a final output tool, so **StampPMsg** should succeed even if there are no other application-specific tools to which the message must be routed.) If it fails, our method returns **S_FREE** so that the message will automatically be discarded.

```

if ((NULL == pMsg->pGraph) ||
    FAILED(pMsg->pGraph->StampPMsg(pMsg)))
{
    return DMUS_S_FREE;
}

```

Now it's time for **CEchoTool** to perform its work on the message. Remember, it is set up to receive messages only of type **DMUS_PMSGT_NOTE**, **DMUS_PMSGT_MIDI**; or **DMUS_PMSGT_PATCH**. (These types are part of the **DMUS_PMSGT_TYPES** enumeration.)

For each successive echo to be created, the tool does the following:

- Creates a new message by calling the **IDirectMusicPerformance::AllocPMsg** method.
- Copies the **DMUS_PMSG** structure for the original message into the new one and updates some members for COM management.
- Assigns the new message to a different performance channel.
- In the case of a music note, progressively reduces the volume of each echo.
- Adds the delay time to the time at which the message is to be played.
- Puts the new message in the pipeline by calling the **IDirectMusicPerformance::SendPMsg** method. Note that because **IDirectMusicGraph::StampPMsg** is not called on the message, it will not be routed to any tools other than the default final output tool.

Here's the sample code that deals with MIDI notes:

```

if( pPMsg->dwType == DMUS_PMSGT_MIDI )
{
    // copy MIDI messages into the echo channels.
    for( dwCount = 1; dwCount <= dwEchoNum; dwCount++ )
    {
        DMUS_MIDI_PMSG* pMidi;
        if( SUCCEEDED( pPerf->AllocPMsg( sizeof(DMUS_MIDI_PMSG),
            (DMUS_PMSG**)&pMidi )))
        {

```

```

// Copy the original message into this message.
memcpy( pMidi, pPMsg, sizeof(DMUS_MIDI_PMSG) );

// Addref or clear out any fields that contain
// or may contain pointers to objects.
if( pMidi->pTool ) pMidi->pTool->AddRef();
if( pMidi->pGraph ) pMidi->pGraph->AddRef();
pMidi->punkUser = NULL;

// Set the PChannel so the message goes to the
// next higher group.
pMidi->dwPChannel = pMidi->dwPChannel +
    (16*dwCount);

// Add to the time of the echoed message.
pMidi->mtTime += (dwCount * mtDelay);

// Set the message so only MUSIC_TIME is valid.
// REFERENCE_TIME will be recomputed inside
// SendPMsg().
pMidi->dwFlags = DMUS_PMSGF_MUSICTIME;

// Send the message
pPerf->SendPMsg( (DMUS_PMSG*)pMidi );
}
}
}

```

Patch changes are also copied and sent to all possible echo channels, even those not currently being used, so that if echoes are added later they will be played by the correct instruments. Note that in the sample EchoTool application, MAX_ECHOES is defined in Echotool.h.

```

else if( pPMsg->dwType == DMUS_PMSGT_PATCH )
{
    for( dwCount = 1; dwCount <= MAX_ECHOES; dwCount++ )
    {
        DMUS_PATCH_PMSG* pPatch;
        if( SUCCEEDED( pPerf->AllocPMsg( sizeof(DMUS_PATCH_PMSG),
            (DMUS_PMSG**)&pPatch )))
        {
            // Copy the original message into this message,
            memcpy( pPatch, pPMsg, sizeof(DMUS_PATCH_PMSG) );

            // Addref or clear out any fields that contain
            // or may contain pointers to objects

```

```

    if( pPatch->pTool ) pPatch->pTool->AddRef();
    if( pPatch->pGraph ) pPatch->pGraph->AddRef();
    pPatch->punkUser = NULL;

    // Set the PChannel so the message goes to the
    // next higher group.
    pPatch->dwPChannel = pPatch->dwPChannel +
        (16*dwCount);

    // Add to the time of the echoed message.
    pPatch->mtTime += (dwCount * mtDelay);

    // Set the message so only MUSIC_TIME is valid.
    // REFERENCE_TIME will be recomputed inside
    // SendPMsg()
    pPatch->dwFlags = DMUS_PMSGF_MUSICTIME;

    // Send the message.
    pPerf->SendPMsg( (DMUS_PMSG*)pPatch );
}
}
}

```

The method deals with music notes much as it did with MIDI notes, but taking the additional step of reducing the volume:

```

else if( pPMsg->dwType == DMUS_PMSGT_NOTE )
{
    // Create a variable to track the next note's velocity
    BYTE bVelocity;
    pNote = (DMUS_NOTE_PMSG*)pPMsg;
    bVelocity = pNote->bVelocity;

    for( dwCount = 1; dwCount <= dwEchoNum; dwCount++ )
    {
        if( SUCCEEDED( pPerf->AllocPMsg( sizeof(DMUS_NOTE_PMSG),
            (DMUS_PMSG**)&pNote )))
        {
            // Copy the original note into this message.
            memcpy( pNote, pPMsg, sizeof(DMUS_NOTE_PMSG) );

            // Addref or clear out any fields that contain
            // or may contain pointers to objects.
            if( pNote->pTool ) pNote->pTool->AddRef();
            if( pNote->pGraph ) pNote->pGraph->AddRef();
            pNote->punkUser = NULL;

```

```
// Add to the time of the echoed note.
pNote->mtTime += (dwCount * mtDelay);

// Reduce the volume of the echoed note.
bVelocity = (BYTE) (bVelocity -
    ((bVelocity * (dwCount * 15))/100));
pNote->bVelocity = bVelocity;

// Set the note so only MUSIC_TIME is valid.
// REFERENCE_TIME will be recomputed inside
// SendPMsg().
pNote->dwFlags = DMUS_PMSGF_MUSICTIME;
pNote->dwPChannel = pNote->dwPChannel +
    (16*dwCount);

// Send the message.
pPerf->SendPMsg( (DMUS_PMSG*)pNote );
}
}
}
```

Finally, the ProcessPMsg method returns DMUS_S_REQUEUE so the original message will be put back in the pipeline.

```
    return DMUS_S_REQUEUE;
} // End CEchoTool::ProcessPMsg()
```

Next: Step 5: Define the Class Methods

Step 5: Define the Class Methods

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

Besides its inherited **IUnknown** and **IDirectMusicTool** methods, the sample CEchoTool class has a constructor and destructor as well as two public methods for setting the parameters of the echo. The definition of these methods is somewhat peripheral to this tutorial, but is included here for completeness.

```
CEchoTool::CEchoTool()
{
```



```
m_cRef = 1;           // So one Release() will free this
m_dwEchoNum = 3;       // Default to 3 echoes per note
m_mtDelay = DMUSPPQ / 2; // Default to 8th-note echoes
InitializeCriticalSection(&m_CrSec);
}

CEchoTool::~CEchoTool()
{
    DeleteCriticalSection(&m_CrSec);
}

void CEchoTool::SetEchoNum(DWORD dwEchoNum)
{
    // ProcessPMsg() uses m_dwEchoNum, so use a critical
    // section to make it thread-safe.
    if( dwEchoNum <= MAX_ECHOES )
    {
        EnterCriticalSection(&m_CrSec);
        m_dwEchoNum = dwEchoNum;
        LeaveCriticalSection(&m_CrSec);
    }
}

void CEchoTool::SetDelay(MUSIC_TIME mtDelay)
{
    // ProcessPMsg() uses m_mtDelay, so use a critical
    // section to make it thread-safe.
    EnterCriticalSection(&m_CrSec);
    m_mtDelay = mtDelay;
    LeaveCriticalSection(&m_CrSec);
}
```

Next: Step 6: Add the Tool to the Performance

Step 6: Add the Tool to the Performance

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

Once it has defined the CEchoTool class, the application can create a tool object, insert it into a graph, and add it to the performance so that it will intercept appropriate messages in the pipeline.

