Getting Started Using MCIWnd

MCIWnd is a window class for controlling multimedia devices. A library of functions, messages, and macros associated with MCIWnd provides a simple method to add multimedia playback or recording capabilities to your applications.

Using a single function, your application can create a control that plays devices such as video, CD audio, waveform audio, MIDI (Musical Instrument Digital Interface), or any device that uses the Media Control Interface (MCI). Automating playback is also quick and easy. Using a function and two macros, an application can create an MCIWnd window with the appropriate media device, play the device, and close both the device and the window when the content has finished playing.

Note Some devices, such as CD audio devices, play content that is stored on a medium. Other devices play content that is stored in files. For purposes of clarity, this book refers to both circumstances as "playing the device."

MCIWnd Window User Interface

MCIWnd provides additional features to adjust the look of the MCIWnd window, customize the behavior of your application, and tune playback performance. The major features of the MCIWnd window include the following:

- A toolbar with Play, Stop, Record and Menu buttons
- A trackbar that controls positioning within the playback content
- A pop-up menu containing common commands
- A playback area for video and other devices requiring it

The MCIWnd window includes a playback area for video, animation, and other devices that display images during playback. MCIWnd omits the playback area from waveform-audio devices, MIDI sequencers, and other devices that do not write to the display.

The Play button is located in the lower-left corner of the MCIWnd window. It appears when the content is stopped. The user can play the content in the following ways:

- To play the content from the current position, select the Play button.
- To play the content full-screen from the current position, select the Play button while holding down the CTRL key.
- To play the content backward from the current position, select the Play button while holding down the SHIFT key.

The Menu button, located next to the Play button, activates a menu that allows the user to open and close audio-video interleaved (AVI) files, and to adjust the image size, playback speed, and volume. (The user can also activate the menu by clicking the right mouse button whenever the cursor is in the client area of the window.) The menu also includes commands to change the configuration of the current device, to copy the playback content to the clipboard, and to issue MCI commands.

The trackbar to the right of the Menu button represents the duration of the playback (or recorded) content. The slider on the trackbar represents the current position within the content. When the slider is positioned at the left end of the trackbar, the current position is the beginning of the content. The user can move to different locations in the content by dragging the slider along the trackbar.

The MCIWnd controls can also include a Record button for devices that can record. The Record button is marked with a red circle and appears only when the device is capable of recording.

Note The playback window must be aligned on a four-pixel boundary for the best video playback performance. Typically, the Microsoft Windows operating system aligns the window automatically when it is created. If a user moves or stretches the window from its initial position, video playback speed might be reduced by half.

Multimedia Playback

The **MCIWndCreate** function is the basis for controlling an MCIWnd window and the device associated with it. In general, this function registers the MCIWnd window class and creates an MCIWnd window for using MCI services. This section describes how to use this function to perform the following tasks:

- Add user-controlled playback to an application
- Automate playback in an application
- Use window styles to change the appearance and behavior of an MCIWnd window
- Allow the user to select files and MCI devices for playback

User-Controlled Playback

You can add user-controlled playback to an existing application by calling the **MCIWndCreate** function as follows:

```
MCIWndCreate(hwndParent, hInstModule, NULL, "filename.typ");
```

The **MCIWndCreate** parameters identify handles to the parent window and to the module instance associated with the MCIWnd window. They also specify window styles and the filename (or device name) to associate with the MCIWnd window.

MCIWndCreate automatically performs the following steps that, for other window classes, you would normally have to code in your application:

1. Registers the MCIWnd window class.
2. Creates the MCIWnd window.
3. Loads the specified content.
4. Establishes the current position at the beginning of the content.
5. Displays the device control.
6. Displays the playback area of the window if needed.

