Windows 3.1 Multimedia Reference

This reference is an online reference to the multimedia application programming interface (API) of Windows 3.1. The multimedia API includes the functions, messages, and data structures you can use to create multimedia applications for Windows.

Function Overview

Click one of the following categories to display a brief description of the multimedia functions; you can then display the full information about each individual function.

High-Level Audio Services

High-level audio services allow applications to play audio files directly, while Windows manages audio playback. Use the following functions to play memory-resident waveform sounds specified by filename, system-alert level, or WIN.INI entries:

sndPlaySound	Plays the sound that corresponds to the given filename or WIN.INI entry.

The Media Control Interface also provides high-level audio services. For an overview of the MCI functions, see Media Control Interface Services.

Low-Level Waveform Audio Services

Low-level waveform audio services allow applications to manage waveform audio playback and recording. Click one of the following categories to display a brief description of the low-level waveform audio functions; you can then display the full information about each individual function.

Working with Waveform Devices

- Querying waveform devices
- Opening and closing waveform devices
- Getting the device ID of waveform devices
- Handling waveform errors
- Getting the current position of waveform devices
- Sending custom messages to waveform drivers

Playing Waveform Data

- Playing waveform data
- Controlling waveform playback
- Changing pitch and playback rate
- Changing playback volume and caching patches

Recording Waveform Data

- Recording waveform data
- Controlling waveform recording

Querying Waveform Devices

Before playing or recording a waveform, you must determine the capabilities of the waveform hardware present in the system. Use the following functions to retrieve the number of waveform devices and the capabilities of each device:

waveInGetNumDevs	Retrieves the number of waveform input devices present in the system.
waveInGetDevCaps	Retrieves the capabilities of a given waveform input device.
waveOutGetNumDevs	Retrieves the number of waveform output devices present in the system.
waveOutGetDevCaps	Retrieves the capabilities of a given waveform output device.

Opening and Closing Waveform Devices

You must open a device before you can begin waveform playback or recording. Once you finish using a device, you must close it so that it will be available to other applications. Use the following functions to open and close waveform devices:

waveInOpen	Opens a waveform input device for recording.
waveInClose	Closes a specified waveform input device.
waveOutOpen	Opens a waveform output device for playback.
waveOutClose	Closes a specified waveform output device.

Getting Waveform Device IDs

Using a waveform device handle, you can retrieve the device ID for an open waveform device. Use the following functions to get the device ID:

waveInGetID	Gets the device ID for a waveform input device.
waveOutGetID	Gets the device ID for a waveform output device.

Handling Waveform Errors

Most of the low-level waveform audio functions return error codes. Use these functions to convert the error codes returned from waveform functions into a textual description of the error:

waveInGetErrorText	Retrieves a textual description of a specified waveform input error.
waveOutGetErrorText	Retrieves a textual description of a specified waveform output error.

Getting Waveform Device Positions

While playing or recording waveform audio, you can query the device for the current playback or recording position. Use the following functions to determine the current position of a waveform device:

waveInGetPosition	Retrieves the current recording position of a waveform input device.
waveOutGetPosition	Retrieves the current playback position of a waveform output device.

Playing Waveform Data

After opening a waveform output device, you can begin sending data blocks to the device. Before sending data blocks to the device, each data block must be prepared. Use the following functions to prepare data blocks and send them to a waveform output device:

waveOutPrepareHeader	Informs the waveform output device driver that the given data block should be prepared for playback.
waveOutWrite	Writes a data block to a waveform output device.
waveOutUnprepareHeader	Informs the waveform output device driver that the preparation performed on the given data block can be cleaned up.

Controlling Waveform Playback

Waveform playback begins as soon as you begin sending data to the waveform output device. Use the following functions to pause, restart, or stop playback and to break loops on a waveform device:

waveOutBreakLoop	Breaks a loop on a waveform output device.
waveOutPause	Pauses playback on a waveform output device.
waveOutRestart	Resumes playback on a paused waveform output device.
waveOutReset	Stops playback on a waveform output device. Marks all pending data blocks as done.

Changing Waveform Pitch and Playback Rate

Some waveform output devices can scale the pitch and the playback rate when playing a waveform. Both of these operations have the effect of changing the pitch of the waveform. Use these functions to query and set waveform pitch and playback rate scale factors:

waveOutGetPitch	Queries the pitch scale factor for a waveform output device.
waveOutGetPlaybackRate	Queries the playback rate scale factor for a waveform output device.
waveOutSetPitch	Sets the pitch scale factor for a waveform output device.
waveOutSetPlaybackRate	Sets the playback rate scale factor for a waveform output device.

Changing Waveform Playback Volume

Some waveform output devices support changes to the playback volume level. Use these functions to query and set the volume level of waveform output devices:

waveOutGetVolume	Queries the current volume level of a waveform output device.
waveOutSetVolume	Sets the volume level of a waveform output device.

Recording Waveform Data

After opening a waveform input device, you can begin recording waveform data. To record waveform data, you must supply the waveform input device with data buffers. These data buffers must be prepared before being sent to the waveform device. Use the following functions to prepare data buffers and send them to waveform input devices:

waveInAddBuffer	Sends a data buffer to a waveform output device. The data buffer is filled with recorded waveform data and sent back to the application.
waveInPrepareHeader	Informs the waveform input device driver that the given data buffer should be prepared for recording.
waveInUnprepareHeader	Informs the waveform input device driver that the preparation performed on the given data buffer can be cleaned up.

Controlling Waveform Recording

When recording waveform audio, you can control when recording starts and stops. Use the following functions to start and stop recording on a waveform input device:

waveInStart	Starts recording on a waveform input device.
waveInStop	Stops recording on a waveform input device.
waveInReset	Stops recording on a waveform input device. Marks all pending data blocks as done.

Sending Custom Messages to Waveform Drivers

The following functions let you send messages directly to waveform drivers:

waveInMessage	Sends a message directly to a waveform input device driver.
waveOutMessage	Sends a message directly to a waveform input device driver.

Low-Level MIDI Audio Services

The low-level MIDI audio services allow applications to communicate directly with device drivers to manage MIDI audio playback and recording. Click one of the following categories to display a brief description of the low-level MIDI audio functions; you can then display the full information about each individual function.

Querying MIDI Devices

Before playing MIDI audio, you must determine the capabilities of the MIDI hardware that is present in the system. Use the following functions to get the number of MIDI devices and the capabilities of these devices:

midiInGetNumDevs	Retrieves the number of MIDI input devices present in the system.
midiInGetDevCaps	Retrieves the capabilities of a given MIDI input device.
midiOutGetNumDevs	Retrieves the number of MIDI output devices present in the system.
midiOutGetDevCaps	Retrieves the capabilities of a given MIDI output device.

Opening and Closing MIDI Devices

After getting the MIDI capabilities, you must open a MIDI device to play or record MIDI messages. After using the device, you should close it to make it available to other applications. Use the following functions to open and close MIDI devices:

midiInOpen	Opens a MIDI input device for recording.
midiInClose	Closes a specified MIDI input device.
midiOutOpen	Opens a MIDI output device for playback.
midiOutClose	Closes a specified MIDI output device.

Getting MIDI Device IDs

Using a MIDI device handle, you can retrieve the device ID for an open MIDI device. Use the following functions to get the device ID:

midiInGetID	Gets the device ID for a MIDI input device.
midiOutGetID	Gets the device ID for a MIDI output device.

Sending MIDI Messages

Once you have opened a MIDI output device, you can send it MIDI messages. MIDI system exclusive messages are sent in data blocks that must be prepared before being sent to an output device. Use the following functions to send MIDI messages to output devices and to prepare system exclusive data blocks:

midiOutLongMsg	Sends a buffer containing MIDI data to a specified MIDI output device.
midiOutShortMsg	Sends any MIDI message other than a system exclusive message to the specified MIDI output device.
midiOutPrepareHeader	Informs the MIDI output device driver that the given MIDI data buffer should be prepared for playback.
midiOutReset	Turns off all notes on all channels for a specified MIDI output device.
midiOutUnprepareHeader	Informs the MIDI output device driver that the preparation performed on the given MIDI data buffer can be cleaned up.

Receiving MIDI Messages

Once you have opened a MIDI input device, you can begin receiving MIDI input. MIDI messages other than system exclusive messages are sent directly to a callback. To receive system exclusive messages, you must pass data buffers to the input device. These data buffers must be prepared before being sent to the device. Use the following messages to prepare system exclusive data buffers and pass these buffers to a MIDI input device:

midiInAddBuffer	Sends an input buffer for system exclusive messages to a specified MIDI input device. The buffer is sent back to the application when it is filled with system exclusive data.
midiInPrepareHeader	Informs a MIDI input device that the given data buffer should be prepared for recording.
midiInUnprepareHeader	Informs a MIDI input device that the preparation performed on the given data buffer can be cleaned up.

Controlling MIDI Input

When receiving MIDI input, you can control when the input starts and stops. Use the following functions to start and stop input on a MIDI input device:

midiInStart	Starts input on a MIDI input device.
midiInStop	Stops input on a MIDI input device.
midiInReset	Stops input on a MIDI input device. Marks all pending data buffers as being done.

Changing MIDI Volume and Caching Patches

Some internal MIDI synthesizers support volume level changes and patch caching. Use the following functions to query and set the volume level and to cache and uncache patches with internal MIDI synthesizer devices:

midiOutCacheDrumPatches	Requests that an internal MIDI synthesizer device preload a specified set of key-based percussion patches.
midiOutCachePatches	Requests that an internal MIDI synthesizer device preload a specified set of patches.
midiOutGetVolume	Queries the current volume level of an internal MIDI synthesizer device.
midiOutSetVolume	Sets the volume level of an internal MIDI synthesizer device.

Handling MIDI Errors

Most of the low-level MIDI audio functions return error codes. Use the following functions to convert the error codes returned from MIDI functions into a textual description of the error:

midiInGetErrorText	Retrieves a textual description of a specified MIDI input error.
midiOutGetErrorText	Retrieves a textual description of a specified MIDI output error.

Sending Custom Messages to MIDI Drivers

The following functions let you send messages directly to MIDI drivers:

midiInMessage	Sends a message directly to a MIDI input device driver.
midiOutMessage	Sends a message directly to a MIDI input device driver.

Auxiliary Audio Services

Auxiliary audio devices are audio devices whose output is mixed with the output of waveform and MIDI synthesizer devices. Use the following functions to query the capabilities of auxiliary audio devices and to query and set their volume level:

auxGetDevCaps	Retrieves the capabilities of a given auxiliary audio device.
auxGetNumDevs	Retrieves the number of auxiliary audio devices present in a system.
auxGetVolume	Queries the volume level of an auxiliary audio device.
auxOutMessage	Sends a message to an auxiliary output device.
auxSetVolume	Sets the volume level of an auxiliary audio device.

File I/O Services

The multimedia file I/O services provide buffered and unbuffered file I/O, and support for standard Resource Interchange File Format (RIFF) files. The services are extensible with custom I/O procedures that can be shared among applications. Click one of the following categories to display a brief description of the multimedia file I/O functions; you can then display the full information about each individual function.

Performing Basic File I/O

Using the basic file I/O services is very similar to using other file I/O services such as the C runtime file I/O services. Files must be opened before they can be read or written. After reading or writing, the file must be closed. You can seek to a specified position in an open file. Use the following functions for basic file I/O:

mmioClose	Closes an opened file.
mmioOpen	Opens a file for reading and/or writing, and returns a handle to the opened file.
mmioRead	Reads a specified number of bytes from an opened file.
mmioRename	Renames a file.
mmioSeek	Changes the current position for reading and/or writing in an opened file.
mmioWrite	Writes a specified number of bytes to an opened file.

Performing Buffered File I/O

Using the basic buffered file I/O services is very similar to using the unbuffered services. Specify the MMIO_ALLOCBUF option with the mmioOpen function to open a file for buffered I/O. The file I/O manager will maintain an internal buffer which is transparent to the application.

You can also change the size of the internal buffer, allocate your own buffer, and directly access a buffer for optimal I/O performance. Use the following functions for I/O buffer control and direct I/O buffer access:

---

mmioAdvance
  Fills and/or flushes the I/O buffer of a file set up for direct I/O buffer access.

mmioFlush
  Writes the contents of the I/O buffer to disk.

mmioGetInfo
  Gets information about the file I/O buffer of a file opened for buffered I/O.

mmioSetBuffer
  Changes the size of the I/O buffer, and allows applications to supply their own buffer.

mmioSetInfo
  Changes information about the file I/O buffer of a file opened for buffered I/O.

Working with RIFF Files

The preferred format for multimedia files is the Microsoft Resource Interchange File Format (RIFF). The RIFF format is based on a tagged-file structure using chunks identified by four-character codes. You can use the multimedia file I/O services to open, read, and write RIFF files the same way as you would any other type of file. You can also use the following functions to create chunks, convert characters and strings to four-character codes, and navigate between chunks in RIFF files:

---

mmioAscend
  Ascends out of a RIFF file chunk to the next chunk in the file.

mmioCreateChunk
  Creates a chunk in a RIFF file.

mmioDescend
  Descends into a RIFF file chunk starting at the current file position, or searches for a specified chunk.

mmioFOURCC
  Converts four individual characters into a FOURCC code.

mmioStringToFOURCC
  Converts a NULL-terminated string into a FOURCC code.

Using Custom I/O Procedures

The multimedia file I/O services use I/O procedures to handle the physical input and output associated with reading and writing different types of storage systems. I/O procedures know how to open, close, read, write, and seek a particular type of storage system. Applications can supply custom I/O procedures for accessing unique storage systems such as databases or file archives. Use the following functions for working with custom I/O procedures:

mmioInstallIOProc	Installs, removes, or locates an I/O procedure.
mmioSendMessage	Sends a message to an I/O procedure associated with a specified file.

Media Control Interface Services

The Media Control Interface (MCI) provides a high-level generalized interface for controlling both internal and external media devices. MCI uses device handlers to interpret and execute high-level MCI commands. Applications can communicate with MCI device handlers by sending messages or command strings. MCI also provides macros for working with the time and position information encoded in a packed DWORD.

The MCI command messages contain most of the MCI functionality. See Media Control Interface Messages for details.

Click one of the following categories to display a brief description of the MCI functions and macros; you can then display the full information about each individual function.

Communicating with MCI Devices
MCI Macros for Encoding and Decoding Time Data

Communicating with MCI Devices

You can communicate with MCI devices using messages or command strings. Messages are used directly by MCI; MCI converts command strings into messages that it then sends to the device handler. Use these functions to send messages or command strings to MCI, to get the ID assigned to a device, and to get a textual description of an MCI error:

---

mciSendCommand
  Sends a command message to MCI.

mciSendString
  Sends a command string to MCI.

mciGetDeviceID
  Returns the device ID assigned when the device was opened.

mciGetCreatorTask
  Returns a handle to the process that opened a device.

mciGetErrorString
  Returns the error string corresponding to an MCI error return value.

mciSetYieldProc
  Specifies a callback procedure to be called while an MCI device is completing a command specified with the wait flag.

mciGetYieldProc
  Returns the curent yield procedure for an MCI device.

Most of the MCI functionality is expressed in its command messages. See Media Control Interface Messages for a reference to all MCI command messages. MCI command messages are prefixed with **MCI**.

In addition to its message-based interface, MCI has a string-based interface. A separate help file, MCISTR.HLP, describes the MCI command strings.

MCI Macros for Encoding and Decoding Time Data

MMSYSTEM.H defines a set of macros that extract information from the packed DWORD that MCI uses to encode time information. Use these macros to extract time and position information from the DWORD:

MCI_HMS_HOUR
Returns the hours field of an argument packed with hours, minutes, and seconds.

MCI_HMS_MINUTE
Returns the minutes field of an argument packed with hours, minutes, and seconds.

MCI_HMS_SECOND
Returns the seconds field of an argument packed with hours, minutes, and seconds.

MCI_MSF_FRAME
Returns the frames field of an argument packed with minutes, seconds, and frames.

MCI_MSF_MINUTE
Returns the minutes field of an argument packed with minutes, seconds, and frames.

MCI_MSF_SECOND
Returns the seconds field of an argument packed with minutes, seconds, and frames.

MCI_TMSF_FRAME
Returns the frames field of an argument packed with tracks, minutes, seconds, and frames.

MCI_TMSF_MINUTE
Returns the minutes field of an argument packed with tracks, minutes, seconds, and frames.

MCI_TMSF_SECOND
Returns the seconds field of an argument packed with tracks, minutes, seconds, and frames.

MCI_TMSF_TRACK
Returns the tracks field of an argument packed with tracks, minutes, seconds, and frames.

MMSYSTEM.H also defines the following macros that combine separate time and position values into the packed DWORD format:

MCI_MAKE_HMS
Creates a DWORD time value in hours/minutes/seconds format from the given hours, minutes, and seconds values.

MCI_MAKE_MSF
Creates a DWORD time value in minutes/seconds/frames format from the given minutes, seconds, and frames values.

MCI_MAKE_TMSF
Creates a DWORD time value in tracks/minutes/seconds/frames format from the given tracks, minutes, seconds, and frames values.

Joystick Services

The joystick services provide support for up to two joystick devices. Use the following functions to get information about joystick devices, to control joystick sensitivity, and to receive messages related to joystick movement and button activity:

joyGetDevCaps
Returns the capabilities of a joystick device.

joyGetNumDevs
Returns the number of devices supported by the joystick driver.

joyGetPos
Returns the position and button state of a joystick.

joyGetThreshold
Returns the movement threshold of a joystick.

joyReleaseCapture
Releases the joystick captured with joySetCapture.

joySetCapture
Causes periodic joystick messages to be sent to a window.

joySetThreshold
Sets the movement threshold of a joystick.

Timer Services

The timer services allow applications to schedule asynchronous timed periodic or one-time events at a higher resolution than is available through the standard Windows timer services. Use the following functions to request and receive timer messages:

timeBeginPeriod
Establishes the timer resolution an application intends to use.

timeEndPeriod
Clears a previously set timer resolution.

timeGetDevCaps
Returns the capabilities of the timer driver.

timeGetSystemTime
Fills an MMTIME structure with the system time in milliseconds.

timeGetTime
Returns the system time in milliseconds.

timeKillEvent
Cancels a timer event previously created with timeSetEvent.

timeSetEvent
Creates a timer event which will call a specified function at periodic intervals or after a single period.

Debugging Services

The debugging services provide support for debugging applications. Use the following functions to get the current version of MMSYSTEM.DLL and to send debugging messages from an application:

mmsystemGetVersion	Gets the version number of MMSYSTEM.DLL.
OutputDebugStr	Sends a debug string to either the COM1 port or to a monochrome display adapter.

Message Overview

Click one of the following categories to display a brief description of the multimedia messages; you can then display the full information about each individual message.

Audio Messages

Audio messages are sent by low-level audio device drivers to an application so that the application can manage audio playback and recording. An application may choose to have audio messages sent either to a window, or to a low-level callback function. There is a set of messages for windows and a parallel set of messages for low-level callback functions.

Click one of the following categories to display a brief description of the audio messages; you can then display the full information about each individual message.

Waveform Output Messages

Waveform output messages are sent by audio device drivers to an application to inform the application about the status of waveform output operations. By specifying flags with the waveOutOpen function, applications may choose to have messages sent either to a window or to a low-level callback function. Use these messages to manage waveform playback:

MM_WOM_CLOSE	Sent to a window when a waveform output device is closed.
MM_WOM_DONE	Sent to a window when a data block has been played and is being returned to the application.
MM_WOM_OPEN	Sent to a window when a waveform output device is opened.
WOM_CLOSE	Sent to a low-level callback function when a waveform output device is closed.
WOM_DONE	Sent to a low-level callback function when a data block has been played and is being returned to the application.
WOM_OPEN	Sent to a low-level callback function when a waveform output device is opened.

Waveform Input Messages

Waveform input messages are sent by audio device drivers to an application to inform the application about the status of waveform input operations. By specifying flags with the waveInOpen function, applications may choose to have messages sent either to a window or to a low-level callback function. Use these messages to manage waveform audio recording:

MM_WIM_CLOSE	Sent to a window when a waveform input device is closed.
MM_WIM_DATA	Sent to a window when an input data buffer is full and is being returned to the application.
MM_WIM_OPEN	Sent to a window when a waveform input device is opened.
WIM_CLOSE	Sent to a low-level callback function when a waveform input device is closed.
WIM_DATA	Sent to a low-level callback function when an input data buffer is full and is being returned to the application.
WIM_OPEN	Sent to a low-level callback function when a waveform input device is opened.

MIDI Output Messages

MIDI output messages are sent by audio device drivers to an application to inform the application about the status of MIDI output operations. By specifying flags with the midiOutOpen function, applications may choose to have messages sent either to a window or to a low-level callback function. Use these messages to manage MIDI output:

MM_MOM_CLOSE
: Sent to a window when a MIDI output device is closed.

MM_MOM_DONE
: Sent to a window when a MIDI system exclusive data block has been played and is being returned to the application.

MM_MOM_OPEN
: Sent to a window when a MIDI output device is opened.

MOM_CLOSE
: Sent to a low-level callback function when a MIDI output device is closed.

MOM_DONE
: Sent to a low-level callback function when a MIDI system exclusive data block has been played and is being returned to the application.

MOM_OPEN
: Sent to a low-level callback function when a MIDI output device is opened.

MIDI Input Messages

MIDI input messages are sent by audio device drivers to an application to inform the application about the status of MIDI input operations. By specifying flags with the midiInOpen function, applications may choose to have messages sent either to a window or to a low-level callback function. Use these messages to manage MIDI input:

MM_MIM_CLOSE
Sent to a window when a MIDI input device is closed.

MM_MIM_DATA
Sent to a window when a MIDI message is received by the device.

MM_MIM_ERROR
Sent to a window when an invalid MIDI message is received by the device.

MM_MIM_LONGERROR
Sent to a window when an invalid MIDI system exclusive message is received by the device.

MM_MIM_LONGDATA
Sent to a window when a MIDI system exclusive data buffer is filled and is being returned to the application.

MM_MIM_OPEN
Sent to a low-level callback function when a MIDI input device is opened.

MIM_OPEN
Sent to a window when a MIDI input device is opened.

MIM_CLOSE
Sent to a low-level callback function when a MIDI input device is closed.

MIM_DATA
Sent to a low-level callback function when a MIDI message is received by the device. The parameters to this message include a timestamp specifying the time that the MIDI message was received.

MIM_ERROR
Sent to a low-level callback function when an invalid MIDI message is received by the device.

MIM_LONGERROR
Sent to a low-level callback function when an invalid MIDI system exclusive message is received by the device.

MIM_LONGDATA
Sent to a low-level callback function when a MIDI system exclusive message is received by the device. The parameters to this message include a timestamp specifying the time that the MIDI message was received.

Media Control Interface Messages

Media Control Interface (MCI) messages control MCI devices and obtain information about device configuration and capabilities. Applications use the mciSendCommand function to send MCI command messages to MCI devices.

Click one of the following categories to display a brief description of the MCI messages; you can then display the full information about each individual message.

System Command Messages

System command messages are interpreted directly by MCI. These messages do not rely on the ability of a device to respond to them.

MCI_BREAK	Sent by an application to set a break key for a specified device.
MCI_SYSINFO	Sent by an application to obtain system-related information about a device.

Required Command Messages

Required command messages are supported by all MCI devices. These messages open, close, and obtain information about devices.

MCI_CLOSE
Sent by an application to request that the specified device be closed.

MCI_GETDEVCAPS
Sent by an application to obtain information about device capabilities.

MCI_INFO
Sent by an application to obtain information about a device.

MCI_OPEN
Sent by an application to open a device and get an MCI device identifier for use with other commands.

MCI_STATUS
Sent by an application to obtain status information about a device.

Basic Command Messages

Basic command messages are recognized by all MCI devices. The use of these commands by a device is optional. If a device does not support a basic command, it returns MCIERR_UNSUPPORTED_FUNCTION.

MCI_LOAD
Sent by an application to load a file.

MCI_PAUSE
Sent by an application to pause a device.

MCI_PLAY
Sent by an application to start a device playing.

MCI_RECORD
Sent by an application to start recording with a device.

MCI_RESUME
Sent by an application to resume playback or recording after a pause.

MCI_SAVE
Sent by an application to save the current file.

MCI_SEEK
Sent by an application to change locations within a media element.

MCI_SET
Sent by an application to set parameters for a device.

MCI_STOP
Sent by an application to stop a device from playing or recording.

Extended Command Messages

Extended command messages apply to particular device types such as animation devices. Device types with extended commands have capabilities that are not present in most types of MCI devices.

Click one of the following categories to display a brief description of the extended command messages; you can then display the full information about each individual command message.

Extended Commands for Working with MCI Element Files

MCI devices that let you edit MCI data can have extended commands for manipulating data. The following commands apply to devices that support editing:

MCI_COPY	Sent by an application to copy data from the MCI element to the Clipboard.
MCI_CUT	Sent by an application to move data from the MCI element to the Clipboard.
MCI_DELETE	Sent by an application to remove data from the MCI element.
MCI_PASTE	Sent by an application to paste data from the Clipboard to the MCI element.

Extended Commands for Device Operation and Positioning

MCI devices can have operating capabilities that apply only to a device type or that apply to a device with unique features. The following commands apply to devices that have specialized operating capabilities:

MCI_CUE
: Sent by an application to cue a device for playback or recording.

MCI_ESCAPE
: Sent by an application to send a string command to a device handler.

MCI_RESUME
: Sent by an application to continue playback or recording previously paused.

MCI_SPIN
: Sent by an application to start or stop spinning a rotating media device such as a laserdisc.

MCI_STEP
: Sent by an application to step a device one or more frames.

Extended Commands for Windowed Video Devices

Video devices that display data in a window on the computer display can have MCI commands for controlling the video display and window. These devices include animation movie players and video overlay devices. The following commands apply to windowed video devices:

MCI_FREEZE
Sent by an application to stop capture.

MCI_PUT
Sent by an application to define a source or destination clipping rectangle.

MCI_REALIZE
Sent by an application to tell a graphic device to realize its palette.

MCI_UNFREEZE
Sent by an application to restore capture.

MCI_UPDATE
Sent by an application to tell a graphic device to update or paint the current frame.

MCI_WHERE
Sent by an application to determine the extent of a clipping rectangle.

MCI_WINDOW
Sent by an application to specify a window and the characteristics of the window for a graphic device to use for its display.

Window Notification Message

Window notification messages are sent by MCI to a window function when an application wants to be notified of the completion of a command. If you want MCI notification, your application must specify a window to handle the notification message; specify the window handle in the data structure sent with mciSendCommand.

MM_MCINOTIFY	Notifies the window function of the command status. The *wParam* parameter of this message contains the status of the command.

Joystick Messages

Joystick messages are sent to an application to notify the application that a joystick has moved or that one of its buttons has been pressed or released. Use these messages to get input from a joystick:

MM_JOY1BUTTONDOWN
Sent to a window that has captured joystick 1 when a button has been pressed.

MM_JOY1BUTTONUP
Sent to a window that has captured joystick 1 when a button has been released.

MM_JOY1MOVE
Sent to a window that has captured joystick 1 when the joystick position has changed.

MM_JOY1ZMOVE
Sent to a window that has captured joystick 1 when the joystick z-axis position has changed.

MM_JOY2BUTTONDOWN
Sent to a window that has captured joystick 2 when a button has been pressed.

MM_JOY2BUTTONUP
Sent to a window that has captured joystick 2 when a button has been released.

MM_JOY2MOVE
Sent to a window that has captured joystick 2 when the joystick position has changed.

MM_JOY2ZMOVE
Sent to a window that has captured joystick 2 when the joystick z-axis position has changed.

File I/O Messages

File I/O messages are sent to custom I/O procedures to request I/O operations on a file. I/O procedures must respond to all of the following messages:

MMIOM_CLOSE
Sent to an I/O procedure to request that a file be closed.

MMIOM_OPEN
Sent to an I/O procedure to request that a file be opened.

MMIOM_READ
Sent to an I/O procedure to request that data be read from a file.

MMIOM_RENAME
Sent to an I/O procedure to request that a file be renamed.

MMIOM_SEEK
Sent to an I/O procedure to request that the current position for reading and writing be changed.

MMIOM_WRITE
Sent to an I/O procedure to request that data be written to a file.

MMIOM_WRITEFLUSH
Sent to an I/O procedure to request that an I/O buffer be flushed to disk.

Data Types and Structures

This chapter describes data types and data structures used in the multimedia APIs. Click one of the following categories to display a brief description of the multimedia data types and structures; you can then display the full information about each individual data type or structure.

Each data structure has an associated long pointer data type with prefix LP.

Auxiliary Audio Data Structures

The following data structure is used with auxiliary audio devices:

AUXCAPS	A data structure that describes the capabilities of an auxiliary audio device.

Joystick Data Structures

The following data structures are used with joystick functions:

JOYCAPS	A data structure that defines joystick capabilities.
JOYINFO	A data structure for joystick information.

Media Control Interface (MCI) Data Structures

Click one of the following categories to display a brief description of the Media Control Interface (MCI) data structures; you can then display the full information about each individual data type or structure.

Data structures for system commands
Data structures for required commands
Data structures for basic commands
Data structures for extended commands

Some MCI commands have several associated data structures; for example, the MCI_PLAY command message is used with a generic MCI_PLAY_PARMS structure and three extended data structures for the multimedia movie, video overlay, and waveform audio devices. Also, the MCI_GENERIC_PARMS data structure is used with several MCI command message.

Data Structures for MCI System Commands

The following data structures are used to specify parameter blocks for system command messages (message handled directly by MCI):

MCI_BREAK_PARMS
A data structure that specifies parameters for the MCI_BREAK command.

MCI_SYSINFO_PARMS
A data structure that specifies parameters for the MCI_SYSINFO command.

Data Structures for MCI Required Commands

The following data structures are used to specify parameter blocks for required command messages (messages handled by all MCI devices):

MCI_GENERIC_PARMS
A data structure that specifies parameters for the MCI_CLOSE command.

MCI_GETDEVCAPS_PARMS
A data structure that specifies parameters for the MCI_GETDEVCAPS command.

MCI_INFO_PARMS
A data structure that specifies parameters for the MCI_INFO command.

MCI_OPEN_PARMS
MCI_ANIM_OPEN_PARMS (multimedia movie device)
MCI_OVLY_OPEN_PARMS (video overlay device)
MCI_WAVE_OPEN_PARMS (waveform audio device)
Data structures that specify parameters for the MCI_OPEN command.

MCI_STATUS_PARMS
A data structure that specifies parameters for the MCI_STATUS command.

Data Structures for MCI Basic Commands

The following data structures are used to specify parameter blocks for basic command messages (messages recognized by all MCI devices):

MCI_GENERIC_PARMS
A data structure that specifies parameters for the MCI_PAUSE, MCI_RESUME, and MCI_STOP commands.

MCI_LOAD_PARMS
MCI_OVLY_LOAD_PARMS (video overlay device)
A data structure that specifies parameters for the MCI_LOAD command.

MCI_PLAY_PARMS
MCI_ANIM_PLAY_PARMS (multimedia movie device)
MCI_VD_PLAY_PARMS (videodisc device)
Data structures that specify parameters for the MCI_PLAY command.

MCI_RECORD_PARMS
A data structure that specifies parameters for the MCI_RECORD command.

MCI_SAVE_PARMS
MCI_OVLY_SAVE_PARMS (video overlay device)
A data structure that specifies parameters for the MCI_SAVE command.

MCI_SEEK_PARMS
A data structure that specifies parameters for the MCI_SEEK command.

MCI_SET_PARMS
MCI_SEQ_SET_PARMS (sequencer device)
MCI_WAVE_SET_PARMS (waveform audio device)
Data structures that specify parameters for the MCI_SET command.

Data Structures for MCI Extended Commands

The following data structures are used to specify parameter blocks for MCI extended command messages (messages defined for specific MCI device types):

MCI_WAVE_DELETE_PARMS (waveform audio device)
A data structure that specifies parameters for the MCI_DELETE command.

MCI_VD_ESCAPE_PARMS (video overlay device)
A data structure that specifies parameters for the MCI_ESCAPE command.

MCI_ANIM_RECT_PARMS (multimedia movie device)
MCI_OVLY_RECT_PARMS (video overlay device)
Data structures that specify parameters for the MCI_PUT and MCI_WHERE commands.

MCI_ANIM_STEP_PARMS (multimedia movie device)
MCI_VD_STEP_PARMS (videodisc device)
Data structures that specify parameters for the MCI_STEP command used with the multimedia movie and video overlay devices.

MCI_ANIM_UPDATE_PARMS (multimedia movie device)
A data structure that specifies parameters for the MCI_UPDATE command used with the multimedia movie and video overlay devices.

MCI_ANIM_WINDOW_PARMS (multimedia movie device)
MCI_OVLY_WINDOW_PARMS (video overlay device)
Data structures that specify parameters for the MCI_WINDOW command used with the multimedia movie and video overlay devices.

MIDI Audio Data Structures

The following data structures are used with MIDI functions:

MIDIHDR	A data structure representing a header for MIDI input and output data blocks.
MIDIINCAPS	A data structure that describes the capabilities of a MIDI input device.
MIDIOUTCAPS	A data structure that describes the capabilities of a MIDI output device.

Multimedia File I/O Data Structures

The following data structures are used with the multimedia file I/O functions:

MMIOINFO	A data structure for information about an open file.
MMCKINFO	A data structure for information about a RIFF chunk in an open file.

Timer Data Structures

The following data structures are used with timer functions:

MMTIME	A data structure that represents time in one of several different formats.
TIMECAPS	A data structure that defines timer capabilities.

Waveform Audio Data Structures

The following data structures are used with waveform functions:

MMTIME	A data structure used to represent time to waveform functions.
PCMWAVEFORMAT	A data structure representing the format of PCM waveform data.
WAVEFORMAT	A data structure representing generic format information common to all types of waveform data.
WAVEHDR	A data structure representing a header for waveform input and output data blocks.
WAVEINCAPS	A data structure that describes the capabilities of a waveform input device.
WAVEOUTCAPS	A data structure that describes the capabilities of a waveform output device.

MIM_CLOSE

This message is sent to a MIDI input callback function when a MIDI input device is closed. The device handle is no longer valid once this message has been sent.

Parameters

DWORD *dwParam1*

Not used.

DWORD *dwParam2*

Not used.

Return Value

None.

See Also

MM_MIM_CLOSE

MIM_DATA

This message is sent to a MIDI input callback function when a MIDI message is received by a MIDI input device.

Parameters

DWORD *dwParam1*

Specifies the MIDI message that was received. The message is packed into a DWORD with the first byte of the message in the low-order byte.

DWORD *dwParam2*

Specifies the time that the message was received by the input device driver. The timestamp is specified in milliseconds, beginning at 0 when midiInStart was called.

Return Value

None.

Comments

MIDI messages received from a MIDI input port have running status disabled; each message is expanded to include the MIDI status byte.

This message is not sent when a MIDI system-exclusive message is received.

See Also

MM_MIM_DATA, MIM_LONGDATA

MIM_ERROR

This message is sent to a MIDI input callback function when an invalid MIDI message is received.

Parameters

DWORD *dwParam1*

Specifies the invalid MIDI message that was received. The message is packed into a DWORD with the first byte of the message in the low-order byte.

DWORD *dwParam2*

Specifies the time that the message was received by the input device driver. The timestamp is specified in milliseconds, beginning at 0 when midiInStart was called.

Return Value

None.

See Also

MM_MIM_ERROR

MIM_LONGDATA

This message is sent to a MIDI input callback function when an input buffer has been filled with MIDI system-exclusive data and is being returned to the application.

Parameters

DWORD *dwParam1*

Specifies a far pointer to a MIDIHDR structure identifying the input buffer.

DWORD *dwParam2*

Specifies the time that the data was received by the input device driver. The timestamp is specified in milliseconds, beginning at 0 when midiInStart was called.

Return Value

None.

Comments

The returned buffer might not be full. To determine the number of bytes recorded into the returned buffer, use the dwBytesRecorded field of the MIDIHDR structure specified by *dwParam1*.

See Also

MIM_DATA, MM_MIM_LONGDATA

MIM_LONGERROR

This message is sent to a MIDI input callback function when an invalid MIDI system-exclusive message is received.

Parameters

DWORD *dwParam1*

Specifies a pointer to a MIDIHDR structure identifying the buffer containing the invalid message.

DWORD *dwParam2*

Specifies the time that the data was received by the input device driver. The timestamp is specified in milliseconds, beginning at 0 when midiInStart was called.

Return Value

None.

Comments

The returned buffer might not be full. To determine the number of bytes recorded into the returned buffer, use the dwBytesRecorded field of the MIDIHDR structure specified by *dwParam1*.

See Also

MM_MIM_LONGERROR

MIM_OPEN

This message is sent to a MIDI input callback function when a MIDI input device is opened.

Parameters

DWORD *dwParam1*

Not used.

DWORD *dwParam2*

Not used.

Return Value

None.

See Also

MM_MIM_OPEN

MM_JOY1BUTTONDOWN

This message is sent to the window that has captured joystick 1 when a button is pressed.

Parameters

WPARAM *wParam*

Indicates which button has changed state. It can be any one of the following combined with any of the flags defined in MM_JOY1MOVE.

JOY_BUTTON1CHG

Set if first joystick button has changed.

JOY_BUTTON2CHG

Set if second joystick button has changed.

JOY_BUTTON3CHG

Set if third joystick button has changed.

JOY_BUTTON4CHG

Set if fourth joystick button has changed.

LPARAM *lParam*

The low-order word contains the current X position of the joystick. The high-order word contains the current Y position.

Return Value

None.

See Also

MM_JOY1BUTTONUP

MM_JOY1BUTTONUP

This message is sent to the window that has captured joystick 1 when a button is released.