The following code creates an instance of the CEchoTool class, initializes the number of echoes to be produced by the tool, creates a graph to contain the tool, and adds the graph to the performance. (In the EchoTool sample application, some of this work is done in a helper function, AddTool. The code has been modified here to make it easier to follow.)

```

/* It is assumed that pPerf is a valid pointer to the
   IDirectMusicPerformance interface of a performance object. */

CEchoTool      *pEchoTool;
IDirectMusicGraph* pGraph;

pEchoTool = new CEchoTool;
if (pEchoTool)
{
    pEchoTool->SetEchoNum(0);

    // Create an IDirectMusicGraph object to hold the tool.
    if (SUCCEEDED(CoCreateInstance(
        CLSID_DirectMusicGraph,
        NULL,
        CLSCTX_INPROC,
        IID_IDirectMusicGraph,
        (void**)&pGraph)))
    {
        // Add the tool to the graph.
        if (SUCCEEDED(pGraph->InsertTool(
            (IDirectMusicTool*)pEchoTool,
            NULL, // Apply to all PChannels
            0,    // How many PChannels otherwise
            0))) // Index of tool in graph
        {
            // Add the graph to the performance. This increments the
            // reference count, so the original graph can then be
            // released.
            pPerf->SetGraph(pGraph);
        }
        pGraph->Release();
    }
}

```

Note that setting the graph on the performance ensures that messages from all segments will have the opportunity to be processed by its tools. Alternatively, you

could use the **IDirectMusicSegment::SetGraph** method to apply tools only to a particular segment.

Tutorial 3: Using Compositions

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

This tutorial shows how to set up a music performance that is based on elements authored in DirectMusic Producer, and how to have the music performed interactively—that is, in such a way that it responds to program events.

The code samples are based on the DMDonuts sample in the DirectX SDK. Because DMDonuts is a large and complex application, only the main points will be covered here. For the complete DirectMusic implementation, see the Donuts.cpp source file and its included headers, where all the relevant code is delimited by DMUSIC BEGIN and DMUSIC END comments.

The tutorial is broken down into the following steps:

- Step 1: Defines and Globals
 - Step 2: Initialize the Performance
 - Step 3: Load the Music Elements
 - Step 4: Set Up Notifications
 - Step 5: Create the Primary Segments
 - Step 6: Play a Primary Segment
 - Step 7: Transition to Another Primary Segment
 - Step 8: Play a Motif
 - Step 9: Handle Notifications
 - Step 10: Shut Down DirectMusic
-

Step 1: Defines and Globals

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[C++]

The following defines and global variables are declared at the beginning of Donuts.cpp. They are given here for you to refer to as you work through the tutorial.

```

    BOOL      bMusicEnabled = TRUE;

// Various constants
const GUID guidZero = {0};
#define MOTIF_BOUNCE    0
#define MOTIF_DEATH     1
#define MOTIF_SHIELD    2
#define MOTIF_BLOWUP    4
#define MOTIF_BLOWUPLITE 3
#define NUM_MOTIFS      5
#define NUM_STYLES      2
#define SEGMENT_1       0
#define SEGMENT_2       1
#define SEGMENT_TRANS_1 2
#define SEGMENT_TRANS_2 3
#define NUM_SEGMENTS    4
#define NUM_CHORDMAP    4
#define BLOWUPS_PER_BEAT 2
#define MEASURE_LENGTH  DMUS_PPQ * 4
// parts/quarter * quarters/measure (assumes 4/4)

// Global interfaces
IDirectMusicStyle*    gapStyle[NUM_STYLES];
IDirectMusicChordMap* gapChordMap[NUM_STYLES][NUM_CHORDMAP];
IDirectMusicSegment*  gapMotif[NUM_STYLES][NUM_MOTIFS];
IDirectMusicSegment*  gapSegment[NUM_SEGMENTS] =
    {NULL, NULL, NULL, NULL};
IDirectMusicComposer * gpComposer = NULL;
IDirectMusicPerformance* gpPerformance = NULL;
IDirectMusic*          gpDirectMusic = NULL;
IDirectMusicLoader*    gpLoader = NULL;
IDirectMusicSegment*   gpIntroTemplate = NULL;
IDirectMusicSegment*   gpGameTemplate = NULL;
IDirectMusicSegment*   gapShieldSegment[NUM_STYLES] = {NULL, NULL};
IDirectMusicSegment*   gapDefaultSegment[NUM_STYLES] =
    {NULL, NULL};
IDirectMusicBand*      gapShieldBand[NUM_STYLES] = {NULL, NULL};
IDirectMusicBand*      gapDefaultBand[NUM_STYLES] = {NULL, NULL};

// Global variables
BOOL      bAnyHits = FALSE;
int       gnCurrentStyle = 0;

```

```
int      gnCurrentChordMap = 0;
int      gnLastStyle = 0;
int      gnLastChordMap = 0;

BOOL     gbShieldsOn = FALSE;
static int snLastTempo;
static int snSubLevel;
static int snMaxBlowUps = BLOWUPS_PER_BEAT;
```

Next: Step 2: Initialize the Performance

Step 2: Initialize the Performance

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

The DirectMusic performance is set up in the InitializeGame function in Donuts.cpp.

First the application queries the registry to obtain the search path for the DirectX sample music files, by calling the GetSearchPath function. Then it creates COM objects for the composer and the performance, as follows:

```
CoInitialize(NULL);

if ( !SUCCEEDED(::CoCreateInstance(
    CLSID_DirectMusicComposer,
    NULL,
    CLSCTX_INPROC,
    IID_IDirectMusicComposer,
    (void**)&gpComposer
)))
{
    return CleanupAndExit("Couldn't create a composer object");
}

if ( !SUCCEEDED( CoCreateInstance( CLSID_DirectMusicPerformance,
    NULL,
    CLSCTX_INPROC,
    IID_IDirectMusicPerformance,
    (void**)&gpPerformance )))
{
    return CleanupAndExit("Couldn't create a performance object");
}
```

```
}

```

The application then initializes the performance by calling **IDirectMusicPerformance::Init**, which creates a DirectMusic object that can be used to create and activate its ports. Depending on whether or not DirectSound is being used for other sound effects, the call either attaches the existing DirectSound object to the performance by passing in *lpDS* or creates one by passing in NULL.

```
#ifdef USE_DSOUND
    if( !SUCCEEDED(gpPerformance->Init(&gpDirectMusic,
        lpDS, hWndMain)))
    {
        return CleanupAndExit("Couldn't initialize the performance");
    }
#else
    if( !SUCCEEDED(gpPerformance->Init(&gpDirectMusic,
        NULL, hWndMain)))
    {
        return CleanupAndExit("Couldn't initialize the performance");
    }
#endif