Automated Playback

You can automate playback in your application by using **MCIWndCreate** and the **MCIWndPlay** macro, along with either the **MCIWndDestroy** or the **MCIWndClose** macro. To automate playback, specify the MCIWNDF_NOPLAYBAR and MCIWNDF_NOTIFYMODE styles in the *dwStyle* parameter. MCIWNDF_NOPLAYBAR hides the toolbar, and MCIWNDF_NOTIFYMODE issues an appropriate notification message when the device stops playing.

You can play the device or file specified in **MCIWndCreate** by using **MCIWndPlay**. **MCIWndPlay** starts playing the content from its current position and continues to its end.

You can destroy or close an MCIWnd window by using the **MCIWndDestroy** or **MCIWndClose** macro. **MCIWndDestroy** closes the device or file and destroys the MCIWnd window by invalidating its handle. If your application can reuse the MCIWnd window, use **MCIWndClose** to close the device without destroying the window.

Your application can detect when the device stops playing and automatically close the window. To do this, specify the MCIWNDF_NOTIFYMODE style for the *dwStyle* parameter of **MCIWndCreate**. This causes the device to send a MCIWNDM_NOTIFYMODE message whenever it changes modes. Your application can trap this message to determine whether the device has stopped playing and, if so, close the window.

MCIWnd Window Styles

As with any window, you can change the appearance and behavior of an MCIWnd window by choosing from the standard Microsoft Win32 window styles. In addition, you can choose from several other window styles that are specific to MCIWnd windows. With these styles, your application can change these MCIWnd windows in the following ways:

- Change window size.
- Hide or display controls.
- Issue notification messages.
- Display information in the title bar.

You can set window styles by specifying them in the **MCIWndCreate** function, or you can use the **MCIWndChangeStyles** macro to change the style of an existing MCIWnd window. You can also query an MCIWnd window for its current styles by using the **MCIWndGetStyles** macro.

For a list of the MCIWnd-specific window styles, see the description of **MCIWndCreate** in the Reference section.

Additional Methods to Specify Files

You can associate a device or file with an existing MCIWnd window by using the **MCIWndOpenDialog**, **MCIWndOpen**, and **MCIWndOpenInterface** macros, and the **GetOpenFileNamePreview** function.

To let a user of your application select a file to play, use **MCIWndOpenDialog**. This macro displays the Open dialog box for choosing a file and associates the selected file with the current MCIWnd window.

You can let a user of your application select a file to associate with an MCIWnd window and preview that file by using **GetOpenFileNamePreview** and **MCIWndOpen**. **GetOpenFileNamePreview** displays the Open dialog box for choosing a file and lets the user preview (play) its contents. When the name of an existing file is specified in the dialog box, **GetOpenFileNamePreview** provides a small control to let the user preview the contents of the file. You can associate a specified file, selected with **GetOpenFileNamePreview** or specified in another manner, with an MCIWnd window by using **MCIWndOpen**.

You can also specify a device, such as "CDAudio," to associate with an MCIWnd window by using **MCIWndOpen**.

To associate an MCIWnd window with a file interface or data-stream interface to multimedia data, use **MCIWndOpenInterface**. For more information about file and data-stream interfaces, see AVIFile Functions and Macros.

Note Before associating a new file or device with an MCIWnd window, **MCIWndOpenDialog** and **MCIWndOpen** implicitly close any device currently associated with the window. Your application does not need to close any open devices before using these macros.

Playback Controls

MCIWnd includes several macros for controlling playback. This section describes how to use these macros to perform the following tasks:

- Determine and change the current position.
- Start, pause, and resume playback.
- Play a portion of the content (scope).
- Play backward.
- Play in a continuous loop.

Current Position

When a file or device is associated with an MCIWnd window, the position is initially set at the start of the content, regardless of the media type. During playback, the position moves linearly through the content and, if playback is uninterrupted, eventually reaches the end of the content. If an interruption occurs, the current position is the location in the content where playback was stopped or paused.