Parameters

WPARAM *wParam*

Indicates which button has changed state. It can be any one of the following combined with any of the flags defined in MM_JOY1MOVE.

JOY_BUTTON1CHG

Set if first joystick button has changed.

JOY_BUTTON2CHG

Set if second joystick button has changed.

JOY_BUTTON3CHG

Set if third joystick button has changed.

JOY_BUTTON4CHG

Set if fourth joystick button has changed.

LPARAM *lParam*

The low-order word contains the current X position of the joystick. The high-order word contains the current Y position.

Return Value

None.

See Also

MM_JOY1BUTTONDOWN

MM_JOY1MOVE

This message is sent to the window that has captured joystick 1 when the joystick position changes.

Parameters

WPARAM *wParam*

Indicates which joystick buttons are pressed. It can be any combination of the following values:

JOY_BUTTON1
Set if first joystick button is pressed.

JOY_BUTTON2
Set if second joystick button is pressed.

JOY_BUTTON3
Set if third joystick button is pressed.

JOY_BUTTON4
Set if fourth joystick button is pressed.

LPARAM *lParam*

The low-order word contains the current X position of the joystick. The high-order word contains the current Y position.

Return Value

None.

See Also

MM_JOY1ZMOVE

MM_JOY1ZMOVE

This message is sent to the window that has captured joystick 1 when the z-axis position changes.

Parameters

WPARAM *wParam*

Indicates which joystick buttons are pressed. It can be any combination of the following values:

JOY_BUTTON1
Set if first joystick button is pressed.

JOY_BUTTON2
Set if second joystick button is pressed.

JOY_BUTTON3
Set if third joystick button is pressed.

JOY_BUTTON4
Set if fourth joystick button is pressed.

LPARAM *lParam*

The low-order word contains the current Z position of the joystick.

Return Value

None.

See Also

MM_JOY1MOVE

MM_JOY2BUTTONDOWN

This message is sent to the window that has captured joystick 2 when a button is pressed.

Parameters

WPARAM *wParam*

Indicates which button has changed state. It can be any one of the following combined with any of the flags defined in MM_JOY1MOVE.

JOY_BUTTON1CHG

Set if first joystick button has changed.

JOY_BUTTON2CHG

Set if second joystick button has changed.

JOY_BUTTON3CHG

Set if third joystick button has changed.

JOY_BUTTON4CHG

Set if fourth joystick button has changed.

LPARAM *lParam*

The low-order word contains the current X position of the joystick. The high-order word contains the current Y position.

Return Value

None.

See Also

MM_JOY2BUTTONUP

MM_JOY2BUTTONUP

This message is sent to the window that has captured joystick 2 when a button is released.

Parameters

WPARAM *wParam*

Indicates which button has changed state. It can be any one of the following combined with any of the flags defined in MM_JOY1MOVE.

JOY_BUTTON1CHG

Set if first joystick button has changed.

JOY_BUTTON2CHG

Set if second joystick button has changed.

JOY_BUTTON3CHG

Set if third joystick button has changed.

JOY_BUTTON4CHG

Set if fourth joystick button has changed.

LPARAM *lParam*

The low-order word contains the current X position of the joystick. The high-order word contains the current Y position.

Return Value

None.

See Also

MM_JOY2BUTTONDOWN

MM_JOY2MOVE

This message is sent to the window that has captured joystick 2 when the joystick position changes.

Parameters

WPARAM *wParam*

Indicates which joystick buttons are pressed. It can be any combination of the following values:

JOY_BUTTON1
Set if first joystick button is pressed.

JOY_BUTTON2
Set if second joystick button is pressed.

JOY_BUTTON3
Set if third joystick button is pressed.

JOY_BUTTON4
Set if fourth joystick button is pressed.

LPARAM *lParam*

The low-order word contains the current X position of the joystick. The high-order word contains the current Y position.

Return Value

None.

See Also

MM_JOY2ZMOVE

MM_JOY2ZMOVE

This message is sent to the window that has captured joystick 2 when the z-axis position changes.

Parameters

WPARAM *wParam*

Indicates which joystick buttons are pressed. It can be any combination of the following values:

JOY_BUTTON1
Set if first joystick button is pressed.

JOY_BUTTON2
Set if second joystick button is pressed.

JOY_BUTTON3
Set if third joystick button is pressed.

JOY_BUTTON4
Set if fourth joystick button is pressed.

LPARAM *lParam*

The low-order word contains the current Z position of the joystick.

Return Value

None.

See Also

MM_JOY2MOVE

MM_MIM_CLOSE

This message is sent to a window when a MIDI input device is closed. The device handle is no longer valid once this message has been sent.

Parameters

WPARAM *wParam*

Specifies a handle to the MIDI input device that was closed.

LPARAM *lParam*

Not used.

Return Value

None.

See Also

MIM_CLOSE

MM_MIM_DATA

This message is sent to a window when a MIDI message is received by a MIDI input device.

Parameters

WPARAM *wParam*

Specifies a handle to the MIDI input device that received the MIDI message.

LPARAM *lParam*

Specifies the MIDI message that was received. The message is packed into a LPARAM with the first byte of the message in the low-order byte.

Return Value

None.

Comments

MIDI messages received from a MIDI input port have running status disabled; each message is expanded to include the MIDI status byte.

This message is not sent when a MIDI system-exclusive message is received. No timestamp is available with this message. For timestamped input data, you must use the messages that are sent to low-level callback functions.

See Also

MIM_DATA, MM_MIM_LONGDATA

MM_MIM_ERROR

This message is sent to a window when an invalid MIDI message is received.

Parameters

WPARAM *wParam*

Specifies a handle to the MIDI input device that received the invalid message.

LPARAM *lParam*

Specifies the invalid MIDI message. The message is packed into a LPARAM with the first byte of the message in the low-order byte.

Return Value

None.

See Also

MIM_ERROR

MM_MIM_LONGDATA

This message is sent to a window when an input buffer has been filled with MIDI system-exclusive data and is being returned to the application.

Parameters

WPARAM *wParam*

Specifies a handle to the MIDI input device that received the data.

LPARAM *lParam*

Specifies a far pointer to a MIDIHDR structure identifying the buffer.

Return Value

None.

Comments

The returned buffer might not be full. To determine the number of bytes recorded into the returned buffer, use the dwBytesRecorded field of the MIDIHDR structure specified by *lParam*.

No timestamp is available with this message. For timestamped input data, you must use the messages that are sent to low-level callback functions.

See Also

MM_MIM_DATA, MIM_LONGDATA

MM_MIM_LONGERROR

This message is sent to a window when an invalid MIDI system-exclusive message is received.

Parameters

WPARAM *wParam*

Specifies a handle to the MIDI input device that received the invalid message.

LPARAM *lParam*

Specifies a far pointer to a MIDIHDR structure identifying buffer containing the invalid message.

Return Value

None.

Comments

The returned buffer might not be full. Use the dwBytesRecordedfield of the MIDIHDR structure specified by *lParam* to determine the number of bytes recorded into the returned buffer.

See Also

MIM_LONGERROR

MM_MIM_OPEN

This message is sent to a window when a MIDI input device is opened.

Parameters

WPARAM *wParam*

Specifies the handle to the MIDI input device that was opened.

LPARAM *lParam*

Not used.

Return Value

None.

See Also

MIM_OPEN

MM_MOM_CLOSE

This message is sent to a window when a MIDI output device is closed. The device handle is no longer valid once this message has been sent.

Parameters

WPARAM *wParam*

Specifies the handle to the MIDI output device.

LPARAM *lParam*

Not used.

Return Value

None.

See Also

MOM_CLOSE

MM_MOM_DONE

This message is sent to a window when the specified system-exclusive buffer has been played and is being returned to the application.

Parameters

WPARAM *wParam*

Specifies a handle to the MIDI output device that played the buffer.

LPARAM *lParam*

Specifies a far pointer to a MIDIHDR structure identifying the buffer.

Return Value

None.

See Also

MOM_DONE

MM_MOM_OPEN

This message is sent to a window when a MIDI output device is opened.

Parameters

WPARAM *wParam*

Specifies the handle to the MIDI output device.

LPARAM *lParam*

Not used.

Return Value

None.

See Also

MOM_OPEN

MM_WIM_CLOSE

This message is sent to a window when a waveform input device is closed. The device handle is no longer valid once this message has been sent.

Parameters

WPARAM *wParam*

Specifies a handle to the waveform input device that was closed.

LPARAM *lParam*

Not used.

Return Value

None.

See Also

WIM_CLOSE

MM_WIM_DATA

This message is sent to a window when waveform data is present in the input buffer and the buffer is being returned to the application. The message can be sent either when the buffer is full, or after the waveInReset function is called.

Parameters

WPARAM *wParam*

Specifies a handle to the waveform input device that received the waveform data.

LPARAM *lParam*

Specifies a far pointer to a WAVEHDR structure identifying the buffer containing the waveform data.

Return Value

None.

Comments

The returned buffer might not be full. Use the dwBytesRecordedfield of the WAVEHDR structure specified by *lParam* to determine the number of bytes recorded into the returned buffer.

See Also

WIM_DATA

MM_WIM_OPEN

This message is sent to a window when a waveform input device is opened.

Parameters

WPARAM *wParam*

Specifies a handle to the waveform input device that was opened.

LPARAM *lParam*

Not used.

Return Value

None.

See Also

WIM_OPEN

MM_WOM_CLOSE

This message is sent to a window when a waveform output device is closed. The device handle is no longer valid once this message has been sent.

Parameters

WPARAM *wParam*

Specifies a handle to the waveform output device that was closed.

LPARAM *lParam*

Not used.

Return Value

None.

See Also

WOM_CLOSE

MM_WOM_DONE

This message is sent to a window when the specified output buffer is being returned to the application. Buffers are returned to the application when they have been played, or as the result of a call to waveOutReset.

Parameters

WPARAM *wParam*

Specifies a handle to the waveform output device that played the buffer.

LPARAM *lParam*

Specifies a far pointer to a WAVEHDR structure identifying the buffer.

Return Value

None.

See Also

WOM_DONE

MM_WOM_OPEN

This message is sent to a window when a waveform output device is opened.

Parameters

WPARAM *wParam*

Specifies a handle to the waveform output device that was opened.

LPARAM *lParam*

Not used.

Return Value

None.

See Also

WOM_OPEN

MMIOM_CLOSE

This message is sent to an I/O procedure by mmioClose to request that a file be closed.

Parameters

LPARAM *lParam1*

Specifies options contained in the *wFlags* parameter of mmioClose.

LPARAM *lParam2*

Not used.

Return Value

The return value is zero if the file is successfully closed. Otherwise, the return value specifies an error code.

See Also

mmioClose, MMIOM_OPEN

MMIOM_OPEN

This message is sent to an I/O procedure by mmioOpen to request that a file be opened or deleted.

Parameters

LPARAM *lParam1*

Specifies a null-terminated string containing the name of the file to open.

LPARAM *lParam2*

Not used.

Return Value

The return value is zero if the operation is successful. Otherwise, the return value specifies an error value. Possible error returns are:

MMIOM_CANNOTOPEN

Specified file could not be opened.

MMIOM_OUTOFMEMORY

Not enough memory to perform operation.

Comments

The dwFlags field of the MMIOINFO structure contains option flags passed to the mmioOpen function. The lDiskOffset field of the MMIOINFO structure is initialized to zero. If this value is incorrect, then the I/O procedure must correct it.

If the caller passed a MMIOINFO structure to mmioOpen, the return value will be returned in the wErrorRet field.

See Also

mmioOpen, MMIOM_CLOSE

MMIOM_READ

This message is sent to an I/O procedure by mmioRead to request that a specified number of bytes be read from an open file.

Parameters

LPARAM *lParam1*

Specifies a huge pointer to the buffer to be filled with data read from the file.

LPARAM *lParam2*

Specifies the number of bytes to read from the file.

Return Value

The return value is the number of bytes actually read from the file. If no more bytes can be read, the return value is zero. If there is an error, the return value is -1.

Comments

The I/O procedure is responsible for updating the lDiskOffset field of the MMIOINFO structure to reflect the new file position after the read operation.

See Also

mmioRead, MMIOM_WRITE, MMIOM_WRITEFLUSH

MMIOM_RENAME

This message is sent to an I/O procedure by mmioRename to request that the specified file be renamed.

Parameters

LPARAM *lParam1*

Specifies a far pointer to a string containing the filename of the file to rename.

LPARAM *lParam2*

Specifies a far pointer to a string containing the new filename.

Return Value

If the file is renamed successfully, the return value is zero. If the specified file was not found, the return value is MMIOERR_FILENOTFOUND.

See Also

mmioRename

MMIOM_SEEK

This message is sent to an I/O procedure by mmioSeek to request that the current file position be moved.

Parameters

LPARAM *lParam1*

Specifies the new file position according to the option flag specified in *lParam2*.

LPARAM *lParam2*

Specifies how the file position is changed. Only one of the following flags can be specified:

SEEK_SET

Move the file position to be *lParam1* bytes from the beginning of the file.

SEEK_CUR

Move the file position to be *lParam1* bytes from the current position. *lParam1* may be positive or negative.

SEEK_END

Move the file position to be *lParam1* bytes from the end of the file.

Return Value

The return value is the new file position. If there is an error, the return value is -1.

Comments

The I/O procedure is responsible for maintaining the current file position in the lDiskOffset field of the MMIOINFO structure.

See Also

mmioSeek

MMIOM_WRITE

This message is sent to an I/O procedure by mmioWrite to request that data be written to an open file.

Parameters

LPARAM *lParam1*

Specifies a huge pointer to a buffer containing the data to write to the file.

LPARAM *lParam2*

Specifies the number of bytes to write to the file.

Return Value

The return value is the number of bytes actually written to the file. If there is an error, the return value is -1.

Comments

The I/O procedure is responsible for updating the lDiskOffset field of the MMIOINFO structure to reflect the new file position after the write operation.

See Also

mmioWrite, MMIOM_READ, MMIOM_WRITEFLUSH

MMIOM_WRITEFLUSH

This message is sent to an I/O procedure by mmioWrite to request that data be written to an open file and then that any internal buffers used by the I/O procedure be flushed to disk.

Parameters

LPARAM *lParam1*

Specifies a huge pointer to a buffer containing the data to write to the file.

LPARAM *lParam2*

Specifies the number of bytes to write to the file.

Return Value

The return value is the number of bytes actually written to the file. If there is an error, the return value is -1.

Comments

The I/O procedure is responsible for updating the lDiskOffset field of the MMIOINFO structure to reflect the new file position after the write operation.

Note that this message is equivalent to the MMIOM_WRITE message except that it additionally requests that the I/O procedure flush its internal buffers, if any. Unless an I/O procedure performs internal buffering, this message can be handled exactly like the MMIOM_WRITE message.

See Also

mmioWrite, mmioFlush, MMIOM_READ, MMIOM_WRITE

MOM_CLOSE

This message is sent to a MIDI output callback function when a MIDI output device is closed. The device handle is no longer valid once this message has been sent.

Parameters

DWORD *dwParam1*

Not used.

DWORD *dwParam2*

Not used.

Return Value

None.

See Also

MM_MOM_CLOSE

MOM_DONE

This message is sent to a MIDI output callback function when the specified system-exclusive buffer has been played and is being returned to the application.

Parameters

DWORD *dwParam1*

Specifies a far pointer to a MIDIHDR structure identifying the buffer.

DWORD *dwParam2*

Not used.

Return Value

None.

See Also

MM_MOM_DONE

MOM_OPEN

This message is sent to a MIDI output callback function when a MIDI output device is opened.

Parameters

DWORD *dwParam1*

Not used.

DWORD *dwParam2*

Not used.

Return Value

None.

See Also

MM_MOM_OPEN

WIM_CLOSE

This message is sent to a waveform input callback function when a waveform input device is closed. The device handle is no longer valid once this message has been sent.

Parameters

DWORD *dwParam1*

Not used.

DWORD *dwParam2*

Not used.

Return Value

None.

See Also

MM_WIM_CLOSE

WIM_DATA

This message is sent to a waveform input callback function when waveform data is present in the input buffer and the buffer is being returned to the application. The message can be sent either when the buffer is full, or after the waveInReset function is called.

Parameters

DWORD *dwParam1*

Specifies a far pointer to a WAVEHDR structure identifying the buffer containing the waveform data.

DWORD *dwParam2*

Not used.

Return Value

None.

Comments

The returned buffer might not be full. Use the dwBytesRecordedfield of the WAVEHDR structure specified by *dwParam1* to determine the number of bytes recorded into the returned buffer.

See Also

MM_WIM_DATA

WIM_OPEN

This message is sent to a waveform input callback function when a waveform input device is opened.

Parameters

DWORD *dwParam1*
Not used.

DWORD *dwParam2*
Not used.

Return Value

None.

See Also

MM_WIM_OPEN

WOM_CLOSE

This message is sent to a waveform output callback function when a waveform output device is closed. The device handle is no longer valid once this message has been sent.

Parameters

DWORD *dwParam1*

Not used.

DWORD *dwParam2*

Not used.

Return Value

None.

See Also

MM_WOM_CLOSE

WOM_DONE

This message is sent to a waveform output callback function when the specified output buffer is being returned to the application. Buffers are returned to the application when they have been played, or as the result of a call to waveOutReset.

Parameters

DWORD *dwParam1*

Specifies a far pointer to a WAVEHDR structure identifying the buffer.

DWORD *dwParam2*

Not used.

Return Value

None.

See Also

MM_WOM_DONE

WOM_OPEN

This message is sent to a waveform output callback function when a waveform output device is opened.

Parameters

DWORD *dwParam1*

Not used.

DWORD *dwParam2*

Not used.

Return Value

None.

See Also

MM_WOM_OPEN

auxGetDevCaps

Syntax

UINT **auxGetDevCaps**(*wDeviceID*, *lpCaps*, *wSize*)

This function queries a specified auxiliary output device to determine its capabilities.

Parameters

UINT *wDeviceID*

Identifies the auxiliary output device to be queried. Specify a valid device ID (see the following "Comments" section), or use the following constant:

AUX_MAPPER

Auxiliary audio mapper. The function will return an error if no auxiliary audio mapper is installed.

LPAUXCAPS *lpCaps*

Specifies a far pointer to an AUXCAPS structure. This structure is filled with information about the capabilities of the device.

UINT *wSize*

Specifies the size of the AUXCAPS structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID

Specified device ID is out of range.

MMSYSERR_NODRIVER

The driver failed to install.

Comments

The device ID specified by *wDeviceID* varies from zero to one less than the number of devices present. Use auxGetNumDevs to determine the number of auxiliary output devices present in the system.

See Also

auxGetNumDevs

auxGetNumDevs

Syntax

UINT **auxGetNumDevs**()

This function retrieves the number of auxiliary output devices present in the system.

Return Value

Returns the number of auxiliary output devices present in the system.

See Also

auxGetDevCaps

auxGetVolume

Syntax

UINT **auxGetVolume**(*wDeviceID*, *lpdwVolume*)

This function returns the current volume setting of an auxiliary output device.

Parameters

UINT *wDeviceID*

Identifies the auxiliary output device to be queried.

LPDWORD *lpdwVolume*

Specifies a far pointer to a location to be filled with the current volume setting. The low-order word of this location contains the left channel volume setting, and the high-order word contains the right channel setting. A value of 0xFFFF represents full volume, and a value of 0x0000 is silence.

If a device does not support both left and right volume control, the low-order word of the specified location contains the volume level.

The full 16-bit setting(s)set with auxSetVolume are returned, regardless of whether the device supports the full 16 bits of volume level control.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID

Specified device ID is out of range.

MMSYSERR_NODRIVER

The driver failed to install.

Comments

Not all devices support volume control. To determine whether the device supports volume control, use the AUXCAPS_VOLUME flag to test the dwSupport field of the AUXCAPS structure (filled by auxGetDevCaps).

To determine whether the device supports volume control on both the left and right channels, use the AUXCAPS_LRVOLUME flag to test the dwSupport field of the AUXCAPS structure (filled by auxGetDevCaps).

See Also

auxSetVolume

auxOutMessage

Syntax

UINT **auxOutMessage**(*wDeviceID*, msg, dw1, dw2)

This function sends a message to an auxiliary output device. It also performs error checking on the device ID.

Parameters

UINT wDeviceID
Identifies the auxiliary output device to receive the message.

UINT msg
Identifies the message to send.

DWORD *wDeviceID*
Specifies the first message parameter.

DWORD *wDeviceID*
Specifies the seond message parameter.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID
Specified device ID is out of range.

MMSYSERR_NODRIVER
The driver failed to install.

auxSetVolume

Syntax

UINT **auxSetVolume**(*wDeviceID*, *dwVolume*)

This function sets the volume in an auxiliary output device.

Parameters

UINT *wDeviceID*

Identifies the auxiliary output device to be queried. Device IDs are determined implicitly from the number of devices present in the system. Device ID values range from zero to one less than the number of devices present. Use auxGetNumDevsto determine the number of auxiliary devices in the system.

DWORD *dwVolume*

Specifies the new volume setting. The low-order word specifies the left channel volume setting, and the high-order word specifies the right channel setting. A value of 0xFFFF represents full volume, and a value of 0x0000 is silence.

If a device does not support both left and right volume control, the low-order word of *dwVolume* specifies the volume level, and the high-order word is ignored.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID

Specified device ID is out of range.

MMSYSERR_NODRIVER

The driver failed to install.

Comments

Not all devices support volume control. To determine whether the device supports volume control, use the AUXCAPS_VOLUME flag to test the dwSupport field of the AUXCAPS structure (filled by auxGetDevCaps).

To determine whether the device supports volume control on both the left and right channels, use the AUXCAPS_LRVOLUME flag to test the dwSupport field of the AUXCAPS structure (filled by auxGetDevCaps).

Most devices do not support the full 16 bits of volume level control and will use only the high-order bits of the requested volume setting. For example, for a device that supports 4 bits of volume control, requested volume level values of 0x4000, 0x4fff, and 0x43be will all produce the same physical volume setting, 0x4000. The auxGetVolume function will return the full 16-bit setting set with **auxSetVolume**.

Volume settings are interpreted logarithmically. This means the perceived volume increase is the same when increasing the volume level from 0x5000 to 0x6000 as it is from 0x4000 to 0x5000.

See Also

auxGetVolume

joyGetDevCaps

Syntax

UINT **joyGetDevCaps**(*wJoyId*, *lpCaps*, *wSize*)

This function queries a joystick device to determine its capabilities.

Parameters

UINT *wJoyId*

Identifies the device to be queried. This value is either JOYSTICKID1 or JOYSTICKID2.

LPJOYCAPS *lpCaps*

Specifies a far pointer to a JOYCAPS structure. This structure is filled with information about the capabilities of the joystick device.

UINT *wSize*

Specifies the size of the JOYCAPS structure.

Return Value

Returns JOYERR_NOERROR if successful. Otherwise, returns one of the following error codes:

MMSYSERR_NODRIVER

The joystick driver is not present.

JOYERR_PARMS

The specified joystick device ID *wId* is invalid.

Comments

Use joyGetNumDevs to determine the number of joystick devices supported by the driver.

See Also

joyGetNumDevs

joyGetNumDevs

Syntax

UINT **joyGetNumDevs**()

This function returns the number of joystick devices supported by the system.

Parameters

None.

Return Value

Returns the number of joystick devices supported by the joystick driver. If no driver is present, the function returns zero.

Comments

Use joyGetPos to determine whether a given joystick is actually attached to the system. The joyGetPos function returns a JOYERR_UNPLUGGED error code if the specified joystick is not connected.

See Also

joyGetDevCaps, joyGetPos

joyGetPos

Syntax

UINT **joyGetPos**(*wJoyId*, *lpInfo*)

This function queries for the position and button activity of a joystick device.

Parameters

UINT *wJoyId*

Identifies the joystick device to be queried. This value is either JOYSTICKID1 or JOYSTICKID2.

LPJOYINFO *lpInfo*

Specifies a far pointer to a JOYINFO structure. This structure is filled with information about the position and button activity of the joystick device.

Return Value

Returns JOYERR_NOERROR if successful. Otherwise, returns one of the following error codes:

MMSYSERR_NODRIVER

The joystick driver is not present.

JOYERR_PARMS

The specified joystick device ID *wId* is invalid.

JOYERR_UNPLUGGED

The specified joystick is not connected to the system.

joyGetThreshold

Syntax

UINT **joyGetThreshold**(*wJoyId*, *lpwThreshold*)

This function queries the current movement threshold of a joystick device.

Parameters

UINT *wJoyId*

Identifies the joystick device to be queried. This value is either JOYSTICKID1 or JOYSTICKID2.

LPWORD *lpwThreshold*

Specifies a far pointer to a UINT variable that is filled with the movement threshold value.

Return Value

Returns JOYERR_NOERROR if successful. Otherwise, returns one of the following error codes:

MMSYSERR_NODRIVER

The joystick driver is not present.

JOYERR_PARMS

The specified joystick device ID *wId* is invalid.

Comments

The movement threshold is the distance the joystick must be moved before a WM_JOYMOVE message is sent to a window that has captured the device. The threshold is initially zero.

See Also

joySetThreshold

joyReleaseCapture

Syntax

UINT **joyReleaseCapture**(*wJoyId*)

This function releases the capture set by joySetCapture on the specified joystick device.

Parameters

UINT *wJoyId*

Identifies the joystick device to be released. This value is either JOYSTICKID1 or JOYSTICK2.

Return Value

Returns JOYERR_NOERROR if successful. Otherwise, returns one of the following error codes:

MMSYSERR_NODRIVER

The joystick driver is not present.

JOYERR_PARMS

The specified joystick device ID *wId* is invalid.

See Also

joySetCapture

joySetCapture

Syntax

UINT **joySetCapture**(*hWnd*, *wJoyId*, *wPeriod*, *bChanged*)

This function causes joystick messages to be sent to the specified window.

Parameters

HWND *hWnd*
Specifies a handle to the window to which messages are to be sent.

UINT *wJoyId*
Identifies the joystick device to be captured. This value is either JOYSTICKID1 or JOYSTICKID2.

UINT *wPeriod*
Specifies the polling rate, in milliseconds.

BOOL *bChanged*
If this parameter is set to TRUE, then messages are sent only when the position changes by a value greater than the joystick movement threshold.

Return Value

Returns JOYERR_NOERROR if successful. Otherwise, returns one of the following error codes:

MMSYSERR_NODRIVER
The joystick driver is not present.

JOYERR_PARMS
The specified window handle *hWnd* or joystick device ID *wId* is invalid.

JOYERR_NOCANDO
Cannot capture joystick input because some required service (for example, a Windows timer) is unavailable.

JOYERR_UNPLUGGED
The specified joystick is not connected to the system.

Comments

This function fails if the specified joystick device is currently captured. You should call the joyReleaseCapture function when the joystick capture is no longer needed. If the window is destroyed, the joystick will be released automatically.

See Also

joyReleaseCapture, joySetThreshold, joyGetThreshold

joySetThreshold

Syntax

UINT **joySetThreshold**(*wJoyId*, *wThreshold*)

This function sets the movement threshold of a joystick device.

Parameters

UINT *wJoyId*

Identifies the joystick device. This value is either JOYSTICKID1 or JOYSTICKID2.

UINT *wThreshold*

Specifies the new movement threshold.

Return Value

Returns JOYERR_NOERROR if successful. Otherwise, returns one of the following error codes:

MMSYSERR_NODRIVER

The joystick driver is not present.

JOYERR_PARMS

The specified joystick device ID *wId* is invalid.

Comments

The movement threshold is the distance the joystick must be moved before a MM_JOYMOVE message is sent to a window that has captured the device.

See Also

joyGetThreshold, joySetCapture

mciGetCreatorTask

Syntax

HTASK WINAPI **mciGetCreatorTask**(*uDeviceID*)

This function retrieves a handle to the process responsible for opening a device.

Parameters

UINT *udeviceID*

Specifies the device ID whose creator task is to be returned.

Return Value

Returns a handle to the creator task if successful. Otherwise, returns NULL.

mciGetDeviceID

Syntax

UINT FAR **mciGetDeviceID**(*lpstrName*)

This function retrieves the device ID corresponding to the name of an open MCI device.

Parameters

LPCSTR *lpstrName*

Specifies the device name used to open the MCI device.

Return Value

Returns the device ID assigned when the device was opened. Returns zero if the device name isn't known, if the device isn't open, or if there was insufficient memory to complete the operation. Each compound device element has a unique device ID. The ID of the "all" device is MCI_ALL_DEVICE_ID

See Also

MCI_OPEN

mciGetErrorString

Syntax

UINT **mciGetErrorString**(*dwError*, *lpstrBuffer*, *wLength*)

This function returns a textual description of the specified MCI error.

Parameters

DWORD *dwError*

Specifies the error code returned by mciSendCommand or mciSendString.

LPSTR *lpstrBuffer*

Specifies a pointer to a buffer that is filled with a textual description of the specified error.

UINT *wLength*

Specifies the length of the buffer pointed to by *lpstrBuffer*.

Return Value

Returns TRUE if successful. Otherwise, the given error code was not known.

mciGetYieldProc

Syntax

YIELDPROC WINAPI **mciGetYieldProc**(*wDeviceID, lpdwYieldData*)

This function returns the address of the callback procedure associated with the mci WAIT flag; the callback procedure is called periodically while an MCI device waits for a command specified with the WAIT flag to complete.

Parameters

UINT *wDeviceID*

Specifies the ID of the MCI device being monitored while it performs an MCI command.

LPDWORD *lpdwYieldData*

Optionally specifies a buffer to hold the yield data passed to the function. If the parameter is NULL, it is ignored.

Return Value

Returns the current yield proc, if it exists. Otherwise, returns NULL for an invalid device ID.

mciSendCommand

Syntax

DWORD **mciSendCommand**(*wDeviceID*, *wMessage*, *dwParam1*, *dwParam2*)

This function sends a command message to the specified MCI device.

Parameters

UINT *wDeviceID*
Specifies the device ID of the MCI device to receive the command. This parameter is not used with the MCI_OPEN command.

UINT *wMessage*
Specifies the command message.

DWORD *dwParam1*
Specifies flags for the command.

DWORD *dwParam2*
Specifies a pointer to a parameter block for the command.

Return Value

Returns zero if the function was successful. Otherwise, it returns error information. The low-order word of the returned DWORD is the error return value. If the error is device-specific, the high-order word contains the driver ID; otherwise the high-order word is zero.

To get a textual description of mciSendCommand return values, pass the return value to mciGetErrorString.

Error values that are returned when a device is being opened are listed with the MCI_OPEN message. In addition to the MCI_OPEN error returns, this function can return the following values:

MCIERR_BAD_TIME_FORMAT
Illegal value for time format.

MCIERR_CANNOT_LOAD_DRIVER
The specified device driver will not load properly.

MCIERR_CANNOT_USE_ALL
The device name "all" is not allowed for this command.

MCIERR_CREATEWINDOW
Could not create or use window.

MCIERR_DEVICE_LENGTH
The device or driver name is too long. specify a device or driver name that is less than 79 characters.

MCIERR_DEVICE_LOCKED
The device is now being closed. Wat a few seconds, then try again.

MCIERR_DEVICE_NOT_INSTALLED
The specified device is not installed on the system. Use the Drivers option from the Control Panel to install the device.

MCIERR_DEVICE_NOT_READY
the device driver is not ready.

MCIERR_DEVICE_OPEN

The device name is already used as an alias by this application. Use a unique alias.

MCIERR_DEVICE_ORD_LENGTH
The device or driver name is too long. Specify a device or driver name that is less than 79 characters.

MCIERR_DEVICE_TYPE_REQUIRED
The specified device cannot be found on the system. Check that the device is installed and the device name is spelled correctly.

MCIERR_DRIVER
The device driver exhibits a problem. Check with the device manufacturer about obtaining a new driver.

MCIERR_DRIVER_INTERNAL
The device driver exhibits a problem. Check with the device manufacturer about obtaining a new driver.

MCIERR_DUPLICATE_ALIAS
The specified alias is already used in this application. Use a unique alias.

MCIERR_EXTENSION_NOT_FOUND
The specified extension has no device type associated with it. Specify a device type.

MCIERR_EXTRA_CHARACTERS
You must enclose a string with quotation marks; characters following the closing quotation mark are not valid.

MCIERR_FILE_NOT_FOUND
The requested file was not found. Check that the path and filename are correct.

MCIERR_FILE_NOT_SAVED
The file was not saved. Make sure your system has sufficient disk space or has an intact network connection.

MCIERR_FILE_READ
A read from the file failed. Make sure the file is present on your system or that your system has an intact network connection.

MCIERR_FILE_WRITE
A write to the file failed. Make sure your system has sufficient disk space or has an intact network connection.

MCIERR_FLAGS_NOT_COMPATIBLE
The specified parameters cannot be used together.

MCIERR_FILENAME_REQUIRED
The filename is invalid. Make sure the filename is no longer than eight characters, followed by a period and an extension.

MCIERR_GET_CD
The requested file or MCI device was not found. Try changing directories or restarting your system.

MCIERR_HARDWARE
The specified device exhibits a problem. Check that the device is working correctly or contact the device manufacturer.

MCIERR_ILLEGAL_FOR_AUTO_OPEN
MCI will not perform the specified command on an automatically opened device. Wait until the device is closed, then try to perform the command.

MCIERR_INTERNAL
A problem occurred in initializing MCI. Try restarting the Windows operating system.

MCIERR_INVALID_DEVICE_ID
Invalid device ID. Use the ID givien to the device when the device was opened.

MCIERR_INVALID_DEVICE_NAME
The specified device is not open nor recognized by MCI.

MCIERR_INVALID_FILE
The specified file cannot be played on the specified MCI device. The file may be corrupt or may use an incorrect file format.

MCIERR_INVALID_SETUP
The current MIDI setup is damaged. Copy the original midimap.cfg file to the Windows SYSTEM directory; then, try to perform the command again.

MCIERR_MISSING_INTEGER
The specified command requires an integer parameter, which you must supply.

MCIERR_MISSING_PARAMETER
The specified command requires a parameter, which you must supply.

MCIERR_MULTIPLE
Errors occurred in more than one device. Specify each command and device separately to identify the devices causing the errors.

MCIERR_MUST_USE_SHAREABLE
The device driver is already in use. You must specify the "shareable" parameter with each open command to share the device.

MCIERR_NO_ELEMENT_ALLOWED
The specified device does not use a filename.

MCIERR_NO_INTEGER
The parameter for this MCI command must be an integer value.

MCIERR_NO_WINDOW
There is no display window.

MCIERR_NONAPPLICABLE_FUNCTION
The specified MCI command sequence cannot be performed in the given order. Correct the command sequence; then, try again.

MCIERR_NULL_PARAMETER_BLOCK
A null parameter block was passed to MCI.

MCIERR_OUT_OF_MEMORY
Your system does not have enough memory for this task. Quit one or more applications to increase the available memory; then, try to perform the task again.

MCIERR_OUTOFRANGE
The specified parameter value is out of range for this MCI command.

MCIERR_SET_CD
The specified file or MCI device is inaccessible because the application cannot change directories.

MCIERR_SET_DRIVE
The specified file or MCI device is inaccessible because the application cannot change drives.

MCIERR_UNNAMED_RESOURCE
You cannot store an unnamed file. Specify a filename.

MCIERR_UNRECOGNIZED_COMMAND
The driver cannot recognize the specified command.

MCIERR_UNSUPPORTED_FUNCTION
The MCI device driver that the system is using does not support the specified command.

Return Values for MCI Sequencers

The following additional return values are defined for the sequencer device type:

MCIERR_SEQ_DIV_INCOMPATIBLE
The time formats of the "song pointer" and SMPTE are mutually exclusive. You can't use them together.

MCIERR_SEQ_NOMIDIPRESENT
This system has no installed MIDI devices. Use the Drivers option from the coNtrol Panel to install a MIDI driver.

MCIERR_SEQ_PORT_INUSE
The specified MIDI port is already in use. Wait until it is free; then, try again.

MCIERR_SEQ_PORT_MAPNODEVICE
The current MIDI Mapper setup refers to a MIDI device that is not installed on the system. Use the MIDI Mapper option from the Control Panel to edit the setup.

MCIERR_SEQ_PORT_MISCERROR
An error occurred with the specified port.

MCIERR_SEQ_PORT_NONEXISTENT
The specified MIDI device is not installed on the system. Use the Drivers option from the Control Panel to install a MIDI device.

MCIERR_SEQ_PORTUNSPECIFIED
The system does not have a current MIDI port specified.

MCIERR_SEQ_TIMER
All multimedia timers are being used by other applications. Quit one of these applications; then, try again.

Return Values for MCI Waveform Audio Devices

The following additional return values are defined for the waveaudio device type:

MCIERR_WAVE_INPUTSINUSE
All waveform devices that can record files in the current format are in use. Wait until one of these devices is free; then, try again.

MCIERR_WAVE_INPUTSUNSUITABLE
No installed waveform device can record files in the current format. Use the Drivers option from

the Control Panel to install a suitable waveform recording device.

MCIERR_WAVE_INPUTUNSPECIFIED
You can specify any compatible waveform recording device.

MCIERR_WAVE_OUTPUTSINUSE
All waveform devices that can play files in the current format are in use. Wait until one of these devices is free; then, try again.

MCIERR_WAVE_OUTPUTSUNSUITABLE
No installed waveform device can play files in the current format. Use the Drivers option from the Control Panel to install a suitable waveform recording device.

MCIERR_WAVE_OUTPUTUNSPECIFIED
You can specify any compatible waveform playback device.

MCIERR_WAVE_SETINPUTINUSE
The current waveform device is in use. Wait until the device is free; then, try again to set the device for recording.

MCIERR_WAVE_SETINPUTUNSUITABLE
The device you are using to record a waveform cannot recognize the data format.

MCIERR_WAVE_SETOUTPUTINUSE
The current waveform device is in use. Wait until the device is free; the, try again to set the device for playback.