```

The application now gets the default port, creates an instance of it with one channel group, and retrieves its capabilities:

```
IDirectMusicPort* pPort = NULL;
DMUS_PORTPARAMS dmos;
DMUS_PORTCAPS dmpc;
GUID guidSynthGUID;
HRESULT hr = S_OK;

if ( !SUCCEEDED(gpDirectMusic->GetDefaultPort(&guidSynthGUID))
{
    return CleanupAndExit("Could't GetDefaultPort on IDirectMusic");
}

ZeroMemory(&dmos, sizeof(dmos));
dmos.dwSize = sizeof(DMUS_PORTPARAMS);
dmos.dwChannelGroups = 1;
dmos.dwValidParams = DMUS_PORTPARAMS_CHANNELGROUPS;

if( !SUCCEEDED(gpDirectMusic->CreatePort(guidSynthGUID,
    &dmos,
    &pPort,
    NULL)))
{
    return CleanupAndExit("Couldn't CreatePort on IDirectMusic");
}

```

```

}

ZeroMemory(&dmpc, sizeof(dmpc));
dmpc.dwSize = sizeof(DMUS_PORTCAPS);

if( !SUCCEEDED(pPort->GetCaps(&dmpc)))
{
    if (pPort) pPort->Release();
    return CleanupAndExit("Couldn't GetCaps on IDirectMusicPort");
}

```

The behavior of the application now varies depending on whether `_SOFTWARE_SYNTH_` is defined. If it is, a synthesizer with DLS capabilities is wanted, so the application checks for the `DMUS_PC_DLS` capabilities flag on the default port. If it fails to find that flag, it goes on to free the default port and enumerate available ports until it finds an output port that has the `DMUS_PC_DLS` capability. Finally, it creates an instance of that port.

```

if ((dmpc.dwClass != DMUS_PC_OUTPUTCLASS)
    || !(dmpc.dwFlags & DMUS_PC_DLS))
{
    pPort->Release();
    pPort = NULL;
}

if (!pPort)
{
    for (DWORD index = 0; hr == S_OK; index++)
    {
        ZeroMemory(&dmpc, sizeof(dmpc));
        dmpc.dwSize = sizeof(DMUS_PORTCAPS);

        hr = gpDirectMusic->EnumPort(index, &dmpc);
        if(hr == S_OK)
        {
            if ( (dmpc.dwClass == DMUS_PC_OUTPUTCLASS) &&
                (dmpc.dwFlags & DMUS_PC_DLS) )
            {
                CopyMemory(&guidSynthGUID, &dmpc.guidPort,
                    sizeof(GUID));

                ZeroMemory(&dmos, sizeof(dmos));
                dmos.dwSize = sizeof(DMUS_PORTPARAMS);
                dmos.dwChannelGroups = 1;
                dmos.dwValidParams = DMUS_PORTPARAMS_CHANNELGROUPS;
            }
        }
    }
}

```

```
        hr = gpDirectMusic->CreatePort(guidSynthGUID,
            &dmos, &pPort, NULL);
        break;
    }
}
}
if (hr != S_OK)
{
    if (pPort) pPort->Release();
    return CleanupAndExit("Couldn't initialize the Synth port");
}
}
```

If, on the other hand, `_SOFTWARE_SYNTH_` is not defined, a legacy hardware port is wanted, and the application goes on to enumerate ports until it finds the MIDI mapper. (That code is omitted here.)

Now the port is activated and attached to the performance.

```
pPort->Activate(TRUE);
gpPerformance->AddPort(pPort);
```

The next call maps PChannels 0-15 to the first group of MIDI channels on the port. Note that this step is necessary because the application did not pass `NULL` to `AddPort`.

```
gpPerformance->AssignPChannelBlock(0, pPort, 1);
```

The original reference to the port can now be released. This call doesn't remove the port from the performance.

```
if (pPort) pPort->Release();
```

Next: Step 3: Load the Music Elements

Step 3: Load the Music Elements

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

The next step in setting up the DirectMusic functionality of the DMDonuts sample application is to load the music elements—styles, templates, motifs, chordmaps, and

bands—from file. Like the previous step, this one is carried out in the `InitializeGame` function.

First the application creates the loader object:

```
if (!SUCCEEDED(::CoCreateInstance(
    CLSID_DirectMusicLoader,
    NULL,
    CLSCTX_INPROC,
    IID_IDirectMusicLoader,
    (void**)&gpLoader
)))
{
    return CleanupAndExit("Couldn't create a loader object");
}
```

Then it sets the search directory for all object types and enables the object cache. This second call simply confirms the default cache status.

```
hr = E_FAIL;

/* The GetSearchPath function gets the media directory
   from the registry and returns it in wszSearchPath. */

if (GetSearchPath(wszSearchPath))
{
    hr = gpLoader->SetSearchDirectory(
        GUID_DirectMusicAllTypes, wszSearchPath, FALSE);
}

/* If that directory doesn't exist, try the current directory. */

if (FAILED(hr))
{
    hr = gpLoader->SetSearchDirectory(GUID_DirectMusicAllTypes,
        L".", FALSE);
}

if (FAILED(hr))
{
    return CleanupAndExit("Couldn't set the search directory \
        for the DirectMusic loader");
}

gpLoader->EnableCache(GUID_DirectMusicAllTypes, TRUE);
```

The following code snippet loads the style named "Donuts", which is in the `Donuts.sty` file. Note that because the application hasn't called **IDirectMusicLoader::ScanDirectory** to build a database of objects that can be loaded by internal name, the first call to **IDirectMusicLoader::GetObject** will fail.

The fallback procedure is to identify the object by file name and call **GetObject** again.

```

IDirectMusicObject* pObject = NULL;
DMUS_OBJECTDESC ObjectDescript;

ObjectDescript.dwSize = sizeof(DMUS_OBJECTDESC);
ObjectDescript.guidClass = CLSID_DirectMusicStyle;
wcscpy(ObjectDescript.wszName, L"Donuts");
ObjectDescript.dwValidData = DMUS_OBJ_CLASS | DMUS_OBJ_NAME;

if (!SUCCEEDED(gpLoader->GetObject(&ObjectDescript,
    IID_IDirectMusicStyle, (void**)&gapStyle[1])))
{
    wcscpy(ObjectDescript.wszFileName, L"Donuts.sty");
    ObjectDescript.dwValidData = DMUS_OBJ_CLASS | DMUS_OBJ_FILENAME;
    if (!SUCCEEDED(gpLoader->GetObject(&ObjectDescript,
        IID_IDirectMusicStyle, (void**)&gapStyle[1])))
    {
        return CleanupAndExit("Couldn't load style object 1");
    }
}

```

The **InitializeGame** function then goes on to load another style and two templates in similar fashion.

The next step is to initialize an array of motifs that will be used to mark certain events in the game. Each motif is contained in a style and is obtained by calling the **IDirectMusicStyle::GetMotif** method, which creates a segment for the motif. The method must be supplied with the internal name of the motif, as in the following example from **Donuts.cpp**.

```

WCHAR awszMotifs[NUM_MOTIFS][64];

wcscpy(awszMotifs[MOTIF_BOUNCE], L"Bounce");
.
.
.
hr = gapStyle[0]->GetMotif(awszMotifs[MOTIF_BOUNCE],
    &(gapMotif[0][MOTIF_BOUNCE]));

```

If you look at the complete code, you'll see that the application loads two sets of motifs with the same names, one set from the "Donuts" style and the other from "Donutz." **DMDonuts** switches between these two styles as the player moves from level to level. When a *gapMotif* is played, its first index is determined by the value of the global *gnCurrentStyle*, ensuring that it is the correct motif for that level.