You can retrieve the locations for the beginning and end of the content by using the **MCIWndGetStart** and **MCIWndGetEnd** macros. You can determine the length of the content by subtracting the value returned by **MCIWndGetStart** from the value returned by **MCIWndGetEnd**, or by using the **MCIWndGetLength** macro. You can retrieve the current position by using the **MCIWndGetPosition** macro, or you can retrieve the position as a null-terminated string by using the **MCIWndGetPositionString** macro.

To change the current position, use the **MCIWndHome**, **MCIWndEnd**, and **MCIWndSeek** macros. You can move the playback position to the start of the content by using **MCIWndHome** or to the end of the content by using **MCIWndEnd**. Use **MCIWndSeek** to move the playback position to any location in the content.

You can also step through the content by using the **MCIWndStep** macro. Beginning from the current position, this macro moves the position forward or backward by a specified increment.

Note The units used to specify position vary among the different media types and devices. For example, the position for AVI files used by the MCIAVI device is measured in frames; the position for CD audio, waveform-audio, and MIDI files is measured in milliseconds.

Devices for other media types and third-party devices might use other units. For information about determining these units, see Playback Enhancements.

Starting, Pausing, and Resuming Playback

MCIWndPlay is the most general playback macro. This macro lets you play a file or device from the current position. Playback continues through the end of the content unless it is interrupted.

You can temporarily interrupt a device that is playing by using the **MCIWndPause** macro. You can resume playback from the paused position by using the **MCIWndResume** macro. Some devices do not support the pause and resume commands. These devices usually map **MCIWndPause** to the **MCIWndStop** macro, which stops playback or recording. You can restart a device that does not support pause or resume by using **MCIWndPlay**, which starts playback from the current position.

Playback Scope

MCIWnd provides macros that allow you to define the playback *scope.* The scope is the portion of the playback you want to play. For example, you can play the content from a position other than the beginning position by using the **MCIWndPlayFrom** macro. This macro seeks the specified location, begins playback, and continues to the end of the content. Similarly, you can play the content to a specified end point by using the **MCIWndPlayTo** macro. **MCIWndPlayTo** starts at the current position and plays until the specified location or the end of the content is reached, whichever comes first.

Also, you can define both the beginning and ending positions by using the **MCIWndPlayFromTo** macro. This macro seeks the specified beginning location and plays until the specified ending location or the end of the content is reached.

Reverse Playback

Some devices support playback in the reverse direction. You can play the content of such a device in the reverse direction by using the **MCIWndPlayReverse** macro. This macro defines the playback scope from the current position to the beginning of the content. The digital-video device, MCIAVI, can play backward. Devices that cannot play backward, such as CD audio, can issue an error message when **MCIWndPlayReverse** is invoked.

Playback Loops

MCIWnd supports playback as a continuous loop. You can play the content of a file or device repeatedly as a loop by using the **MCIWndSetRepeat** macro in combination with the Play button on the toolbar. The video playback device, MCIAVI, supports playback loops. To determine if continuous playback has been activated, use the **MCIWndGetRepeat** macro.

Multimedia Recording

You can implement recording capabilities in your application by using the user interface built into MCIWnd. You can use the **MCIWndCreate** function and the **MCIWndNew** macro to provide controls for starting and stopping recording and for saving the recorded information. Using **MCIWndCreate**, you can specify window styles to display an MCIWnd window and to include the Record button on the toolbar. Using **MCIWndNew**, you can specify the device type that is being recorded and that the information is to be captured in a new file.

If your application requires more sophistication, you can automate and customize the recording by using the **MCIWndRecord** macro. For additional information about customizing the recording process, see Customizing the Recording Process.

Note Some devices, such as CD audio and MCIAVI, are used for playback only. Other devices, such as waveform-audio devices, can be used for recording. If you specify a device that cannot record, MCIWnd omits the Record button from the toolbar.