MCIERR_WAVE_SETOUTPUTUNSUITABLE
The device you are using to play back a waveform cannot recognize the data format.

Comments
Use the MCI_OPEN command to obtain the device ID specified by *wDeviceID*.

See Also
mciGetErrorString, mciSendString

mciSendString

Syntax

DWORD **mciSendString**(*lpstrCommand*, *lpstrReturnString*, *wReturnLength*, *hCallback*)

This function sends a command string to an MCI device. The device that the command is sent to is specified in the command string.

Parameters

LPCSTR *lpstrCommand*
Specifies an MCI command string.

LPSTR *lpstrReturnString*
Specifies a buffer for return information. If no return information is needed, you can specify NULL for this parameter.

UINT *wReturnLength*
Specifies the size of the return buffer specified by *lpstrReturnString*.

HANDLE *hCallback*
Specifies a handle to a window to call back if "notify" was specified in the command string.

Return Value

Returns zero if the function was successful. Otherwise, it returns error information. The low-order word of the returned DWORD contains the error return value.

To get a textual description of mciSendString return values, pass the return value to mciGetErrorString.

The error returns listed for mciSendCommand also apply to **mciSendString**. The following error returns are unique to mciSendString:

MCIERR_BAD_CONSTANT
The specified constant is invalid for this command.

MCIERR_BAD_INTEGER
The specified integer is invalid for this command.

MCIERR_DUPLICATE_FLAGS
The parameter or value was specified twice. Remove the duplicate occurrence of the parameter or value.

MCIERR_MISSING_COMMAND_STRING
No command was specified.

MCIERR_MISSING_DEVICE_NAME
The specified command requires an alias or the name of a file, driver, or device, which you must specify.

MCIERR_MISSING_STRING_ARGUMENT
The specified command requires a string parameter, which you must supply.

MCIERR_NEW_REQUIRES_ALIAS
You must specify an alias when using the "new" parameter.

MCIERR_NO_CLOSING_QUOTE
The string parameter is missing a closing double quotation mark, which you must supply.

MCIERR_NOTIFY_ON_AUTO_OPEN

You cannot use the "notify" flag with automatically opened device.

MCIERR_PARAM_OVERFLOW
The output string was too large to fit in the return buffer. Increase the size of the buffer.

MCIERR_PARSER_INTERNAL
The device driver returned an invalid return type. Check with the device manufacturer about obtaining a new driver.

MCIERR_UNRECOGNIZED_KEYWORD
The driver cannot recognize the specified command parameter.

See Also
mciGetErrorString, mciSendCommand

mciSetYieldProc

Syntax

BOOL **mciSetYieldProc**(*wDeviceID*, *fpYieldProc*, *dwYieldData*)

This function sets the address of a callback procedure to be called periodically when an MCI device is completing a command specified with the WAIT flag.

Parameters

UINT wDeviceID

Specifies the device ID of the MCI device to which the yield procedure is to be assigned.

YIELDPROC *fpYieldProc*

Specifies the callback procedure to be called when the given device is yielding. Specify a NULL value to disable any existing yield procedure.

DWORD *dwYieldData*

Specifies the data sent to the yield procedure when it is called for the given device.

Return Value

Returns TRUE if successful. Returns FALSE for an invalid device ID.

Callback

int CALLBACK **YieldProc**(*wDeviceID*, *dwData*)

YieldProc is a placeholder for the application-supplied function name. Export the actual name by including it in the EXPORTS statement in your module-definition file.

Callback Parameters

UINT *wDeviceID*

Specifies the device ID of the MCI device.

DWORD *dwData*

Specifies the application-supplied yield data originally supplied in the *dwYieldData* parameter.

Callback Return Value

Return zero to continue the operation. To cancel the operation, return a nonzero value.

Comments

This call overrides any previous yield procedure for this device.

midiInAddBuffer

Syntax

UINT **midiInAddBuffer**(*hMidiIn*, *lpMidiInHdr*, *wSize*)

This function sends an input buffer to a specified opened MIDI input device. When the buffer is filled, it is sent back to the application. Input buffers are used only for system-exclusive messages.

Parameters

HMIDIIN *hMidiIn*

Specifies a handle to the MIDI input device.

LPMIDIHDR *lpMidiInHdr*

Specifies a far pointer to a MIDIHDRstructure that identifies the buffer.

UINT *wSize*

Specifies the size of the MIDIHDR structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MIDIERR_UNPREPARED

lpMidiInHdr hasn't been prepared.

Comments

The data buffer must be prepared with midiInPrepareHeader before it is passed to **midiInAddBuffer**. The MIDIHDR data structure and the data buffer pointed to by its lpData field must be allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with **GlobalLock**.

See Also

midiInPrepareHeader

midiInClose

Syntax

UINT **midiInClose**(*hMidiIn*)

This function closes the specified MIDI input device.

Parameters

HMIDIIN *hMidiIn*

Specifies a handle to the MIDI input device. If the function is successful, the handle is no longer valid after this call.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MIDIERR_STILLPLAYING

There are still buffers in the queue.

Comments

If there are input buffers that have been sent with midiInAddBuffer and haven't been returned to the application, the close operation will fail. Call midiInReset to mark all pending buffers as being done.

See Also

midiInOpen, midiInReset

midiInGetDevCaps

Syntax

UINT **midiInGetDevCaps**(*wDeviceID*, *lpCaps*, *wSize*)

This function queries a specified MIDI input device to determine its capabilities.

Parameters

UINT wDeviceID

Identifies the MIDI input device to query. Specify a valid MIDI input device ID (see the following "Comments" section) or the following constant:

MIDI_MAPPER

MIDI mapper. The function will return an error if no MIDI mapper is installed.

LPMIDIINCAPS *lpCaps*

Specifies a far pointer to a MIDIINCAPSdata structure. This structure is filled with information about the capabilities of the device.

UINT *wSize*

Specifies the size of the MIDIINCAPS structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID

Specified device ID is out of range.

MMSYSERR_NODRIVER

The driver was not installed.

Comments

The device ID specified by *wDeviceID* varies from zero to one less than the number of devices present. Use midiInGetNumDevs to determine the number of MIDI input devices present in the system.

Only *wSize* bytes (or less) of information is copied to the location pointed to by *lpCaps*. If *wSize* is zero, nothing is copied, and the function returns zero.

See Also

midiInGetNumDevs

midiInGetErrorText

Syntax

UINT **midiInGetErrorText**(*wError*, *lpText*, *wSize*)

This function retrieves a textual description of the error identified by the specified error number.

Parameters

UINT *wError*
Specifies the error number.

LPSTR *lpText*
Specifies a far pointer to the buffer to be filled with the textual error description.

UINT *wSize*
Specifies the length of buffer pointed to by *lpText*.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADERRNUM
Specified error number is out of range.

Comments

If the textual error description is longer than the specified buffer, the description is truncated. The returned error string is always null-terminated. If *wSize* is zero, nothing is copied, and the function returns zero. All error descriptions are less than MAXERRORLENGTH characters long.

midiInGetID

Syntax

UINT **midiInGetID**(*hMidiIn*, *lpwDeviceID*)

This function gets the device ID for a MIDI input device.

Parameters

HMIDIIN *hMidiIn*

Specifies the handle to the MIDI input device.

LPWORD *lpwDeviceID*

Specifies a pointer to the UINT-sized memory location to be filled with the device ID.

Return Value

Returns zero if successful. Otherwise, returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

The *hMidiIn* parameter specifies an invalid handle.

midiInGetNumDevs

Syntax

UINT **midiInGetNumDevs**()

This function retrieves the number of MIDI input devices in the system.

Parameters

None.

Return Value

Returns the number of MIDI input devices present in the system.

See Also

midiInGetDevCaps

midiInMessage

Syntax

DWORD **midiInMessage**(hMidiIn, msg, dwParam1, dwParam2)

This function sends a message to a MIDI input device driver. Use it to send driver-specific messages that aren't supported by the MIDI APIs.

Parameters

HMIDIIN *hMidiIn*

Specifies the handle to the audio device driver.

UINT m*sg*

Specifies the message to send.

DWORD *dwParam1*

Specifies the first message parameter.

DWORD *dwParam2*

Specifies the second message parameter.

Return Value

Returns the value returned by the audio device driver.

Comments

Do not use this function to send standard messages to an audio device driver.

See Also

midiOutMessage

midiInOpen

Syntax

UINT **midiInOpen**(*lphMidiIn*, *wDeviceID*, *dwCallback*, *dwCallbackInstance*, *dwFlags*)

This function opens a specified MIDI input device.

Parameters

LPHMIDIIN *lphMidiIn*

Specifies a far pointer to an HMIDIIN handle. This location is filled with a handle identifying the opened MIDI input device. Use the handle to identify the device when calling other MIDI input functions.

UINT *wDeviceID*

Identifies the MIDI input device to be opened. Specify a valid MIDI input device ID (see the following "Comments" section) or the following constant:

MIDI_MAPPER

MIDI mapper. The function will return an error if no MIDI mapper is installed.

DWORD *dwCallback*

Specifies the address of a fixed callback function or a handle to a window called with information about incoming MIDI messages.

DWORD *dwCallbackInstance*

Specifies user instance data passed to the callback function. This parameter is not used with window callbacks.

DWORD *dwFlags*

Specifies a callback flag for opening the device.

CALLBACK_WINDOW

If this flag is specified, *dwCallback* is assumed to be a window handle.

CALLBACK_FUNCTION

If this flag is specified, *dwCallback* is assumed to be a callback procedure address.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID

Specified device ID is out of range.

MMSYSERR_ALLOCATED

Specified resource is already allocated.

MMSYSERR_NOMEM

Unable to allocate or lock memory.

Callback

void CALLBACK **MidiInFunc**(*hMidiIn*, *wMsg*, *dwInstance*, *dwParam1*, *dwParam2*)

MidiInFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an EXPORTS statement in the DLL's module definition file.

Callback Parameters

HMIDIIN *hMidiIn*

Specifies a handle to the MIDI input device.

UINT *wMsg*
Specifies a MIDI input message.

DWORD *dwInstance*
Specifies the instance data supplied with **midiInOpen**.

DWORD *dwParam1*
Specifies a parameter for the message.

DWORD *dwParam2*
Specifies a parameter for the message.

Comments

The device ID specified by *wDeviceID* varies from zero to one less than the number of devices present. Use midiInGetNumDevs to determine the number of MIDI input devices present in the system.

If a window is chosen to receive callback information, the following messages are sent to the window procedure function to indicate the progress of MIDI input:

* MM_MIM_OPEN
* MM_MIM_CLOSE
* MM_MIM_DATA
* MM_MIM_LONGDATA
* MM_MIM_ERROR
* MM_MIM_LONGERROR

If a function is chosen to receive callback information, the following messages are sent to the function to indicate the progress of MIDI input:

* MIM_OPEN
* MIM_CLOSE
* MIM_DATA
* MIM_LONGDATA
* MIM_ERROR
* MIM_LONGERROR

The callback function must reside in a DLL. You do not have to use **MakeProcInstance** to get a procedure-instance address for the callback function.

Because the callback is accessed at interrupt time, it must reside in a DLL, and its code segment must be specified as FIXED in the module-definition file for the DLL. Any data that the callback accesses must be in a FIXED data segment as well. The callback may not make any system calls except for **PostMessage**, timeGetSystemTime, timeGetTime, timeSetEvent, timeKillEvent, midiOutShortMsg, midiOutLongMsg, and OutputDebugStr.

See Also

midiInClose

midiInPrepareHeader

Syntax

UINT **midiInPrepareHeader**(*hMidiIn*, *lpMidiInHdr*, *wSize*)

This function prepares a buffer for MIDI input.

Parameters

HMIDIIN *hMidiIn*

Specifies a handle to the MIDI input device.

LPMIDIHDR *lpMidiInHdr*

Specifies a pointer to a MIDIHDRstructure that identifies the buffer to be prepared.

UINT *wSize*

Specifies the size of the MIDIHDR structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MMSYSERR_NOMEM

Unable to allocate or lock memory.

Comments

The MIDIHDR data structure and the data block pointed to by its lpData field must be allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with **GlobalLock**. Preparing a header that has already been prepared has no effect, and the function returns zero.

See Also

midiInUnprepareHeader

midiInReset

Syntax

UINT **midiInReset**(*hMidiIn*)

This function stops input on a given MIDI input device and marks all pending input buffers as done.

Parameters

HMIDIIN *hMidiIn*

Specifies a handle to the MIDI input device.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

See Also

midiInStart, midiInStop, midiInAddBuffer, midiInClose

midiInStart

Syntax

UINT **midiInStart**(*hMidiIn*)

This function starts MIDI input on the specified MIDI input device.

Parameters

HMIDIIN *hMidiIn*

Specifies a handle to the MIDI input device.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

Comments

This function resets the timestamps to zero; timestamp values for subsequently received messages are relative to the time this function was called.

All messages other than system-exclusive messages are sent directly to the client when received. System-exclusive messages are placed in the buffers supplied by midiInAddBuffer;if there are no buffers in the queue, the data is thrown away without notification to the client, and input continues.

Buffers are returned to the client when full, when a complete system-exclusive message has been received, or when midiInReset is called. The dwBytesRecorded field in the header will contain the actual length of data received.

Calling this function when input is already started has no effect, and the function returns zero.

See Also

midiInStop, midiInReset

midiInStop

Syntax

UINT **midiInStop**(*hMidiIn*)

This function terminates MIDI input on the specified MIDI input device.

Parameters

HMIDIIN *hMidiIn*

Specifies a handle to the MIDI input device.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

Comments

Current status (running status, parsing state, etc.) is maintained across calls to **midiInStop** and midiInStart. If there are any system-exclusive message buffers in the queue, the current buffer is marked as done (the dwBytesRecorded field in the header will contain the actual length of data), but any empty buffers in the queue remain there. Calling this function when input is not started has no effect, and the function returns zero.

See Also

midiInStart, midiInReset

midiInUnprepareHeader

Syntax

UINT **midiInUnprepareHeader**(*hMidiIn*, *lpMidiInHdr*, *wSize*)

This function cleans up the preparation performed by midiInPrepareHeader. The **midiInUnprepareHeader** function must be called after the device driver fills a data buffer and returns it to the application. You must call this function before freeing the data buffer.

Parameters

HMIDIIN *hMidiIn*

Specifies a handle to the MIDI input device.

LPMIDIHDR *lpMidiInHdr*

Specifies a pointer to a MIDIHDRstructure identifying the data buffer to be cleaned up.

UINT *wSize*

Specifies the size of the MIDIHDR structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MIDIERR_STILLPLAYING

lpMidiInHdr is still in the queue.

Comments

This function is the complementary function to midiInPrepareHeader. You must call this function before freeing the data buffer with **GlobalFree**. After passing a buffer to the device driver with midiInAddBuffer, you must wait until the driver is finished with the buffer before calling **midiInUnprepareHeader**. Unpreparing a buffer that has not been prepared has no effect, and the function returns zero.

See Also

midiInPrepareHeader

midiOutCacheDrumPatches

Syntax

UINT **midiOutCacheDrumPatches**(*hMidiOut*, *wPatch*,*lpKeyArray*, *wFlags*)

This function requests that an internal MIDI synthesizer device preload a specified set of key-based percussion patches. Some synthesizers are not capable of keeping all percussion patches loaded simultaneously. Caching patches ensures specified patches are available.

Parameters

HMIDIOUT *hMidiOut*
Specifies a handle to the opened MIDI output device. This device should be an internal MIDI synthesizer.

UINT *wPatch*
Specifies which drum patch number should be used. To specify caching of the default drum patches, set this parameter to zero.

LPKEYARRAY *lpKeyArray*
Specifies a pointer to a **KEYARRAY** array indicating the key numbers of the specified percussion patches to be cached or uncached.

UINT *wFlags*
Specifies options for the cache operation. Only one of the following flags can be specified:

MIDI_CACHE_ALL
Cache all of the specified patches. If they can't all be cached, cache none, clear the **KEYARRAY** array, and return MMSYSERR_NOMEM.

MIDI_CACHE_BESTFIT
Cache all of the specified patches. If all patches can't be cached, cache as many patches as possible, change the **KEYARRAY** array to reflect which patches were cached, and return MMSYSERR_NOMEM.

MIDI_CACHE_QUERY
Change the **KEYARRAY** array to indicate which patches are currently cached.

MIDI_UNCACHE
Uncache the specified patches and clear the **KEYARRAY** array.

Return Value

Returns zero if the function was successful. Otherwise, it returns one of the following error codes:

MMSYSERR_INVALHANDLE
The specified device handle is invalid.

MMSYSERR_NOTSUPPORTED
The specified device does not support patch caching.

MMSYSERR_NOMEM
The device does not have enough memory to cache all of the requested patches.

Comments

The KEYARRAY data type is defined as:

```
typedef WORD KEYARRAY[128];
```

Each element of the array represents one of the 128 key-based percussion patches and has bits set for each of the 16 MIDI channels that use that particular patch. The least-significant bit represents physical channel 0; the most-significant bit represents physical channel 15. For example, if the patch on key number 60 is used by physical channels 9 and 15, element 60 would be set to 0x8200.

This function applies only to internal MIDI synthesizer devices. Not all internal synthesizers support patch caching. Use the MIDICAPS_CACHE flag to test the dwSupport field of the MIDIOUTCAPS structure filled by midiOutGetDevCaps to see if the device supports patch caching.

See Also

midiOutCachePatches

midiOutCachePatches

Syntax

UINT **midiOutCachePatches**(*hMidiOut*, *wBank*, *lpPatchArray*, *wFlags*)

This function requests that an internal MIDI synthesizer device preload a specified set of patches. Some synthesizers are not capable of keeping all patches loaded simultaneously and must load data from disk when they receive MIDI program change messages. Caching patches ensures specified patches are immediately available.

Parameters

HMIDIOUT *hMidiOut*

Specifies a handle to the opened MIDI output device. This device must be an internal MIDI synthesizer.

UINT *wBank*

Specifies which bank of patches should be used. To specify caching of the default patch bank, set this parameter to zero.

LPPATCHARRAY *lpPatchArray*

Specifies a pointer to a **PATCHARRAY** array indicating the patches to be cached or uncached.

UINT *wFlags*

Specifies options for the cache operation. Only one of the following flags can be specified:

MIDI_CACHE_ALL

Cache all of the specified patches. If they can't all be cached, cache none, clear the **PATCHARRAY** array, and return MMSYSERR_NOMEM.

MIDI_CACHE_BESTFIT

Cache all of the specified patches. If all patches can't be cached, cache as many patches as possible, change the **PATCHARRAY** array to reflect which patches were cached, and return MMSYSERR_NOMEM.

MIDI_CACHE_QUERY

Change the **PATCHARRAY** array to indicate which patches are currently cached.

MIDI_UNCACHE

Uncache the specified patches and clear the **PATCHARRAY** array.

Return Value

Returns zero if the function was successful. Otherwise, it returns one of the following error codes:

MMSYSERR_INVALHANDLE

The specified device handle is invalid.

MMSYSERR_NOTSUPPORTED

The specified device does not support patch caching.

MMSYSERR_NOMEM

The device does not have enough memory to cache all of the requested patches.

Comments

The **PATCHARRAY** data type is defined as:

```
typedef WORD PATCHARRAY[128];
```

Each element of the array represents one of the 128 patches and has bits set for each of the 16 MIDI

channels that use that particular patch. The least-significant bit represents physical channel 0; the most-significant bit represents physical channel 15 (0x0F). For example, if patch 0 is used by physical channels 0 and 8, element 0 would be set to 0x0101.

This function only applies to internal MIDI synthesizer devices. Not all internal synthesizers support patch caching. Use the MIDICAPS_CACHE flag to test the dwSupport field of the MIDIOUTCAPS structure filled by midiOutGetDevCaps to see if the device supports patch caching.

See Also

midiOutCacheDrumPatches

midiOutClose

Syntax

UINT **midiOutClose**(*hMidiOut*)

This function closes the specified MIDI output device.

Parameters

HMIDIOUT *hMidiOut*

Specifies a handle to the MIDI output device. If the function is successful, the handle is no longer valid after this call.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MIDIERR_STILLPLAYING

There are still buffers in the queue.

Comments

If there are output buffers that have been sent with midiOutLongMsg and haven't been returned to the application, the close operation will fail. Call midiOutReset to mark all pending buffers as being done.

See Also

midiOutOpen, midiOutReset

midiOutGetDevCaps

Syntax

UINT **midiOutGetDevCaps**(*wDeviceID*, *lpCaps*, *wSize*)

This function queries a specified MIDI output device to determine its capabilities.

Parameters

UINT wDeviceID

Identifies the MIDI output device to query. Specify a valid MIDI output device ID (see the following "Comments" section) or the following constant:

MIDI_MAPPER

MIDI mapper. The function will return an error if no MIDI mapper is installed.

LPMIDIOUTCAPS *lpCaps*

Specifies a far pointer to a MIDIOUTCAPSstructure. This structure is filled with information about the capabilities of the device.

UINT *wSize*

Specifies the size of the MIDIOUTCAPS structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID

Specified device ID is out of range.

MMSYSERR_NODRIVER

The driver was not installed.

Comments

The device ID specified by *wDeviceID* varies from zero to one less than the number of devices present. Use midiOutGetNumDevs to determine the number of MIDI output devices present in the system.

Only *wSize* bytes (or less) of information is copied to the location pointed to by *lpCaps*. If *wSize* is zero, nothing is copied, and the function returns zero.

See Also

midiOutGetNumDevs

midiOutGetErrorText

Syntax

UINT **midiOutGetErrorText**(*wError*, *lpText*, *wSize*)

This function retrieves a textual description of the error identified by the specified error number.

Parameters

UINT *wError*

Specifies the error number.

LPSTR *lpText*

Specifies a far pointer to a buffer to be filled with the textual error description.

UINT *wSize*

Specifies the length of the buffer pointed to by *lpText*.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADERRNUM

Specified error number is out of range.

Comments

If the textual error description is longer than the specified buffer, the description is truncated. The returned error string is always null-terminated. If *wSize* is zero, nothing is copied, and the function returns MMSYSERR_NOERROR. All error descriptions are less than MAXERRORLENGTH characters long.

midiOutGetID

Syntax

UINT **midiOutGetID**(*hMidiOut*, *lpwDeviceID*)

This function gets the device ID for a MIDI output device.

Parameters

HMIDIOUT *hMidiOut*

Specifies the handle to the MIDI output device.

LPWORD *lpwDeviceID*

Specifies a pointer to the UINT-sized memory location to be filled with the device ID.

Return Value

Returns MMSYSERR_NOERROR if successful. Otherwise, returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

The *hMidiOut* parameter specifies an invalid handle.

midiOutGetNumDevs

Syntax

UINT **midiOutGetNumDevs**()

This function retrieves the number of MIDI output devices present in the system.

Parameters

None.

Return Value

Returns the number of MIDI output devices present in the system.

See Also

midiOutGetDevCaps

midiOutGetVolume

Syntax

UINT **midiOutGetVolume**(*wDeviceID*, *lpdwVolume*)

This function returns the current volume setting of a MIDI output device.

Parameters

UINT wDeviceID

Identifies the MIDI output device.

LPDWORD *lpdwVolume*

Specifies a far pointer to a location to be filled with the current volume setting. The low-order word of this location contains the left channel volume setting, and the high-order UINT contains the right channel setting. A value of 0xFFFF represents full volume, and a value of 0x0000 is silence.

If a device does not support both left and right volume control, the low-order word of the specified location contains the mono volume level.

The full 16-bit setting(s)set with midiOutSetVolume is returned, regardless of whether the device supports the full 16 bits of volume level control.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MMSYSERR_NOTSUPPORTED

Function isn't supported.

MMSYSERR_NODRIVER

The driver was not installed.

Comments

Not all devices support volume control. To determine whether the device supports volume control, use the MIDICAPS_VOLUME flag to test the dwSupport field of the MIDIOUTCAPSstructure (filled by midiOutGetDevCaps).

To determine whether the device supports volume control on both the left and right channels, use the MIDICAPS_LRVOLUME flag to test the dwSupport field of the MIDIOUTCAPSstructure (filled by midiOutGetDevCaps).

See Also

midiOutSetVolume

midiOutLongMsg

Syntax

UINT **midiOutLongMsg**(*hMidiOut*, *lpMidiOutHdr*, *wSize*)

This function sends a buffer of MIDI data to the specified MIDI output device. Use this function to send multiple MIDI events, including system-exclusive messages, to a device.

Parameters

HMIDIOUT *hMidiOut*

Specifies a handle to the MIDI output device.

LPMIDIHDR *lpMidiOutHdr*

Specifies a far pointer to a MIDIHDRstructure that identifies the MIDI data buffer.

UINT *wSize*

Specifies the size of the MIDIHDR structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MIDIERR_UNPREPARED

lpMidiOutHdr hasn't been prepared.

MIDIERR_NOTREADY

The hardware is busy with other data.

Comments

The data buffer must be prepared with midiOutPrepareHeader before it is passed to **midiOutLongMsg**. The MIDIHDR data structure and the data buffer pointed to by its lpData field must be allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with **GlobalLock**. The MIDI output device driver determines whether the data is sent synchronously or asynchronously.

MIDI status is maintained across consecutive calls to **midiOutLongMsg** and midiOutShortMsg.

See Also

midiOutShortMsg, midiOutPrepareHeader

midiOutMessage

Syntax

DWORD **midiOutMessage**(hMidiOut, wMsg, dwParam1, dwParam2)

This function sends a message to a MIDI output device driver. Use it to send driver-specific messages that aren't supported by the MIDI APIs.

Parameters

HMIDIOUT *hMidiOut*

Specifies the handle to the audio device driver.

UINT *wMsg*

Specifies the message to send.

DWORD *dwParam1*

Specifies the first message parameter.

DWORD *dwParam2*

Specifies the second message parameter.

Return Value

Returns the value returned by the audio device driver.

Comments

Do not use this function to send standard messages to an audio device driver.

See Also

midiInMessage

midiOutOpen

Syntax

UINT **midiOutOpen**(*lphMidiOut*, *wDeviceID*, *dwCallback*, *dwCallbackInstance*, *dwFlags*)

This function opens a specified MIDI output device for playback.

Parameters

LPHMIDIOUT *lphMidiOut*

Specifies a far pointer to an HMIDIOUT handle. This location is filled with a handle identifying the opened MIDI output device. Use the handle to identify the device when calling other MIDI output functions.

UINT *wDeviceID*

Identifies the MIDI output device that is to be opened. Specify a valid MIDI output device ID (see the following "Comments" section) or the following constant:

MIDI_MAPPER

MIDI mapper. The function will return an error if no MIDI mapper is installed.

DWORD *dwCallback*

Specifies the address of a fixed callback function or a handle to a window called during MIDI playback to process messages related to the progress of the playback. Specify NULL for this parameter if no callback is desired.

DWORD *dwCallbackInstance*

Specifies user instance data passed to the callback. This parameter not used with window callbacks.

DWORD *dwFlags*

Specifies a callback flag for opening the device.

CALLBACK_WINDOW

If this flag is specified, *dwCallback* is assumed to be a window handle.

CALLBACK_FUNCTION

If this flag is specified, *dwCallback* is assumed to be a callback procedure address.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are as follows:

MMSYSERR_BADDEVICEID

Specified device ID is out of range.

MMSYSERR_ALLOCATED

Specified resource is already allocated.

MMSYSERR_NOMEM

Unable to allocate or lock memory.

MIDIERR_NOMAP

There is no current MIDI map. This occurs only when opening the mapper.

MIDIERR_NODEVICE

A port in the current MIDI map doesn't exist. This occurs only when opening the mapper.

Callback

void CALLBACK **MidiOutFunc**(*hMidiOut*, *wMsg*, *dwInstance*, *dwParam1*, *dwParam2*)

MidiOutFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an EXPORTS statement in the DLL's module-definition file.

Callback Parameters

HMIDIOUT *hMidiOut*
Specifies a handle to the MIDI device associated with the callback.

UINT *wMsg*
Specifies a MIDI output message.

DWORD *dwInstance*
Specifies the instance data supplied with **midiOutOpen**.

DWORD *dwParam1*
Specifies a parameter for the message.

DWORD *dwParam2*
Specifies a parameter for the message.

Comments

The device ID specified by *wDeviceID* varies from zero to one less than the number of devices present. Use midiOutGetNumDevs to determine the number of MIDI output devices present in the system.

If a window is chosen to receive callback information, the following messages are sent to the window procedure function to indicate the progress of MIDI output:

* MM_MOM_OPEN
* MM_MOM_CLOSE
* MM_MOM_DONE

If a function is chosen to receive callback information, the following messages are sent to the function to indicate the progress of MIDI output:

* MOM_OPEN
* MOM_CLOSE
* MOM_DONE

The callback function must reside in a DLL. You do not have to use **MakeProcInstance** to get a procedure-instance address for the callback function.

Because the callback is accessed at interrupt time, it must reside in a DLL and its code segment must be specified as FIXED in the module-definition file for the DLL. Any data that the callback accesses must be in a FIXED data segment as well. The callback may not make any system calls except for **PostMessage**, timeGetSystemTime, timeGetTime, timeSetEvent, timeKillEvent, midiOutShortMsg, midiOutLongMsg, and OutputDebugStr.

See Also

midiOutClose

midiOutPrepareHeader

Syntax

UINT **midiOutPrepareHeader**(*hMidiOut*, *lpMidiOutHdr*, *wSize*)

This function prepares a MIDI system-exclusive data block for output.

Parameters

HMIDIOUT *hMidiOut*

Specifies a handle to the MIDI output device.

LPMIDIHDR *lpMidiOutHdr*

Specifies a far pointer to a MIDIHDRstructure that identifies the data block to be prepared.

UINT *wSize*

Specifies the size of the MIDIHDR structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MMSYSERR_NOMEM

Unable to allocate or lock memory.

Comments

The MIDIHDR data structure and the data block pointed to by its lpData field must be allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags and locked with **GlobalLock**. Preparing a header that has already been prepared has no effect, and the function returns zero.

See Also

midiOutUnprepareHeader

midiOutReset

Syntax

UINT **midiOutReset**(*hMidiOut*)

This function turns off all notes on all MIDI channels for the specified MIDI output device. If there are any system-exclusive output buffers pending, they are marked as done and returned to the application.

Parameters

HMIDIOUT *hMidiOut*

Specifies a handle to the MIDI output device.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

Comments

To turn off all notes, a note-off message for each note for each channel is sent. In addition, the sustain controller is turned off for each channel.

See Also

midiOutLongMsg, midiOutClose

midiOutSetVolume

Syntax

UINT **midiOutSetVolume**(*wDeviceID*, *dwVolume*)

This function sets the volume of a MIDI output device.

Parameters

UINT wDeviceID

Identifies the MIDI output device.

DWORD *dwVolume*

Specifies the new volume setting. The low-order word contains the left channel volume setting, and the high-order word contains the right channel setting. A value of 0xFFFF represents full volume, and a value of 0x0000 is silence.

If a device does not support both left and right volume control, the low-order word of *dwVolume* specifies the volume level, and the high-order word is ignored.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MMSYSERR_NOTSUPPORTED

Function isn't supported.

MMSYSERR_NODRIVER

The driver was not installed.

Comments

Not all devices support volume changes. To determine whether the device supports volume control, use the MIDICAPS_VOLUME flag to test the dwSupport field of the MIDIOUTCAPSstructure (filled by midiOutGetDevCaps).

To determine whether the device supports volume control on both the left and right channels, use the MIDICAPS_LRVOLUME flag to test the dwSupport field of the MIDIOUTCAPSstructure (filled by midiOutGetDevCaps).

Most devices do not support the full 16 bits of volume level control and will use only the high-order bits of the requested volume setting. For example, for a device that supports 4 bits of volume control, requested volume level values of 0x4000, 0x4fff, and 0x43be will all produce the same physical volume setting, 0x4000. The midiOutGetVolume function will return the full 16-bit setting set with **midiOutSetVolume**.

Volume settings are interpreted logarithmically. This means the perceived increase in volume is the same when increasing the volume level from 0x5000 to 0x6000 as it is from 0x4000 to 0x5000.

See Also

midiOutGetVolume

midiOutShortMsg

Syntax

UINT **midiOutShortMsg**(*hMidiOut*, *dwMsg*)

This function sends a short MIDI message to the specified MIDI output device. Use this function to send MIDI messages other than system-exclusive messages.

Parameters

HMIDIOUT *hMidiOut*

Specifies a handle to the MIDI output device.

DWORD *dwMsg*

Specifies the MIDI message. The message is packed into a DWORD with the first byte of the message in the low-order byte.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MIDIERR_NOTREADY

The hardware is busy with other data.

Comments

A system-exclusive message can be started or completed using **midiOutShortMsg** by sending a start system exlcusive message (0x000000F0) or an end of system-exclusive message (0x000000F7); but, the system-exclusive data bytes must be sent with midiOutLongMsg.

MIDI status is maintained across consecutive calls to **midiOutShortMsg** and midiOutLongMsg; but,a **midiOutShortMsg** message must contain all data bytes for a MIDI event.

midiOutShortMsg supports, as recommended usage, the status byte associated with each MIDI message.

This function might not return until the message has been sent to the output device.

See Also

midiOutLongMsg

midiOutUnprepareHeader

Syntax

UINT **midiOutUnprepareHeader**(*hMidiOut*, *lpMidiOutHdr*, *wSize*)

This function cleans up the preparation performed by midiOutPrepareHeader. The **midiOutUnprepareHeader** function must be called after the device driver fills a data buffer and returns it to the application. You must call this function before freeing the data buffer.

Parameters

HMIDIOUT *hMidiOut*

Specifies a handle to the MIDI output device.

LPMIDIHDR *lpMidiOutHdr*

Specifies a pointer to a MIDIHDRstructure identifying the buffer to be cleaned up.

UINT *wSize*

Specifies the size of the MIDIHDR structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MIDIERR_STILLPLAYING

lpMidiOutHdr is still in the queue.

Comments

This function is the complementary function to midiOutPrepareHeader. You must call this function before freeing the data buffer with **GlobalFree**. After passing a buffer to the device driver with midiOutLongMsg, you must wait until the driver is finished with the buffer before calling **midiOutUnprepareHeader**.

Unpreparing a buffer that has not been prepared has no effect, and the function returns zero.

See Also

midiOutPrepareHeader

mmioAdvance

Syntax

UINT **mmioAdvance**(*hmmio*, *lpmmioinfo*, *wFlags*)

This function advances the I/O buffer of a file set up for direct I/O buffer access with mmioGetInfo. If the file is opened for reading, the I/O buffer is filled from the disk. If the file is opened for writing and the MMIO_DIRTY flag is set in the dwFlags field of the MMIOINFO structure, the buffer is written to disk. The pchNext, pchEndRead, and pchEndWrite fields of the MMIOINFO structure are updated to reflect the new state of the I/O buffer.

Parameters

HMMIO *hmmio*

Specifies the file handle for a file opened with mmioOpen.

LPMMIOINFO *lpmmioinfo*

Specifies a far pointer to the MMIOINFO structure obtained with mmioGetInfo.

UINT *wFlags*

Specifies options for the operation. Contains exactly one of the following two flags:

MMIO_READ

The buffer is filled from the file.

MMIO_WRITE

The buffer is written to the file.

Return Value

The return value is zero if the operation is successful. Otherwise, the return value specifies an error code. The error code can be one of the following codes:

MMIOERR_CANNOTWRITE

The contents of the buffer could not be written to disk.

MMIOERR_CANNOTREAD

An error occurred while re-filling the buffer.

MMIOERR_UNBUFFERED

The specified file is not opened for buffered I/O.

MMIOERR_CANNOTEXPAND

The specified memory file cannot be expanded, probably because the adwInfo[0] field was set to zero in the initial call to mmioOpen.

MMIOERR_OUTOFMEMORY

There was not enough memory to expand a memory file for further writing.

Comments

If the specified file is opened for writing or for both reading and writing, the I/O buffer will be flushed to disk before the next buffer is read. If the I/O buffer cannot be written to disk because the disk is full, then **mmioAdvance** will return MMIOERR_CANNOTWRITE.

If the specified file is only open for writing, the MMIO_WRITE flag must be specified.

If you have written to the I/O buffer, you must set the MMIO_DIRTY flag in the dwFlags field of the MMIOINFO structure before calling **mmioAdvance**. Otherwise, the buffer will not be written to disk.

If the end of file is reached, **mmioAdvance** will still return success, even though no more data can be read. Thus, to check for the end of the file, it is necessary to see if the pchNext and pchEndRead

fields of the MMIOINFO structure are equal after calling **mmioAdvance**.

See Also

mmioGetInfo, MMIOINFO

mmioAscend

Syntax

UINT **mmioAscend**(*hmmio*, *lpck*, *wFlags*)

This function ascends out of a chunk in a RIFF file descended into with mmioDescend or created with mmioCreateChunk.

Parameters

HMMIO *hmmio*

Specifies the file handle of an open RIFF file.

LPMMCKINFO *lpck*

Specifies a far pointer to a caller-supplied MMCKINFO structure previously filled by mmioDescend or mmioCreateChunk.

UINT *wFlags*

Is not used and should be set to zero.

Return Value

The return value is zero if the function is successful. Otherwise, the return value specifies an error code. The error code can be one of the following codes:

MMIOERR_CANNOTWRITE

The contents of the buffer could not be written to disk.

MMIOERR_CANNOTSEEK

There was an error while seeking to the end of the chunk.

Comments

If the chunk was descended into using mmioDescend, then **mmioAscend** seeks to the location following the end of the chunk (past the extra pad byte, if any).