The application now initializes four chordmaps for each style. These are obtained from separate files.

```
WCHAR awszChordMap[NUM_STYLES][NUM_CHORDMAP][64];
wcscpy(awszChordMap[0][0], L"minaeo.per");
wcscpy(awszChordMap[0][1], L"minfunc.per");
wcscpy(awszChordMap[0][2], L"mipedpt.per");
wcscpy(awszChordMap[0][3], L"tension.per");
wcscpy(awszChordMap[1][0], L"dianoble.per");
wcscpy(awszChordMap[1][1], L"minpedpt.per");
wcscpy(awszChordMap[1][2], L"mippjazz.per");
wcscpy(awszChordMap[1][3], L"minjazz.per");

ObjectDescript.guidClass = CLSID_DirectMusicChordMap;
ObjectDescript.dwValidData = DMUS_OBJ_CLASS | DMUS_OBJ_FILENAME;

for (short n = 0; n < NUM_STYLES; n++)
{
    for (short m = 0; m < NUM_CHORDMAP; m++)
    {
        if (hr == S_OK)
        {
            wcscpy(ObjectDescript.wszFileName,
                awszChordMap[n][m]);
            hr = gpLoader->GetObject(&ObjectDescript,
                IID_IDirectMusicChordMap,
                (void**)&gapChordMap[n][m]);
        }
    }
}

if (hr != S_OK)
{
    return CleanupAndExit("Couldn't load a ChordMap");
}
```

The last elements to be retrieved from the styles are the bands. Each style has two different bands: one for when the player's ship isn't shielded and one for when it is. Note that the names of the bands are allocated differently than were the motifs and chordmaps—each is a **BSTR** rather than a local array of **WCHAR**—but the effect is the same, because in the Win32® API a **BSTR** is a pointer to a **WCHAR** array.

As each band is loaded, it is downloaded to the performance, making available the DLS data for its instruments.

```
BSTR bstrDefault = SysAllocString(L"Default 2");
BSTR bstrShields = SysAllocString(L"Shields");
```

```
for (n = 0; n < NUM_STYLES; n++)
{
    if (hr == S_OK)
    {
        hr = gapStyle[n]->GetBand(bstrShields, &gapShieldBand[n]);
    }
    if (hr == S_OK)
    {
        hr = gapShieldBand[n]->Download(gpPerformance);
    }
    if (hr == S_OK)
    {
        hr = gapStyle[n]->GetBand(bstrDefault, &gapDefaultBand[n]);
    }
    if (hr == S_OK)
    {
        hr = gapDefaultBand[n]->Download(gpPerformance);
    }
}

SysFreeString(bstrDefault);
SysFreeString(bstrShields);
```

After some error-checking code, the InitializeGame function goes on to create segments from the four bands it has obtained. These segments will be used to "play" the band changes at the appropriate times. Once the segments have been created, the band interfaces are released.

```
for (n = 0; n < NUM_STYLES; n++)
{
    if (hr == S_OK)
    {
        hr = gapShieldBand[n]->CreateSegment(&gapShieldSegment[n]);
    }
    if (hr == S_OK)
    {
        hr = gapDefaultBand[n]->CreateSegment(&gapDefaultSegment[n]);
    }
    apShieldBand[n]->Release();
    apDefaultBand[n]->Release();
}
```

Next: Step 4: Set Up Notifications

Step 4: Set Up Notifications

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

The last bit of DirectMusic code inside the InitializeGame function is a request for notification when playback of the primary segment reaches a beat boundary or the end of the segment. As you'll see later in this tutorial, the notification mechanism is used to limit the number of "blowup" motifs that can be played at the same time.

The following code adds the notification types. Note that the notification GUIDs have to be placed in a variable, because they are passed by reference.

```
GUID guid;  
guid = GUID_NOTIFICATION_SEGMENT;  
gpPerformance->AddNotificationType( guid );  
guid = GUID_NOTIFICATION_MEASUREANDBEAT;  
gpPerformance->AddNotificationType( guid );
```

Next: Step 5: Create the Primary Segments

Step 5: Create the Primary Segments

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

The one-time initialization of DirectMusic in the DMDonuts sample application is now complete. Further initialization is done in the setup_game function each time the game is started or the player advances to the next level.

The setup_game function first calls another function called ComposeNewSegments, which selects a style and chordmap based on the current game level, releases any previously created segments, and then composes two new segments based on the templates previously loaded from file. One of these segments is an introductory theme that plays until a donut has been hit; the other plays from that point until the player beats the level. The *snSubLevel* variable tracks which part of the level we are in and is used elsewhere to determine which segment should be played when music is toggled on with the F7 key.

```
snSubLevel = 1;
```

```

snLastTempo = 0;
gnLastStyle = gnCurrentStyle;
gnLastChordMap = gnCurrentChordMap;
gnCurrentStyle = level % NUM_STYLES;
gnCurrentChordMap = ((level - 1) / 2) % NUM_CHORDMAP;
HRESULT hr = S_OK;
MUSIC_TIME mtSegmentLength;
if (gapSegment[SEGMENT_1])
{
    gapSegment[SEGMENT_1]->Release();
}
hr = gpComposer->ComposeSegmentFromTemplate(
    gapStyle[gnCurrentStyle],
    gpIntroTemplate, 0,
    gapChordMap[gnCurrentStyle][gnCurrentChordMap],
    &gapSegment[SEGMENT_1]
);
if (!SUCCEEDED(hr))
{
    CleanupAndExit("Segment 1 composition failed");
}

```

We happen to know that the template has a signpost on the last measure that matches the signpost on the first measure, for graceful looping. Once the segment is composed, the last measure is lopped off. Note that the value of `MEASURE_LENGTH` was calculated in Step 1: Defines and Globals as `(DMUS_PPQ * 4)`—that is, four quarter-notes of music time. This is valid because the time signature is 4/4.

```

gapSegment[SEGMENT_1]->GetLength(&mtSegmentLength);
gapSegment[SEGMENT_1]->SetLength(
    mtSegmentLength - MEASURE_LENGTH);

```

Finally, the segment is set to loop repeatedly.

```

gapSegment[SEGMENT_1]->SetRepeats(999);

```

The second segment, based on *gpGameTemplate*, is composed and set up the same way.

Next: Step 6: Play a Primary Segment

Step 6: Play a Primary Segment

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[C++]

In the previous step, the `setup_game` function in `Donuts.cpp` called the `ComposeNewSegments` function, which created two primary segments to be used for the current game level. Now the `setup_game` function plays the first of these segments:

```
gpPerformance->PlaySegment(  
    gapSegment[SEGMENT_1], 0, 0, NULL);
```

Note that the absence of any flag causes the segment to be played immediately.