Saving Recorded Content

After completing the recording, you can save the content by using the **MCIWndSave** or **MCIWndSaveDialog** macro, or by using the **GetSaveFileNamePreview** function with **MCIWndSave**. **MCIWndSave** saves data in the file associated with the MCIWnd window. **MCIWndSaveDialog** lets the user specify a filename and save the recorded data in the specified file. **GetSaveFileNamePreview** displays the SaveAs dialog box for choosing a file and lets the user preview (play) the file. When the name of an existing file is specified in the SaveAs dialog, **GetSaveFileNamePreview** provides a small control in the dialog box to let the user preview the contents of the file. You can save the recorded data in a file selected with **GetSaveFileNamePreview** by using **MCIWndSave**.

Playback Enhancements

When your application can play multimedia data using an MCIWnd window, you can enhance and adjust the window's appearance and behavior. This section describes how to perform the following tasks:

- Specify time formats.
- Adjust speed, volume, and zoom.
- Provide controls for cropping and stretching images.
- Use palettes.
- Provide status updates.
- Use a multiple document interface.

Time Formats

Multimedia data types typically can use time to identify significant positions within their content. Common time formats are milliseconds, tracks, and frames; other less common time formats, such as SMPTE 24, also exist. Time is the format and reference system for waveform-audio, MIDI, and CD audio data. Video supports time even though it is recorded as a sequence of frames (stream) that is normally played at a specific speed. Several macros are available for designating time format.

You can retrieve the current time format for a file or device by using the **MCIWndGetTimeFormat** macro. You can change the current time format to any other time format supported by a device by using the **MCIWndSetTimeFormat** macro. Or you can the set the time format to milliseconds or frames by using the **MCIWndUseTime** or **MCIWndUseFrames** macros.

Note Noncontinuous formats, such as tracks and SMPTE (Society of Motion Picture and Television Engineers), can cause the toolbar to behave erratically. For these time formats, you might want to turn off the toolbar by specifying the MCIWNDF_NOPLAYBAR window style when creating an MCIWnd window.

Speed, Volume, and Zoom

The speed, volume, and zoom macros provide the functionality of the View, Volume, and Speed commands of the MCIWnd menu. The macros in this section are generally used with video and other devices that display images during playback.

Some devices support multiple playback speed changes. You can set the playback speed for these devices by using the **MCIWndSetSpeed** macro. This macro defines the normal playback speed as 1000. Higher values indicate faster speeds. Lower values indicate slower speeds.

You can retrieve the current playback speed by using the **MCIWndGetSpeed** macro. This macro uses the same numerical values and range as **MCIWndSetSpeed**.

Some devices support volume changes. You can adjust or set the volume by using the **MCIWndSetVolume** macro. This macro defines the normal volume level as 1000. Higher values indicate louder volumes. Lower values indicate quieter volumes.

You can retrieve the current volume level by using the **MCIWndGetVolume** macro. This macro uses the same numerical values and range as **MCIWndSetVolume**.

For devices that use a playback window, MCIWnd supports a zoom feature that sets the size of the playback image. You can set the playback image size by using the **MCIWndSetZoom** macro. The macro redefines the playback image size while maintaining a constant aspect ratio for the image. The zoom value is defined as a percentage of the original image size. Thus, 100 represents the original image size, 50 indicates the image is shown half its original size, and 200 indicates the image is shown twice its original size.

You can retrieve the current zoom value by using the **MCIWndGetZoom** macro. This macro uses the same numerical values and range as **MCIWndSetZoom**.

Note The standard MCI CD audio and waveform-audio drivers do not support volume or speed changes.

Cropping and Stretching Images

MCIWnd allows you to crop and stretch images of a video clip. To understand these features, you need to understand the relationships between *frame size*, *source rectangle*, *destination rectangle*, and *playback area*.

A video clip consists of several frames, each containing one image. The frame size of a video clip is the size of the image in the current frame. Typically, a video clip has one frame size because all the images in the clip are the same size.