If the chunk was created and descended into using mmioCreateChunk, or if the MMIO_DIRTY flag is set in the dwFlags field of the MMCKINFO structure referenced by *lpck*, then the current file position is assumed to be the end of the data portion of the chunk. If the chunk size is not the same as the value stored in the cksize field when mmioCreateChunkwas called, then **mmioAscend** corrects the chunk size in the file before ascending from the chunk. If the chunk size is odd, **mmioAscend** writes a null pad byte at the end of the chunk. After ascending from the chunk, the current file position is the location following the end of the chunk (past the extra pad byte, if any).

See Also

mmioDescend, mmioCreateChunk, MMCKINFO

mmioClose

Syntax

UINT **mmioClose**(*hmmio*, *wFlags*)

This function closes a file opened with mmioOpen.

Parameters

HMMIO *hmmio*

Specifies the file handle of the file to close.

UINT *wFlags*

Specifies options for the close operation.

MMIO_FHOPEN

If the file was opened by passing the MS-DOS file handle of an already-opened file to mmioOpen, then using this flag tells **mmioClose** to close the MMIO file handle, but not the MS-DOS file handle.

Return Value

The return value is zero if the function is successful. Otherwise, the return value is an error code, either from mmioFlush or from the I/O procedure. The error code can be one of the following codes:

MMIOERR_CANNOTWRITE

The contents of the buffer could not be written to disk.

See Also

mmioOpen, mmioFlush

mmioCreateChunk

Syntax

UINT **mmioCreateChunk**(*hmmio*, *lpck*, *wFlags*)

This function creates a chunk in a RIFF file opened with mmioOpen. The new chunk is created at the current file position. After the new chunk is created, the current file position is the beginning of the data portion of the new chunk.

Parameters

HMMIO *hmmio*

Specifies the file handle of an open RIFF file.

LPMMCKINFO *lpck*

Specifies a pointer to a caller-supplied MMCKINFO structure containing information about the chunk to be created. The MMCKINFO structure should be set up as follows:

* The ckid field specifies the chunk ID of the chunk. If *wFlags* includes MMIO_CREATERIFF or MMIO_CREATELIST, this field will be filled by **mmioCreateChunk**.
* The cksize field specifies the size of the data portion of the chunk, including the form type or list type (if any). If this value is not correct when mmioAscend is called to mark the end of the chunk, them mmioAscend will correct the chunk size.
* The fccType field specifies the form type or list type if the chunk is a "RIFF" or "LIST" chunk. If the chunk is not a "RIFF" or "LIST" chunk, this field need not be filled in.
* The dwDataOffset field need not be filled in. The **mmioCreateChunk** function will fill this field with the file offset of the data portion of the chunk.
* The dwFlags field need not be filled in. The **mmioCreateChunk** function will set the MMIO_DIRTY flag in dwFlags.

UINT *wFlags*

Specifies flags to optionally create either a "RIFF" chunk or a "LIST" chunk. Can contain one of the following flags:

MMIO_CREATERIFF

Creates a "RIFF" chunk.

MMIO_CREATELIST

Creates a "LIST" chunk.

Return Value

The return value is zero if the function is successful. Otherwise, the return value specifies an error code. The error code can be one of the following codes:

MMIOERR_CANNOTWRITE

Unable to write the chunk header.

MMIOERR_CANNOTSEEK

Uanble to determine offset of data portion of the chunk.

Comments

This function cannot insert a chunk into the middle of a file. If a chunk is created anywhere but the end of a file, **mmioCreateChunk** will overwrite existing information in the file.

mmioDescend

Syntax

UINT **mmioDescend**(*hmmio*, *lpck*, *lpckParent*, *wFlags*)

This function descends into a chunk of a RIFF file opened with mmioOpen. It can also search for a given chunk.

Parameters

HMMIO *hmmio*

Specifies the file handle of an open RIFF file.

LPMMCKINFO *lpck*

Specifies a far pointer to a caller-supplied MMCKINFO structure that **mmioDescend** fills with the following information:

* The ckid field is the chunk ID of the chunk.
* The cksize field is the size of the data portion of the chunk. The data size includes the form type or list type (if any), but does not include the 8-byte chunk header or the pad byte at the end of the data (if any).
* The fccType field is the form type if ckid is "RIFF", or the list type if ckid is "LIST". Otherwise, it is NULL.
* The dwDataOffset field is the file offset of the beginning of the data portion of the chunk. If the chunk is a "RIFF" chunk or a "LIST" chunk, then dwDataOffsetis the offset of the form type or list type.
* The dwFlags contains other information about the chunk. Currently, this information is not used and is set to zero.

  If the MMIO_FINDCHUNK, MMIO_FINDRIFF, or MMIO_FINDLIST flag is specified for *wFlags*, then the MMCKINFO structure is also used to pass parameters to **mmioDescend**:
* The ckid field specifies the four-character code of the chunk ID, form type, or list type to search for.

LPMMCKINFO *lpckParent*

Specifies a far pointer to an optional caller-supplied MMCKINFO structure identifying the parent of the chunk being searched for. A parent of a chunk is the enclosing chunk--only "RIFF" and "LIST"chunks can be parents. If *lpckParent* is not NULL, then **mmioDescend** assumes the MMCKINFO structure it refers to was filled when **mmioDescend** was called to descend into the parent chunk, and **mmioDescend** will only search for a chunk within the parent chunk. Set *lpckParent* to NULL if no parent chunk is being specified.

UINT *wFlags*

Specifies search options. Contains up to one of the following flags. If no flags are specified, **mmioDescend** descends into the chunk beginning at the current file position.

MMIO_FINDCHUNK

Searches for a chunk with the specified chunk ID.

MMIO_FINDRIFF

Searches for a chunk with chunk ID "RIFF"and with the specified form type.

MMIO_FINDLIST

Searches for a chunk with chunk ID "LIST"and with the specified form type.

Return Value

The return value is zero if the function is successful. Otherwise, the return value specifies an error code. If the end of the file (or the end of the parent chunk, if given) is reached before the desired chunk is found, the return value is MMIOERR_CHUNKNOTFOUND.

Comments

A RIFF chunk consists of a four-byte chunk ID (type FOURCC), followed by a four-byte chunk size (type DWORD), followed by the data portion of the chunk, followed by a null pad byte if the size of the data portion is odd. If the chunk ID is "RIFF" or "LIST", the first four bytes of the data portion of the chunk are a form type or list type (type FOURCC).

If **mmioDescend** is used to search for a chunk, the file position should be at the beginning of a chunk before calling **mmioDescend**. The search begins at the current file position and continues to the end of the file. If a parent chunk is specified, the file position should be somewhere within the parent chunk before calling **mmioDescend**. In this case, the search begins at the current file position and continues to the end of the parent chunk.

If **mmioDescend** is unsuccessful in searching for a chunk, the current file position is undefined. If **mmioDescend** is successful, the current file position is changed. If the chunk is a "RIFF" or "LIST" chunk, the new file position will be just after the form type or list type (12 bytes from the beginning of the chunk). For other chunks, the new file position will be the start of the data portion of the chunk (8 bytes from the beginning of the chunk).

For efficient RIFF file I/O, use buffered I/O.

See Also

mmioAscend, MMCKINFO

mmioFlush

Syntax

UINT **mmioFlush**(*hmmio*, *wFlags*)

This function writes the I/O buffer of a file to disk, if the I/O buffer has been written to.

Parameters

HMMIO *hmmio*

Specifies the file handle of a file opened with mmioOpen.

UINT *wFlags*

Is not used and should be set to zero.

Return Value

The return value is zero if the function is successful. Otherwise, the return value specifies an error code. The error code can be one of the following codes:

MMIOERR_CANNOTWRITE

The contents of the buffer could not be written to disk.

Comments

Closing a file with mmioClose will automatically flush its buffer.

If there is insufficient disk space to write the buffer, **mmioFlush** will fail, even if the preceding mmioWritecalls were successful.

mmioFOURCC

Syntax

FOURCC **mmioFOURCC**(*ch0*, *ch1*, *ch2*, *ch3*)

This macro converts four characters to to a four-character code.

Parameters

CHAR *ch0*

The first character of the four-character code.

CHAR *ch1*

The second character of the four-character code.

CHAR *ch2*

The third character of the four-character code.

CHAR *ch3*

The fourth character of the four-character code.

Return Value

The return value is the four-character code created from the given characters.

Comments

This macro does not check to see if the four character code follows any conventions regarding which characters to include in a four-character code.

See Also

mmioStringToFOURCC

mmioGetInfo

Syntax

UINT **mmioGetInfo**(*hmmio*, *lpmmioinfo*, *wFlags*)

This function retrieves information about a file opened with mmioOpen. This information allows the caller to directly access the I/O buffer, if the file is opened for buffered I/O.

Parameters

HMMIO *hmmio*

Specifies the file handle of the file.

LPMMIOINFO *lpmmioinfo*

Specifies a far pointer to a caller-allocated MMIOINFO structure that **mmioGetInfo** fills with information about the file. See the MMIOINFO structure and the mmioOpen function for information about the fields in this structure.

UINT *wFlags*

Is not used and should be set to zero.

Return Value

The return value is zero if the function is successful.

Comments

To directly access the I/O buffer of a file opened for buffered I/O, use the following fields of the MMIOINFO structure filled by **mmioGetInfo**:

* The pchNext field points to the next byte in the buffer that can be read or written. When you read or write, increment pchNext by the number of bytes read or written.
* The pchEndRead field points to one byte past the last valid byte in the buffer that can be read.
* The pchEndWrite field points to one byte past the last location in the buffer that can be written.

Once you read or write to the buffer and modify pchNext, do not call any MMIO function except mmioAdvance until you call mmioSetInfo. Call mmioSetInfowhen you are finished directly accessing the buffer.

When you reach the end of the buffer specified by pchEndRead or pchEndWrite, call mmioAdvance to fill the buffer from the disk, or write the buffer to the disk. The mmioAdvance function will update the pchNext, pchEndRead, and pchEndWrite fields in the MMIOINFO structure for the file.

Before calling mmioAdvance or mmioSetInfo to flush a buffer to disk, set the MMIO_DIRTY flag in the dwFlagsfield of the MMIOINFO structure for the file. Otherwise, the buffer will not get written to disk.

Do not decrement pchNext or modify any fields in the MMIOINFO structure other than pchNext and dwFlags. Do not set any flags in dwFlagsexcept MMIO_DIRTY.

See Also

mmioSetInfo, MMIOINFO

mmioInstallIOProc

Syntax

LPMMIOPROC **mmioInstallIOProc**(*fccIOProc*, *pIOProc*, *dwFlags*)

This function installs or removes a custom I/O procedure. It will also locate an installed I/O procedure, given its corresponding four-character code.

Parameters

FOURCC *fccIOProc*

Specifies a four-character code identifying the I/O procedure to install, remove, or locate. All characters in this four-character code should be uppercase characters.

LPMMIOPROC *pIOProc*

Specifies the address of the I/O procedure to install. To remove or locate an I/O procedure, set this parameter to NULL.

DWORD *dwFlags*

Specifies one of the following flags indicating whether the I/O procedure is being installed, removed, or located:

MMIO_INSTALLPROC

Installs the specified I/O procedure. To allow other procedures to use the specified I/O procedure, also specify the MMIO_GLOBALPROC flag.

MMIO_REMOVEPROC

Removes the specified I/O procedure. When removing a global I/O procedure, only the task that refisters a blobal I/O procedure can unregister that procedure.

MMIO_FINDPROC

Searches local, then global procedures for the specified I/O procedure.

MMIO_GLOBALPROC

Identifies the I/O procedure being installed as a global procedure.

Return Value

The return value is the address of the I/O procedure installed, removed, or located. If there is an error, the return value is NULL.

Callback

LRESULT FAR PASCAL **IOProc**(*lpmmioinfo*, *wMsg*, *lParam1*, *lParam2*)

IOProc is a placeholder for the application-supplied function name. The actual name must be exported by including it in a EXPORTS statement in the application's module-definitions file.

Callback Parameters

LPSTR *lpmmioinfo*

Specifies a far pointer to an MMIOINFO structure containing information about the open file. The I/O procedure must maintain the lDiskOffsetfield in this structure to indicate the file offset to the next read or write location. The I/O procedure can use the adwInfo[] field to store state information. The I/O procedure should not modify any other fields of the MMIOINFO structure.

UINT *wMsg*

Specifies a message indicating the requested I/O operation. Messages that can be received include MMIOM_OPEN, MMIOM_CLOSE, MMIOM_READ, MMIOM_WRITE, and MMIOM_SEEK.

LPARAM *lParam1*

Specifies a parameter for the message.

LPARAM *lParam2*
Specifies a parameter for the message.

Callback Return Value

The return value depends on the message specified by *wMsg*. If the I/O procedure does not recognize a message, it should return zero.

Comments

If the I/O procedure resides in the application, use **MakeProcInstance** to get a procedure-instance address and specify this address for *pIOProc*. You don't need to get a procedure-instance address if the I/O procedure resides in a DLL.

The four-character code specified by the fccIOProc field in the MMIOINFO structure associated with a file identifies a filename extension for a custom storage system. When an application calls mmioOpen with a filename such as "FNAME.XYZ!boo", the I/O procedure associated with the four-character code "XYZ " is called to open the "boo" element of the file FNAME.XYZ.

The **mmioInstallIOProc** function maintains a separate list of installed I/O procedures for each Windows application. Therefore, different applications can use the same I/O procedure identifier for different I/O procedures without conflict.

To share an I/O procedure among applications, each application can install and use local copies of the I/O procedure or one application can install a blobal copy of the I/O procedure for one or more applications to use. To use multiple, local copies of an I/O procedure among several applications, the I/O procedure must reside in a DLL called by each application using it. Each application using the shared I/O procedure must call **mmioInstallIOProc** to install the procedure (or call the DLL to install the procedure on behalf of the application). Each application must call **mmioInstallIOProc** to remove the I/O procedure before terminating.

If an application calls **mmioInstallIOProc** more than once to register the same I/O procedure, then it must call **mmioInstallIOProc** to remove the procedure once for each time it installed the procedure.

mmioInstallIOProc will not prevent an application from installing two different I/O procedures with the same identifier, or installing an I/O procedure with one of the predefined identifiers ("DOS ", "MEM ", or "BND "). The most recently installed procedure takes precedence, and the most recently installed procedure is the first one to get removed.

To use a single copy of an I/O procedure among several applications, one application must install the I/O procedure as a global procedure. The other applications locate the global procedure before they use it. An application that installs a global I/O procedure can, without regard to other applications using the procedure, unregister that procedure at any time.

An application installs a global copy of an I/O procedure by calling **mmioInstallIOProc** with the flags MMIO_INSTALLPROC and MMIO_GLOBALPROC. Once an application globally installs a procedure, that application can use the global procedure. To unregister a procedure, the application that installed the procedure must call **mmioInstallIOProc.**

Other applications must locate an installed, global I/O procedure before using it. To locate a global procedure, an application calls **mmioInstallIOProc** with the flag MMIO_FINDPROC. Once an application locates the global procedure, it can call the procedure as needed. Applications that use, but do not install, a global I/O procedure, are exempt from actions to unregister that procedure.

See Also

mmioOpen

mmioOpen

Syntax

HMMIO **mmioOpen**(*szFilename*, *lpmmioinfo*, *dwOpenFlags*)

This function opens a file for unbuffered or buffered I/O. The file can be a MS-DOS file, a memory file, or an element of a custom storage system.

Parameters

LPSTR szFilename

Specifies a far pointer to a string containing the filename of the file to open. If no I/O procedure is specified to open the file, then the filename determines how the file is opened, as follows:

* If the filename does not contain "+", then it is assumed to be the name of a MS-DOS file.
* If the filename is of the form "FNAME.EXT+boo", then the extension "EXT " is assumed to identify an installed I/O procedure which is called to perform I/O on the file (see mmioInstallIOProc).
* If the filename is NULL and no I/O procedure is given, then adwInfo[0] is assumed to be the MS-DOS file handle of a currently open file.

The MS-DOS filename should not be longer than 128 bytes, including the terminating NULL.

When opening a memory file, set *szFilename* to NULL.

LPMMIOINFO *lpmmioinfo*

Specifies a far pointer to an MMIOINFO structure containing extra parameters used by **mmioOpen**. Unless you are opening a memory file, specifying the size of a buffer for buffered I/O, or specifying an uninstalled I/O procedure to open a file, this parameter should be NULL.

If *lpmmioinfo* is not NULL, all unused fields of the MMIOINFO structure it references must be set to zero, including the reserved fields.

DWORD *dwOpenFlags*

Specifies option flags for the open operation. The MMIO_READ, MMIO_WRITE, and MMIO_READWRITE flags are mutually exclusive--only one should be specified. The MMIO_COMPAT, MMIO_EXCLUSIVE, MMIO_DENYWRITE, MMIO_DENYREAD, and MMIO_DENYNONE flags are MS-DOS file-sharing flags, and can only be used after the MS-DOS command SHARE has been executed.

MMIO_READ

Opens the file for reading only. This is the default, if MMIO_WRITE and MMIO_READWRITE are not specified.

MMIO_WRITE

Opens the file for writing. You should not read from a file opened in this mode.

MMIO_READWRITE

Opens the file for both reading and writing.

MMIO_CREATE

Creates a new file. If the file already exists, it is truncated to zero length. For memory files, MMIO_CREATE indicates the end of the file is initially at the start of the buffer.

MMIO_DELETE

Deletes a file. If this flag is specified, *szFilename* should not be NULL. The return value will be TRUE (cast to HMMIO) if the file was deleted successfully, FALSE otherwise. Do not call mmioClose for a file that has been deleted. If this flag is specified, all other flags are ignored.

MMIO_PARSE

Creates a fully qualified filename from the path specified in *szFileName.* The fully qualified filename is placed back into *szFileName*. The return value will be TRUE (cast to HMMIO) if the qualification was successful, FALSE otherwise. The file is not opened, and the function does not return a valid MMIO file handle, so do not attempt to close the file. If this flag is specified, all other file opening flags are ignored.

MMIO_EXIST

Determines whether the specified file exists and creates a fully qualified filename from the path specified in *szFileName*. The fully qualified filename is placed back into *szFileName*. The return value will be TRUE (cast to HMMIO) if the qualification was successful and the file exists, FALSE otherwise. The file is not opened, and the function does not return a valid MMIO file handle, so do not attempt to close the file.

MMIO_ALLOCBUF

Opens a file for buffered I/O. To allocate a buffer larger or smaller than the default buffer size (8K), set the cchBuffer field of the MMIOINFO structure to the desired buffer size. If cchBuffer is zero, then the default buffer size is used. If you are providing your own I/O buffer, then the MMIO_ALLOCBUF flag should not be used.

MMIO_COMPAT

Opens the file with compatibility mode, allowing any process on a given machine to open the file any number of times. **mmioOpen** fails if the file has been opened with any of the other sharing modes.

MMIO_EXCLUSIVE

Opens the file with exclusive mode, denying other processes both read and write access to the file. **mmioOpen** fails if the file has been opened in any other mode for read or write access, even by the current process.

MMIO_DENYWRITE

Opens the file and denies other processes write access to the file. **mmioOpen** fails if the file has been opened in compatibility or for write access by any other process.

MMIO_DENYREAD

Opens the file and denies other processes read access to the file. **mmioOpen** fails if the file has been opened in compatibility mode or for read access by any other process.

MMIO_DENYNONE

Opens the file without denying other processes read or write access to the file. **mmioOpen** fails if the file has been opened in compatibility mode by any other process.

MMIO_GETTEMP

Creates a temporary filename, optionally using the parameters passed in *szFileName* to determine the temporary name. For example, you can specify "C:F" to create a filename for a temporary file residing on drive C, with the filename starting with letter F. The resulting filename is placed in the buffer pointed to by *szFileName*. The return value will be TRUE (cast to HMMIO) if the temporary filename was created successfully, FALSE otherwise. The file is not opened, and the function does not return a valid MMIO file handle, so do not attempt to close the file. This flag overrides all other flags.

Return Value

The return value is a handle to the opened file. This handle is not a MS-DOS file handle--do not use it with any file I/O functions other than MMIO functions.

If the file cannot be opened, the return value is NULL. If *lpmmioinfo* is not NULL, then its wErrorRet field will contain extended error information returned by the I/O procedure.

Comments

If *lpmmioinfo* references an MMIOINFO structure, set up the fields as described below. All unused fields must be set to zero, including reserved fields.

* To request that a file be opened with an installed I/O procedure, set the fccIOProc field to the four-character code of the I/O procedure, and set the pIOProc field to NULL.

* To request that a file be opened with an uninstalled I/O procedure, set the pIOProc field to point to the I/O procedure, and set fccIOProc to NULL.

* To request that **mmioOpen** determine which I/O procedure to use to open the file based on the filename contained in *szFilename*, set both fccIOProc and pIOProc to NULL. This is the default behavior if no MMIOINFO structure is specified.

* To open a memory file using an internally allocated and managed buffer, set the pchBuffer field to NULL, fccIOProc to FOURCC_MEM, cchBuffer to the initial size of the buffer, and adwInfo[0] to the incremental expansion size of the buffer. This memory file will automatically be expanded in increments of adwInfo[0] bytes when necessary. Specify the MMIO_CREATE flag for the *dwOpenFlags* parameter to initially set the end of the file to be the beginning of the buffer.

* To open a memory file using a caller-supplied buffer, set the pchBuffer field to point to the memory buffer, fccIOProc to FOURCC_MEM, cchBuffer to the size of the buffer, and adwInfo[0] to the incremental expansion size of the buffer. The expansion size in adwInfo[0] should only be non-zero if pchBuffer is a pointer obtained by calling **GlobalAlloc** and **GlobalLock**, since **GlobalReAlloc** will be called to expand the buffer. In particular, if pchBuffer points to a local or global array, a block of memory in the local heap, or a block of memory allocated by **GlobalDosAlloc**, adwInfo[0] must be zero.

  Specify the MMIO_CREATE flag for the *dwOpenFlags* parameter to initially set the end of the file to be the beginning of the buffer;otherwise, the entire block of memory will be considered readable.

* To use a currently open MS-DOS file handle with MMIO, set the fccIOProc field to FOURCC_DOS, pchBuffer to NULL, and adwInfo[0] to the MS-DOS file handle. Note that offsets within the file will be relative to the beginning of the file, and will not depend on the MS-DOS file position at the time **mmioOpen** is called; the initial MMIO offset will be the same as the MS-DOS offset when **mmioOpen** is called. Later, to close the MMIO file handle without closing the MS-DOS file handle, pass the MMIO_FHOPEN flag to mmioClose.

You must call mmioClose to close a file opened with **mmioOpen**. Open files are not automatically closed when an application exits.

See Also

mmioClose

mmioRead

Syntax

LONG **mmioRead**(*hmmio*, *pch*, *cch*)

This function reads a specified number of bytes from a file opened with mmioOpen.

Parameters

HMMIO *hmmio*

Specifies the file handle of the file to be read.

HPSTR *pch*

Specifies a huge pointer to a buffer to contain the data read from the file.

LONG *cch*

Specifies the number of bytes to read from the file.

Return Value

The return value is the number of bytes actually read. If the end of the file has been reached and no more bytes can be read, the return value is zero. If there is an error reading from the file, the return value is -1.

See Also

mmioWrite

mmioRename

Syntax

UINT **mmioRename**(*szFilename*, *szNewFileName*, *lpmmioinfo, dwRenameFlags*)

This function renames the specified file.

Parameters

LPCSTR szFilename

Specifies a far pointer to a string containing the filename of the file to rename.

LPCSTR *szNewFileName*

Specifies a far pointer to a string containing the new filename.

LPMMIOINFO *lpmmioinfo*

Specifies a far pointer to an MMIOINFO structure containing extra parameters used by **mmioRename**.

If *lpmmioinfo* is not NULL, all unused fields of the MMIOINFO structure it references must be set to zero, including the reserved fields.

DWORD\b *dwRenameFlags*

Specifies option flags for the rename operation. This should be set to zero.

Return Value

The return value is zero if the file was renamed. Otherwise, the return value is an error code returned from **mmioRename** or from the I/O procedure.

mmioSeek

Syntax

LONG **mmioSeek**(*hmmio*, *lOffset*, *iOrigin*)

This function changes the current file position in a file opened with mmioOpen. The current file position is the location in the file where data is read or written.

Parameters

HMMIO *hmmio*

Specifies the file handle of the file to seek in.

LONG *lOffset*

Specifies an offset to change the file position.

int *iOrigin*

Specifies how the offset specified by *lOffset* is interpreted. Contains one of the following flags:

SEEK_SET

Seeks to *lOffset* bytes from the beginning of the file.

SEEK_CUR

Seeks to *lOffset* bytes from the current file position.

SEEK_END

Seeks to *lOffset* bytes from the end of the file.

Return Value

The return value is the new file position in bytes, relative to the beginning of the file. If there is an error, the return value is -1.

Comments

Seeking to an invalid location in the file, such as past the end of the file, may not cause **mmioSeek** to return an error, but may cause subsequent I/O operations on the file to fail.

To locate the end of a file, call **mmioSeek** with *lOffset*set to zero and *iOrigin* set to SEEK_END.

mmioSendMessage

Syntax

LRESULT **mmioSendMessage**(*hmmio*, *wMsg*, *lParam1*, *lParam2*)

This function sends a message to the I/O procedure associated with the specified file.

Parameters

HMMIO *hmmio*
Specifies the file handle for a file opened with mmioOpen.

UINT *wMsg*
Specifies the message to send to the I/O procedure.

LPARAM *lParam1*
Specifies a parameter for the message.

LPARAM *lParam2*
Specifies a parameter for the message.

Return Value

The return value depends on the message. If the I/O procedure does not recognize the message, the return value is zero.

Comments

Use this function to send custom user-defined messages. Do not use it to send the MMIOM_OPEN, MMIOM_CLOSE, MMIOM_READ, MMIOM_WRITE, MMIOM_WRITEFLUSH, or MMIOM_SEEK messages. Define custom messages to be greater than or equal to the MMIOM_USER constant.

See Also

mmioInstallIOProc

mmioSetBuffer

Syntax

UINT **mmioSetBuffer**(*hmmio, pchBuffer, cchBuffer, wFlags*)

This function enables or disables buffered I/O, or changes the buffer or buffer size for a file opened with mmioOpen.

Parameters

HMMIO *hmmio*

Specifies the file handle of the file.

LPSTR *pchBuffer*

Specifies a far pointer to a caller-supplied buffer to use for buffered I/O. If NULL, **mmioSetBuffer** allocates an internal buffer for buffered I/O.

LONG *cchBuffer*

Specifies the size of the caller-supplied buffer, or the size of the buffer for **mmioSetBuffer** to allocate.

UINT *wFlags*

Is not used and should be set to zero.

Return Value

The return value is zero if the function is successful. Otherwise, the return value specifies an error code. If an error occurs, the file handle remains valid. The error code can be one of the following codes:

MMIOERR_CANNOTWRITE

The contents of the old buffer could not be written to disk, so the operation was aborted.

MMIOERR_OUTOFMEMORY

The new buffer could not be allocated, probably due to a lack of available memory.

Comments

To enable buffering using an internal buffer, set *pchBuffer* to NULL and *cchBuffer* to the desired buffer size.

To supply your own buffer, set *pchBuffer* to point to the buffer, and set *cchBuffer* to the size of the buffer.

To disable buffered I/O, set *pchBuffer* to NULL and *cchBuffer* to zero.

If buffered I/O is already enabled using an internal buffer, you can reallocate the buffer to a different size by setting *pchBuffer* to NULL and *cchBuffer* to the new buffer size. The contents of the buffer may be changed after resizing.

mmioSetInfo

Syntax

UINT **mmioSetInfo**(*hmmio*, *lpmmioinfo*, *wFlags*)

This function updates the information retrieved by mmioGetInfo about a file opened with mmioOpen. Use this function to terminate direct buffer access of a file opened for buffered I/O.

Parameters

HMMIO *hmmio*

Specifies the file handle of the file.

LPMMIOINFO *lpmmioinfo*

Specifies a far pointer to an MMIOINFO structure filled with information with mmioGetInfo.

UINT *wFlags*

Is not used and should be set to zero.

Return Value

The return value is zero if the function is successful.

Comments

If you have written to the file I/O buffer, set the MMIO_DIRTY flag in the dwFlags field of the MMIOINFOstructure before calling **mmioSetInfo** to terminate direct buffer access. Otherwise, the buffer will not get flushed to disk.

See Also

mmioGetInfo, MMIOINFO

mmioStringToFOURCC

Syntax

FOURCC **mmioStringToFOURCC**(*sz*, *wFlags*)

This function converts a null-terminated string to a four-character code.

Parameters

LPCSTR *sz*

Specifies a far pointer to a null-terminated string to a four-character code.

UINT *wFlags*

Specifies options for the conversion:

MMIO_TOUPPER

Converts all characters to uppercase.

Return Value

The return value is the four character code created from the given string.

Comments

This function does not check to see if the string referenced by *sz* follows any conventions regarding which characters to include in a four-character code. The string is simply copied to a four-character code, padding with blanks to the right if required, and truncated to four characters if required.

See Also

mmioFOURCC

mmioWrite

Syntax

LONG **mmioWrite**(*hmmio*, *pch*, *cch*)

This function writes a specified number of bytes to a file opened with mmioOpen.

Parameters

HMMIO *hmmio*

Specifies the file handle of the file.

HPSTR *pch*

Specifies a huge pointer to the buffer to be written to the file.

LONG *cch*

Specifies the number of bytes to write to the file.

Return Value

The return value is the number of bytes actually written. If there is an error writing to the file, the return value is -1.

Comments

The current file position is incremented by the number of bytes written.

See Also

mmioRead

mmsystemGetVersion

Syntax

UINT **mmsystemGetVersion**()

This function returns the current version number of MMSYSTEM.DLL.

Parameters

None.

Return Value

The return value specifies the major and minor version numbers of MMSYSTEM.DLL. The high-order byte specifies the major version number. The low-order byte specifies the minor version number.

OutputDebugStr

Syntax

void **OutputDebugStr**(*lpOutputString*)

This function sends a debugging message directly to the COM1 port or to a secondary monochrome display adapter. Because it bypasses MS-DOS, it can be called by low-level callback functions and other code at interrupt time.

Parameters

LPCSTR lpOutputString

Specifies a far pointer to a null-terminated string.

Comments

This function is available only in the debugging version of Windows. The DebugOutput keyname in the [mmsystem]section of SYSTEM.INI controls where the debugging information is sent. If DebugOutput is 0, all debug output is disabled. If DebugOutput is 1, debug output is sent to the COM1 port. If DebugOutput is 2, debug output is sent to a secondary monochrome display adapter.

To print the contents of a register, use the pound sign ("#") followed by one of the following register designations: "ax", "bx", "cx", "dx", "si", "di", "bp", "sp", "al", "bl", "cl", "dl". For systems that support the 80386 architecture, OutputDebugStr also supports the following register designations: "fs", "gs","edi","esi","eax","ebx","ecx","edx".

For example, to print the stack pointer and accumulator registers, pass the following string to **OutputDebugStr**: "SP=#sp\r\nAX=#ax\r\n".

sndPlaySound

Syntax

BOOL **sndPlaySound**(*lpszSoundName*, *wFlags*)

This function plays a waveform sound specified by a filename or by an entry in the [sounds] section of WIN.INI. If the sound can't be found, it plays the default sound specified by the SystemDefault entry in the [sounds] section of WIN.INI. If there is no SystemDefault entry or if the default sound can't be found, the function makes no sound and returns FALSE.

Parameters

LPCSTR lpszSoundName

Specifies the name of the sound to play. The function searches the [sounds] section of WIN.INI for an entry with this name and plays the associated waveform file. If no entry by this name exists, then it assumes the name is the name of a waveform file. If this parameter is NULL, any currently playing sound is stopped.

UINT *wFlags*

Specifies options for playing the sound using one or more of the following flags:

SND_SYNC

The sound is played synchronously and the function does not return until the sound ends.

SND_ASYNC

The sound is played asynchronously and the function returns immediately after beginning the sound. To terminate an asynchronously-played sound, call **sndPlaySound** with *lpszSoundName* set to NULL.

SND_NODEFAULT

If the sound can't be found, the function returns silently without playing the default sound.

SND_MEMORY

The parameter specified by *lpszSoundName*points to an in-memory image of a waveform sound.

SND_LOOP

The sound will continue to play repeatedly until **sndPlaySound** is called again with the *lpszSoundName* parameter set to NULL. You must also specify the SND_ASYNC flag to loop sounds.

SND_NOSTOP

If a sound is currently playing, the function will immediately return FALSE without playing the requested sound.

Return Value

Returns TRUE if the sound is played, otherwise returns FALSE.

Comments

The sound must fit in available physical memory and be playable by an installed waveform audio device driver. The directories searched for sound files are, in order: the current directory;the Windows directory; the Windows system directory; the directories listed in the PATH environment variable; the list of directories mapped in a network. See the Windows **OpenFile** function for more information about the directory search order.

If you specify the SND_MEMORY flag, *lpszSoundName* must point to an in-memory image of a waveform sound. If the sound is stored as a resource, use **LoadResource** and **LockResource** to load and lock the resource and get a pointer to it. If the sound is not a resource, you must use **GlobalAlloc** with the GMEM_MOVEABLE and GMEM_SHARE flags set and then **GlobalLock** to

allocate and lock memory for the sound.

timeBeginPeriod

Syntax

UINT **timeBeginPeriod**(*wPeriod*)

This function sets the minimum (lowest number of milliseconds) timer resolution that an application or driver is going to use. Call this function immediately before starting to use timer-event services, and call timeEndPeriod immediately after finishing with the timer-event services.

Parameters

UINT *wPeriod*

Specifies the minimum timer-event resolution that the application or driver will use.

Return Value

Returns zero if successful. Returns TIMERR_NOCANDO if the specified *wPeriod* resolution value is out of range.

Comments

For each call to **timeBeginPeriod**, you must call timeEndPeriod with a matching *wPeriod* value. An application or driver can make multiple calls to **timeBeginPeriod**, as long as each **timeBeginPeriod** call is matched with a timeEndPeriod call.

See Also

timeEndPeriod, timeSetEvent

timeEndPeriod

Syntax

UINT **timeEndPeriod**(*wPeriod*)

This function clears a previously set minimum (lowest number of milliseconds) timer resolution that an application or driver is going to use. Call this function immediately after using timer event services.

Parameters

UINT *wPeriod*

Specifies the minimum timer-event resolution value specified in the previous call to timeBeginPeriod.

Return Value

Returns zero if successful. Returns TIMERR_NOCANDO if the specified *wPeriod* resolution value is out of range.

Comments

For each call to timeBeginPeriod, you must call **timeEndPeriod** with a matching *wPeriod* value. An application or driver can make multiple calls to timeBeginPeriod, as long as each timeBeginPeriod call is matched with a **timeEndPeriod** call.

See Also

timeBeginPeriod, timeSetEvent

timeGetDevCaps

Syntax

UINT **timeGetDevCaps**(*lpTimeCaps*, *wSize*)

This function queries the timer device to determine its capabilities.

Parameters

LPTIMECAPS *lpTimeCaps*

Specifies a far pointer to a TIMECAPS structure. This structure is filled with information about the capabilities of the timer device.

UINT *wSize*

Specifies the size of the TIMECAPS structure.

Return Value

Returns zero if successful. Returns TIMERR_NOCANDO if it fails to return the timer device capabilities.

timeGetSystemTime

Syntax

UINT **timeGetSystemTime**(*lpTime*, *wSize*)

This function retrieves the system time in milliseconds. The system time is the time elapsed since Windows was started.

Parameters

LPMMTIME *lpTime*

Specifies a far pointer to an MMTIME data structure.

UINT *wSize*

Specifies the size of the MMTIME structure.

Return Value

Returns zero. The system time is returned in the ms field of the MMTIMEstructure.

Comments

The time is always returned in milliseconds.

See Also

timeGetTime

timeGetTime

Syntax

DWORD **timeGetTime**()

This function retrieves the system time in milliseconds. The system time is the time elapsed since Windows was started.

Parameters

None.

Return Value

The return value is the system time in milliseconds.

Comments

The only difference between this function and the timeGetSystemTime function is timeGetSystemTimeuses the standard multimedia time structure MMTIME to return the system time. The **timeGetTime** function has less overhead than timeGetSystemTime.

See Also

timeGetSystemTime

timeKillEvent

Syntax

UINT **timeKillEvent**(*wTimerID*)

This functions destroys a specified timer callback event.

Parameters

UINT *wTimerID*

Identifies the event to be destroyed.

Return Value

Returns zero if successful. Returns TIMERR_NOCANDO if the specified timer event does not exist.

Comments

The timer event ID specified by *wID* must be an ID returned by timeSetEvent.

See Also

timeSetEvent

timeSetEvent

Syntax

UINT **timeSetEvent**(*wDelay, wResolution, lpFunction, dwUser, wFlags*)

This function sets up a timed callback event. The event can be a one-time event or a periodic event. Once activated, the event calls the specified callback function.

Parameters

UINT *wDelay*

Specifies the event period in milliseconds. If the delay is less than the minimum period supported by the timer, or greater than the maximum period supported by the timer, the function returns an error.

UINT *wResolution*

Specifies the accuracy of the delay in milliseconds. The resolution of the timer event increases with smaller *wResolution* values. To reduce system overhead, use the maximum *wResolution* value appropriate for your application.

LPTIMECALLBACK *lpFunction*

Specifies the procedure address of a callback function that is called once upon expiration of a one-shot event or periodically upon expiration of periodic events.

DWORD *dwUser*

Contains user-supplied callback data.

UINT *wFlags*

Specifies the type of timer event, using one of the following flags:

TIME_ONESHOT

Event occurs once, after *wDelay* milliseconds.

TIME_PERIODIC

Event occurs every *wDelay* milliseconds.

Return Value

Returns an ID code that identifies the timer event. Returns NULL if the timer event was not created. The ID code is also passed to the callback function.