The function now plays the default band segment that was created for the current style. (Remember, there are two different styles that alternate when the game level changes. Each style has two bands, the default band and the "shield" band.) Playing the band segment ensures that the correct set of instruments is playing the music. Note that the band segment must be flagged as secondary. No start time is given in the third parameter because the changes are to be made immediately, and the last parameter is `NULL` because we don't require a pointer to the segment state.

```
gpPerformance->PlaySegment(  
    gapDefaultSegment[gnCurrentStyle],  
    DMUS_SEGF_SECONDARY,  
    0, NULL);
```

The default band for the style is used when the player's ship is unshielded, which is always the case at the beginning of a level. The application plays the second band inside the `UpdateDisplayList` function in response to the shields being turned on, and plays the default band again when the shields are turned off.

Next: Step 7: Transition to Another Primary Segment

Step 7: Transition to Another Primary Segment

[Visual Basic]

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[C++]

DMDonuts cues a different primary segment in two places: when the game level changes, and when the first hit is made on a donut on any level. The following code is from the `CheckForHits` function. When the function finds that a hit is the first to have

occurred on this level, it tells the DirectMusic composer to create a transition from the current (introductory) theme to the action theme. The call to **IDirectMusicComposer::AutoTransition** also cues the transition and the following action segment so that they play automatically; in this case, the transition will start on the next measure boundary.

```
if (gapSegment[SEGMENT_TRANS_1])
{
    gapSegment[SEGMENT_TRANS_1]->Release();
}
gpComposer->AutoTransition(
    gpPerformance,           // The performance
    gapSegment[SEGMENT_2],   // The next primary segment
    DMUS_COMMANDT_FILL,      // Embellishment type
    DMUS_COMPOSEF_MODULATE | // Modulate to new key
    DMUS_COMPOSEF_MEASURE,    // and start on measure
    gapChordMap[gnCurrentStyle] [gnCurrentChordMap],
                             // Use current chordmap
    &gapSegment[SEGMENT_TRANS_1], // Created transition segment
    NULL, NULL                // No segment states needed
);
```

Note that you don't have to stop the first segment. It stops automatically when a transition or new primary segment is played.

Next: Step 8: Play a Motif

Step 8: Play a Motif

[Visual Basic]

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[C++]

DMDonuts uses several motifs to accent the game action: for example, when the player's ship bounces off the edge of the screen and when an object is hit by the player's fire.

Playing a motif is very simple, as the following code from the CheckForHits function shows. The MOTIF_BLOWUP motif is played whenever a donut is hit.

```
gpPerformance->PlaySegment(
    gapMotif[gnCurrentStyle][MOTIF_BLOWUP],
    DMUS_SEGF_SECONDARY | DMUS_SEGF_GRID, 0, NULL);
```


The motif must be played as a secondary segment so that it does not interrupt the main theme being played as the primary segment. It is cued to play on a grid boundary, the lowest resolution at which it can play without being out of step with the primary segment. The *rtStartTime* parameter is 0, indicating that the segment should play as soon as the first boundary is reached, and the *ppSegmentState* parameter is NULL because there's no need to access the segment while it is playing.

Next: Step 9: Handle Notifications

Step 9: Handle Notifications

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

Back in Step 4: Set Up Notifications we saw how DMDonuts requests notification whenever a segment ends and whenever a beat or measure boundary is reached. The purpose of this notification is to help limit the number of "blowup" motifs playing at any one time.

Notifications are retrieved in the **WinMain** function. After dealing with any messages in the queue, the application checks to see if there are any pending notifications. It does this by calling the **IDirectMusicPerformance::GetNotificationPMsg** method. If no notification message is pending, the method returns **S_FALSE** and no further action is needed. A return of **S_OK** indicates that a notification message has been placed in **pEvent*. Because it has only requested notifications relevant to the beat, the application doesn't actually need to examine the message, which is immediately discarded. It then resets *snMaxBlowUps*, which tracks how many more blowup motifs can be played till the next beat.

```
if (gpPerformance)
{
    DMUS_NOTIFICATION_PMSG* pEvent;
    while ( S_OK == gpPerformance->GetNotificationPMsg(&pEvent))
    {
        gpPerformance->FreePMsg((DMUS_PMSG*)pEvent);
        snMaxBlowUps = BLOWUPS_PER_BEAT;
    }
}
```

Next: Step 10: Shut Down DirectMusic

Step 10: Shut Down DirectMusic

[\[Visual Basic\]](#)

This tutorial pertains only to applications written in C++. See DirectMusic Visual Basic Tutorials.

[\[C++\]](#)

Closing down DirectMusic is a matter of clearing the loader cache, stopping the performance, releasing all the objects that have been created, and finally dereferencing COM (remember, every call to **CoInitialize** must have a matching call to **CoUninitialize**).

The following function performs the necessary cleanup in Donuts.cpp:

```
void CleanUpDMusic()
{
    if (gpLoader)
    {
        gpLoader->ClearCache(GUID_DirectMusicAllTypes);
        gpLoader->Release();
    }

    if (gpComposer)
    {
        gpComposer->Release();
    }

    if (gpIntroTemplate)
    {
        gpIntroTemplate->Release();
    }

    if (gpGameTemplate)
    {
        gpGameTemplate->Release();
    }

    for (short n = 0; n < NUM_STYLES; n++)
    {
        if (gapShieldBand[n])
        {
            gapShieldBand[n]->Unload(gpPerformance);
            gapShieldBand[n]->Release();
        }
        if (gapDefaultBand[n])
        {

```

```
        gapDefaultBand[n]->Unload(gpPerformance);
        gapDefaultBand[n]->Release();
    }
    if (gapShieldSegment[n])
    {
        gapShieldSegment[n]->Release();
    }

    if (gapDefaultSegment[n])
    {
        gapDefaultSegment[n]->Release();
    }

    if (gapStyle[n])
    {
        gapStyle[n]->Release();
    }

    for (short m = 0; m < NUM_CHORDMAP; m++)
    {
        if (gapChordMap[n][m])
        {
            gapChordMap[n][m]->Release();
        }
    }

    for (m = 0; m < NUM_MOTIFS; m++)
    {
        if (gapMotif[n][m])
        {
            gapMotif[n][m]->Release();
        }
    }
}

if (gpPerformance)
{
    gpPerformance->Stop( NULL, NULL, 0, 0 );
    gpPerformance->CloseDown();
    gpPerformance->Release();
}

for (n = 0; n < NUM_SEGMENTS; n++)
{
    if (gapSegment[n])
    {
```

```
        gapSegment[n]->Release();
    }
}

if (gpDirectMusic)
{
    gpDirectMusic->Release();
}

CoUninitialize();
}
```

DirectMusic Visual Basic Tutorials

[C++]

This section pertains only to DirectX for Visual Basic. See DirectMusic C/C++ Tutorials.

[Visual Basic]

This section contains the following tutorials showing how to implement DirectMusic in a Visual Basic application:

- Tutorial 1: Playing a MIDI File
 - Tutorial 2: Using DLS Sound Effects
-

Tutorial 1: Playing a MIDI File

[C++]

This section pertains only to DirectX for Visual Basic. A similar tutorial is available for C/C++; see Tutorial 1: Playing a MIDI File.

[Visual Basic]

This tutorial is a guide to setting up the simplest possible DirectMusic application, one that just plays a primary segment. In this example, the segment represents a MIDI file, but the process of loading and playing the data would be exactly the same if the source were a segment authored in a tool such as DirectMusic Producer.