The source rectangle is a rectangular area that overlays the frames of a video clip. The source rectangle defines the portion of each frame that is displayed during playback. When a video clip is loaded with MCIWnd, the source rectangle is initialized to the same dimensions and position as the initial frame of the video clip.

The destination rectangle is a rectangular area that defines a virtual playback window. The destination rectangle receives the image data from the source rectangle for each frame of the video clip. When the source and destination rectangle dimensions are different, MCIWnd adjusts the image data horizontally and vertically as needed to fill the destination rectangle. When a video clip is loaded with MCIWnd, the destination rectangle is initialized to the same dimensions and position as the initial frame of the video clip.

The playback area is the portion of an MCIWnd window an application uses to display the video clip. The playback area is the client area of an MCIWnd window or the portion of the client area that excludes the MCIWnd toolbar. When a video clip is loaded with MCIWnd, the playback area is initialized to the same dimensions and position as the initial frame of the video clip.

You can crop a video clip by using the **MCIWndGetSource** and **MCIWndPutSource** macros to alter the source rectangle. Cropping an image determines only which portion of the frames are displayed during playback; it does not alter the content of the file being played. Before you crop an image, you can retrieve the current size of the source rectangle by using **MCIWndGetSource**. After the new size and location of the source rectangle are calculated, you can set the cropping boundaries of the source rectangle by using **MCIWndPutSource**.

You can stretch a video clip by using the **MCIWndGetDest** and **MCIWndPutDest** macros to alter the destination rectangle. When you stretch a video clip, you lengthen or shorten the frame size of a video clip vertically, horizontally, or in both directions. Before you stretch an image, you can retrieve the current size and location of the destination rectangle by using **MCIWndGetDest**. **MCIWndPutDest** allows you to redefine the destination rectangle. Stretching can distort the image during playback, but it does not alter the content of the file being played.

If the size of the destination rectangle becomes larger than the playback area, you can specify which portion of the playback area will display the video clip by using **MCIWndPutDest**.

Note **MCIWndPutDest** does not change the size of the playback area. To stretch the MCIWnd window along with the destination rectangle, you need to know the current size of the MCIWnd window and issue new window dimensions based on the destination rectangle. You can retrieve the MCIWnd window dimensions by using the **GetWindowRect** function and resize the MCIWnd window by using the **SetWindowPos** function.

MCIWnd Palettes

Playing video clips or animations with 8-bit color depth (256-color capacity) requires a palette to define the colors being used. Sometimes, the palette included with a video clip is not the most appropriate palette to use during playback. In this case, MCIWnd provides three ways to manage palettes for playback.

You can retrieve a handle to the palette associated with an MCIWnd window by using the **MCIWndGetPalette** macro. The palette is not necessarily associated exclusively with the MCIWnd window. Other applications can access, and even invalidate, the palette handle. Consequently, your application should anticipate the global use of the palette and, when finished with the palette, should not free it.

You can also specify a new palette to use with the video clip or animation associated with an MCIWnd window by using the **MCIWndSetPalette** macro.

Finally, you can realize the palette associated with an MCIWnd window to the system palette by using the **MCIWndRealize** macro. This macro calls the **RealizePalette** function with the palette associated with the MCIWnd window. If your application message handlers for WM_PALETTECHANGED and WM_QUERYNEWPALETTE call only **RealizePalette** or **MCIWndRealize**, you must forward these messages to MCIWnd if you do not handle them.

Note When a video clip or animation with 8-bit color depth is loaded into the MCIWnd window, the palette included with that clip replaces the palette associated with the MCIWnd window.

Status Updates

MCIWnd uses timers to periodically update information in the window title bar and scroll bar, and to send notification messages to the parent window. One timer controls the update period of the active MCIWnd window, and a second timer controls the update period for MCIWnd windows that are inactive. Your application can use the MCIWnd timer macros to retrieve the current timer settings and to adjust the update periods.