Callback

void CALLBACK **TimeFunc**(*wTimerID*, *wMsg*, *dwUser*, *dw1*, *dw2*)

TimeFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in the EXPORTS statement of the module-definition file for the DLL.

Callback Parameters

UINT *wTimerID*

The ID of the timer event. This is the ID returned by **timeSetEvent**.

UINT *wMsg*

Not used.

DWORD *dwUser*

User instance data supplied to the *dwUser* parameter of **timeSetEvent**.

DWORD *dw1*

Not used.

DWORD *dw2*
Not used.

Comments

Using this function to generate a high-frequency periodic-delay event (with a period less than 10 milliseconds) can consume a significant portion of the system CPU bandwidth. Any call to **timeSetEvent** for a periodic-delay timer must be paired with a call to timeKillEvent.

The callback function must reside in a DLL. You don't have to use **MakeProcInstance** to get a procedure-instance address for the callback function.

Because the callback is accessed at interrupt time, it must reside in a DLL, and its code segment must be specified as FIXED in the module-definition file for the DLL. Any data that the callback accesses must be in a FIXED data segment as well. The callback may not make any system calls except for **PostMessage**, timeGetSystemTime, timeGetTime, **timeSetEvent**, timeKillEvent, midiOutShortMsg, midiOutLongMsg, and OutputDebugStr.

See Also

timeKillEvent, timeBeginPeriod, timeEndPeriod

waveInAddBuffer

Syntax

UINT **waveInAddBuffer**(*hWaveIn*, *lpWaveInHdr*, *wSize*)

This function sends an input buffer to a waveform input device. When the buffer is filled, it is sent back to the application.

Parameters

HWAVEIN *hWaveIn*

Specifies a handle to the waveform input device.

LPWAVEHDR *lpWaveInHdr*

Specifies a far pointer to a WAVEHDRstructure that identifies the buffer.

UINT *wSize*

Specifies the size of the WAVEHDR structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

WAVERR_UNPREPARED

lpWaveInHdr hasn't been prepared.

Comments

The data buffer must be prepared with waveInPrepareHeader before it is passed to **waveInAddBuffer**. The WAVEHDR data structure and the data buffer pointed to by its lpData field must be allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with **GlobalLock**.

See Also

waveInPrepareHeader

waveInClose

Syntax

UINT **waveInClose**(*hWaveIn*)

This function closes the specified waveform input device.

Parameters

HWAVEIN *hWaveIn*

Specifies a handle to the waveform input device. If the function is successful, the handle is no longer valid after this call.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

WAVERR_STILLPLAYING

There are still buffers in the queue.

Comments

If there are input buffers that have been sent with waveInAddBuffer, and haven't been returned to the application, the close operation will fail. Call waveInReset to mark all pending buffers as done.

See Also

waveInOpen, waveInReset

waveInGetDevCaps

Syntax

UINT **waveInGetDevCaps**(*wDeviceID*, *lpCaps*, *wSize*)

This function queries a specified waveform input device to determine its capabilities.

Parameters

UINT wDeviceID

Identifies the waveform input device to query. Use a valid waveform input device ID (see the following "Comments" section) or the following constant:

WAVE_MAPPER

Wave mapper. If no wave mapper is installed, the function returns an error number.

LPWAVEINCAPS *lpCaps*

Specifies a far pointer to a WAVEINCAPSstructure. This structure is filled with information about the capabilities of the device.

UINT *wSize*

Specifies the size of the WAVEINCAPS structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID

Specified device ID is out of range.

MMSYSERR_NODRIVER

The driver was not installed.

Comments

The device ID specified by *wDeviceID* varies from zero to one less than the number of devices present. Use waveInGetNumDevs to determine the number of waveform input devices present in the system.

Only *wSize* bytes (or less) of information is copied to the location pointed to by *lpCaps*. If *wSize* is zero, nothing is copied, and the function returns zero.

See Also

waveInGetNumDevs

waveInGetErrorText

Syntax

UINT **waveInGetErrorText**(*wError*, *lpText*, *wSize*)

This function retrieves a textual description of the error identified by the specified error number.

Parameters

UINT *wError*

Specifies the error number.

LPSTR *lpText*

Specifies a far pointer to the buffer to be filled with the textual error description.

UINT *wSize*

Specifies the size of the buffer pointed to by *lpText*.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADERRNUM

Specified error number is out of range.

Comments

If the textual error description is longer than the specified buffer, the description is truncated. The returned error string is always null-terminated. If *wSize* is zero, nothing is copied, and the function returns zero. All error descriptions are less than MAXERRORLENGTH characters long.

waveInGetID

Syntax

UINT **waveInGetID**(*hWaveIn*, *lpwDeviceID*)

This function gets the device ID for a waveform input device.

Parameters

HWAVEIN *hWaveIn*

Specifies the handle to the waveform input device.

UINT FAR* *lpwDeviceID*

Specifies a pointer to the UINT-sized memory location to be filled with the device ID.

Return Value

Returns zero if successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

The *hWaveIn* parameter specifies an invalid handle.

waveInGetNumDevs

Syntax

UINT **waveInGetNumDevs**()

This function returns the number of waveform input devices.

Parameters

None.

Return Value

Returns the number of waveform input devices present in the system.

See Also

waveInGetDevCaps

waveInGetPosition

Syntax

UINT **waveInGetPosition**(*hWaveIn*, *lpInfo*, *wSize*)

This function retrieves the current input position of the specified waveform input device.

Parameters

HWAVEIN *hWaveIn*

Specifies a handle to the waveform input device.

LPMMTIME *lpInfo*

Specifies a far pointer to an MMTIMEstructure.

UINT *wSize*

Specifies the size of the MMTIME structure.

Return Value

Returns zero if the function was successful. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

Comments

Before calling **waveInGetPosition**, set the wType field of the MMTIME structure to indicate the time format that you desire. After calling **waveInGetPosition**, be sure to check the wType field to determine if the desired time format is supported. If the desired format is not supported, wType will specify an alternative format.

The position is set to zero when the device is opened or reset.

waveInMessage

Syntax

DWORD **waveInMessage**(*hWaveIn*, m*sg*, *dwParam1*, *dwParam2*)

This function sends a message to a waveform input device driver. Use it to send driver-specific messages that aren't supported by the waveform APIs.

Parameters

HWAVEOUT *hWaveIn*

Specifies the handle to the audio device driver.

UINT *msg*

Specifies the message to send.

DWORD *dwParam1*

Specifies the first message parameter.

DWORD *dwParam2*

Specifies the second message parameter.

Return Value

Returns the value returned by the audio device driver.

Comments

Do not use this function to send standard messages to an audio device driver.

See Also

waveOutMessage

waveInOpen

Syntax

UINT **waveInOpen**(*lphWaveIn, wDeviceID, lpFormat, dwCallback, dwCallbackInstance, dwFlags*)

This function opens a specified waveform input device for recording.

Parameters

LPHWAVEIN *lphWaveIn*

Specifies a far pointer to a HWAVEIN handle. This location is filled with a handle identifying the opened waveform input device. Use this handle to identify the device when calling other waveform input functions. This parameter may be NULL if the WAVE_FORMAT_QUERY flag is specified for *dwFlags*.

UINT *wDeviceID*

Identifies the waveform input device to open. Use a valid waveform input device ID (see the following "Comments" section) or the following constant:

WAVE_MAPPER

Wave mapper. If no wave mapper is installed, the system selects a waveform input device capable of recording in the given format.

LPWAVEFORMAT *lpFormat*

Specifies a pointer to a WAVEFORMATdata structure that identifies the desired format for recording waveform data.

DWORD *dwCallback*

Specifies the address of a callback function or a handle to a window called during waveform recording to process messages related to the progress of recording.

DWORD *dwCallbackInstance*

Specifies user instance data passed to the callback. This parameter is not used with window callbacks.

DWORD *dwFlags*

Specifies flags for opening the device.

WAVE_FORMAT_QUERY

If this flag is specified, the device will be queried to determine if it supports the given format but will not actually be opened.

WAVE_ALLOWSYNC

Allows a synchronous (blocking) waveform driver to be opened. If this flag is not set while opening a synchronous driver, the open will fail.

CALLBACK_WINDOW

If this flag is specified, *dwCallback* is assumed to be a window handle.

CALLBACK_FUNCTION

If this flag is specified, *dwCallback* is assumed to be a callback procedure address.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_NODRIVER

The driver was not installed.

MMSYSERR_BADDEVICEID
Specified device ID is out of range.

MMSYSERR_ALLOCATED
Specified resource is already allocated.

MMSYSERR_NOMEM
Unable to allocate or lock memory.

WAVERR_BADFORMAT
Attempted to open with an unsupported wave format.

WAVERR_SYNC
Attempted to open a synchronous driver without specifying the WAVE_ALLOWSYNC flag.

Callback

void CALLBACK **WaveInFunc**(*hWaveIn*, *wMsg*, *dwInstance*, *dwParam1*, *dwParam2*)

WaveInFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an EXPORTS statement in the DLL's module-definition file.

Parameters

HWAVEIN *hWaveIn*
Specifies a handle to the waveform device associated with the callback.

UINT *wMsg*
Specifies a waveform input device.

DWORD *dwInstance*
Specifies the user instance data specified with **waveInOpen**.

DWORD *dwParam1*
Specifies a parameter for the message.

DWORD *dwParam2*
Specifies a parameter for the message.

Comments

The device ID specified by *wDeviceID* varies from zero to one less than the number of devices present. Use waveInGetNumDevs to determine the number of waveform input devices present in the system.

If a window is chosen to receive callback information, the following messages are sent to the window procedure function to indicate the progress of waveform input:

* MM_WIM_OPEN
* MM_WIM_CLOSE
* MM_WIM_DATA

If a function is chosen to receive callback information, the following messages are sent to the function to indicate the progress of waveform input:

* WIM_OPEN
* WIM_CLOSE
* WIM_DATA

The callback function must reside in a DLL. You do not have to use **MakeProcInstance** to get a procedure-instance address for the callback function.

Because the callback is accessed at interrupt time, it must reside in a DLL and its code segment must be specified as FIXED in the module-definition file for the DLL. Any data that the callback accesses must be in a FIXED data segment as well. The callback may not make any system calls except for **PostMessage**, timeGetSystemTime, timeGetTime, timeSetEvent, timeKillEvent, midiOutShortMsg, midiOutLongMsg, and OutputDebugStr.

See Also

waveInClose

waveInPrepareHeader

Syntax

UINT **waveInPrepareHeader**(*hWaveIn*, *lpWaveInHdr*, *wSize*)

This function prepares a buffer for waveform input.

Parameters

HWAVEIN *hWaveIn*
Specifies a handle to the waveform input device.

LPWAVEHDR *lpWaveInHdr*
Specifies a pointer to a WAVEHDR structure that identifies the buffer to be prepared.

UINT *wSize*
Specifies the size of the WAVEHDR structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE
Specified device handle is invalid.

MMSYSERR_NOMEM
Unable to allocate or lock memory.

Comments

The WAVEHDR data structure and the data block pointed to by its lpData field must be allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with **GlobalLock**. Preparing a header that has already been prepared will have no effect, and the function will return zero.

See Also

waveInUnprepareHeader

waveInReset

Syntax

UINT **waveInReset**(*hWaveIn*)

This function stops input on a given waveform input device and resets the current position to 0. All pending buffers are marked as done and returned to the application.

Parameters

HWAVEIN *hWaveIn*

Specifies a handle to the waveform input device.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

See Also

waveInStart, waveInStop, waveInAddBuffer, waveInClose

waveInStart

Syntax

UINT **waveInStart**(*hWaveIn*)

This function starts input on the specified waveform input device.

Parameters

HWAVEIN *hWaveIn*

Specifies a handle to the waveform input device.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

Comments

Buffers are returned to the client when full or when waveInResetis called (the dwBytesRecorded field in the header will contain the actual length of data). If there are no buffers in the queue, the data is thrown away without notification to the client, and input continues.

Calling this function when input is already started has no effect, and the function returns zero.

See Also

waveInStop, waveInReset

waveInStop

Syntax

UINT **waveInStop**(*hWaveIn*)

This function stops waveform input.

Parameters

HWAVEIN *hWaveIn*

Specifies a handle to the waveform input device.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

Comments

If there are any buffers in the queue, the current buffer will be marked as done (the dwBytesRecorded field in the header will contain the actual length of data), but any empty buffers in the queue will remain there. Calling this function when input is not started has no effect, and the function returns zero.

See Also

waveInStart, waveInReset

waveInUnprepareHeader

Syntax

UINT **waveInUnprepareHeader**(*hWaveIn*, *lpWaveInHdr*, *wSize*)

This function cleans up the preparation performed by waveInPrepareHeader. The function must be called after the device driver fills a data buffer and returns it to the application. You must call this function before freeing the data buffer.

Parameters

HWAVEIN *hWaveIn*

Specifies a handle to the waveform input device.

LPWAVEHDR *lpWaveInHdr*

Specifies a pointer to a WAVEHDRstructure identifying the data buffer to be cleaned up.

UINT *wSize*

Specifies the size of the WAVEHDR structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

WAVERR_STILLPLAYING

lpWaveInHdr is still in the queue.

Comments

This function is the complementary function to waveInPrepareHeader. You must call this function before freeing the data buffer with **GlobalFree**. After passing a buffer to the device driver with waveInAddBuffer, you must wait until the driver is finished with the buffer before calling **waveInUnprepareHeader**. Unpreparing a buffer that has not been prepared has no effect, and the function returns zero.

See Also

waveInPrepareHeader

waveOutBreakLoop

Syntax

UINT **waveOutBreakLoop**(*hWaveOut*)

This function breaks a loop on a given waveform output device and allows playback to continue with the next block in the driver list.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

Comments

Waveform looping is controlled by the dwLoops and dwFlags fields in the WAVEHDR structures passed to the device with waveOutWrite. Use the WHDR_BEGINLOOP and WHDR_ENDLOOP flags in the dwFlags field to specify the beginning and ending data blocks for looping.

To loop on a single block, specify both flags for the same block. To specify the number of loops, use the dwLoops field in the WAVEHDR structure for the first block in the loop.

The blocks making up the loop are played to the end before the loop is terminated.

Calling this function when the nothing is playing or looping has no effect, and the function returns zero.

See Also

waveOutWrite, waveOutPause, waveOutRestart

waveOutClose

Syntax

UINT **waveOutClose**(*hWaveOut*)

This function closes the specified waveform output device.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device. If the function is successful, the handle is no longer valid after this call.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

WAVERR_STILLPLAYING

There are still buffers in the queue.

Comments

If the device is still playing a waveform, the close operation will fail. Use waveOutReset to terminate waveform playback before calling **waveOutClose**.

See Also

waveOutOpen, waveOutReset

waveOutGetDevCaps

Syntax

UINT **waveOutGetDevCaps**(*wDeviceID*, *lpCaps*, *wSize*)

This function queries a specified waveform device to determine its capabilities.

Parameters

UINT wDeviceID

Identifies the waveform output device to query. Use a valid waveform output device ID (see the following "Comments" section) or the following constant:

WAVE_MAPPER

Wave mapper. If no wave mapper is installed, the function returns and error number.

LPWAVEOUTCAPS *lpCaps*

Specifies a far pointer to a WAVEOUTCAPSstructure. This structure is filled with information about the capabilities of the device.

UINT *wSize*

Specifies the size of the WAVEOUTCAPS structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID

Specified device ID is out of range.

MMSYSERR_NODRIVER

The driver was not installed.

Comments

The device ID specified by *wDeviceID* varies from zero to one less than the number of devices present. Use waveOutGetNumDevs to determine the number of waveform output devices present in the system.

Only *wSize* bytes (or less) of information is copied to the location pointed to by *lpCaps*. If *wSize* is zero, nothing is copied, and the function returns zero.

See Also

waveOutGetNumDevs

waveOutGetErrorText

Syntax

UINT **waveOutGetErrorText**(*wError*, *lpText*, *wSize*)

This function retrieves a textual description of the error identified by the specified error number.

Parameters

UINT *wError*

Specifies the error number.

LPSTR *lpText*

Specifies a far pointer to a buffer to be filled with the textual error description.

UINT *wSize*

Specifies the length of the buffer pointed to by *lpText*.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADERRNUM

Specified error number is out of range.

Comments

If the textual error description is longer than the specified buffer, the description is truncated. The returned error string is always null-terminated. If *wSize* is zero, nothing is copied, and the function returns zero. All error descriptions are less than MAXERRORLENGTH characters long.

waveOutGetID

Syntax

UINT **waveOutGetID**(*hWaveOut*, *lpwDeviceID*)

This function gets the device ID for a waveform output device.

Parameters

HWAVEOUT *hWaveOut*

Specifies the handle to the waveform output device.

UINT FAR* *lpwDeviceID*

Specifies a pointer to the UINT-sized memory location to be filled with the device ID.

Return Value

Returns zero if successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

The *hWaveOut* parameter specifies an invalid handle.

waveOutGetNumDevs

Syntax

UINT **waveOutGetNumDevs**()

This function retrieves the number of waveform output devices present in the system.

Parameters

None.

Return Value

Returns the number of waveform output devices present in the system.

See Also

waveOutGetDevCaps

waveOutGetPitch

Syntax

UINT **waveOutGetPitch**(*hWaveOut*, *lpdwPitch*)

This function queries the current pitch setting of a waveform output device.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

LPDWORD *lpdwPitch*

Specifies a far pointer to a location to be filled with the current pitch multiplier setting. The pitch multiplier indicates the current change in pitch from the original authored setting. The pitch multiplier must be a positive value.

The pitch multiplier is specified as a fixed-point value. The high-order word of the DWORD location contains the signed integer part of the number, and the low-order word contains the fractional part. The fraction is expressed as a UINT in which a value of 0x8000 represents one half, and 0x4000 represents one quarter. For example, the value 0x00010000 specifies a multiplier of 1.0 (no pitch change), and a value of 0x000F8000 specifies a multiplier of 15.5.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MMSYSERR_NOTSUPPORTED

Function isn't supported.

Comments

Changing the pitch does not change the playback rate, sample rate, or playback time. Not all devices support pitch changes. To determine whether the device supports pitch control, use the WAVECAPS_PITCH flag to test the dwSupportfield of the WAVEOUTCAPS structure (filled by waveOutGetDevCaps).

See Also

waveOutSetPitch, waveOutGetPlaybackRate, waveOutSetPlaybackRate

waveOutGetPlaybackRate

Syntax

UINT **waveOutGetPlaybackRate**(*hWaveOut*, *lpdwRate*)

This function queries the current playback rate setting of a waveform output device.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

LPDWORD *lpdwRate*

Specifies a far pointer to a location to be filled with the current playback rate. The playback rate setting is a multiplier indicating the current change in playback rate from the original authored setting. The playback rate multiplier must be a positive value.

The rate is specified as a fixed-point value. The high-order word of the DWORD location contains the signed integer part of the number, and the low-order word contains the fractional part. The fraction is expressed as a UINT in which a value of 0x8000 represents one half, and 0x4000 represents one quarter. For example, the value 0x00010000 specifies a multiplier of 1.0 (no playback rate change), and a value of 0x000F8000 specifies a multiplier of 15.5.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MMSYSERR_NOTSUPPORTED

Function isn't supported.

Comments

Changing the playback rate does not change the sample rate but does change the playback time.

Not all devices support playback rate changes. To determine whether a device supports playback rate changes, use the WAVECAPS_PLAYBACKRATE flag to test the dwSupport field of the WAVEOUTCAPS structure (filled by waveOutGetDevCaps).

See Also

waveOutSetPlaybackRate, waveOutSetPitch, waveOutGetPitch

waveOutGetPosition

Syntax

UINT **waveOutGetPosition**(*hWaveOut*, *lpInfo*, *wSize*)

This function retrieves the current playback position of the specified waveform output device.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

LPMMTIME *lpInfo*

Specifies a far pointer to an MMTIMEstructure.

UINT *wSize*

Specifies the size of the MMTIME structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

Comments

Before calling **waveOutGetPosition**, set the wType field of the MMTIME structure to indicate the time format that you desire. After calling **waveOutGetPosition**, check the wType field to determine if the desired time format is supported. If the desired format is not supported, wType will specify an alternative format.

The position is set to zero when the device is opened or reset.

waveOutGetVolume

Syntax

UINT **waveOutGetVolume**(*wDeviceID*, *lpdwVolume*)

This function queries the current volume setting of a waveform output device.

Parameters

UINT wDeviceID

Identifies the waveform output device.

LPDWORD *lpdwVolume*

Specifies a far pointer to a location to be filled with the current volume setting. The low-order word of this location contains the left channel volume setting, and the high-order UINT contains the right channel setting. A value of 0xFFFF represents full volume, and a value of 0x0000 is silence.

If a device does not support both left and right volume control, the low-order word of the specified location contains the mono volume level.

The full 16-bit setting(s)set with waveOutSetVolume is returned, regardless of whether the device supports the full 16 bits of volume-level control.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MMSYSERR_NOTSUPPORTED

Function isn't supported.

MMSYSERR_NODRIVER

The driver was not installed.

Comments

Not all devices support volume changes. To determine whether the device supports volume control, use the WAVECAPS_VOLUME flag to test the dwSupport field of the WAVEOUTCAPSstructure (filled by waveOutGetDevCaps).

To determine whether the device supports volume control on both the left and right channels, use the WAVECAPS_VOLUME flag to test the dwSupport field of the WAVEOUTCAPSstructure (filled by waveOutGetDevCaps).

See Also

waveOutSetVolume

waveOutMessage

Syntax

DWORD **waveOutMessage**(*hWaveOut, msg, dwParam1, dwParam2*)

This function sends a message to a waveform output device driver. Use it to send driver-specific messages that aren't supported by the waveform APIs.

Parameters

HWAVEOUT *hWaveOut*

Specifies the handle to the audio device driver.

UINT *msg*

Specifies the message to send.

DWORD *dwParam1*

Specifies the first message parameter.

DWORD *dwParam2*

Specifies the second message parameter.

Return Value

Returns the value returned by the audio device driver.

Comments

Do not use this function to send standard messages to an audio device driver.

See Also

waveInMessage

waveOutOpen

Syntax

UINT **waveOutOpen**(*lphWaveOut, wDeviceID, lpFormat, dwCallback, dwCallbackInstance, dwFlags*)

This function opens a specified waveform output device for playback.

Parameters

LPHWAVEOUT *lphWaveOut*

Specifies a far pointer to an HWAVEOUT handle. This location is filled with a handle identifying the opened waveform output device. Use the handle to identify the device when calling other waveform output functions. This parameter may be NULL if the WAVE_FORMAT_QUERY flag is specified for *dwFlags*.

UINT *wDeviceID*

Identifies the waveform output device to open. Use a valid waveform output device ID (see the following "Comments" section) or the following constant:

WAVE_MAPPER

Wave mapper. If no wave mapper is installed, the system selects a waveform output device capable of playing the given format.

LPWAVEFORMAT *lpFormat*

Specifies a pointer to a WAVEFORMATstructure that identifies the format of the waveform data to be sent to the waveform output device.

DWORD *dwCallback*

Specifies the address of a callback function or a handle to a window called during waveform playback to process messages related to the progress of the playback. Specify NULL for this parameter if no callback is desired.

DWORD *dwCallbackInstance*

Specifies user instance data passed to the callback. This parameter is not used with window callbacks.

DWORD *dwFlags*

Specifies flags for opening the device.

WAVE_FORMAT_QUERY

If this flag is specified, the device is be queried to determine if it supports the given format but is not actually opened.

WAVE_ALLOWSYNC

Allows a synchronous (blocking) waveform driver to be opened. If this flag is not set while opening a synchronous driver, the open will fail.

CALLBACK_WINDOW

If this flag is specified, *dwCallback* is assumed to be a window handle.

CALLBACK_FUNCTION

If this flag is specified, *dwCallback* is assumed to be a callback procedure address.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_BADDEVICEID

Specified device ID is out of range.

MMSYSERR_ALLOCATED
Specified resource is already allocated.

MMSYSERR_NOMEM
Unable to allocate or lock memory.

WAVERR_BADFORMAT
Attempted to open with an unsupported wave format.

WAVERR_SYNC
Attempted to open a synchronous driver without specifying the WAVE_ALLOWSYNC flag.

Callback

void CALLBACK **WaveOutFunc**(*hWaveOut*, *wMsg*, *dwInstance*, *dwParam1*, *dwParam2*)

WaveOutFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an EXPORTS statement in the DLL's module-definition file.

Parameters

HWAVEOUT *hWaveOut*
Specifies a handle to the waveform device associated with the callback.

UINT *wMsg*
Specifies a waveform output message.

DWORD *dwInstance*
Specifies the user instance data specified with **waveOutOpen**.

DWORD *dwParam1*
Specifies a parameter for the message.

DWORD *dwParam2*
Specifies a parameter for the message.

Comments

The device ID specified by *wDeviceID* varies from zero to one less than the number of devices present. Use waveOutGetNumDevs to determine the number of waveform output devices present in the system.

The WAVEFORMAT structure pointed to by *lpFormat* may be extended to include type-specific information for certain data formats. For example, for PCM data, an extra UINT is added to specify the number of bits per sample. Use the PCMWAVEFORMAT structure in this case.

If a window is chosen to receive callback information, the following messages are sent to the window procedure function to indicate the progress of waveform output:

* MM_WOM_OPEN
* MM_WOM_CLOSE
* MM_WOM_DONE

If a function is chosen to receive callback information, the following messages are sent to the function to indicate the progress of waveform output:

* WOM_OPEN
* WOM_CLOSE
* WOM_DONE

The callback function must reside in a DLL. You do not have to use **MakeProcInstance** to get a procedure-instance address for the callback function.

Because the callback is accessed at interrupt time, it must reside in a DLL and its code segment must be specified as FIXED in the module-definition file for the DLL. Any data that the callback accesses must be in a FIXED data segment as well. The callback may not make any system calls except for **PostMessage**, timeGetSystemTime, timeGetTime, timeSetEvent, timeKillEvent, midiOutShortMsg, midiOutLongMsg, and OutputDebugStr.

See Also

waveOutClose

waveOutPause

Syntax

UINT **waveOutPause**(*hWaveOut*)

This function pauses playback on a specified waveform output device. The current playback position is saved. Use waveOutRestart to resume playback from the current playback position.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

Comments

Calling this function when the output is already paused has no effect, and the function returns zero.

See Also

waveOutRestart, waveOutBreakLoop

waveOutPrepareHeader

Syntax

UINT **waveOutPrepareHeader**(*hWaveOut*, *lpWaveOutHdr*, *wSize*)

This function prepares a waveform data block for playback.

Parameters

HWAVEOUT *hWaveOut*
Specifies a handle to the waveform output device.

LPWAVEHDR *lpWaveOutHdr*
Specifies a pointer to a WAVEHDR structure that identifies the data block to be prepared.

UINT *wSize*
Specifies the size of the WAVEHDR structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE
Specified device handle is invalid.

MMSYSERR_NOMEM
Unable to allocate or lock memory.

Comments

The WAVEHDR data structure and the data block pointed to by its lpData field must be allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with **GlobalLock**. Preparing a header that has already been prepared has no effect, and the function returns zero.

See Also

waveOutUnprepareHeader

waveOutReset

Syntax

UINT **waveOutReset**(*hWaveOut*)

This function stops playback on a given waveform output device and resets the current position to 0. All pending playback buffers are marked as done and returned to the application.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

See Also

waveOutWrite, waveOutClose

waveOutRestart

Syntax

UINT **waveOutRestart**(*hWaveOut*)

This function restarts a paused waveform output device.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

Comments

Calling this function when the output is not paused has no effect, and the function returns zero.

See Also

waveOutPause, waveOutBreakLoop

waveOutSetPitch

Syntax

UINT **waveOutSetPitch**(*hWaveOut*, *dwPitch*)

This function sets the pitch of a waveform output device.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

DWORD *dwPitch*

Specifies the new pitch multiplier setting. The pitch multiplier setting indicates the current change in pitch from the original authored setting. The pitch multiplier must be a positive value.

The pitch multiplier is specified as a fixed-point value. The high-order word location contains the signed integer part of the number, and the low-order word contains the fractional part. The fraction is expressed as a UINT in which a value of 0x8000 represents one half, and 0x4000 represents one quarter. For example, the value 0x00010000 specifies a multiplier of 1.0 (no pitch change), and a value of 0x000F8000 specifies a multiplier of 15.5.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MMSYSERR_NOTSUPPORTED

Function isn't supported.

Comments

Changing the pitch does not change the playback rate or the sample rate. The playback time is also unchanged. Not all devices support pitch changes. To determine whether the device supports pitch control, use the WAVECAPS_PITCH flag to test the dwSupportfield of the WAVEOUTCAPS structure (filled by waveOutGetDevCaps).

See Also

waveOutGetPitch, waveOutSetPlaybackRate, waveOutGetPlaybackRate

waveOutSetPlaybackRate

Syntax

UINT **waveOutSetPlaybackRate**(*hWaveOut*, *dwRate*)

This function sets the playback rate of a waveform output device.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

DWORD *dwRate*

Specifies the new playback rate setting. The playback rate setting is a multiplier indicating the current change in playback rate from the original authored setting. The playback rate multiplier must be a positive value.

The rate is specified as a fixed-point value. The high-order word contains the signed integer part of the number, and the low-order word contains the fractional part. The fraction is expressed as a UINT in which a value of 0x8000 represents one half, and 0x4000 represents one quarter. For example, the value 0x00010000 specifies a multiplier of 1.0 (no playback rate change), and a value of 0x000F8000 specifies a multiplier of 15.5.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

MMSYSERR_NOTSUPPORTED

Function isn't supported.

Comments

Changing the playback rate does not change the sample rate but does change the playback time.

Not all devices support playback rate changes. To determine whether a device supports playback rate changes, use the WAVECAPS_PLAYBACKRATE flag to test the dwSupport field of the WAVEOUTCAPS structure (filled by waveOutGetDevCaps).

See Also

waveOutGetPlaybackRate, waveOutSetPitch, waveOutGetPitch

waveOutSetVolume

Syntax

UINT **waveOutSetVolume**(*wDeviceID*, *dwVolume*)

This function sets the volume of a waveform output device.

Parameters

UINT wDeviceID
Identifies the waveform output device.

DWORD *dwVolume*
Specifies the new volume setting. The low-order word contains the left channel volume setting, and the high-order word contains the right channel setting. A value of 0xFFFF represents full volume, and a value of 0x0000 is silence.

If a device does not support both left and right volume control, the low-order word of *dwVolume* specifies the volume level, and the high-order word is ignored.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE
Specified device handle is invalid.

MMSYSERR_NOTSUPPORTED
Function isn't supported.

MMSYSERR_NODRIVER
The driver was not installed.

Comments

Not all devices support volume changes. To determine whether the device supports volume control, use the WAVECAPS_VOLUME flag to test the dwSupport field of the WAVEOUTCAPSstructure (filled by waveOutGetDevCaps).

To determine whether the device supports volume control on both the left and right channels, use the WAVECAPS_LRVOLUME flag flag to test the dwSupport field of the WAVEOUTCAPSstructure (filled by waveOutGetDevCaps).

Most devices don't support the full 16 bits of volume level control and will not use the high-order bits of the requested volume setting. For example, for a device that supports 4 bits of volume control, requested volume level values of 0x4000, 0x4fff, and 0x43be all produce the same physical volume setting, 0x4000. The waveOutGetVolume function returns the full 16-bit setting set with **waveOutSetVolume**.

Volume settings are interpreted logarithmically. This means the perceived increase in volume is the same when increasing the volume level from 0x5000 to 0x6000 as it is from 0x4000 to 0x5000.

See Also

waveOutGetVolume

waveOutUnprepareHeader

Syntax

UINT **waveOutUnprepareHeader**(*hWaveOut*, *lpWaveOutHdr*, *wSize*)

This function cleans up the preparation performed by waveOutPrepareHeader. The function must be called after the device driver is finished with a data block. You must call this function before freeing the data buffer.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

LPWAVEHDR *lpWaveOutHdr*

Specifies a pointer to a WAVEHDRstructure identifying the data block to be cleaned up.

UINT *wSize*

Specifies the size of the WAVEHDR structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

WAVERR_STILLPLAYING

lpWaveOutHdr is still in the queue.

Comments

This function is the complementary function to waveOutPrepareHeader. You must call this function before freeing the data buffer with **GlobalFree**. After passing a buffer to the device driver with waveOutWrite, you must wait until the driver is finished with the buffer before calling **waveOutUnprepareHeader**.

Unpreparing a buffer that has not been prepared has no effect, and the function returns zero.

See Also

waveOutPrepareHeader

waveOutWrite

Syntax

UINT **waveOutWrite**(*hWaveOut, lpWaveOutHdr, wSize*)

This function sends a data block to the specified waveform output device.

Parameters

HWAVEOUT *hWaveOut*

Specifies a handle to the waveform output device.

LPWAVEHDR *lpWaveOutHdr*

Specifies a far pointer to a WAVEHDRstructure containing information about the data block.

UINT *wSize*

Specifies the size of the WAVEHDR structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. Possible error returns are:

MMSYSERR_INVALHANDLE

Specified device handle is invalid.

WAVERR_UNPREPARED

lpWaveOutHdr hasn't been prepared.

Comments

The data buffer must be prepared with waveOutPrepareHeader before it is passed to **waveOutWrite**. The WAVEHDR data structure and the data buffer pointed to by its lpData field must be allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with **GlobalLock**. Unless the device is paused by calling waveOutPause, playback begins when the first data block is sent to the device.

See Also

waveOutPrepareHeader, waveOutPause, waveOutReset, waveOutRestart

AUXCAPS

The **AUXCAPS** structure describes the capabilities of an auxiliary output device.

```
typedef struct auxcaps_tag {
    UINT wMid;
    UINT wPid;
    VERSION vDriverVersion;
    char szPname[MAXPNAMELEN];
    UINT wTechnology;
    DWORD dwSupport;
} AUXCAPS;
```

Fields

The **AUXCAPS** structure contains the following fields:

wMid

Specifies a manufacturer ID for the device driver for the auxiliary audio device. Manufacturer IDs are defined in Manufacturer and Product IDs.

wPid

Specifies a product ID for the auxiliary audio device. Currently, no product IDs are defined for auxiliary audio devices.

vDriverVersion

Specifies the version number of the device driver for the auxiliary audio device. The high-order byte is the major version number, and the low-order byte is the minor version number.

szPname[MAXPNAMELEN]

Specifies the product name in a NULL-terminated string.

wTechnology

Describes the type of the auxiliary audio output according to one of the following flags:

AUXCAPS_CDAUDIO

Audio output from an internal CD-ROM drive.

AUXCAPS_AUXIN

Audio output from auxiliary input jacks.

dwSupport

Describes optional functionality supported by the auxiliary audio device.

AUXCAPS_VOLUME

Supports volume control.

AUXCAPS_LRVOLUME

Supports separate left and right volume control.

Comments

If a device supports volume changes, the AUXCAPS_VOLUME flag will be set for the **dwSupport** field. If a device supports separate volume changes on the left and right channels, both the AUXCAPS_VOLUME and the AUXCAPS_LRVOLUME flags will be set for this field.

See Also

auxGetDevCaps

JOYCAPS

The **JOYCAPS** structure contains the fields describing the joystick capabilities.

```
typedef struct joycaps_tag {
    UINT wMid;
    UINT wPid;
    char szPname[MAXPNAMELEN];
    UINT wXmin;
    UINT wXmax;
    UINT wYmin;
    UINT wYmax;
    UINT wZmin;
    UINT wZmax;
    UINT wNumButtons;
    UINT wPeriodMin;
    UINT wPeriodMax;
} JOYCAPS;
```

Fields

The **JOYCAPS** structure contains the following fields:

wMid
Specifies the manufacturer ID of the joystick. Manufacturer IDs are defined in Manufacturer and Product IDs.

wPid
Specifies the product ID of the joystick. Product IDs are defined in Manufacturer and Product IDs.

szPname[MAXPNAMELEN]
Specifies the product name of the joystick. This information is stored as a null-terminated string.

wXmin
Specifies the minimum x position value of the joystick.

wXmax
Specifies the maximum x position value of the joystick.

wYmin
Specifies the minimum y position value of the joystick.

wYmax
Specifies the maximum y position value of the joystick.

wZmin
Specifies the minimum z position value of the joystick.

wZmax
Specifies the maximum z position value of the joystick.

wNumButtons
Specifies the number of buttons on the joystick.

wPeriodMin
Specifies the smallest polling interval supported when captured by joySetCapture.

wPeriodMax
Specifies the largest polling interval supported when captured by joySetCapture.

See Also

joyGetDevCaps

JOYINFO

The **JOYINFO** structure contains fields for storing joystick position and button state information.

```
typedef struct joyinfo_tag {
    UINT wXpos;
    UINT wYpos;
    UINT wZpos;
    UINT wButtons;
} JOYINFO;
```

Fields

The **JOYINFO** structure contains the following fields:

wXpos

Specifies the current x-position of joystick.

wYpos

Specifies the current y-position of joystick.

wZpos

Specifies the current z-position of joystick.

wButtons

Specifies the current state of joystick buttons. It can be any combination of the following bit flags:

JOY_BUTTON1

Set if button 1 is pressed.

JOY_BUTTON2

Set if button 2 is pressed.

JOY_BUTTON3

Set if button 3 is pressed.

JOY_BUTTON4

Set if button 4 is pressed.