The tutorial is divided into the following steps:

- Step 1: Create the Performance and Loader
 - Step 2: Load the MIDI File
 - Step 3: Play the MIDI File
-

Step 1: Create the Performance and Loader

[C++]

This topic pertains only to DirectX for Visual Basic. See DirectMusic C/C++ Tutorials.

[Visual Basic]

The central object of any DirectMusic application is an instance of the **DirectMusicPerformance** class. Most applications require only a single performance object, although it is possible to have multiple performances playing to different ports, with different tempos, and so on. To load any musical data, the application also needs a **DirectMusicLoader** object.

To create these objects, you must first declare a global **DirectX7** object.

The following global declarations create the DirectX7 object and variables for the other necessary objects:

```
Public gObjDX As New DirectX7
Public gObjDMLoader As DirectMusicLoader
Public gObjDMPPerformance As DirectMusicPerformance
```

In the startup code, create the performance and loader objects as follows:

```
Set gObjDMLoader = gObjDX.DirectMusicLoaderCreate
Set gObjDMPPerformance = gObjDX.DirectMusicPerformanceCreate
```

You must also initialize the performance and assign a port to it. The following code has the performance create its own **DirectSound** object and assigns the default port, which is always either the Microsoft Software Synthesizer or a DLS-capable hardware synthesizer. A single channel group is assigned to the port because a MIDI file requires a maximum of 16 channels. If you are writing an application that plays other kinds of segments as well, you might need to increase this number.

```
' hWnd is the window handle of the main form
Call gObjDMPPerformance.Init(Nothing, hWnd)
Call gObjDMPPerformance.SetPort(-1, 1)
```

Next: Step 2: Load the MIDI File

Step 2: Load the MIDI File

[C++]

This topic pertains only to DirectX for Visual Basic. See DirectMusic C/C++ Tutorials.

[Visual Basic]

To load a MIDI file, you need three things:

- The **DirectMusicLoader** object created in the previous step.
- A **DirectMusicSegment** variable.
- The name of the MIDI file.

If your music files are not in the working directory of the application, call **DirectMusicLoader.SetSearchDirectory** so that the loader knows where to look, or provide a full path for each file.

The following code sets the search directory for the loader and loads a file from that directory.

```
Call gObjDMLoader.SetSearchDirectory("c:\midiplay\media")  
' This needs to be done only once.
```

```
Dim objSeg As DirectMusicSegment
```

```
' FileName is a string containing the short name of the file  
Set objSeg = gObjDMLoader.LoadSegment(FileName)
```

Next: Step 3: Play the MIDI File

Step 3: Play the MIDI File

[C++]

This topic pertains only to DirectX for Visual Basic. See DirectMusic C/C++ Tutorials.

[Visual Basic]

Now that you have a **DirectMusicSegment** object representing the MIDI file, you need to do a little more preparation before you can play it.

First, let the segment know that it represents a standard MIDI file. Although the segment can be played without this step, certain details of playback might not be

handled properly. The following code performs this step for files with the conventional MIDI file extension:

```
If StrConv(Right(FileName, 4), vbLowerCase) = ".mid" Then
    Call objSeg.SetStandardMidiFile
End If
```

Second, ensure that the DLS data for the instruments is downloaded to the port. The simplest way to do this is to turn on automatic downloading for the entire performance, as might be done in the setup code, as follows:

```
Call gObjDMPPerformance.SetMasterAutoDownload(True)
```

The disadvantage of this method is that the instruments are unloaded from the port as soon as the segment stops, and must then be downloaded again. If your application is playing different segments in rapid succession, this behavior might cause unacceptable delays. For greater control, you can download the instruments each time a segment is about to be played. The instruments remain on the port until you choose to unload them, or until the performance is closed down. Any redundant downloads are ignored.

```
Call objSeg.Download(gObjDMPPerformance)
```

Now, the segment can be played:

```
Call gObjDMPPerformance.PlaySegment(objSeg, 0, 0)
```

If you want to track the state of playback (for example, to be able to pause and resume), you can save the return value of **DirectMusicPerformance.PlaySegment** in a **DirectMusicSegmentState** object:

```
Dim objSegState as DirectMusicSegmentState
Set objSegState = gObjDMPPerformance.PlaySegment(gObjPrimarySeg, 0, 0)
```

Tutorial 2: Using DLS Sound Effects

[C++]

This topic pertains only to application development in DirectX for Visual Basic.

[Visual Basic]

This tutorial shows how to use DLS to create nonmusical sound effects, using DirectMusic. The code example is based on the DLSEffects sample, which uses the DLS collection file Boids.dls. If you like, you can use DirectMusic Producer to examine this collection to better understand some of the concepts discussed in the

tutorial. Consult the DirectMusic Producer documentation for more information on how to view and edit a DLS collection.

The tutorial is divided into the following steps:

- Step 1: Load the Sounds
 - Step 2: Play Effects
 - Step 3: Modify and Stop Effects
-

Step 1: Load the Sounds

[C++]

This topic pertains only to application development in DirectX for Visual Basic.

[Visual Basic]

After creating and initializing the performance, the DLSEffects sample loads the DLS file and a segment file, both from the directory represented by *mediapath*.

```
' dx is a DirectX7 object.  
' perf is an initialized DirectMusicPerformance object.  
' seg is declared as DirectMusicSegment.  
  
mediapath = FindMediaDir("sample.sgt", "dmusic")  
If mediapath <> vbNullString Then ChDir mediapath  
  
Dim loader As DirectMusicLoader  
Set loader = dx.DirectMusicLoaderCreate  
Set coll = loader.LoadCollection(mediapath & "boids.dls")  
Set seg = loader.LoadSegment(mediapath & "sample.sgt")
```

It does not matter what segment is loaded. The application does not play the segment, but it is necessary to have a **DirectMusicSegment** object to download DLS data.

Now, the application associates the **DirectMusicCollection** object with the segment and downloads it to the port. Any other instruments referred to by the segment are downloaded as well.

```
Call seg.ConnectToCollection(coll)  
Call seg.Download(perf)
```

The Boids.dls collection contains only a single instrument, called Vocals, which has patch number 127. You can assign this instrument to any channel by sending a MIDI patch message. The application assigns the instrument to two channels so that one of the samples that is to be played, the heartbeat sound, can be given a pitch bend without affecting other samples.


```
Const channel = 1
Const hbchannel = 2
Const patch = 127
Call perf.SendPatchPMSG(0, DMUS_PMSGF_REFTIME, channel, patch, 0, 0)
Call perf.SendPatchPMSG(0, DMUS_PMSGF_REFTIME, hbchannel, patch, 0, 0)
```

Now, any note played on channels 1 or 2 uses the Vocals instrument. How this instrument is used to get different sound effects is covered in Step 2: Play Effects.

Step 2: Play Effects

[C++]

This topic pertains only to application development in DirectX for Visual Basic.

[Visual Basic]

In the DLSEffects sample, the user can hear a variety of sound effects, including laughter, fragments of speech in several languages, and an infant's heartbeat. All these sounds are contained in a single DLS instrument.

Completely different sounds can be played by a single instrument because different note ranges, or *regions*, can be associated with different wave samples. In Boids.dls, for example, the first speech fragment is assigned to the pitch range from C0 through B3. Any note in that range is based on the same wave sample. The root note is C3, meaning that when a C3 note is played, the wave is heard at its original frequency. All other notes cause the pitch of the wave to be modified.