You can set the update period used by the active window timer by using the **MCIWndSetActiveTimer** macro. This macro sets the period used by MCIWnd to update the trackbar, to update the position reported in the window title bar, and to notify the parent window that the media has changed. You can retrieve the current update period used by the active window timer by using the **MCIWndGetActiveTimer** macro. The default update period for the active window timer is 500 milliseconds.

You can set the update period used by the inactive window timer by using the **MCIWndSetInactiveTimer** macro. This macro sets the period used by MCIWnd to update the trackbar, to update the position reported in the window caption, and to notify the parent window that the media has changed. You can retrieve the current update period used by the inactive window timer by using the **MCIWndGetInactiveTimer** macro. The default update period for the inactive window timer is 2000 milliseconds.

Your application can simultaneously set the update period for both timers by using the **MCIWndSetTimers** macro. This macro limits the size of each update period to 16 bits. If a larger quantity for either update period is needed, set the timers individually.

Multiple Document Interface Windows

Applications that use a multiple document interface (MDI) might need to specify window styles that are not available through the **MCIWndCreate** function. For these applications, you can register and create an MCIWnd window by using the **MCIWndRegisterClass** function with the **CreateWindowEx** function. **MCIWndRegisterClass** registers the MCIWND_WINDOW_CLASS window class and then **CreateWindowEx** creates an instance of an MCIWnd window.

Error Messages and Notifications

MCIWnd uses MCI to control the devices that play and record multimedia data. In general, MCIWnd displays MCI errors in an error dialog box. An MCI error is generated whenever an MCI command fails. For example, if your application tries to resume paused playback by using the **MCIWndResume** macro and the current device does not support resume, an error is reported to the user.

MCIWnd allows you two choices for handling error messages: you can prevent error messages from reaching the user, or you can redirect them to your application for display. To prevent the display of MCI error messages, specify the MCIWNDF_NOERRORDLG window style when you create an instance of an MCIWnd window by using the **MCIWndCreate** or **CreateWindowEx** function. To redirect MCI error messages to your application, specify the MCIWNDF_NOTIFYERROR window style when you create an instance of an MCIWnd window by using **MCIWndCreate** or **CreateWindowEx**.

When error notification is enabled, MCIWnd sends each notification message (MCIWNDM_NOTIFYERROR) to the main message handler of the parent of the MCIWnd window. Your application must have a message handler to process the notification messages it receives.

You can obtain a textual description of the most recent MCI error message by using the **MCIWndGetError** macro. This macro returns the text in an application-defined buffer. If the error string is longer than the buffer, MCIWnd truncates the string.

You can route all notifications to another window by using the **MCIWndSetOwner** macro.

Communicating with MCI Devices

The driver of each MCI device maintains a list of its current settings and capabilities, so it can issue an accurate response when it is queried for information. When you want to communicate with an MCI device, you can use MCIWnd macros and functions, or you can send MCI commands directly to the device by using either the message or string form of the commands. For more information about MCI, see Chapter 3, "MCI Overview."

Many of the most common MCI commands and queries are defined as macros; however, if the task you want to perform is unavailable as a function or macro, you can issue MCI commands directly to the device driver by using the **MCIWndSendString** macro. This macro is equivalent to using the **mciSendString** function as follows:

```
mciSendString(sz, Null, 0, Null)
```

The parameters of **MCIWndSendString** include only the window handle and the string form of the command. To retrieve the information returned by a string command, use the **MCIWndReturnString** macro.

Note You must exclude the device alias from the MCI command when you use **MCIWndSendString**. The MCIWnd library adds this alias when it sends the command to the MCI device.

Communication with MCI Devices

Each MCI device can have several identifications associated with it that include the following: a device identifier, a device name, an alias, and the filename of the currently loaded content. MCIWnd provides macros you can use to retrieve this information. You can then use this information to communicate through MCI directly with MCI devices associated with MCIWnd windows.