See Also

joyGetPos

MIDIHDR

The **MIDIHDR** structure defines the header used to identify a MIDI system-exclusive data buffer.

```
typedef struct midihdr_tag {
    LPSTR lpData;
    DWORD dwBufferLength;
    DWORD dwBytesRecorded;
    DWORD dwUser;
    DWORD dwFlags;
    struct midihdr_tag far * lpNext;
    DWORD reserved;
} MIDIHDR;
```

Fields

The **MIDIHDR** structure contains the following fields:

lpData

Specifies a far pointer to the system-exclusive data buffer.

dwBufferLength

Specifies the length of the data buffer.

dwBytesRecorded

When the header is used in input, this specifies how much data is in the buffer.

dwUser

Specifies user data.

dwFlags

Specifies flags giving information about the data buffer.

MHDR_DONE

Set by the device driver to indicate that it is finished with the data buffer and is returning it to the application.

MHDR_PREPARED

Set by Windows to indicate that the data buffer has been prepared with midiInPrepareHeader or midiOutPrepareHeader. This flag is reserved for the driver and should not be set by the application.

lpNext

Is reserved and should not be used.

reserved

Is reserved and should not be used.

MIDIINCAPS

The **MIDIINCAPS** structure describes the capabilities of a MIDI input device.

```
typedef struct midiincaps_tag {
    UINT wMid;
    UINT wPid;
    VERSION vDriverVersion;
    char szPname[MAXPNAMELEN];
} MIDIINCAPS;
```

Fields

The **MIDIINCAPS** structure contains the following fields:

wMid

Specifies a manufacturer ID for the device driver for the MIDI input device. Manufacturer IDs are defined in Manufacturer and Product IDs.

wPid

Specifies a product ID for the MIDI input device. Product IDs are defined in Manufacturer and Product IDs.

vDriverVersion

Specifies the version number of the device driver for the MIDI input device. The high-order byte is the major version number, and the low-order byte is the minor version number.

szPname[MAXPNAMELEN]

Specifies the product name in a NULL-terminated string.

See Also

midiInGetDevCaps

MIDIOUTCAPS

The **MIDIOUTCAPS** structure describes the capabilities of a MIDI output device.

```
typedef struct midioutcaps_tag {
    UINT wMid;
    UINT wPid;
    VERSION vDriverVersion;
    char szPname[MAXPNAMELEN];
    UINT wTechnology;
    UINT wVoices;
    UINT wNotes;
    UINT wChannelMask;
    DWORD dwSupport;
} MIDIOUTCAPS;
```

Fields

The **MIDIOUTCAPS** structure contains the following fields:

wMid

Specifies a manufacturer ID for the device driver for the MIDI output device. Manufacturer IDs are defined in Manufacturer and Product IDs.

wPid

Specifies a product ID for the MIDI output device. Product IDs are defined in Manufacturer and Product IDs.

vDriverVersion

Specifies the version number of the device driver for the MIDI output device. The high-order byte is the major version number, and the low-order byte is the minor version number.

szPname[MAXPNAMELEN]

Specifies the product name in a NULL-terminated string.

wTechnology

Describes the type of the MIDI output device according to one of the following flags:

MOD_MIDIPORT

Indicates the device is a MIDI hardware port.

MOD_SQSYNTH

Indicates the device is a square wave synthesizer.

MOD_FMSYNTH

Indicates the device is an FM synthesizer.

MOD_MAPPER

Indicates the device is the Microsoft MIDI Mapper.

wVoices

Specifies the number of voices supported by an internal synthesizer device. If the device is a port, the field is not meaningful and will be set to 0.

wNotes

Specifies the maximum number of simultaneous notes that may be played by an internal synthesizer device. If the device is a port, the field is not meaningful and will be set to 0.

wChannelMask

Specifies the channels that an internal synthesizer device responds to, where the least significant bit refers to channel 0 and the most significant bit to channel 15. Port devices transmit on all channels and so will set this field to 0xFFFF.

dwSupport

Specifies optional functionality supported by the device.

MIDICAPS_VOLUME

Supports volume control.

MIDICAPS_LRVOLUME

Supports separate left and right volume control.

MIDICAPS_CACHE

Supports patch caching.

Comments

If a device supports volume changes, the MIDICAPS_VOLUME flag will be set for the **dwSupport** field. If a device supports separate volume changes on the left and right channels, both the MIDICAPS_VOLUME and the MIDICAPS_LRVOLUME flags will be set for this field.

See Also

midiOutGetDevCaps

MMCKINFO

This structure contains information about a chunk in a RIFF file.

```
typedef struct _MMCKINFO {
    FOURCC ckid;
    DWORD cksize;
    FOURCC fccType;
    DWORD dwDataOffset;
    DWORD dwFlags;
} MMCKINFO;
```

Fields

The **MMCKINFO** structure contains the following fields:

ckid

Specifies the chunk ID of the chunk.

cksize

Specifies the size of the data field of the chunk. The size of the data field does not include the four-byte chunk ID, the four-byte chunk size, or the optional pad byte at the end of the data field.

fccType

Specifies the form type for "RIFF" chunks or the list type for "LIST" chunks.

dwDataOffset

Specifies the file offset of the beginning of the chunk's data field, relative to the beginning of the file.

dwFlags

Specifies flags giving additional information about the chunk. Contains zero or more of the following flags:

MMIO_DIRTY

Indicates that the length of the chunk may have changed and should be updated by mmioAscend. This flag is set when a chunk is created by mmioCreateChunk.

MMIOINFO

This structure contains the current state of a file opened with mmioOpen.

```
typedef struct _MMIOINFO {
    DWORD dwFlags;
    FOURCC fccIOProc;
    LPMMIOPROC pIOProc;
    UINT wErrorRet;
    HTASK htask;
    LONG cchBuffer;
    HPSTR pchBuffer;
    HPSTR pchNext;
    HPSTR pchEndRead;
    HPSTR pchEndWrite;
    LONG lBufOffset;
    LONG lDiskOffset;
    DWORD adwInfo[4];
    DWORD dwReserved1;
    DWORD dwReserved2;
    HMMIO hmmio;
} MMIOINFO;
```

Fields

The **MMIOINFO** structure contains the following fields:

dwFlags

Specifies options indicating how a file was opened:

MMIO_READ
: The file was opened only for reading.

MMIO_WRITE
: The file was opened only for writing.

MMIO_READWRITE
: The file was opened for both reading and writing.

MMIO_COMPAT
: The file was opened with compatibility mode, allowing any process on a given machine to open the file any number of times.

MMIO_EXCLUSIVE
: The file was opened with exclusive mode, denying other processes both read and write access to the file.

MMIO_DENYWRITE
: Other processes are denied write access to the file.

MMIO_DENYREAD
: Other processes are denied read access to the file.

MMIO_DENYNONE
: Other processes are not denied read or write access to the file.

MMIO_CREATE
: mmioOpen was directed to create the file, or truncate it to zero length if it already existed.

MMIO_ALLOCBUF
The file's I/O buffer was allocated by mmioOpen or mmioSetBuffer.

fccIOProc
Specifies the four-character code identifying the file's I/O procedure. If the I/O procedure is not an installed I/O procedure, **fccIOProc** is NULL.

pIOProc
Specifies the address of the file's I/O procedure.

wErrorRet
Holds the extended error value from mmioOpenif mmioOpen returns NULL. **wErrorRet** is not used to return extended error information from any other functions.

htask
handle to a local I/O procedure. MCI devices that perform file I/O in the background and need an I/O procedure can locate a local I/O procedure with this handle.

cchBuffer
Specifies the size of the file's I/O buffer in bytes. If the file does not have an I/O buffer, this field is zero.

pchBuffer
Specifies the address of the file's I/O buffer. If the file is unbuffered, **pchBuffer** is NULL.

pchNext
Specifies a huge pointer to the next location in the I/O buffer to be read or written. If no more bytes can be read without calling mmioAdvance or mmioRead, then this field points to **pchEndRead**. If no more bytes can be written without calling mmioAdvanceor mmioWrite, then this field points to **pchEndWrite**.

pchEndRead
Specifies a pointer to the location that is one byte past the last location in the buffer that can be read.

pchEndWrite
Specifies a pointer to the location that is one byte past the last location in the buffer that can be written.

lBufOffset
Reserved for internal use by MMIO functions.

lDiskOffset
Specifies the current file position. The current file position is an offset in bytes from the beginning of the file. I/O procedures are responsible for maintaining this field.

adwInfo[4]
Contained state information maintained by the I/O procedure. I/O procedures can also use these fields to transfer information from the caller to the I/O procedure when the caller opens a file.

dwReserved1
Reserved for internal use by MMIO functions.

dwReserved2
Reserved for internal use by MMIO functions.

hmmio
Specifies the MMIO handle to the open file. I/O procedures can use this handle when calling

other MMIO functions.

See Also

mmioGetInfo

MMTIME

General purpose structure for timing information.

```
typedef struct mmtime_tag {
    UINT wType;
    union {
        DWORD ms;
        DWORD sample;
        DWORD cb;
        struct {
            BYTE hour;
            BYTE min;
            BYTE sec;
            BYTE frame;
            BYTE fps;
            BYTE dummy;
        } smpte;
        struct {
            DWORD songptrpos;
        } midi;
    } u;
} MMTIME;
```

Fields

The **MMTIME** structure contains the following fields:

wType

Specifies the type of the union. This field must contain one of the following values:

TIME_MS
: Time counted in milliseconds.

TIME_SAMPLES
: Number of wave samples.

TIME_BYTES
: Current byte offset.

TIME_SMPTE
: SMPTE time.

TIME_MIDI
: MIDI time.

u

The contents of the union. The following fields are contained in union **u**:

ms
: Milliseconds. Used when **wType** is TIME_MS.

sample
: Samples. Used when **wType** is TIME_SAMPLES.

cb
: Byte count. Used when **wType** is TIME_BYTES.

smpte

SMPTE time. Used when **wType** is TIME_SMPTE. The following fields are contained in structure **smpte**:

hour
Hours.

min
Minutes.

sec
Seconds.

frame
Frames.

fps
Frames per second (24, 25, 29(30 drop) or 30).

dummy
Dummy byte for alignment.

midi
MIDI time. Used when **wType** is TIME_MIDI. The following field is contained in structure **midi**:

songptrpos
Song pointer position.

PCMWAVEFORMAT

The **PCMWAVEFORMAT** structure describes the data format for PCM waveform data.

```
typedef struct pcmwaveformat_tag {
    WAVEFORMAT wf;
    WORD wBitsPerSample;
} PCMWAVEFORMAT;
```

Fields

The **PCMWAVEFORMAT** structure contains the following fields:

wf

Specifies a WAVEFORMAT structure containing general information about the format of the waveform data.

wBitsPerSample

Specifies the number of bits per sample.

See Also

WAVEFORMAT

TIMECAPS

Structure for returning information about the resolution of the timer.

```
typedef struct timecaps_tag {
    UINT wPeriodMin;
    UINT wPeriodMax;
} TIMECAPS;
```

Fields

The **TIMECAPS** structure contains the following fields:

wPeriodMin

Minimum period supported by timer.

wPeriodMax

Maximum period supported by timer.

See Also

timeGetDevCaps

WAVEFORMAT

The **WAVEFORMAT** structure describes the format of waveform data. Only format information common to all waveform data formats is included in this structure. For formats that require additional information, this structure is included as a field in another data structure along with the additional information.

```
typedef struct waveformat_tag {
    WORD wFormatTag;
    WORD nChannels;
    DWORD nSamplesPerSec;
    DWORD nAvgBytesPerSec;
    WORD nBlockAlign;
} WAVEFORMAT;
```

Fields

The **WAVEFORMAT** structure contains the following fields:

wFormatTag

Specifies the format type. Currently defined format types are as follows:

WAVE_FORMAT_PCM
Waveform data is PCM.

nChannels

Specifies the number of channels in the waveform data. Mono data uses 1 channel and stereo data uses 2 channels.

nSamplesPerSec

Specifies the sample rate in samples per second.

nAvgBytesPerSec

Specifies the required average data transfer rate in bytes per second.

nBlockAlign

Specifies the block alignment in bytes. The block alignment is the minimum atomic unit of data.

Comments

For PCM data, the block alignment is the number of bytes used by a single sample, including data for both channels if the data is stereo. For example, the block alignment for 16-bit stereo PCM is 4 bytes (2 channels, 2 bytes per sample).

See Also

PCMWAVEFORMAT

WAVEHDR

The **WAVEHDR** structure defines the header used to identify a waveform data buffer.

```
typedef struct wavehdr_tag {
    LPSTR lpData;
    DWORD dwBufferLength;
    DWORD dwBytesRecorded;
    DWORD dwUser;
    DWORD dwFlags;
    DWORD dwLoops;
    struct wavehdr_tag far * lpNext;
    DWORD reserved;
} WAVEHDR;
```

Fields

The **WAVEHDR** structure contains the following fields:

lpData
Specifies a far pointer to the waveform data buffer.

dwBufferLength
Specifies the length of the data buffer.

dwBytesRecorded
When the header is used in input, this specifies how much data is in the buffer.

dwUser
Specifies 32 bits of user data.

dwFlags
Specifies flags giving information about the data buffer.

WHDR_DONE
Set by the device driver to indicate that it is finished with the data buffer and is returning it to the application.

WHDR_BEGINLOOP
Specifies that this buffer is the first buffer in a loop. This flag is only used with output data buffers.

WHDR_ENDLOOP
Specifies that this buffer is the last buffer in a loop. This flag is only used with output data buffers.

WHDR_PREPARED
Set by Windows to indicate that the data buffer has been prepared with waveInPrepareHeader or waveOutPrepareHeader.

dwLoops
Specifies the number of times to play the loop. This parameter is used only with output data buffers.

lpNext
Is reserved and should not be used.

reserved
Is reserved and should not be used.

Comments

Use the WHDR_BEGINLOOP and WHDR_ENDLOOP flags in the **dwFlags**field to specify the beginning and ending data blocks for looping. To loop on a single block, specify both flags for the same block. Use the **dwLoops** field in the **WAVEHDR** structure for the first block in the loop to specify the number of times to play the loop.

WAVEINCAPS

The **WAVEINCAPS** structure describes the capabilities of a waveform input device.

```
typedef struct waveincaps_tag {
    UINT wMid;
    UINT wPid;
    VERSION vDriverVersion;
    char szPname[MAXPNAMELEN];
    DWORD dwFormats;
    UINT wChannels;
} WAVEINCAPS;
```

Fields

The **WAVEINCAPS** structure contains the following fields:

wMid

Specifies a manufacturer ID for the device driver for the waveform input device. Manufacturer IDs are defined in Manufacturer and Product IDs.

wPid

Specifies a product ID for the waveform input device. Product IDs are defined in Manufacturer and Product IDs.

vDriverVersion

Specifies the version number of the device driver for the waveform input device. The high-order byte is the major version number, and the low-order byte is the minor version number.

szPname[MAXPNAMELEN]

Specifies the product name in a NULL-terminated string.

dwFormats

Specifies which standard formats are supported. The supported formats are specified with a logical OR of the following flags:

WAVE_FORMAT_1M08
    11.025 kHz, Mono, 8-bit

WAVE_FORMAT_1S08
    11.025 kHz, Stereo, 8-bit

WAVE_FORMAT_1M16
    11.025 kHz, Mono, 16-bit

WAVE_FORMAT_1S16
    11.025 kHz, Stereo, 16-bit

WAVE_FORMAT_2M08
    22.05 kHz, Mono, 8-bit

WAVE_FORMAT_2S08
    22.05 kHz, Stereo, 8-bit

WAVE_FORMAT_2M16
    22.05 kHz, Mono, 16-bit

WAVE_FORMAT_2S16
    22.05 kHz, Stereo, 16-bit

WAVE_FORMAT_4M08
44.1 kHz, Mono, 8-bit

WAVE_FORMAT_4S08
44.1 kHz, Stereo, 8-bit

WAVE_FORMAT_4M16
44.1 kHz, Mono, 16-bit

WAVE_FORMAT_4S16
44.1 kHz, Stereo, 16-bit

wChannels
Specifies whether the device supports mono (1) or stereo (2) input.

See Also
waveInGetDevCaps

WAVEOUTCAPS

The **WAVEOUTCAPS** structure describes the capabilities of a waveform output device.

```
typedef struct waveoutcaps_tag {
    UINT wMid;
    UINT wPid;
    VERSION vDriverVersion;
    char szPname[MAXPNAMELEN];
    DWORD dwFormats;
    UINT wChannels;
    DWORD dwSupport;
} WAVEOUTCAPS;
```

Fields

The **WAVEOUTCAPS** structure contains the following fields:

wMid

Specifies a manufacturer ID for the device driver for the waveform output device. Manufacturer IDs are defined in Manufacturer and Product IDs.

wPid

Specifies a product ID for the waveform output device. Product IDs are defined in Manufacturer and Product IDs.

vDriverVersion

Specifies the version number of the device driver for the waveform output device. The high-order byte is the major version number, and the low-order byte is the minor version number.

szPname[MAXPNAMELEN]

Specifies the product name in a NULL-terminated string.

dwFormats

Specifies which standard formats are supported. The supported formats are specified with a logical OR of the following flags:

WAVE_FORMAT_1M08
: 11.025 kHz, Mono, 8-bit

WAVE_FORMAT_1S08
: 11.025 kHz, Stereo, 8-bit

WAVE_FORMAT_1M16
: 11.025 kHz, Mono, 16-bit

WAVE_FORMAT_1S16
: 11.025 kHz, Stereo, 16-bit

WAVE_FORMAT_2M08
: 22.05 kHz, Mono, 8-bit

WAVE_FORMAT_2S08
: 22.05 kHz, Stereo, 8-bit

WAVE_FORMAT_2M16
: 22.05 kHz, Mono, 16-bit

WAVE_FORMAT_2S16
: 22.05 kHz, Stereo, 16-bit

WAVE_FORMAT_4M08
44.1 kHz, Mono, 8-bit

WAVE_FORMAT_4S08
44.1 kHz, Stereo, 8-bit

WAVE_FORMAT_4M16
44.1 kHz, Mono, 16-bit

WAVE_FORMAT_4S16
44.1 kHz, Stereo, 16-bit

wChannels
Specifies whether the device supports mono (1) or stereo (2) output.

dwSupport
Specifies optional functionality supported by the device.

WAVECAPS_PITCH
Supports pitch control.

WAVECAPS_PLAYBACKRATE
Supports playback rate control.

WAVECAPS_SYNC
Specifies that the driver is synchronous and will block while playing a buffer.

WAVECAPS_VOLUME
Supports volume control.

WAVECAPS_LRVOLUME
Supports separate left and right volume control.

Comments
If a device supports volume changes, the WAVECAPS_VOLUME flag will be set for the **dwSupport** field. If a device supports separate volume changes on the left and right channels, both the WAVECAPS_VOLUME and the WAVECAPS_LRVOLUME flags will be set for this field.

See Also
waveOutGetDevCaps

MM_MCINOTIFY

This message is sent to a window to notify an application that an MCI device has completed an operation. MCI devices send this message only when the MCI_NOTIFY flag is used with an MCI command message or when the notify flag is used with an MCI command string.

Parameters

WPARAM *wParam*

Contains one of the following message:

MCI_NOTIFY_ABORTED

Specifies that the device received a command that prevented the current conditions for initiating the callback from being met. If a new command interrupts the current command and it also requests notification, the device will send only this message and not MCI_NOTIFY_SUPERCEDED.

MCI_NOTIFY_SUCCESSFUL

Specifies that the conditions initiating the callback have been met.

MCI_NOTIFY_SUPERSEDED

Specifies that the device received another command with the MCI_NOTIFY flag set and the current conditions for initiating the callback have been superseded.

MCI_NOTIFY_FAILURE

Specifies that a device error occurred while the device was executing the command.

LPARAM *lParam*

The low-order word specifies the ID of the device initiating the callback.

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

A device returns the flag MCI_NOTIFY_SUCCESSFUL with MM_MCINOTIFY when the action for a command finishes. For example, a CD audio device uses this flag for notification for MCI_PLAY when the device finishes playing. The MCI_PLAY command completes successfully only when it reaches the specified end position or reaches the end of the media. Similarly, MCI_SEEK and MCI_RECORD do not return MCI_NOTIFY_SUCCESSFUL until they reach the specified end position or reach the end of the media.

A device returns the flag MCI_NOTIFY_ABORTED with MM_MCINOTIFY only when it receives a command that prevents it from meeting the notification conditions. For example, the command MCI_PLAY would not abort notification for a previous play command provided that the new command does not change the play direction or change the ending position for the play command with an active notify. The MCI_RECORD and MCI_SEEK commands behave similarly.

MCI also does not send MCI_NOTIFY_ABORTED when MCI_PLAY or MCI_RECORD is paused with MCI_PAUSE. Sending the MCI_RESUME command will let them continue to meet the callback conditions.

When your application requests notification for a command, check the error return of mciSendCommand or mciSendString. If these functions encounter an error and return a nonzero value,MCI will not set notification for the command.

MCI_BREAK

This MCI command message sets a break key for an MCI device. MCI supports this message directly rather than passing it to the device.

Parameters

DWORD *dwParam1*

The following flags apply to all devices:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpBreak*.

MCI_WAIT

Specifies that the break operation should finish before MCI returns control to the application.

MCI_BREAK_KEY

Indicates the nVirtKey field of the data structure identified by *lpBreak* specifies the virtual key code used for the break key. By default, MCI assigns CTRL+BREAK as the break key. This flag is required if MCI_BREAK_OFF is not specified.

MCI_BREAK_HWND

Indicates the hwndBreak field of the data structure identified by *lpBreak* contains a window handle which must be the current window in order to enable break detection for that MCI device. This is usually the application's main window. If omitted, MCI does not check the window handle of the current window.

MCI_BREAK_OFF

Used to disable any existing break key for the indicated device.

DWORD *dwParam2*

Specifies a far pointer to the MCI_BREAK_PARMS data structure.

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

You might have to press the break key multiple times to interrupt a wait operation. Pressing the break key after a device wait is broken can send the break to an application. If an application has an action defined for the virtual key code, then it can inadvertantly respond to the break. For example, an application using VK_CANCEL for an accelerator key can respond to the default CTRL+BREAK key if it is pressed after a wait is canceled.

MCI_CLOSE

This MCI command message releases access to a device or device element. All devices respond to this message.

Parameters

DWORD *dwFlags*

The following flags apply to all devices:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpDefault*.

MCI_WAIT

Specifies that the close operation should finish before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpDefault*

Specifies a far pointer to the MCI_GENERIC_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

Exiting an application without closing any MCI devices it has opened can leave the device opened and unaccessible. Your application should explicitly close each device or device element when it is finished with it. MCI unloads the device when all instances of the device or all device elements are closed.

See Also

MCI_OPEN

MCI_COPY

This MCI command message copies data to the Clipboard. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_COPY:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpCopy*.

MCI_WAIT

Specifies that the copy should finish before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpCopy*

Specifies a far pointer to an MCI_GENERIC_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

See Also

MCI_CUT, MCI_DELETE, MCI_PASTE

MCI_CUE

This MCI command message cues a device so that playback or recording begins with minimum delay. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_CUE:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpDefault*.

MCI_WAIT

Specifies that the cue operation should finish before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpDefault*

Specifies a far pointer to the MCI_GENERIC_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Waveform Audio Extensions

DWORD *dwFlags*

The following additional flags apply to wave audio devices:

MCI_WAVE_INPUT

Specifies that a wave input device should be cued.

MCI_WAVE_OUTPUT

Specifies that a wave output device should be cued. This is the default flag if a flag is not specified.

LPMCI_GENERIC_PARMS *lpDefault*

Specifies a far pointer to the MCI_GENERIC_PARMS data structure.

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

See Also

MCI_SEEK, MCI_PLAY, MCI_RECORD

MCI_CUT

This MCI command message removes data from the MCI element and copies it to the Clipboard. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_CUT:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpCut*.

MCI_WAIT

Specifies that the cut operation should finish before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpCut*

Specifies a far pointer to an MCI_GENERIC_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

See Also

MCI_COPY, MCI_DELETE, MCI_PASTE

MCI_DELETE

This MCI command message removes data from the MCI element. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_DELETE:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpDelete*.

MCI_WAIT

Specifies that the delete operation should finish before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpCut*

Specifies a far pointer to an MCI_GENERIC_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Wave Audio Extensions

DWORD *dwFlags*

The following extensions apply to wave audio devices:

MCI_FROM

Specifies that a beginning position is included in the dwFrom field of the data structure identified by *lpDelete*. The units assigned to the position values is specified with the MCI_SET_TIME_FORMAT flag of the MCI_SET command.

MCI_TO

Specifies that an ending position is included in the dwTo field of the data structure identified by *lpDelete*. The units assigned to the position values is specified with the MCI_SET_TIME_FORMAT flag of the MCI_SET command.

LPMCI_WAVE_DELETE_PARMS *lpDelete*

Specifies a far pointer to an MCI_WAVE_DELETE_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

See Also

MCI_COPY, MCI_DELETE, MCI_PASTE

MCI_ESCAPE (VIDEODISC)

This MCI command message sends a string directly to the device. This message is part of the videodisc command set. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_COMMAND:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpEscape*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

MCI_VD_ESCAPE_STRING

Indicates a command string is specified in the lpstrCommand field of the data structure identified by *lpEscape*. This flag is required.

LPMCI_VD_ESCAPE_PARMS *lpEscape*

Specifies a far pointer to the MCI_VD_ESCAPE_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Comments

The data sent with **MCI_ESCAPE** is device dependent and is usually passed directly to the hardware associated with the device.

Return value

Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_FREEZE (VIDEO OVERLAY)

This MCI command message freezes motion on the display. This command is part of the video overlay command set. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_FREEZE:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpFreeze*.

MCI_WAIT

Specifies that the freeze operation should finish before MCI returns control to the application.

MCI_OVLY_RECT

Specifies that the rc field of the data structure identified by *lpFreeze* contains a valid rectangle. If this flag is not specified, the device driver will freeze the entire frame.

LPMCI_OVLY_RECT_PARMS *lpFreeze*

Specifies a far pointer to a MCI_OVLY_RECT_PARMS data structure. (Devices with additional parameters might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

See Also

MCI_UNFREEZE

MCI_GETDEVCAPS

This MCI command message is used to obtain static information about a device. All devices must respond to this message. The parameters and flags available for this message depend on the selected device. Information is returned in the dwReturn field of the data structure identified by *lpCapsParms*.

Parameters

DWORD *dwFlags*

The following standard and command-specific flags apply to all devices:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpCapsParms*.

MCI_WAIT

Specifies that the query operation should finish before MCI returns control to the application.

MCI_GETDEVCAPS_ITEM

Specifies that the dwItem field of the data structure identified by *lpCapsParms* contains a constant specifying which device capability to obtain. The following constants define which capability to return in the dwReturn field of the data structure:

MCI_GETDEVCAPS_CAN_EJECT

The dwReturn field is set to TRUE if the device can eject the media; otherwise, it is set to FALSE.

MCI_GETDEVCAPS_CAN_PLAY

The dwReturn field is set to TRUE if the device can play the media; otherwise, it is set to FALSE.

If a device specifies TRUE, it implies the device supports MCI_PAUSE and MCI_STOP as well as MCI_PLAY.

MCI_GETDEVCAPS_CAN_RECORD

The dwReturn field is set to TRUE if the device supports recording; otherwise, it is set to FALSE.

If a device specifies TRUE, it implies the device supports MCI_PAUSE and MCI_STOP as well as MCI_RECORD.

MCI_GETDEVCAPS_CAN_SAVE

The dwReturn field is set to TRUE if the device can save a file; otherwise, it is set to FALSE.

MCI_GETDEVCAPS_COMPOUND_DEVICE

The dwReturn field is set to TRUE if the device uses device elements; otherwise, it is set to FALSE.

MCI_GETDEVCAPS_DEVICE_TYPE

The dwReturn field is set to one of the following values indicating the device type:

* MCI_DEVTYPE_ANIMATION
* MCI_DEVTYPE_CD_AUDIO
* MCI_DEVTYPE_DAT

* MCI_DEVTYPE_DIGITAL_VIDEO
* MCI_DEVTYPE_OTHER
* MCI_DEVTYPE_OVERLAY
* MCI_DEVTYPE_SCANNER
* MCI_DEVTYPE_SEQUENCER
* MCI_DEVTYPE_VIDEODISC
* MCI_DEVTYPE_VIDEOTAPE
* MCI_DEVTYPE_WAVEFORM_AUDIO

MCI_GETDEVCAPS_HAS_AUDIO
The dwReturn field is set to TRUE if the device has audio output; otherwise, it is set to FALSE.

MCI_GETDEVCAPS_HAS_VIDEO
The dwReturn field is set to TRUE if the device has video output; otherwise, it is set to FALSE.

For example, the field is set to TRUE for devices that support the animation or videodisc command set.

MCI_GETDEVCAPS_USES_FILES
The dwReturn field is set to TRUE if the device requires a filename as its element name; otherwise, it is set to FALSE.

Only compound devices use files.

LPMCI_GETDEVCAPS_PARMS *lpCapsParms*
Specifies a far pointer to the MCI_GETDEVCAPS_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Animation Extensions

DWORD *dwFlags*
The following extensions apply to animation devices:

MCI_GETDEVCAPS_ITEM
Specifies that the dwItem field of the data structure identified by *lpCapsParms* contains a constant specifying which device capability to obtain. The following additional device-capability constants are defined for animation devices and specify which value to return in the dwReturn field of the data structure:

MCI_ANIM_GETDEVCAPS_CAN_REVERSE
The dwReturn field is set to TRUE if the device can play in reverse; otherwise, it is set to FALSE.

MCI_ANIM_GETDEVCAPS_CAN_STRETCH
The dwReturn field is set to TRUE if the device can stretch the image to fill the frame; otherwise, it is set to FALSE.

MCI_ANIM_GETDEVCAPS_FAST_RATE
The dwReturn field is set to the standard fast play rate in frames per second.

MCI_ANIM_GETDEVCAPS_MAX_WINDOWS
The dwReturn field is set to the maximum number of windows that the device can handle simultaneously.

MCI_ANIM_GETDEVCAPS_NORMAL_RATE
The dwReturn field is set to the normal rate of play in frames per second.

MCI_ANIM_GETDEVCAPS_PALETTES
The dwReturn field is set to TRUE if the device can return a palette handle; otherwise, it is set to FALSE.

MCI_ANIM_GETDEVCAPS_SLOW_RATE
The dwReturn field is set to the standard slow play rate in frames per second.

LPMCI_GETDEVCAPS_PARMS *lpCapsParms*
Specifies a far pointer to the MCI_GETDEVCAPS_PARMS data structure.

Videodisc Extensions

DWORD *dwFlags*
The following extensions apply to videodisc devices:

MCI_GETDEVCAPS_ITEM
Specifies that the dwItem field of the data structure identified by *lpCapsParms* contains a constant specifying which device capability to obtain. The following additional device-capability constants are defined for videodisc devices and specify which value to return in the dwReturn field of the data structure:

MCI_VD_GETDEVCAPS_CAN_REVERSE
The dwReturn field is set to TRUE if the videodisc player can play in reverse; otherwise, it is set to FALSE.

Some players can play CLV discs in reverse as well as CAV discs.

MCI_VD_GETDEVCAPS_FAST_RATE
The dwReturn field is set to the standard fast play rate in frames per second.

MCI_VD_GETDEVCAPS_NORMAL_RATE
The dwReturn field is set to the normal play rate in frames per second.

MCI_VD_GETDEVCAPS_SLOW_RATE
The dwReturn field is set to the standard slow play rate in frames per second.

MCI_VD_GETDEVCAPS_CLV
Indicates the information requested applies to CLV format discs. By default, the capabilities apply to the current disc.

MCI_VD_GETDEVCAPS_CAV
Indicates the information requested applies to CAV format discs. By default, the capabilities apply to the current disc.

LPMCI_GETDEVCAPS_PARMS *lpCapsParms*
Specifies a far pointer to the MCI_GETDEVCAPS_PARMS data structure.

Video Overlay Extensions

DWORD *dwFlags*
The following extensions apply to video overlay devices:

MCI_GETDEVCAPS_ITEM
Specifies that the dwItem field of the data structure identified by *lpCapsParms* contains a constant specifying which device capability to obtain. The following additional device-capability constants are defined for video overlay devices and specify which value to return in the dwReturn field of the data structure:

MCI_OVLY_GETDEVCAPS_CAN_FREEZE
The dwReturn field is set to TRUE if the device can freeze the image; otherwise, it is set to FALSE.

MCI_OVLY_GETDEVCAPS_CAN_STRETCH
The dwReturn field is set to TRUE if the device can stretch the image to fill the frame; otherwise, it is set to FALSE.

MCI_OVLY_GETDEVCAPS_MAX_WINDOWS
The dwReturn field is set to the maximum number of windows that the device can handle simultaneously.

LPMCI_GETDEVCAPS_PARMS *lpCapsParms*
Specifies a far pointer to the MCI_GETDEVCAPS_PARMS data structure.

Waveform Audio Extensions

DWORD *dwFlags*
The following extended flag applies to waveform audio devices:

MCI_GETDEVCAPS_ITEM
Specifies that the dwItem field of the data structure identified by *lpCapsParms* contains a constant specifying which device capability to obtain. The following additional device-capability constants are defined for waveform audio devices and specify which value to return in the dwReturn field of the data structure:

MCI_WAVE_GETDEVCAPS_INPUT
The dwReturn field is set to the total number of waveform input (recording) devices.

MCI_WAVE_GETDEVCAPS_OUTPUT
The dwReturn field is set to the total number of waveform output (playback) devices.

LPMCI_GETDEVCAPS_PARMS *lpCapsParms*
Specifies a far pointer to the MCI_GETDEVCAPS_PARMS data structure.

Return Value
Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_INFO

This MCI command message obtains string information from a device. All devices respond to this message. The parameters and flags available for this message depend on the selected device. Information is returned in the lpstrReturn field of the data structure identified by *lpInfo*. The dwRetSize field specifies the buffer length for the return data.

Parameters

DWORD *dwFlags*

The following standard and command-specific flags apply to all devices:

MCI_NOTIFY
: Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpInfo*.

MCI_WAIT
: Specifies that the query operation should finish before MCI returns control to the application.

MCI_INFO_PRODUCT
: Obtains a description of the hardware associated with a device. Devices should supply a description that identifies both the driver and the hardware used.

LPMCI_INFO_PARMS *lpInfo*

Specifies a far pointer to the MCI_INFO_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Animation Extensions

DWORD *dwFlags*

The following additional flags apply to animation devices:

MCI_INFO_FILE
: Obtains the filename of the current file. This flag is only supported by devices that return TRUE to the MCI_GETDEVCAPS_USES_FILES query.

MCI_ANIM_INFO_TEXT
: Obtains the window caption.

LPMCI_INFO_PARMS *lpInfo*

Specifies a far pointer to the MCI_INFO_PARMS data structure.

Video Overlay Extensions

DWORD *dwFlags*

The following additional flags apply to video overlay devices:

MCI_INFO_FILE
: Obtains the filename of the current file. This flag is only supported by devices that return TRUE to the MCI_GETDEVCAPS_USES_FILES query.

MCI_OVLY_INFO_TEXT
: Obtains the caption of the window associated with the overlay device.

LPMCI_INFO_PARMS *lpInfo*

Specifies a far pointer to the MCI_INFO_PARMS data structure.

Waveform Audio Extensions

DWORD *dwFlags*

The following additional flags apply to waveform audio devices:

MCI_INFO_FILE

Obtains the filename of the current file. This flag is supported by devices that return TRUE to the MCI_GETDEVCAPS_USES_FILES query.

MCI_WAVE_INPUT

Obtains the product name of the current input.

MCI_WAVE_OUTPUT

Obtains the product name of the current output.

LPMCI_INFO_PARMS *lpInfo*

Specifies a far pointer to the MCI_INFO_PARMS data structure.

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_LOAD

This MCI command message loads a file. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_LOAD:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpLoad*.

MCI_WAIT

Specifies that the load operation should finish before MCI returns control to the application.

MCI_LOAD_FILE

Indicates the lpfilename field of the data structure identified by *lpLoad* contains a pointer to a buffer containing the file name.

LPMCI_LOAD_PARMS *lpLoad*

Specifies a far pointer to the MCI_LOAD_PARMS data structure. (Devices with additional parameters might replace this data structure with a device-specific data structure.)

Video Overlay Extensions

DWORD *dwFlags*

The following additional flags apply to video overlay devices supporting MCI_LOAD:

MCI_OVLY_RECT

Specifies that the rc field of the data structure identified by *lpLoad* contains a valid display rectangle that identifies the area of the video buffer to update.

LPMCI_OVLY_LOAD_PARMS *lpLoad*

Specifies a far pointer to a MCI_OVLY_LOAD_PARMS data structure.

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

This command applies to video overlay devices.

See Also

MCI_SAVE

MCI_OPEN

This MCI command message initializes a device or device element. All devices respond to this message. The parameters and flags available for this message depend on the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices:

MCI_NOTIFY
: Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpOpen*.

MCI_WAIT
: Specifies that the open operation should finish before MCI returns control to the application.

MCI_OPEN_ALIAS
: Specifies that an alias is included in the lpstrAlias field of the data structure identified by *lpOpen*.

MCI_OPEN_SHAREABLE
: Specifies that the device or device element should be opened as shareable.

MCI_OPEN_TYPE
: Specifies that a device type name or constant is included in the lpstrDeviceType field of the data structure identified by *lpOpen*.

MCI_OPEN_TYPE_ID
: Specifies that the low-order word of the lpstrDeviceType field of the associated data structure contains a standard MCI device type ID and the high-order word optionally contains the ordinal index for the device. Use this flag with the MCI_OPEN_TYPE flag.

LPMCI_OPEN_PARMS *lpOpen*

Specifies a far pointer to the MCI_OPEN_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Flags for Compound Devices

DWORD *dwFlags*

The following additional flags apply to compound devices:

MCI_OPEN_ELEMENT
: Specifies that an element name is included in the lpstrElementName field of the data structure identified by *lpOpen*.