The following procedure sends a note at the volume indicated by the global variable *gVelocity*:

```
Private Sub SendNote(chan As Integer, pitch As Byte, dur As Long)

    Dim noteMsg As DMUS_NOTE_PMSG
    noteMsg.velocity = gVelocity
    noteMsg.flags = DMUS_NOTEF_NOTEON
    noteMsg.midiValue = pitch
    noteMsg.mtDuration = dur
    Call perf.SendNotePMSG(0, DMUS_PMSGF_REFTIME, chan, noteMsg)

End Sub
```

All but one of the waves in Boids.dls are one-shot samples. This means that when a note is played, the wave sample plays only once, regardless of the duration of the note. The exception is the heartbeat sample, which is looped so that it plays as long as the note is playing. This looping behavior was specified by the author of the instrument collection.

The following procedure sends a C3 note. C3 is the root note of the first speech sample, so it plays at the correct pitch. The constant *NoteDur* is 6000 milliseconds, allowing enough time for this or any of the other waves to play completely before DirectMusic generates a note-off MIDI message. (If you click the button and send an identical note before this time has elapsed, the note-off message might cancel the new note as well, so it might not play fully.)

```
Private Sub cmdC3_Click()  
    SendNote channel, 36, NoteDur  
End Sub
```

The DLSEffects application plays most of the samples at their root note. However, the heartbeat sample is treated a little differently, as an illustration of how the pitch of sound effects can be modified programmatically. This topic is covered in Step 3: Modify and Stop Effects.

Step 3: Modify and Stop Effects

[C++]

This topic pertains only to application development in DirectX for Visual Basic.

[Visual Basic]

In the definition of the Vocals instrument in Boids.dls, the region of the heart sample is in the range from B7 through B8. As the sound is playing in the sample application, the user can choose a note within this region by using the **Note** slider. Each time the slider is changed, the existing note is stopped, and a new one is sent.

Because the duration of the note is indefinite, it is sent in the form of a standard MIDI note-on message by using **DirectMusicPerformance.SendMIDIPIMSG**. (You could also use **DirectMusicPerformance.SendNotePIMSG**, specifying a very large number for the duration of the note.) The current frequency and velocity are given in the last two parameters. Because the method accepts the channel as a separate parameter, the MIDI status byte (&H90) does not contain the channel number in the 4 lower bits, as it would in a standard MIDI message.

```
Call perf.SendMIDIPIMSG(0, DMUS_PMSGF_REFTIME, hbchannel, _  
    &H90, B7Freq, gVelocity)
```

In addition to changing the note for the heartbeat, the user can also assign a pitch bend to the channel on which the note is played. The pitch bend is also sent as a standard MIDI message:

```
Call perf.SendMIDIPIMSG(0, DMUS_PMSGF_REFTIME, hbchannel, _  
    &HE0, lo, hi)
```

The *lo* and *hi* parameters are 7-bit values, so there are 16,384 possible settings for a pitch bend. For more information, see the MIDI specification.

To stop the heartbeat note, the sample application sends a note of the same pitch as the currently playing note, using the **SendNotePMSG** method, with the **flags** member of the **DMUS_NOTE_PMSG** type set to 0, instead of the usual **DMUS_NOTEF_NOTEON**.

```
Private Sub B7NoteOff()  
  
    Dim noteMsg As DMUS_NOTE_PMSG  
    noteMsg.flags = 0  
    noteMsg.midiValue = B7Freq ' B7Freq is pitch of last note-on  
    Call perf.SendNotePMSG(0, DMUS_PMSGF_REFTIME, hbchannel, noteMsg)  
End Sub
```

The pitch bend on the channel does not affect this operation.

DirectMusic Samples

This section describes sample applications included with the DirectX SDK that demonstrate the use of DirectMusic. Descriptions are organized as follows:

- DirectMusic C/C++ Samples
- DirectMusic Visual Basic Samples

DirectMusic C/C++ Samples

[\[Visual Basic\]](#)

This section pertains only to application development in C and C++. See DirectMusic Visual Basic Samples.

[\[C++\]](#)

The following sample applications demonstrate the use and capabilities of the DirectMusic® application programming interface:

- 3DMusic Sample
- DMBoids Sample
- DMDonuts Sample
- DirectMusic MIDI Sample
- DirectMusic Shell Sample
- EchoTool Sample
- MusicLines Sample

- PlayMotif Sample
- PlayPrimary Sample

Although DirectX® samples include Microsoft® Visual C++® project workspace files, you might need to verify other settings in your development environment to ensure that the samples compile properly. For more information, see [Compiling DirectX Samples and Other DirectX Applications](#).

3DMusic Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C and C++. See [DirectMusic Visual Basic Samples](#).

[\[C++\]](#)

Description

The 3DMusic sample shows how to play a MIDI file into a custom DirectSound 3-D buffer.

Path

Source: *(SDK root)\Samples\Multimedia\DMusic\Src\3DMusic*

Executable: *(SDK root)\Samples\Multimedia\DMusic\Bin\3DMusic.exe*

User's Guide

The icons represents a listener and a sound source in a 3-D environment, although movement and orientation are possible in only two dimensions.

Use the arrow keys to move the sound source. Use the numeric keypad to change the orientation of the listener.

Choosing **Play Stereo Drip** from the **File** menu causes a dripping sound to play at random pan and volume, illustrating how pseudo-3-D effects can be achieved with a 2-D buffer.

Programming Notes

The program uses the registry key set up by the DirectX SDK setup to find the media file path.

Helper.cpp contains useful functions that set up DirectMusic. These functions are called from Sound.cpp.

DMBoids Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C and C++. See DirectMusic Visual Basic Samples.

[\[C++\]](#)

Description

DMBoids is a version of Boids that adds DirectMusic support. As objects fly over a simple landscape, the music responds to user input and events on the screen.

Path

Source: *(SDK root)*\Samples\Multimedia\DMusic\Src\DMBoids

Executable: *(SDK root)*\Samples\Multimedia\DMusic\Bin\Dmboids.exe

User's Guide

Press F10 to access the main menu. The **Drivers** menu allows you to change the driver, device, and video mode. The application runs only in full-screen modes.

The A (alignment), C (cohesion) and O (obstacle) keys alter behavior of the boids in various ways as long as they are held down.

Hold down the S key or the spacebar and the birds flock in closer. Release the key and they spread apart. Note the use of motifs to track this behavior.

Hold down the M key and the birds wander off their path. Notice that the music completely changes. Release and the birds will eventually get back on the path.

Press the ESC key to quit.

Programming Notes

DirectMusic features illustrated include the following:

- Software synthesis with DLS. In addition to the musical instruments from the GS sound set, the application uses custom downloadable sounds such as the voices that appear to come from the planets.
- Composing and performing style-based segments.
- Musical transitions using style-based motifs and segment cues.
- Echo/articulation tool coded that uses the proximity of the birds to adjust the echoes and note durations of the music as it plays.

DMDonuts Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C and C++. See DirectMusic Visual Basic Samples.

[\[C++\]](#)

Description

DM Donuts is a variation of the Space Donuts sample with the addition of music that responds to game events.

Path

Source: (*SDK root*)\Samples\Multimedia\Dmusic\Src\DMDonuts

Executable: (*SDK root*)\Samples\Multimedia\DMusic\Bin\Dmdonuts.exe

User's Guide

While the donut floats in space, notice that the music is subtle and spacey. Press the arrow keys to move the ship. Bounce around and see how the music and rhythm respond.