You can retrieve the identifier of the currently open MCI device by using the **MCIWndGetDeviceID** macro. The MCI device identifier is a numerical value that identifies the instance of the MCI device your application is using. Your application can use this identifier when communicating with an MCI device by using the **mciSendCommand** function.

To retrieve the name of the currently open MCI device, use the **MCIWndGetDevice** macro. The MCI device name is a null-terminated string that identifies the device type associated with an MCIWnd window. Your application can use this name when communicating with an MCI device by using **mciSendCommand**.

You can retrieve the alias of the currently open MCI device by using the **MCIWndGetAlias** macro. Your application can use this alias when communicating with an MCI device by using the **mciSendString** function.

Finally, you can retrieve the filename currently used by the MCI device by using the **MCIWndGetFileName** macro. The filename identifies the content currently associated with an MCIWnd window. Your application can use this filename when communicating with a MCI device by using **mciSendCommand** or **mciSendString**.

MCI Device Capabilities

Each MCI device has functions and features that identify how it can be useful. MCIWnd includes the following macros to let you query MCI devices for these capabilities.

Macro	Description
MCIWndCanPlay	Determines whether a device can play the existing content.
MCIWndCanRecord	Determines whether a device can record.
MCIWndCanWindow	Determines whether a device supports MCI window commands (such as **window**, **put** and **where**).
MCIWndCanSave	Determines whether a device can store data.
MCIWndCanEject	Determines whether a device has a software-controlled eject function.
MCIWndCanConfig	Determines whether a device has a configuration dialog box to support multiple configurations, such as the MCIAVI device.

These macros return TRUE if the device supports the specific capability or FALSE otherwise.

Using the MCIWnd Window Class

This section contains examples demonstrating how to perform the following tasks:

- Create an MCIWnd window.
- Automate playback.
- Pause and resume playback.
- Limit the playback scope.
- Record with MCIWnd controls.
- Customize the recording process.
- Crop an image.
- Stretch an image.
- Stretch an image and window.

Creating an MCIWnd Window

The **MCIWndCreate** function registers and creates an MCIWnd window. The window can be a parent, child, or pop-up window. The following example creates an MCIWnd window as a child window and lets the user control playback by providing access to the trackbar and the Play, Stop, and Menu buttons. The example specifies a handle of a parent window and NULL for the window styles, so the default window styles of WS_CHILD, WS_BORDER, and WS_VISIBLE are used to create the MCIWnd window.

```
// Global variable and constants
// extern HINSTANCE g_hinst;       instance handle
// extern HWND g_hwndMCIWnd;       MCIWnd window handle

case WM_COMMAND:
    switch (LOWORD(wParam)) {
    case IDM_CREATEMCIWND:
        g_hwndMCIWnd = MCIWndCreate(hwnd, g_hinst, NULL,
            "sample.avi");
        break;
    }
    break;
```

Note You could also specify NULL for the parent window handle and for the window styles, in which case the default window styles would be WS_OVERLAPPED and WS_VISIBLE.

Automating Playback

You can automate playback for MCIWnd by specifying a few window styles in the **MCIWndCreate** function. To play the device, the window needs a parent window to process notification messages, a playback area to play AVI files, and notification of device mode changes to identify when playback stops. The window does not need a toolbar. You can set these characteristics by specifying the appropriate styles in **MCIWndCreate**.