MCI_OPEN_ELEMENT_ID
: Specifies that the lpstrElementName field of the data structure identified by *lpOpen* is interpreted as a DWORD and has meaning internal to the device. Use this flag with the MCI_OPEN_ELEMENT flag.

LPMCI_OPEN_PARMS *lpOpen*

Specifies a far pointer to the MCI_OPEN_PARMS data structure. (Devices with additional parameters might replace this data structure with a device-specific data structure.)

Animation Extensions

DWORD *dwFlags*

The following flags apply to animation devices:

MCI_ANIM_OPEN_EXPANDDIBS
Specifies that the device should expand bitmaps while loading the animation rather than while playing the animation.

MCI_ANIM_OPEN_NOSTATIC
Specifies that the device should reduce the number of static (system) colors in the palette to two.

MCI_ANIM_OPEN_PARENT
Indicates the parent window handle is specified in the hWndParent field of the data structure identified by *lpOpen*. The parent window handle is required for some window styles.

MCI_ANIM_OPEN_WS
Indicates a window style is specified in the dwStyle field of the data structure identified by *lpOpen*. The dwStyle field specifies the style of the window that the driver will create and display if the application does not provide one. The style parameter takes an integer that defines the window style. These constants are the same as the ones in WINDOWS.H (such as WS_CHILD, WS_OVERLAPPEDWINDOW, or WS_POPUP).

LPMCI_ANIM_OPEN_PARMS *lpOpen*
Specifies a far pointer to the MCI_ANIM_OPEN_PARMS data structure.

Video Overlay Extensions

DWORD *dwFlags*
The following flags apply to video overlay devices:

MCI_OVLY_OPEN_PARENT
Indicates the parent window handle is specified in the hWndParent field of the data structure identified by *lpOpen*.

MCI_OVLY_OPEN_WS
Indicates a window style is specified in the dwStyle field of the data structure identified by *lpOpen*. The dwStyle field specifies the style of the window that the driver will create and display if the application does not provide one. The style parameter takes an integer that defines the window style. These constants are the same as those in WINDOWS.H (for example, WS_CHILD, WS_OVERLAPPEDWINDOW, or WS_POPUP).

LPMCI_OVLY_OPEN_PARMS *lpOpen*
Specifies a far pointer to the MCI_OVLY_OPEN_PARMS data structure.

Waveform Audio Extensions

The MCIWAVE device included with Windows requires an asynchronous waveform driver. It does not work with synchronous drivers like the PC Speaker driver.

DWORD *dwFlags*
The following flags apply to waveform audio devices:

MCI_WAVE_OPEN_BUFFER
Indicates a buffer length is specified in the dwBufferSeconds field of the data structure identified by *lpOpen*. The default size of the buffer is set when the waveform audio device is installed or configured. Typically the buffer size is set to 4 seconds. With the MCIWAVE device, the minimum size is 2 seconds and the maximum size is 9 seconds.

LPMCI_WAVE_OPEN_PARMS *lpOpen*
Specifies a far pointer to the MCI_WAVE_OPEN_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if the open is successful. If an error occurs, it returns the following values:

MCIERR_CANNOT_LOAD_DRIVER
Error loading media device driver.

MCIERR_DEVICE_OPEN
The device name is in use by this task. Use a unique alias.

MCIERR_DUPLICATE_ALIAS
The specified alias is an open device in this task.

MCIERR_EXTENSION_NOT_FOUND
Cannot deduce a device type from the given extension.

MCIERR_FILENAME_REQUIRED
A valid filename is required.

MCIERR_MISSING_PARAMETER
Required parameter is missing.

MCIERR_MUST_USE_SHAREABLE
The device is already open; use the shareable flag with each open.

MCIERR_NO_ELEMENT_ALLOWED
An element name cannot be used with this device.

Comments

If MCI_OPEN_SHAREABLE is not specified when a device or device element is initially opened, then all subsequent MCI_OPEN messages to the device or device element will fail. If the device or device element is already open, and this flag is not specified, the call will fail even if the first open command specified MCI_OPEN_SHAREABLE. Files for the MCISEQ and MCIWAVE devices are nonshareable.

Case is ignored in the device name, but there must not be any leading or trailing blanks.

To use automatic type selection (via the [mci extensions] section of the WIN.INI file), assign the file name (including file extension) to the lpstrElementName field, assign a NULL pointer to the lpstrDeviceType field, and set the MCI_OPEN_ELEMENT flag.

See Also

MCI_CLOSE

MCI_PASTE

This MCI command message pastes data from the Clipboard into a device element.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_PASTE:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpPaste*.

MCI_WAIT

Specifies that the device should complete the operation before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpPaste*

Specifies a far pointer to the MCI_GENERIC_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

See Also

MCI_CUT, MCI_COPY, MCI_DELETE

MCI_PAUSE

This MCI command message pauses the current action.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_PAUSE:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpDefault*.

MCI_WAIT

Specifies that the device should be paused before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpDefault*

Specifies a far pointer to the MCI_GENERIC_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

The difference between MCI_STOP and MCI_PAUSE depends upon the device. If possible, MCI_PAUSE suspends device operation but leaves the device ready to resume play immediately.

See Also

MCI_PLAY, MCI_RECORD, MCI_RESUME, MCI_STOP

MCI_PLAY

This MCI command message signals the device to begin transmitting output data. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_PLAY:

MCI_NOTIFY
: Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpPlay*.

MCI_WAIT
: Specifies that the play operation should finish before MCI returns control to the application.

MCI_FROM
: Specifies that a starting position is included in the dwFrom field of the data structure identified by *lpPlay*. The units assigned to the position values is specified with the MCI_SET_TIME_FORMAT flag of the MCI_SET command. If MCI_FROM is not specified, the starting position defaults to the current location.

MCI_TO
: Specifies that an ending position is included in the dwTo field of the data structure identified by *lpPlay*. The units assigned to the position values is specified with the MCI_SET_TIME_FORMAT flag of the MCI_SET command. If MCI_TO is not specified, the end position defaults to the end of the media.

LPMCI_PLAY_PARMS *lpPlay*

Specifies a far pointer to an MCI_PLAY_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Animation Extensions

DWORD *dwFlags*

The following additional flags apply to animation devices:

MCI_ANIM_PLAY_FAST
: Specifies to play fast.

MCI_ANIM_PLAY_REVERSE
: Specifies to play in reverse.

MCI_ANIM_PLAY_SCAN
: Specifies to scan quickly.

MCI_ANIM_PLAY_SLOW
: Specifies to play slowly.

MCI_ANIM_PLAY_SPEED
: Specifies that the play speed is included in the dwSpeed field in the data structure identified by *lpPlay*.

LPMCI_ANIM_PLAY_PARMS *lpPlay*

Specifies a far pointer to an MCI_ANIM_PLAY_PARMS data structure.

Videodisc Extensions

DWORD *dwFlags*

The following additional flags apply to videodisc devices:

MCI_VD_PLAY_FAST
Specifies to play fast.

MCI_VD_PLAY_REVERSE
Specifies to play in reverse.

MCI_VD_PLAY_SCAN
Specifies to scan quickly.

MCI_VD_PLAY_SLOW
Specifies to play slowly.

MCI_VD_PLAY_SPEED
Specifies that the play speed is included in the dwSpeed field in the data structure identified by *lpPlay*.

LPMCI_VD_PLAY_PARMS *lpPlay*

Specifies a far pointer to an MCI_VD_PLAY_PARMS data structure.

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

See Also

MCI_CUE, MCI_PAUSE, MCI_RECORD, MCI_RESUME, MCI_SEEK, MCI_STOP

MCI_PUT

This MCI command message sets the source, destination, and frame rectangles. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_PUT:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpDest*.

MCI_WAIT

Specifies that the operation should finish before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpDest*

Specifies a far pointer to an MCI_GENERIC_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Animation Extensions

DWORD *dwFlags*

The following additional flags apply to animation devices supporting MCI_PUT:

MCI_ANIM_RECT

Specifies that the rc field of the data structure identified by *lpDest* contains a valid rectangle. If this flag is not specified, the default rectangle matches the coordinates of the image or window being clipped.

MCI_ANIM_PUT_DESTINATION

Indicates the rectangle defined for MCI_ANIM_RECT specifies the area of the client window used to display an image. The rectangle contains the offset and visible extent of the image relative to the window origin. If the frame is being stretched, the source is stretched to the destination rectangle.

MCI_ANIM_PUT_SOURCE

Indicates the rectangle defined for MCI_ANIM_RECT specifies a clipping rectangle for the animation image. The rectangle contains the offset and extent of the image relative to the image origin.

LPMCI_ANIM_RECT_PARMS *lpDest*

Specifies a far pointer to a MCI_ANIM_RECT_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Video Overlay Extensions

DWORD *dwFlags*

The following additional flags apply to video overlay devices supporting MCI_PUT:

MCI_OVLY_RECT

Specifies that the rc field of the data structure identified by *lpDest* contains a valid display rectangle. If this flag is not specified, the default rectangle matches the coordinates of the video buffer or window being clipped.

MCI_OVLY_PUT_DESTINATION

Indicates the rectangle defined for MCI_OVLY_RECT specifies the area of the client window

used to display an image. The rectangle contains the offset and visible extent of the image relative to the window origin. If the frame is being stretched, the source is stretched to the destination rectangle.

MCI_OVLY_PUT_FRAME
Indicates the rectangle defined for MCI_OVLY_RECT specifies the area of the video buffer used to receive the video image. The rectangle contains the offset and extent of the buffer area relative to the video buffer origin.

MCI_OVLY_PUT_SOURCE
Indicates that the rectangle defined for MCI_OVLY_RECT specifies the area of the video buffer used as the source of the digital image. The rectangle contains the offset and extent of the clipping rectangle for the video buffer relative to its origin.

MCI_OVLY_PUT_VIDEO
Indicates that the rectangle defined for MCI_OVLY_RECT specifies the area of the video source capture by the video buffer. The rectangle contains the offset and extent of the clipping rectangle for the video source relative to its origin.

LPMCI_OVLY_RECT_PARMS *lpDest*
Specifies a far pointer to a MCI_OVLY_RECT_PARMS data structure.

Return Value
Returns zero if successful. Otherwise, it returns an MCI error code.

See Also
MCI_WHERE

MCI_REALIZE (ANIMATION)

This MCI command message tells a graphic device to realize its palette into a device context. This is part of the animation command set. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_REALIZE**:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpRealize*.

MCI_WAIT

Specifies that the palette should be realized before MCI returns control to the application.

MCI_ANIM_REALIZE_BKGD

If this flag is set, the palette is realized as a background palette.

MCI_ANIM_REALIZE_NORM

If this flag is set, the palette is realized normally. This is the default action.

LPMCI_GENERIC_PARMS *lpRealize*

Specifies a far pointer to a MCI_GENERIC_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

This command is supported by devices that return true to the MCI_GETDEVCAPS_PALETTES query.

MCI_RECORD

This MCI command message starts recording from the current position or from the specified position until the specified position. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_RECORD:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpRecord*.

MCI_WAIT

Specifies that recording should finish before MCI returns control to the application.

MCI_RECORD_INSERT

Indicates that newly recorded information should be inserted or pasted into the existing data. (Some devices may not support this.) If supported, this is the default.

MCI_FROM

Specifies that a starting position is included in the dwFrom field of the data structure identified by *lpRecord*. The units assigned to the position values is specified with the MCI_SET_TIME_FORMAT flag of the MCI_SET command. If MCI_FROM is not specified, the starting position defaults to the current location.

MCI_RECORD_OVERWRITE

Specifies that data should overwrite existing data.

MCIWAVE returns MCIERR_UNSUPPORTED_FUNCTION in response to this flag.

MCI_TO

Specifies that an ending position is included in the dwTo field of the data structure identified by *lpRecord*. The units assigned to the position values is specified with the MCI_SET_TIME_FORMAT flag of the MCI_SET command. If MCI_TO is not specified, the ending position defaults to the end of the media.

LPMCI_RECORD_PARMS *lpRecord*

Specifies a far pointer to the MCI_RECORD_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

MCISEQ returns MCIERR_UNSUPPORTED_FUNCTION for this command.

Comments

This command is supported by devices that return TRUE to the MCI_GETDEVCAPS_CAN_RECORD query.

See Also

MCI_CUE, MCI_PAUSE, MCI_PLAY, MCI_RESUME, MCI_SEEK

MCI_RESUME

This MCI command message resumes a paused device. Support of this message by a device is optional.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_RESUME:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpDefault*.

MCI_WAIT

Specifies that the device should resume before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpDefault*

Specifies a far pointer to the MCI_GENERIC_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

This command resumes playing and recording without changing the stop position set with MCI_PLAY or MCI_RECORD.

See Also

MCI_STOP, MCI_PLAY, MCI_RECORD

MCI_SAVE

This MCI command message saves the current file. Devices which modify files should not destroy the original copy until they receive the save message. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_SAVE:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpSave*.

MCI_WAIT

Specifies that the save operation should finish before MCI returns control to the application.

MCI_SAVE_FILE

Indicates the lpfilename field of the data structure identified by *lpSave* contains a pointer to a buffer containing the destination file name.

LPMCI_SAVE_PARMS *lpSave*

Specifies a far pointer to the MCI_SAVE_PARMS data structure. (Devices with additional parameters might replace this data structure with a device-specific data structure.)

Video Overlay Extensions

DWORD *dwFlags*

The following additional flags apply to video overlay devices supporting MCI_SAVE:

MCI_OVLY_RECT

Specifies that the rc field of the data structure identified by *lpSave* contains a valid display rectangle indicating the area of the video buffer to save.

LPMCI_OVLY_SAVE_PARMS *lpSave*

Specifies a far pointer to a MCI_OVLY_SAVE_PARMS data structure.

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code. MCISEQ returns MCIERR_UNSUPPORTED_FUNCTION.

Comments

This command is supported by devices that return true to the MCI_GETDEVCAPS_CAN_SAVE query.

See Also

MCI_LOAD

MCI_SEEK

This MCI command message changes the current position of media as quickly as possible. Video and audio output are disable during the seek. After the seek is complete, the device will be stopped. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_SEEK:

MCI_NOTIFY
: Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpSeek*.

MCI_WAIT
: Specifies that the seek operation should finish before MCI returns control to the application.

MCI_SEEK_TO_END
: Specifies to seek to the end of the media.

MCI_SEEK_TO_START
: Specifies to seek to the start of the media.

MCI_TO
: Specifies a position is included in the dwTo field of the MCI_SEEK_PARMS data structure. The units assigned to the position values is specified with the MCI_SET_TIME_FORMAT flag of the MCI_SET command. Do not use this flag with MCI_SEEK_END or MCI_SEEK_START.

LPMCI_SEEK_PARMS *lpSeek*

Specifies a far pointer to the MCI_SEEK_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Videodisc Extensions

DWORD *dwFlags*

The following additional flag applies to videodisc devices.

MCI_VD_SEEK_REVERSE
: Specifies to seek backward.

LPMCI_SEEK_PARMS *lpSeek*

Specifies a far pointer to the MCI_SEEK_PARMS data structure.

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

See Also

MCI_PLAY, MCI_RECORD

MCI_SET

This MCI command message sets device information. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_SET:

MCI_NOTIFY
: Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpSet*.

MCI_WAIT
: Specifies that the set operation should finish before MCI returns control to the application.

MCI_SET_AUDIO
: Specifies an audio channel number is included in the dwAudio field of the data structure identified by *lpSet*. This flag must be used with MCI_SET_ON or MCI_SET_OFF. Use one of the following constants to indicate the channel number:

  MCI_SET_AUDIO_ALL
  : Specifies all audio channels.

  MCI_SET_AUDIO_LEFT
  : Specifies the left channel.

  MCI_SET_AUDIO_RIGHT
  : Specifies the right channel.

MCI_SET_DOOR_CLOSED
: Instructs the device to close the media cover (if any).

MCI_SET_DOOR_OPEN
: Instructs the device to open the media cover (if any).

MCI_SET_TIME_FORMAT
: Specifies a time format parameter is included in the dwTimeFormat field of the data structure identified by *lpSet*. Specifying MCI_FORMAT_MILLISECONDS indicates that subsequent commands that specify time will use milliseconds for both input and output. Other units are device dependent.

MCI_SET_VIDEO
: Sets the video signal on or off. This flag must be used with either MCI_SET_ON or MCI_SET_OFF. Devices that do not have video return MCIERR_UNSUPPORTED_FUNCTION.

MCI_SET_ON
: Enables the specified video or audio channel.

MCI_SET_OFF
: Disables the specified video or audio channel.

LPMCI_SET_PARMS *lpSet*
: Specifies a far pointer to the MCI_SET_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Animation Extensions

DWORD *dwFlags*
The following additional flags apply to animation devices:

MCI_SET_TIME_FORMAT
Specifies a time format parameter is included in the dwTimeFormat field of the data structure identified by *lpSet*. The following constants are defined for the time format:

MCI_FORMAT_MILLISECONDS
Changes the time format to milliseconds.

MCIMMP returns MCIERR_UNSUPPORTED_FUNCTION if the time format is set to MCI_FORMAT_MILLISECONDS.

MCI_FORMAT_FRAMES
Changes the time format to frames.

LPMCI_SET_PARMS *lpSet*
Specifies a far pointer to the MCI_SET_PARMS data structure.

CD Audio Extensions

DWORD *dwFlags*
The following additional flags apply to videodisc devices:

MCI_SET_TIME_FORMAT
Specifies a time format parameter is included in the dwTimeFormat field of the data structure identified by *lpSet*. The following constants are defined for the time format:

MCI_FORMAT_MILLISECONDS
Changes the time format to milliseconds.

MCI_FORMAT_MSF
Changes the time format to minutes, seconds, and frames.

MCI_FORMAT_TMSF
Changes the time format to tracks, minutes, seconds, and frames. (MCI uses continuous track numbers.)

LPMCI_SET_PARMS *lpSet*
Specifies a far pointer to the MCI_SET_PARMS structure.

MIDI Sequencer Extensions

DWORD *dwFlags*
The following additional flags apply to MIDI sequencer devices:

MCI_SEQ_SET_MASTER
Sets the sequencer as a source of synchronization data and indicates that the type of synchronization is specified in the dwMaster field of the data structure identified by *lpSet*.

MCISEQ returns MCIERR_UNSUPPORTED_FUNCTION.

The following constants are defined for the synchronization type:

MCI_SEQ_MIDI
The sequencer will send MIDI format synchronization data.

MCI_SEQ_SMPTE
The sequencer will send SMPTE format synchronization data.

MCI_SEQ_NONE
The sequencer will not send synchronization data.

MCI_SEQ_SET_OFFSET
Changes the SMPTE offset of a sequence to that specified by the dwOffset field of the data structure identified by *lpSet*. This only affects sequences with a SMPTE division type.

MCI_SEQ_SET_PORT
Sets the output MIDI port of a sequence to that specified by the MIDI device ID in the dwPort field of the data structure identified by *lpSet*. The device will close the previous port (if any), and attempt to open and use the new port. If it fails, it will return an error and re-open the previously used port (if any). The following constants are defined for the ports:

MCI_SEQ_NONE
Closes the previously used port (if any). The sequencer will behave exactly the same as if a port were open, except no MIDI message will be sent.

MIDIMAPPER
Sets the port opened to the MIDI Mapper.

MCI_SEQ_SET_SLAVE
Sets the sequencer to receive synchronization data and indicates that the type of synchronization is specified in the dwSlave field of the data structure identified by *lpSet*.

The following constants are defined for the synchronization type:

MCI_SEQ_FILE
Sets the sequencer to receive synchronization data contained in the MIDI file.

MCI_SEQ_SMPTE
Sets the sequencer to receive SMPTE synchronization data. MCISEQ returns MCIERR_UNSUPPORTED_FUNCTION.

MCI_SEQ_MIDI
Sets the sequencer to receive MIDI synchronization data. MCISEQ returns MCIERR_UNSUPPORTED_FUNCTION.

MCI_SEQ_NONE
Sets the sequencer to ignore synchronization data in a MIDI stream.

MCI_SEQ_SET_TEMPO
Changes the tempo of the MIDI sequence to that specified by the dwTempo field of the structure pointed to by *lpSet*. For sequences with division type PPQN, tempo is specified in beats per minute; for sequences with division type SMPTE,tempo is specified in frames per second.

MCI_SET_TIME_FORMAT
Specifies a time format parameter is included in the dwTimeFormat field of the data structure identified by *lpSet*. The following constants are defined for the time format:

MCI_FORMAT_MILLISECONDS
Changes the time format to milliseconds for both input and output.

MCI_FORMAT_SMPTE_24

Sets the time format to 24 frame SMPTE.

MCI_FORMAT_SMPTE_25
Sets the time format to 25 frame SMPTE.

MCI_FORMAT_SMPTE_30
Sets the time format to 30 frame SMPTE.

MCI_FORMAT_SMPTE_30DROP
Sets the time format to 30 drop-frame SMPTE.

MCI_SEQ_FORMAT_SONGPTR
Sets the time format to song-pointer units.

LPMCI_SEQ_SET_PARMS *lpSet*
Specifies a far pointer to the MCI_SEQ_SET_PARMS data structure.

Videodisc Extensions

DWORD *dwFlags*
The following additional flags apply to videodisc devices:

MCI_SET_TIME_FORMAT
Specifies a time format parameter is included in the dwTimeFormat field of the data structure identified by *lpSet*. The following constants are defined for the time format:

MCI_FORMAT_CHAPTERS
Changes the time format to chapters.

MCI_FORMAT_FRAMES
Changes the time format to frames.

MCI_FORMAT_HMS
Changes the time format to hours, minutes, and seconds.

MCI_FORMAT_MILLISECONDS
Changes the time format to milliseconds for both input and output.

MCI_VD_FORMAT_TRACK
Changes the time format to tracks. MCI uses continuous track numbers.

LPMCI_SET_PARMS *lpSet*
Specifies a far pointer to the MCI_SET_PARMS structure. (Devices with additional parameters might replace this data structure with a device-specific data structure.)

Waveform Audio Extensions

DWORD *dwFlags*
The following additional flags apply to waveform audio devices:

MCI_WAVE_INPUT
Sets the input used for recording to the wInput field of the data structure identified by *lpSet*.

MCI_WAVE_OUTPUT
Sets the output used for playing to the wOutput field of the data structure identified by *lpSet*.

MCI_WAVE_SET_ANYINPUT

Specifies that any wave input compatible with the current format can be used for recording.

MCI_WAVE_SET_ANYOUTPUT
Specifies that any wave output compatible with the current format can be used for playing.

MCI_WAVE_SET_AVGBYTESPERSEC
Sets the bytes per second used for playing, recording, and saving to the nAvgBytesPerSec field of the data structure identified by *lpSet*.

MCI_WAVE_SET_BITSPERSAMPLE
Sets the bits per sample used for playing, recording, and saving to the nBitsPerSample field of the data structure identified by *lpSet*.

MCI_WAVE_SET_BLOCKALIGN
Sets the block alignment used for playing, recording, and saving to the nBlockAlign field of the data structure identified by *lpSet*.

MCI_WAVE_SET_CHANNELS
Specifies the number of channels is indicated in the nChannels field of the data structure identified by *lpSet*.

MCI_WAVE_SET_FORMATTAG
Sets the format type used for playing,recording, and saving to the wFormatTag field of the data structure identified by *lpSet*. Specifying WAVE_FORMAT_PCM changes the format to PCM.

MCI_WAVE_SET_SAMPLESPERSEC
Sets the samples per second used for playing, recording, and saving to the nSamplesPerSec field of the data structure identified by *lpSet*.

MCI_SET_TIME_FORMAT
Specifies a time format parameter is included in the dwTimeFormat field of the data structure identified by *lpSet*. The following constants are defined for the time format:

MCI_FORMAT_BYTES
Changes the time format to bytes for input or output.

MCI_FORMAT_MILLISECONDS
Changes the time format to milliseconds for input or output.

MCI_FORMAT_SAMPLES
Changes the time format to samples for input or output.

LPMCI_WAVE_SET_PARMS *lpSet*
Specifies a far pointer to the MCI_WAVE_SET_PARMS data structure. This parameter replaces the standard default parameter data structure identified by *lpDefault*.

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_SPIN (VIDEODISC)

This MCI command message starts the device spinning up or down. This command is part of the videodisc command set. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD dwFlags

The following flags apply to all devices supporting **MCI_SPIN**:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the structure identified by *lpDefault*.

MCI_WAIT

Specifies that the spin up or spin down operation should finish before MCI returns control to the application.

MCI_VD_SPIN_UP

Starts the disc spinning.

MCI_VD_SPIN_DOWN

Stops the disc from spinning.

LPMCI_GENERIC_PARMS *lpDefault*

Specifies a far pointer to the MCI_GENERIC_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

This command applies to videodisc devices.

MCI_STATUS

This MCI command message is used to obtain information about an MCI device. All devices respond to this message. The parameters and flags available for this message depend on the selected device. Information is returned in the dwReturn field of the data structure identified by *lpStatus*.

Parameters

DWORD *dwFlags*

The following standard and command-specific flags apply to all devices:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpStatus*.

MCI_WAIT

Specifies that the status operation should finish before MCI returns control to the application.

MCI_STATUS_ITEM

Specifies that the dwItem field of the data structure identified by *lpStatus* contains a constant specifying which status item to obtain. The following constants define which status item to return in the dwReturn field of the data structure:

MCI_STATUS_CURRENT_TRACK

The dwReturn field is set to the current track number. MCI uses continuous track numbers.

MCI_STATUS_LENGTH

The dwReturn field is set to the total media length.

MCI_STATUS_MODE

The dwReturn field is set to the current mode of the device. The modes include the following:

* MCI_MODE_NOT_READY
* MCI_MODE_PAUSE
* MCI_MODE_PLAY
* MCI_MODE_STOP
* MCI_MODE_OPEN
* MCI_MODE_RECORD
* MCI_MODE_SEEK

MCI_STATUS_NUMBER_OF_TRACKS

The dwReturn field is set to the total number of playable tracks.

MCI_STATUS_POSITION

The dwReturn field is set to the current position.

MCI_STATUS_READY

The dwReturn field is set to TRUE if the device is ready; otherwise, it is set to FALSE.

MCI_STATUS_TIME_FORMAT

The dwReturn field is set to the current time format of the device. The time formats include:

* MCI_FORMAT_BYTES
* MCI_FORMAT_FRAMES

* MCI_FORMAT_HMS
* MCI_FORMAT_MILLISECONDS
* MCI_FORMAT_MSF
* MCI_FORMAT_SAMPLES
* MCI_FORMAT_TMSF

MCI_STATUS_START
Obtains the starting position of the media. To get the starting position, combine this flag with MCI_STATUS_ITEM and set the dwItem field of the data structure identified by *lpStatus* to MCI_STATUS_POSITION.

MCI_TRACK
Indicates a status track parameter is included in the dwTrack field of the data structure identified by *lpStatus*. You must use this flag with the MCI_STATUS_POSITION or MCI_STATUS_LENGTH constants.

When used with MCI_STATUS_POSITION, MCI_TRACK obtains the starting position of the specified track.

When used with MCI_STATUS_LENGTH, MCI_TRACK obtains the length of the specified track. MCI uses continuous track numbers.

LPMCI_STATUS_PARMS *lpStatus*
Specifies a far pointer to the MCI_STATUS_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Animation Extensions

DWORD *dwFlags*
The following extensions apply to animation devices:

MCI_STATUS_ITEM
Specifies that the dwItem field of the data structure identified by *lpStatus* contains a constant specifying which status item to obtain. The following additional status constants are defined for animation devices and indicate which item to return in the dwReturn field of the data structure:

MCI_ANIM_STATUS_FORWARD
The dwReturn field is set to TRUE if playing forward;otherwise, it is set to FALSE.

MCI_ANIM_STATUS_HPAL
The dwReturn field is set to the handle of the palette.

MCI_ANIM_STATUS_HWND
The dwReturn field is set to the handle of the playback window.

MCI_ANIM_STATUS_SPEED
The dwReturn field is set to the animation speed.

MCI_ANIM_STATUS_STRETCH
The dwReturn field is set to TRUE if stretching is enabled;otherwise, it is set to FALSE.

MCI_STATUS_MEDIA_PRESENT
The dwReturn field is set to TRUE if the media is inserted in the device; otherwise, it is set to FALSE.

LPMCI_STATUS_PARMS *lpStatus*
Specifies a far pointer to the MCI_STATUS_PARMS data structure.

CD Audio Extensions

DWORD *dwFlags*

The following extensions applies to CD audio devices:

MCI_STATUS_ITEM

Specifies that the dwItem field of the data structure identified by *lpStatus* contains a constant specifying which status item to obtain. The following additional status constants are defined for CD audio devices and indicate which item to return in the dwReturn field of the data structure:

MCI_STATUS_MEDIA_PRESENT

The dwReturn field is set to TRUE if the media is inserted in the device; otherwise, it is set to FALSE.

LPMCI_STATUS_PARMS *lpStatus*

Specifies a far pointer to the MCI_STATUS_PARMS data structure. This parameter replaces the standard default parameter data structure.

MIDI Sequencer Extensions

DWORD *dwFlags*

The following extensions apply to sequencers:

MCI_STATUS_ITEM

Specifies that the dwItem field of the data structure identified by *lpStatus* contains a constant specifying which status item to obtain. The following additional status constants are defined for sequencers and indicate which item to return in the dwReturn field of the data structure:

MCI_SEQ_STATUS_DIVTYPE

The dwReturn field is set to one of the following values indicating the current division type of a sequence:

* MCI_SEQ_DIV_PPQN
* MCI_SEQ_DIV_SMPTE_24
* MCI_SEQ_DIV_SMPTE_25
* MCI_SEQ_DIV_SMPTE_30
* MCI_SEQ_DIV_SMPTE_30DROP

MCI_SEQ_STATUS_MASTER

The dwReturn field is set to the synchronization type used for master operation.

MCI_SEQ_STATUS_OFFSET

The dwReturn field is set to the current SMPTE offset of a sequence.

MCI_SEQ_STATUS_PORT

The dwReturn field is set to the MIDI device ID for the current port used by the sequence.

MCI_SEQ_STATUS_SLAVE

The dwReturn field is set to the synchronization type used for slave operation.

MCI_SEQ_STATUS_TEMPO

The dwReturn field is set to the current tempo of a MIDI sequence in beats-per-minute for PPQN files, or frames-per-second for SMPTE files.

MCI_STATUS_MEDIA_PRESENT

The dwReturn field is set to TRUE if the media for the device is present; otherwise, it is

set to FALSE.

LPMCI_STATUS_PARMS *lpStatus*
Specifies a far pointer to the MCI_STATUS_PARMS data structure. This parameter replaces the standard default parameter data structure.

Videodisc Extensions

DWORD *dwFlags*
The following additional flags apply to videodisc devices:

MCI_STATUS_ITEM
Specifies that the dwItem field of the data structure identified by *lpStatus* contains a constant specifying which status item to obtain. The following additional status constants are defined for videodisc devices and indicate which item to return in the dwReturn field of the data structure:

MCI_STATUS_MEDIA_PRESENT
The dwReturn field is set to TRUE if the media is inserted in the device; otherwise, it is set to FALSE.

MCI_VD_STATUS_DISC_SIZE
The dwReturn field is set to the size of the loaded disc in inches (8 or 12).

MCI_VD_STATUS_FORWARD
The dwReturn field is set to TRUE if playing forward; otherwise, it is set to FALSE.

MCI_VD_STATUS_MEDIA_TYPE
The dwReturn field is set to the media type of the inserted media. The following media types can be returned:

* MCI_VD_MEDIA_CAV
* MCI_VD_MEDIA_CLV
* MCI_VD_MEDIA_OTHER

MCI_STATUS_MODE
The dwReturn field is set to the current mode of the device. All devices can return the following constants to indicate the current mode:

* MCI_MODE_NOT_READY
* MCI_MODE_PAUSE
* MCI_MODE_PLAY
* MCI_MODE_STOP
* MCI_VD_MODE_PARK (videodisc devices)

MCI_VD_STATUS_SIDE
The dwReturn field is set to 1 or 2 to indicate which side of the disc is loaded. Not all videodisc devices support this flag.

MCI_VD_STATUS_SPEED
The dwReturn field is set to the play (const) speed in frames per second.

MCIPIONR returns MCIERR_UNSUPPORTED_FUNCTION.

LPMCI_STATUS_PARMS *lpStatus*
Specifies a far pointer to the MCI_STATUS_PARMS data structure. This parameter replaces the

standard default parameter data structure.

Waveform Audio Extensions

DWORD *dwFlags*

The following additional flags apply to waveform audio devices:

MCI_STATUS_ITEM

Specifies that the dwItem field of the data structure identified by *lpStatus* contains a constant specifying which status item to obtain. The following additional status constants are defined for waveform audio devices and indicate which item to return in the dwReturn field of the data structure:

MCI_STATUS_MEDIA_PRESENT

The dwReturn field is set to TRUE if the media is inserted in the device; otherwise, it is set to FALSE.

MCI_WAVE_INPUT

The dwReturn field is set to the wave input device used for recording. If no device is in use and no device has been explicitly set, then the error return is MCI_WAVE_INPUTUNSPECIFIED.

MCI_WAVE_OUTPUT

The dwReturn field is set to the wave output device used for playing. If no device is in use and no device has been explicitly set, then the error return is MCI_WAVE_OUTPUTUNSPECIFIED.

MCI_WAVE_STATUS_AVGBYTESPERSEC

The dwReturn field is set to the current bytes per second used for playing, recording, and saving.

MCI_WAVE_STATUS_BITSPERSAMPLE

The dwReturn field is set to the current bits per sample used for playing, recording, and saving.

MCI_WAVE_STATUS_BLOCKALIGN

The dwReturn field is set to the current block alignment used for playing, recording, and saving.

MCI_WAVE_STATUS_CHANNELS

The dwReturn field is set to the current channel count used for playing, recording, and saving.

MCI_WAVE_FORMATTAG

The dwReturn field is set to the current format tag used for playing, recording, and saving.

MCI_WAVE_STATUS_LEVEL

The dwReturn field is set to the current record or playback level. The value is returned as an 8- or 16-bit value, depending on the sample size used. The right or mono channel level is returned in the low-order word. The left channel level is returned in the high-order word.

MCI_WAVE_STATUS_SAMPLESPERSEC

The dwReturn field is set to the current samples per second used for playing, recording, and saving.

LPMCI_STATUS_PARMS *lpStatus*
Specifies a far pointer to the MCI_STATUS_PARMS data structure.

Video Overlay Extensions

DWORD *dwFlags*
The following additional flags apply to video overlay devices:

MCI_STATUS_ITEM
Specifies that the dwItem field of the data structure identified by *lpStatus* contains a constant specifying which status item to obtain. The following additional status constants are defined for video overlay devices and indicate which item to return in the dwReturn field of the data structure:

MCI_OVLY_STATUS_HWND
The dwReturn field is set to the handle of the window associated with the video overlay device.

MCI_OVLY_STATUS_STRETCH
The dwReturn field is set to TRUE if stretching is enabled;otherwise, it is set to FALSE.

MCI_STATUS_MEDIA_PRESENT
The dwReturn field is set to TRUE if the media is inserted in the device; otherwise, it is set to FALSE.

LPMCI_STATUS_PARMS *lpStatus*
Specifies a far pointer to the MCI_STATUS_PARMS data structure.

Return Value
Returns zero if successful. Otherwise, it returns an MCI error code.

MCI_STEP

This MCI command message steps the player one or more frames.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_STEP:

MCI_NOTIFY
: Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpStep*.

MCI_WAIT
: Specifies that the step operation should finish before MCI returns control to the application.

Animation Extensions

DWORD *dwFlags*

The following additional flag applies to animation devices.

MCI_ANIM_STEP_FRAMES
: Indicates that the dwFrames field of the data structure identified by *lpStep* specifies the number of frames to step.

MCI_ANIM_STEP_REVERSE
: Steps in reverse.

LPMCI_ANIM_STEP_PARMS *lpStep*

Specifies a far pointer to the MCI_ANIM_STEP_PARMS data structure.

Videodisc Extensions

DWORD *dwFlags*

The following additional flag applies to videodisc devices.

MCI_VD_STEP_FRAMES
: Indicates that the dwFrames field of the data structure identified by *lpStep* specifies the number of frames to step.

MCI_VD_STEP_REVERSE
: Steps in reverse.

LPMCI_VD_STEP_PARMS *lpStep*

Specifies a far pointer to the MCI_VD_STEP_PARMS data structure.

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

Only devices that return TRUE to the MCI_GETDEVCAPS_HAS_VIDEO capability query support this command at present.

See Also

MCI_CUE, MCI_PLAY, MCI_SEEK

MCI_STOP

This MCI command message stops all play and record sequences, unloads all play buffers, and ceases display of video images. Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_STOP**:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpStop*.

MCI_WAIT

Specifies that the device should stop before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpStop*

Specifies a far pointer to the MCI_GENERIC_PARMS data structure. (Devices with extended command sets might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

The difference between MCI_STOP and MCI_PAUSE depends upon the device. If possible, MCI_PAUSE suspends device operation but leaves the device ready to resume play immediately.

See Also

MCI_PAUSE, MCI_PLAY, MCI_RECORD, MCI_RESUME

MCI_SYSINFO

This MCI command message returns information about MCI devices. MCI supports this message directly rather than passing it to the devices. String information is returned in the application-supplied buffer pointed to by the lpstrReturn field of the data structure identified by *lpSysInfo*. Numeric information is returned as a DWORD placed in the application-supplied buffer. The dwRetSize field specifies the buffer length.

Parameters

DWORD *dwFlags*

The following standard and command-specific flags apply to all devices:

MCI_SYSINFO_INSTALLNAME

Obtains the name (listed in the SYSTEM.INI file) used to install the device.