Shoot the donut by firing with the space bar. Immediately there is an explosion of music and the background music grows in intensity.

Press the 7 key on the numeric keypad. This turns on the shields. Notice how the music becomes muted as if you were listening from inside the shield.

When all the donut fragments are destroyed, notice that the music immediately transitions into an ending, then starts the next level on the start of the next musical theme.

Notice that the music is never the same.

You can control the ship with the joystick if you prefer, by changing the input device under the **Game** menu.

The following is a complete list of game commands. All numbers must be entered from the numeric keypad. "Joy" refers to a joystick button.

Key	Command
ESC, F12	Quit
4	Turn left

6	Turn right
5 (Joy 3)	Stop
8	Accelerate forward
2	Accelerate backward
7 (Joy 2)	Shields
SPACEBAR (JoyFire 1)	
ENTER	Start game
F1	Toggle trailing afterimage effect on/off
F5	Toggle frame rate display on/off
F7	Turn music on/off
F10	Main menu

The display defaults to 640x480 at 256 colors. You can specify a different resolution and pixel depth on the command line.

The game uses the following command line switches, which are case-sensitive:

e	Use software emulation, not hardware acceleration
t	Test mode, no input required
x	Stress mode. Never halt if you can help it

These switches may be followed by three option numbers representing x-resolution, y-resolution, and bits per pixel. For example:

```
donuts -t 800 600 16
```

Programming Notes

Techniques illustrated include the following:

- Composing and performing style-based segments.
- Autotransitions on game state changes.
- Motifs (short musical clips) to highlight actions. Because the motifs track the rhythm and harmony of the underlying music, they add to the music while providing sonic reinforcement.
- Dynamic bands that change the orchestration in response to real-time events.

By default, this sample runs on the software synthesizer. On versions of the operating system that support DLS synthesis in hardware, undefine the `_SOFTWARE_SYNT` compile flag and recompile the sample.

DirectMusic MIDI Sample

[Visual Basic]

This topic pertains only to application development in C and C++. See DirectMusic Visual Basic Samples.

[C++]

Description

The DirectMusic MIDI sample is a simple MIDI file player.

Path

Source: *(SDK root)\Samples\Multimedia\DMusic\Src\DMusMIDI*

Executable: *(SDK root)\Samples\Multimedia\DMusic\Bin\Dmusmidi.exe*

User's Guide

Load a MIDI file by choosing **Open** from the **File** menu. (There is a sample file in the *\Samples\Multimedia\DMusic\Media* folder.) You can also load a file by drag-and-drop. If **Autoplay Dropped Files** is selected on the **Options** menu, a dropped file automatically starts playing.

Once a file is loaded, you can play, pause, or stop it by choosing from the **File** menu or the toolbar.

The **Options** menu provides a choice for adding reverberation to the music, and several choices for the interface. The time can be displayed in hours, minutes, and seconds from the start of play, or in music time ticks.

When no music is playing, you can select a port from the **Device** menu. Note that to hear music from the "MIDI Out" port, you must have an external synthesizer attached.

Programming Notes

The application demonstrates how to load a MIDI file as a segment, play it, stop it, and restart it either at the beginning or at the point where it was stopped. It also shows how to set the reverb property.

The code is written in pure C, so methods are called through pointers to *vtables*. For more information, see *Accessing COM Objects by Using C*.

DirectMusic Shell Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C and C++. See DirectMusic Visual Basic Samples.

[\[C++\]](#)

Description

The DirectMusic Shell sample demonstrates interactive music that responds to Windows system events.

Path

Source: *(SDK root)*\Samples\Multimedia\DMusic\Src\DMShell

Executable: *(SDK root)*\Samples\Multimedia\DMusic\Bin\Dmshell.exe

User's Guide

When you run the program, its icon appears in the tray on the taskbar. Click on the icon to see a menu that allows you to change music schemes, select the output device, start and stop the music, and close the program.

Listen to the music and note how it changes and how motifs are introduced in response to system events such as minimizing, restoring, or closing a window, opening an application menu or the Start menu, and pressing a key (there are special sounds for a few keys).

Programming Notes

The Windows system messages are obtained in Dmhook.dll, the source code for which is found in the *(SDK root)*\Samples\Multimedia\DMusic\Src\DMHook folder. (Dmhook.dll must be compiled with a Microsoft compiler.)

EchoTool Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C and C++. See DirectMusic Visual Basic Samples.

[\[C++\]](#)

Description

The EchoTool sample shows how to implement a tool in DirectMusic.

Path

Source: *(SDK root)\Samples\Multimedia\Dmusic\Src\EchoTool*

Executable: *(SDK root)\Samples\Multimedia\DMusic\Bin*

User's Guide

Select an option button to change the delay of the echoes. Click the **Close** button to exit the application.

Programming Notes

The tool creates up to four delayed echoes of the music playing through it.

Helper.cpp contains useful functions that set up DirectMusic. These functions are called from Main.cpp. Echotool.cpp contains the tool code.

For more information, see Tutorial 2: Using Tools.

MusicLines Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C and C++. See DirectMusic Visual Basic Samples.

[\[C++\]](#)

Description

The MusicLines sample demonstrates interactive music elements in a simple game, and in particular how game elements can be driven by the music.

Path

Source: *(SDK root)\Samples\Multimedia\Dmusic\Src\MusicLines*

Executable: *(SDK root)\Samples\Multimedia\DMusic\Bin*

User's Guide

In the opening dialog box, choose windowed or full-screen mode and a difficulty level, and set the players to human or computer. If two humans are playing, one can use the arrow keys while the other uses the keys AZSW. Start moving by pressing an arrow key.

The object of the game is to force the other player to collide with an existing line.

Change the direction of your line by pressing the arrow keys or the equivalent letter keys. Observe how the main music changes to reflect the current state of play, how motifs are used to signal events such as collisions, and how the speed of the lines is actually controlled by the music.

If you win against the computer, you can continue extending your line or bring the game to an end by deliberately colliding. Play again by pressing the space bar. Quit by pressing ALT+F4.

Programming Notes

The music logic is in Mlmusic.cpp and is amply commented.

PlayMotif Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C and C++. See DirectMusic Visual Basic Samples.

[\[C++\]](#)

Description

This sample application shows how motifs can be played over a primary segment.

Path

Source: *(SDK root)*\Samples\Multimedia\Dmusic\Src\PlayMotf

Executable: *(SDK root)*\Samples\Multimedia\DMusic\Bin

User's Guide

Click the buttons to play the various secondary segments (motifs) on top of the main playing primary segment. Click **Close** to exit the application.

Programming Notes

The program uses the registry key set up by the DirectX SDK setup to find the media file path.

PlayPrimary Sample

[\[Visual Basic\]](#)

This topic pertains only to application development in C and C++. See DirectMusic Visual Basic Samples.

[\[C++\]](#)

Description

The PlayPrimary sample shows how to play a primary segment based on an authored music section.

Path

Source: *(SDK root)\Samples\Multimedia\Dmusic\Src\PlayPri*

Executable: *(SDK root)\Samples\Multimedia\DMusic\Bin*

User's Guide

Click **Close** to quit the program.

Programming Notes

The program uses the registry key set up by the DirectX SDK setup to find the media file path.