MCI_SYSINFO_NAME

Obtains a device name corresponding to the device number specified in the dwNumber field of the data structure identified by *lpSysInfo* . If the MCI_SYSINFO_OPEN flag is set, MCI returns the names of open devices.

MCI_SYSINFO_OPEN

Obtains the quantity or name of open devices.

MCI_SYSINFO_QUANTITY

Obtains the number of devices of the specified type that are listed in the [mci] section of the SYSTEM.INI file. If the MCI_SYSINFO_OPEN flag is set, the number of open devices is returned.

LPMCI_SYSINFO_PARMS *lpSysInfo*

Specifies a far pointer to the MCI_SYSINFO_PARMS structure.

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

The wDeviceType field of the *lpSysInfo*structure indicates the device type of the query. If the *wDeviceID* parameter is set to MCI_ALL_DEVICE_ID, it overrides the value of the wDeviceType field.

Integer return values are DWORDS returned in the buffer pointed to by the lpstrReturn field of MCI_SYSINFO_PARMS.

String return values are NULL-terminated strings returned in the buffer pointed to by the lpstrReturn field.

MCI_UNFREEZE (VIDEO OVERLAY)

This MCI command message restores motion to an area of the video buffer frozen with MCI_FREEZE. This command is part of the video overlay command set. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_UNFREEZE:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpFreeze*.

MCI_WAIT

Specifies that the unfreeze operation should finish before MCI returns control to the application.

MCI_OVLY_RECT

Specifies that the rc field of the data structure identified by *lpFreeze* contains a valid display rectangle. This is a required parameter.

LPMCI_OVLY_RECT_PARMS *lpFreeze*

Specifies a far pointer to a MCI_OVLY_RECT_PARMS data structure. (Devices with additional parameters might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

This command applies to video overlay devices.

See Also

MCI_FREEZE

MCI_UPDATE (ANIMATION)

This MCI command message updates the display rectangle. This command is part of the animation command set. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting **MCI_UPDATE**:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpDest*.

MCI_WAIT

Specifies that the palette should be realized before MCI returns control to the application.

LPMCI_GENERIC_PARMS *lpDest*

Specifies a far pointer to a device specific data structure. For a description of this parameter, see the *lpDest* description included with the device extensions.

Animation Extensions

DWORD *wFlags*

The following additional flags apply to animation devices supporting **MCI_UPDATE**:

MCI_ANIM_RECT

Specifies that the rc field of the data structure identified by *lpDest* contains a valid rectangle. If this flag is not specified, the entire window is updated.

MCI_ANIM_UPDATE_HDC

Specifies that the hDC field of the data structure identified by *lpDest* contains a handle to the display context. This flag is required.

LPMCI_ANIM_UPDATE_PARMS *lpDest*

Specifies a far pointer to a MCI_ANIM_UPDATE_PARMS data structure.

Returns

Returns zero if successful. Otherwise, it returns an MCI error code.

See Also

MCI_PUT, MCI_WHERE

MCI_WHERE (ANIMATION/VIDEO OVERLAY)

This MCI command message obtains the clipping rectangle for the video device. The top and left fields of the returned rectangle contain the origin of the clipping rectangle, and the right and bottom fields contain the width and height of the clipping rectangle. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_WHERE:

MCI_NOTIFY

Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpQuery*.

MCI_WAIT

Specifies that the operation should complete before MCI returns control to the application.

DWORD *lpQuery*

Specifies a far pointer to a device-specific data structure. For a description of this parameter, see the *lpQuery*description included with the device extensions.

Animation Extensions

DWORD *dwFlags*

The following additional flags apply to animation devices supporting MCI_WHERE:

MCI_ANIM_WHERE_DESTINATION

Obtains the destination display rectangle. The rectangle coordinates are placed in the rc field of the data structure identified by *lpQuery*.

MCI_ANIM_WHERE_SOURCE

Obtains the animation source rectangle. The rectangle coordinates are placed in the rc field of the data structure identified by *lpQuery*.

LPMCI_ANIM_RECT_PARMS *lpQuery*

Specifies a far pointer to a MCI_ANIM_RECT_PARMS data structure.

Video Overlay Extensions

DWORD *dwFlags*

The following additional flags apply to video overlay devices supporting MCI_WHERE:

MCI_OVLY_WHERE_DESTINATION

Obtains the destination display rectangle. The rectangle coordinates are placed in the rc field of the data structure identified by *lpQuery*.

MCI_OVLY_WHERE_FRAME

Obtains the overlay frame rectangle. The rectangle coordinates are placed in the rc field of the data structure identified by *lpQuery*.

MCI_OVLY_WHERE_SOURCE

Obtains the source rectangle. The rectangle coordinates are placed in the rc field of the data structure identified by *lpQuery*.

MCI_OVLY_WHERE_VIDEO

Obtains the video rectangle. The rectangle coordinates are placed in the rc field of the data

structure identified by *lpQuery*.

LPMCI_OVLY_RECT_PARMS *lpQuery*
Specifies a far pointer to a MCI_OVLY_RECT_PARMS data structure.

Return Value
Returns zero if successful. Otherwise, it returns an MCI error code.

Comments
This command applies to animation and video overlay devices.

See Also
MCI_PUT

MCI_WINDOW

This MCI command message specifies the window and the window characteristics for graphic devices. Graphic devices should create a default window when a device is opened but should not display it until they receive the play command. The window command is used to supply an application-created window to the device and to change the display characteristics of an application-supplied or default display window. If the application supplies the display window, it should be prepared to update an invalid rectangle on the window.

Support of this message by a device is optional. The parameters and flags for this message vary according to the selected device.

Parameters

DWORD *dwFlags*

The following flags apply to all devices supporting MCI_WINDOW:

MCI_NOTIFY
Specifies that MCI should post the MM_MCINOTIFY message when this command completes. The window to receive this message is specified in the dwCallback field of the data structure identified by *lpWindow*.

MCI_WAIT
Specifies that the operation should finish before MCI returns control to the application.

DWORD *lpWindow*

Specifies a far pointer to a device specific data structure. For a description of this parameter, see the *lpWindow* description included with the device extensions.

Animation Extensions

DWORD *dwFlags*

The following additional flags apply to animation devices supporting MCI_WINDOW:

MCI_ANIM_WINDOW_DISABLE_STRETCH
Disables stretching of the image.

MCI_ANIM_WINDOW_ENABLE_STRETCH
ENables stretching of the image.

MCI_ANIM_WINDOW_HWND
Indicates the handle of the window to use for the destination is included in the hWnd field of the data structure identified by *lpWindow*. Set this to MCI_ANIM_WINDOW_DEFAULT to return to the default window.

MCI_ANIM_WINDOW_STATE
Indicates the nCmdShow field of the MCI_ANIM_WINDOW_PARMS data structure contains parameters for setting the window state. This flag is equivalent to calling ShowWindow with the state parameter. The constants are the same as the ones in WINDOWS.H (such as SW_HIDE,SW_MINIMIZE, or SW_SHOWNORMAL.)

MCI_ANIM_WINDOW_TEXT
Indicates the lpstrText field of the MCI_ANIM_WINDOW_PARMS data structure contains a pointer to a buffer containing the caption used for the window.

LPMCI_ANIM_WINDOW_PARMS *lpWindow*

Specifies a far pointer to a MCI_ANIM_WINDOW_PARMS data structure. (Devices with additional parameters might replace this data structure with a device-specific data structure.)

Video Overlay Extensions

DWORD *dwFlags*

The following additional flags apply to video overlay devices supporting MCI_WINDOW:

MCI_OVLY_WINDOW_DISABLE_STRETCH
Disables stretching of the image.

MCI_OVLY_WINDOW_ENABLE_STRETCH
Enables stretching of the image.

MCI_OVLY_WINDOW_HWND
Indicates the handle of the window used for the destination is included in the hWnd field of the MCI_OVLY_WINDOW_PARMS data structure. Set this to MCI_OVLY_WINDOW_DEFAULT to return to the default window.

MCI_OVLY_WINDOW_STATE
Indicates the nCmdShow field of the *lpWindow* data structure contains parameters for setting the window state. This flag It is equivalent to calling showwindow with the state parameter. The constants are the same as those defined in WINDOWS.H (such as SW_HIDE, SW_MINIMIZE, or SW_SHOWNORMAL.)

MCI_OVLY_WINDOW_TEXT
Indicates the lpstrText field of the MCI_OVLY_WINDOW_PARMS data structure contains a pointer to buffer containing the caption used for the window.

LPMCI_OVLY_WINDOW_PARMS *lpWindow*
Specifies a far pointer to a MCI_OVLY_WINDOW_PARMS data structure. (Devices with additional parameters might replace this data structure with a device-specific data structure.)

Return Value

Returns zero if successful. Otherwise, it returns an MCI error code.

Comments

This command applies to animation and video overlay devices.

MCI_HMS_HOUR

Syntax

BYTE **MCI_HMS_HOUR**(*dwHMS*)

This macro returns the hours field from a DWORD argument containing packed HMS (hours,minutes, seconds) information.

Parameters

DWORD *dwHMS*

Specifies the time in HMS format.

Return Value

The return value is the hours field of the given argument.

Comments

Time in HMS format is expressed as a DWORD with the least significant byte containing hours, the next least significant byte containing minutes, and the next least significant byte containing seconds. The most significant byte is unused.

See Also

MCI_HMS_MINUTE, MCI_HMS_SECOND, MCI_MAKE_HMS

MCI_HMS_MINUTE

Syntax

BYTE **MCI_HMS_MINUTE**(*dwHMS*)

This macro returns the minutes field from a DWORD argument containing packed HMS (hours,minutes, seconds) information.

Parameters

DWORD *dwHMS*

Specifies the time in HMS format.

Return Value

The return value is the minutes field of the given argument.

Comments

Time in HMS format is expressed as a DWORD with the least significant byte containing hours, the next least significant byte containing minutes, and the next least significant byte containing seconds. The most significant byte is unused.

See Also

MCI_HMS_HOUR, MCI_HMS_SECOND, MCI_MAKE_HMS

MCI_HMS_SECOND

Syntax

BYTE **MCI_HMS_SECOND**(*dwHMS*)

This macro returns the seconds field from a DWORD argument containing packed HMS (hours, minutes, seconds) information.

Parameters

DWORD *dwHMS*

Specifies the time in HMS format.

Return Value

The return value is the seconds field of the given argument.

Comments

Time in HMS format is expressed as a DWORD with the least significant byte containing hours, the next least significant byte containing minutes, and the next least significant byte containing seconds. The most significant byte is unused.

See Also

MCI_HMS_HOUR, MCI_HMS_MINUTE, MCI_MAKE_HMS

MCI_MAKE_HMS

Syntax

DWORD **MCI_MAKE_HMS**(*hours*, *minutes*, *seconds*)

This macro returns a time value in HMS (hours, minutes, seconds) format from the given hours, minutes,and seconds values.

Parameters

BYTE *hours*

Specifies the number of hours.

BYTE *minutes*

Specifies the number of minutes.

BYTE *seconds*

Specifies the number of seconds.

Return Value

The return value is a DWORD value containing the time in packed HMS format.

Comments

Time in HMS format is expressed as a DWORD with the least significant byte containing hours, the next least significant byte containing minutes, and the next least significant byte containing seconds. The most significant byte is unused.

See Also

MCI_HMS_HOUR, MCI_HMS_MINUTE, MCI_HMS_SECOND

MCI_MAKE_MSF

Syntax

DWORD **MCI_MAKE_MSF**(*minutes*, *seconds*, *frames*)

This macro returns a time value in MSF (minutes, seconds, frames) format from the given minutes, seconds,and frames values.

Parameters

BYTE *minutes*
Specifies the number of minutes.

BYTE *seconds*
Specifies the number of seconds.

BYTE *frames*
Specifies the number of frames.

Return Value

The return value is a DWORD value containing the time in packed MSF format.

Comments

Time in MSF format is expressed as a DWORD with the least significant byte containing minutes, the next least significant byte containing seconds, and the next least significant byte containing frames. The most significant byte is unused.

See Also

MCI_MSF_MINUTE, MCI_MSF_SECOND, MCI_MSF_FRAME

MCI_MAKE_TMSF

Syntax

DWORD **MCI_MAKE_TMSF**(*tracks*, *minutes*, *seconds*, *frames*)

This macro returns a time value in TMSF (tracks, minutes, seconds, frames) format from the given tracks,minutes, seconds, and frames values.

Parameters

BYTE *tracks*
Specifies the number of tracks.

BYTE *minutes*
Specifies the number of minutes.

BYTE *seconds*
Specifies the number of seconds.

BYTE *frames*
Specifies the number of frames.

Return Value

The return value is a DWORD value containing the time in packed TMSF (tracks, minutes, seconds, frames) format.

Comments

Time in TMSF format is expressed as a DWORD with the least significant byte containing tracks, the next least significant byte containing minutes, the next least significant byte containing seconds, and the most significant byte containing frames.

See Also

MCI_TMSF_MINUTE, MCI_TMSF_SECOND, MCI_TMSF_FRAME

MCI_MSF_FRAME

Syntax

BYTE **MCI_MSF_FRAME**(*dwMSF*)

This macro returns the frames field from a DWORD argument containing packed MSF (minutes, seconds,frames) information.

Parameters

DWORD *dwMSF*

Specifies the time in MSF format.

Return Value

The return value is the frames field of the given argument.

Comments

Time in MSF format is expressed as a DWORD with the least significant byte containing minutes, the next least significant byte containing seconds, and the next least significant byte containing frames. The most significant byte is unused.

See Also

MCI_MSF_MINUTE, MCI_MSF_SECOND, MCI_MAKE_MSF

MCI_MSF_MINUTE

Syntax

BYTE **MCI_MSF_MINUTE**(*dwMSF*)

This macro returns the minutes field from a DWORD argument containing packed MSF (minutes, seconds, frames)information.

Parameters

DWORD *dwMSF*

Specifies the time in MSF format.

Return Value

The return value is the minutes field of the given argument.

Comments

Time in MSF format is expressed as a DWORD with the least significant byte containing minutes, the next least significant byte containing seconds, and the next least significant byte containing frames. The most significant byte is unused.

See Also

MCI_MSF_SECOND, MCI_MSF_FRAME, MCI_MAKE_MSF

MCI_MSF_SECOND

Syntax

BYTE **MCI_MSF_SECOND**(*dwMSF*)

This macro returns the seconds field from a DWORD argument containing packed MSF (minutes, seconds,frames) information.

Parameters

DWORD *dwMSF*

Specifies the time in MSF format.

Return Value

The return value is the seconds field of the given argument.

Comments

Time in MSF format is expressed as a DWORD with the least significant byte containing minutes, the next least significant byte containing seconds, and the next least significant byte containing frames. The most significant byte is unused.

See Also

MCI_MSF_MINUTE, MCI_MSF_FRAME, MCI_MAKE_MSF

MCI_TMSF_FRAME

Syntax

BYTE **MCI_TMSF_FRAME**(*dwTMSF*)

This macro returns the frames field from a DWORD argument containing packed TMSF (tracks, minutes, seconds, frames) information.

Parameters

DWORD *dwTMSF*

Specifies the time in TMSF format.

Return Value

The return value is the frames field of the given argument.

Comments

Time in TMSF format is expressed as a DWORD with the least significant byte containing tracks, the next least significant byte containing minutes, the next least significant byte containing seconds, and the most significant byte containing frames.

See Also

MCI_TMSF_TRACK, MCI_TMSF_MINUTE, MCI_TMSF_SECOND, MCI_MAKE_TMSF

MCI_TMSF_MINUTE

Syntax

BYTE **MCI_TMSF_MINUTE**(*dwTMSF*)

This macro returns the minutes field from a DWORD argument containing packed TMSF (tracs, minutes, seconds, frames) information.

Parameters

DWORD *dwTMSF*

Specifies the time in TMSF format.

Return Value

The return value is the minutes field of the given argument.

Comments

Time in TMSF format is expressed as a DWORD with the least significant byte containing tracks, the next least significant byte containing minutes, the next least significant byte containing seconds, and the most significant byte containing frames.

See Also

MCI_TMSF_TRACK, MCI_TMSF_SECOND, MCI_TMSF_FRAME, MCI_MAKE_TMSF

MCI_TMSF_SECOND

Syntax

BYTE **MCI_TMSF_SECOND**(*dwTMSF*)

This macro returns the seconds field from a DWORD argument containing packed TMSF (tracks, minutes, seconds, frames) information.

Parameters

DWORD *dwTMSF*

Specifies the time in TMSF format.

Return Value

The return value is the seconds field of the given argument.

Comments

Time in TMSF format is expressed as a DWORD with the least significant byte containing tracks, the next least significant byte containing minutes, the next least significant byte containing seconds, and the most significant byte containing frames.

See Also

MCI_TMSF_TRACK, MCI_TMSF_MINUTE, MCI_TMSF_FRAME, MCI_MAKE_TMSF

MCI_TMSF_TRACK

Syntax

BYTE **MCI_TMSF_TRACK**(*dwTMSF*)

This macro returns the tracks field from a DWORD argument containing packed TMSF (tracks, minutes, seconds, frames) information.

Parameters

DWORD *dwTMSF*

Specifies the time in TMSF format.

Return Value

The return value is the tracks field of the given argument.

Comments

Time in TMSF format is expressed as a DWORD with the least significant byte containing tracks, the next least significant byte containing minutes, the next least significant byte containing seconds, and the most significant byte containing frames.

See Also

MCI_TMSF_MINUTE, MCI_TMSF_SECOND, MCI_TMSF_FRAME, MCI_MAKE_TMSF

MCI_ANIM_OPEN_PARMS

The **MCI_ANIM_OPEN_PARMS**structure contains information for the MCI_OPEN message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields. You can use the MCI_OPEN_PARMS data structure in place of **MCI_ANIM_OPEN_PARMS** if you are not using the extended data fields.

```
typedef struct  {
    DWORD dwCallback;
    UINT wDeviceID;
    UINT wReserved0;
    LPCSTR lpstrDeviceType;
    LPCSTR lpstrElementName;
    LPCSTR lpstrAlias;
    DWORD dwStyle;
    HWND hWndParent;
    UINT wReserved1;
} MCI_ANIM_OPEN_PARMS;
```

Fields

The **MCI_ANIM_OPEN_PARMS** structure contains the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

wDeviceID
Specifies the device ID returned to user.

wReserved0
Reserved field.

lpstrDeviceType
Specifies the name or constant ID of the device type.

lpstrElementName
Specifies the device element name (usually a pathname).

lpstrAlias
Specifies an optional device alias.

dwStyle
Specifies the window style.

hWndParent
Specifies the handle to use as the window parent.

wReserved1
Reserved.

See Also

MCI_OPEN

MCI_ANIM_PLAY_PARMS

The **MCI_ANIM_PLAY_PARMS** structure contains parameters for the MCI_PLAY message for animation devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields. You can use the MCI_PLAY_PARMS data structure in place of **MCI_ANIM_PLAY_PARMS** if you are not using the extended data fields.

```
typedef struct  {
    DWORD dwCallback;
    DWORD dwFrom;
    DWORD dwTo;
    DWORD dwSpeed;
} MCI_ANIM_PLAY_PARMS;
```

Fields

The **MCI_ANIM_PLAY_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwFrom

Specifies the position to play from.

dwTo

Specifies the position to play to.

dwSpeed

Specifies the play rate in frames per second.

See Also

MCI_PLAY

MCI_ANIM_RECT_PARMS

The **MCI_ANIM_RECT_PARMS** structure contains parameters for the MCI_PUT and MCI_WHERE messages for animation devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags*parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    RECT rc;
} MCI_ANIM_RECT_PARMS;
```

Fields

The **MCI_ANIM_RECT_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc

Specifies a rectangle.

See Also

MCI_PUT, MCI_WHERE

MCI_ANIM_STEP_PARMS

The **MCI_ANIM_STEP_PARMS** structure contains parameters for the MCI_STEP message for animation devices. When assigning data the fields in this data structure, set the corresponding MCI flags in the *dwFlags*parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    DWORD dwFrames;
} MCI_ANIM_STEP_PARMS;
```

Fields

The **MCI_ANIM_STEP_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwFrames

Specifies the number of frames to step.

See Also

MCI_STEP

MCI_ANIM_UPDATE_PARMS

The **MCI_ANIM_UPDATE_PARMS** structure contains parameters for the MCI_UPDATE message for animation devices. When assigning data to the fields in this data structure,set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    RECT rc;
    HDC hDC;
} MCI_ANIM_UPDATE_PARMS;
```

Fields

The **MCI_ANIM_UPDATE_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc

Specifies a window rectangle.

hDC

Specifies a handle to the device context.

See Also

MCI_UPDATE

MCI_ANIM_WINDOW_PARMS

The **MCI_ANIM_WINDOW_PARMS** structure contains parameters for the MCI_WINDOW message for animation devices. When assigning data to the fields in this data structure,set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    HWND hWnd;
    UINT wReserved1;
    UINT nCmdShow;
    UINT wReserved2;
    LPCSTR lpstrText;
} MCI_ANIM_WINDOW_PARMS;
```

Fields

The **MCI_ANIM_WINDOW_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

hWnd

Specifies a handle to the display window.

wReserved1

Reserved.

nCmdShow

Specifies how the window is displayed.

wReserved2

Reserved.

lpstrText

Specifies a long pointer to a null-terminated string containing the window caption.

See Also

MCI_WINDOW

MCI_BREAK_PARMS

The **MCI_BREAK_PARMS** structure contains parameters for the MCI_BREAK message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    int nVirtKey;
    UINT wReserved0;
    HWND hwndBreak;
    UINT wReserved1;
} MCI_BREAK_PARMS;
```

Fields

The **MCI_BREAK_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

nVirtKey

Specifies the virtual key code used for the break key.

wReserved0

Reserved.

hwndBreak

Specifies a window handle of the window that must be the current window for break detection.

wReserved1

Reserved.

See Also

MCI_BREAK

MCI_GENERIC_PARMS

The **MCI_GENERIC_PARMS** structure contains the information for MCI command messages that have empty parameter lists. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
} MCI_GENERIC_PARMS;
```

Fields

The **MCI_GENERIC_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

MCI_GETDEVCAPS_PARMS

The **MCI_GETDEVCAPS_PARMS** structure contains parameters for the MCI_GETDEVCAPS message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    DWORD dwReturn;
    DWORD dwItem;
} MCI_GETDEVCAPS_PARMS;
```

Fields

The **MCI_GETDEVCAPS_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwReturn

Contains the return information on exit.

dwItem

Identifies the capability being queried.

See Also

MCI_GETDEVCAPS

MCI_INFO_PARMS

The **MCI_INFO_PARMS** structure contains parameters for the MCI_INFO message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    LPSTR lpstrReturn;
    DWORD dwRetSize;
} MCI_INFO_PARMS;
```

Fields

The **MCI_INFO_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpstrReturn

Specifies a long pointer to a user-supplied buffer for the return string.

dwRetSize

Specifies the size in bytes of the buffer for the return string.

See Also

MCI_INFO

MCI_LOAD_PARMS

The **MCI_LOAD_PARMS**structure contains the information for MCI_LOAD message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    LPCSTR lpfilename;
} MCI_LOAD_PARMS;
```

Fields

The **MCI_LOAD_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpfilename

Specifies a far pointer to a null-terminated string containing the filename of the device element to load.

See Also

MCI_LOAD

MCI_OPEN_PARMS

The **MCI_OPEN_PARMS** structure contains information for the MCI_OPEN message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    UINT wDeviceID;
    UINT wReserved0;
    LPCSTR lpstrDeviceType;
    LPCSTR lpstrElementName;
    LPCSTR lpstrAlias;
} MCI_OPEN_PARMS;
```

Fields

The **MCI_OPEN_PARMS** structure contains the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

wDeviceID
Contains the device ID returned to user.

wReserved0
Reserved.

lpstrDeviceType
Specifies the name or constant ID of the device type.

lpstrElementName
Specifies the device element name (usually a path).

lpstrAlias
Specifies an optional device alias.

See Also

MCI_OPEN

MCI_OVLY_LOAD_PARMS

The **MCI_OVLY_LOAD_PARMS** structure contains the information for the MCI_LOAD message for video overlay devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    LPCSTR lpfilename;
    RECT rc;
} MCI_LOAD_PARMS;
```

Fields

DWORD dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

LPCSTR lpfilename

Specifies a far pointer to a null-terminated string containing the filename of the device element to load.

RECT rc

Specifies a valid display rectangle identifying the area of the video buffer to update.

See also

MCI_LOAD

MCI_OVLY_OPEN_PARMS

The **MCI_OVLY_OPEN_PARMS**structure contains information for the MCI_OPEN message for video overlay devices. When assigning data to the fields in this data structure,set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields. You can use the MCI_OPEN_PARMS data structure in place of **MCI_OVLY_OPEN_PARMS** if you are not using the extended data fields.

```
typedef struct  {
    DWORD dwCallback;
    UINT wDeviceID;
    UINT wReserved0;
    LPCSTR lpstrDeviceType;
    LPCSTR lpstrElementName;
    LPCSTR lpstrAlias;
    DWORD dwStyle;
    DWORD hWndParent;
    UINT wReserved1;
} MCI_OVLY_OPEN_PARMS;
```

Fields

The **MCI_OVLY_OPEN_PARMS** structure contains the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

wDeviceID
Specifies the device ID returned to user.

wReserved0
Reserved.

lpstrDeviceType
Specifies the name or contstant ID of the device type obtained from the SYSTEM.INI file.

lpstrElementName
Specifies the device element name (usually a pathname).

lpstrAlias
Specifies an optional device alias.

dwStyle
Specifies the window style.

hWndParent
Specifies the handle to use as the window parent.

wReserved1
Reserved.

See Also

MCI_OPEN

MCI_OVLY_RECT_PARMS

The **MCI_OVLY_RECT_PARMS** structure contains parameters for the MCI_PUT and MCI_WHERE messages for video overlay devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    RECT rc;
} MCI_OVLY_RECT_PARMS;
```

Fields

The **MCI_OVLY_RECT_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc

Specifies a rectangle.

See Also

MCI_PUT, MCI_WHERE

MCI_OVLY_SAVE_PARMS

The **MCI_OVLY_SAVE_PARMS** structure contains the information for the MCI_SAVE message for video overlay devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    LPCSTR lpfilename;
    RECT rc;
} MCI_OVLY_SAVE_PARMS;
```

Fields

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpfilename

Specifies a far pointer to the buffer containing a null-terminated string.

rc

Specifies a valid display rectangle identifying the area of the video buffer to save.

See also

MCI_SAVE

MCI_OVLY_WINDOW_PARMS

The **MCI_OVLY_WINDOW_PARMS** structure contains parameters for the MCI_WINDOW message for video overlay devices. When assigning data to the fields in this data structure,set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    HWND hWnd;
    UINT wReserved1;
    UINT nCmdShow;
    UINT wReserved2;
    LPCSTR lpstrText;
} MCI_OVLY_WINDOW_PARMS;
```

Fields

The **MCI_OVLY_WINDOW_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

hWnd

Specifies a handle to the display window.

wReserved1

Reserved.

nCmdShow

Specifies how the window is displayed.

wReserved2

Reserved.

lpstrText

Specifies a long pointer to a null-terminated buffer containing the window caption.

See Also

MCI_WINDOW

MCI_PLAY_PARMS

The **MCI_PLAY_PARMS** structure contains parameters for the MCI_PLAY message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    DWORD dwFrom;
    DWORD dwTo;
} MCI_PLAY_PARMS;
```

Fields

The **MCI_PLAY_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwFrom

Specifies the position to play from.

dwTo

Specifies the position to play to.

See Also

MCI_PLAY

MCI_RECORD_PARMS

The **MCI_RECORD_PARMS** structure contains parameters for the MCI_RECORD message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    DWORD dwFrom;
    DWORD dwTo;
} MCI_RECORD_PARMS;
```

Fields

The **MCI_RECORD_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwFrom

Specifies the position to play from.

dwTo

Specifies the position to play to.

See Also

MCI_RECORD

MCI_SAVE_PARMS

The **MCI_SAVE_PARMS**structure contains the information for the MCI_SAVE message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    LPCSTR lpfilename;
} MCI_SAVE_PARMS;
```

Fields

The **MCI_SAVE_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpfilename

Specifies a far pointer to the buffer containing a null-terminated string.

See Also

MCI_SAVE

MCI_SEEK_PARMS

The **MCI_SEEK_PARMS** structure contains parameters for the MCI_SEEK message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    DWORD dwTo;
} MCI_SEEK_PARMS;
```

Fields

The **MCI_SEEK_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwTo

Specifies the position to seek to.

See Also

MCI_SEEK

MCI_SEQ_SET_PARMS

The **MCI_SEQ_SET_PARMS** structure contains parameters for the MCI_SET message for MIDI sequencer devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    DWORD dwTimeFormat;
    DWORD dwAudio;
    DWORD dwTempo;
    DWORD dwPort;
    DWORD dwSlave;
    DWORD dwMaster;
    DWORD dwOffset;
} MCI_SEQ_SET_PARMS;
```

Fields

The **MCI_SEQ_SET_PARMS** structure contains the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwTimeFormat
Specifies the time format of the sequencer.

dwAudio
Specifies the audio output channel.

dwTempo
Specifies the tempo.

dwPort
Specifies the output port.

dwSlave
Specifies the type of synchronization used by the sequencer for slave operation.

dwMaster
Specifies the type of synchronization used by the sequencer for master operation.

dwOffset
Specifies the data offset.

See Also

MCI_SET

MCI_SET_PARMS

The **MCI_SET_PARMS** structure contains parameters for the MCI_SET message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    DWORD dwTimeFormat;
    DWORD dwAudio;
} MCI_SET_PARMS;
```

Fields

The **MCI_SET_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwTimeFormat

Specifies the time format used by the device.

dwAudio

Specifies the audio output channel.

See Also

MCI_SET

MCI_STATUS_PARMS

The **MCI_STATUS_PARMS** structure contains parameters for the MCI_STATUS message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    DWORD dwReturn;
    DWORD dwItem;
    DWORD dwTrack;
} MCI_STATUS_PARMS;
```

Fields

The **MCI_STATUS_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwReturn

Contains the return information on exit.

dwItem

Identifies the capability being queried.

dwTrack

Specifies the length or number of tracks.

See Also

MCI_STATUS

MCI_SYSINFO_PARMS

The **MCI_SYSINFO_PARMS** structure contains parameters for the MCI_SYSINFO message. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    LPSTR lpstrReturn;
    DWORD dwRetSize;
    DWORD dwNumber;
    UINT wDeviceType;
    UINT wReserved0;
} MCI_SYSINFO_PARMS;
```

Fields

The **MCI_SYSINFO_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpstrReturn

Specifies a long pointer to a user-supplied buffer for the return string. It is also used to return a DWORD when the MCI_SYSINFO_QUANTITY flag is used.

dwRetSize

Specifies the size in bytes of the buffer for the return string.

dwNumber

Specifies a number indicating the device position in the MCI device table or in the list of open devices if the MCI_SYSINFO_OPEN flag is set.

wDeviceType

Specifies the type of device.

wReserved0

Reserved.

See Also

MCI_SYSINFO

MCI_VD_ESCAPE_PARMS

The **MCI_VD_ESCAPE_PARMS** structure contains parameters for the MCI_ESCAPE message for videodisc devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    LPCSTR lpstrCommand;
} MCI_VD_ESCAPE_PARMS;
```

Fields

The **MCI_VD_ESCAPE_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpstrCommand

Specifies a far pointer to a null-terminated buffer containing the command to send to the device.

See Also

MCI_ESCAPE

MCI_VD_PLAY_PARMS

The **MCI_VD_PLAY_PARMS** structure contains parameters for the MCI_PLAY message for videodiscs. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields. You can use the MCI_PLAY_PARMS data structure in place of **MCI_VD_PLAY_PARMS** if you are not using the extended data fields.

```
typedef struct  {
    DWORD dwCallback;
    DWORD dwFrom;
    DWORD dwTo;
    DWORD dwSpeed;
} MCI_VD_PLAY_PARMS;
```

Fields

The **MCI_VD_PLAY_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwFrom

Specifies the position to play from.

dwTo

Specifies the position to play to.

dwSpeed

Specifies the playing speed in frames per second.

See Also

MCI_PLAY

MCI_VD_STEP_PARMS

The **MCI_VD_STEP_PARMS** structure contains parameters for the MCI_STEP message for videodiscs. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    DWORD dwFrames;
} MCI_VD_STEP_PARMS;
```

Fields

The **MCI_VD_STEP_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwFrames

Specifies the number of frames to step.

See Also

MCI_STEP

MCI_WAVE_DELETE_PARMS

The **MCI_WAVE_DELETE_PARMS** structure contains parameters for the MCI_DELETE message for waveform audio devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    DWORD dwFrom;
    DWORD dwTo;
} MCI_WAVE_DELETE_PARMS;
```

Fields

The **MCI_WAVE_DELETE_PARMS** structure contains the following fields:

dwCallback

The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwFrom

Specifies the starting position for the delete.

dwTo

Specifies the end position for the delete.

See Also

MCI_DELETE

MCI_WAVE_OPEN_PARMS

The **MCI_WAVE_OPEN_PARMS**structure contains information for the MCI_OPEN message for waveform audio devices. When assigning data to the fields in this data structure,set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields. You can use the MCI_OPEN_PARMS data structure in place of **MCI_WAVE_OPEN_PARMS** if you are not using the extended data fields.

```
typedef struct  {
    DWORD dwCallback;
    UINT wDeviceID;
    UINT wReserved0;
    LPCSTR lpstrDeviceType;
    LPCSTR lpstrElementName;
    LPCSTR lpstrAlias;
    DWORD dwBufferSeconds;
} MCI_WAVE_OPEN_PARMS;
```

Fields

The **MCI_WAVE_OPEN_PARMS** structure contains the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

wDeviceID
Specifies the device ID returned to user.

wReserved0
Reserved.

lpstrDeviceType
Specifies the name or constant ID of the device type obtained.

lpstrElementName
Specifies the device element name (usually a pathname).

lpstrAlias
Specifies an optional device alias.

dwBufferSeconds
Specifies the buffer length in seconds.

See Also

MCI_OPEN

MCI_WAVE_SET_PARMS

The **MCI_WAVE_SET_PARMS** structure contains parameters for the MCI_SET message for waveform audio devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *dwFlags* parameter of mciSendCommand to validate the fields.

```
typedef struct  {
    DWORD dwCallback;
    DWORD dwTimeFormat;
    DWORD dwAudio;
    UINT wInput;
    UINT wReserved0;
    UINT wOutput;
    UINT wReserved1;
    UINT wFormatTag;
    UINT wReserved2;
    UINT nChannels;
    UINT wReserved3;
    DWORD nSamplesPerSec;
    DWORD nAvgBytesPerSec;
    UINT nBlockAlign;
    UINT wReserved4;
    UINT wBitsPerSample;
    UINT wReserved5;
} MCI_WAVE_SET_PARMS;
```

Fields

The **MCI_WAVE_SET_PARMS** structure contains the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwTimeFormat
Specifies the time format used by by the device.

dwAudio
Specifies the channel used for audio output.

wInput
Specifies the channel used for audio input.

wReserved0
Reserved.

wOutput
Specifies the channel used for output.

wReserved1
Reserved.

wFormatTag
Species the interpretation of the waveform data.

wReserved2
Reserved.

nChannels
Specifies mono (1) or stereo (2).

wReserved3
Reserved.

nSamplesPerSec
Specifies the samples per second used for the waveform.

nAvgBytesPerSec
Specifies the sample rate in bytes per second.

nBlockAlign
Specifies the block alignment of the data.

wReserved4
Reserved.

wBitsPerSample
Specifies the number of bits per sample.

wReserved5
Reserved.

See Also
MCI_SET

Manufacturer and Product IDs

This appendix provides lists of the manufacturer and product IDs currently used with the multimedia APIs. This list will grow as more manufacturers create multimedia products for Windows.

To get a current list of multimedia manufacturer and product IDs, and to register new ones, request a *Multimedia Developer Registration Kit* from the following group:

Microsoft Corporation
Multimedia Systems Group
Product Marketing
One Microsoft Way
Redmond, WA 98052-6399

Multimedia Extensions Manufacturer IDs

The current manufacturer IDs are as follows:

Constant Name	Value	Description
MM_MICROSOFT	1	Drivers developed by Microsoft Corporation

Multimedia Extensions Product IDs

The current product IDs are as follows:

Constant Name	Value	Description
MM_MIDI_MAPPER	1	Microsoft MIDI Mapper
MM_WAVE_MAPPER	2	Microsoft Wave Mapper
MM_SNDBLST_MIDIOUT	3	Sound Blaster MIDI output port
MM_SNDBLST_MIDIIN	4	Sound Blaster MIDI input port
MM_SNDBLST_SYNTH	5	Sound Blaster internal synthesizer
MM_SNDBLST_WAVEOUT	6	Sound Blaster waveform input port
MM_SNDBLST_WAVEIN	7	Sound Blaster waveform input port
MM_ADLIB	9	AdLib-compatible synthesizer
MM_MPU401_MIDIOUT	10	MPU401 MIDI output port
MM_MPU401_MIDIIN	11	MPU401 MIDI input port
MM_PC_JOYSTICK	12	IBM Game Control Adapter

Data Types

The multimedia APIs use the following data types:

FOURCC
A 32-bit value representing a four-character code.

HPSTR
A huge pointer to a character string.

HMIDIIN
A handle to a MIDI input device.

HMIDIOUT
A handle to a MIDI output device.

HMMIO
A handle to an open file.

HWAVEIN
A handle to a waveform input device.

HWAVEOUT
A handle to a waveform output device.

The MMSYSTEM.H header file also defines a series of pointer types associated with multimedia data structures. Each of these pointer types is named with an LP prefix followed by the name of the corresponding data structure. For example, the MMTIME data structure has an associated **LPMMTIME** pointer type.