The following example uses menu commands to create an MCIWnd window to play content from several different types of devices. **MCIWndCreate** creates the MCIWnd window, and devices and files are loaded by using the **MCIWndOpen** macro in the device-specific commands. When a device finishes playing, it is closed by trapping the MCIWNDM_NOTIFYMODE message and issuing the **MCIWndClose** macro.

```
case WM_COMMAND:
    switch (LOWORD(wParam)) {
    case IDM_CREATEMCIWND:
           dwMCIWndStyle = WS_CHILD |                 // child window
            WS_VISIBLE |                                  // visible
            MCIWNDF_NOTIFYMODE |                       // notifies of mode
changes
            MCIWNDF_NOPLAYBAR;                         // hides toolbar
        g_hwndMCIWnd = MCIWndCreate(hwnd,
            g_hinst, dwMCIWndStyle, NULL);
        break;
    case IDM_PLAYCDA:
        LoadNGoMCIWnd(hwnd, "CDAudio");
           break;
    case IDM_PLAYWAVE:
        LoadNGoMCIWnd(hwnd, "SoundWave.WAV");
        break;
    case IDM_PLAYMIDI:
        LoadNGoMCIWnd(hwnd, "MIDIFile.MID");
        break;
    case IDM_PLAYAVI:
        LoadNGoMCIWnd(hwnd, "AVIFile.AVI");
        break;
    case IDM_EXIT:
        MCIWndDestroy(g_hwndMCIWnd);
        DestroyWindow(hwnd);
        break;
     }
    break;

case MCIWNDM_NOTIFYMODE:
    if (lParam == MCI_MODE_STOP){        // device stopped
        MessageBox(hwnd,"","Closing Device",MB_OK);
        MCIWndClose(g_hwndMCIWnd);
    }
    break;
.
. // Handle other messages here.
.

// LoadNGoMCIWnd - Automatically loads and plays a multimedia device.
```

```
//
// hwnd -  handle to the parent window
// lpstr - pointer to device or filename played by device
//
// Global variable
// extern HINSTANCE g_hwndMCIWnd;  instance handle to MCIWnd window

VOID LoadNGoMCIWnd(HWND hwnd, LPSTR lpstr)
{
    MessageBox(hwnd, lpstr, "Loading Device", MB_OK);
    MCIWndOpen(g_hwndMCIWnd, lpstr, NULL);   // new device in window
    MCIWndPlay(g_hwndMCIWnd);                // plays device
}
```

Pausing and Resuming Playback

You can interrupt playback of a device or file associated with an MCIWnd window by using the **MCIWndPause** macro. You can then restart playback by using the **MCIWndResume** macro. If the device does not support resume or an error occurs, you can use the **MCIWndPlay** macro to restart playback.

The following example creates an MCIWnd window and plays an AVI file. Pause and resume menu commands are available to the user to interrupt and restart playback.

MCIWnd window styles are changed temporarily by using the **MCIWndChangeStyles** macro to inhibit an MCI error dialog box from being displayed if **MCIWndResume** fails.

```
case WM_COMMAND:
    switch (LOWORD(wParam)) {
           case IDM_CREATEMCIWND:              // creates and plays clip
            g_hwndMCIWnd = MCIWndCreate(hwnd,
                g_hinst,
                WS_CHILD | WS_VISIBLE |     // standard styles
                MCIWNDF_NOPLAYBAR |         // hides toolbar
                MCIWNDF_NOTIFYMODE,         // notifies of mode changes
                "sample.avi");

            MCIWndPlay(g_hwndMCIWnd);
            break;
           case IDM_PAUSEMCIWND:               // pauses playback
            MCIWndPause(g_hwndMCIWnd);
            MessageBox(hwnd, "MCIWnd", "Pausing Playback", MB_OK);
            break;
           case IDM_RESUMEMCIWND:               // resumes playback
            MCIWndChangeStyles(                 // hides error dialog
messages
                g_hwndMCIWnd,             // MCIWnd window
                MCIWNDF_NOERRORDLG,       // mask of style to change
                MCIWNDF_NOERRORDLG);            // suppresses MCI error
dialogs

            lResult = MCIWndResume(g_hwndMCIWnd);

            if(lResult){                    // device doesn't resume
                MessageBox(hwnd, "MCIWnd",
                    "Resume with Stop and Play", MB_OK);
                MCIWndStop(g_hwndMCIWnd);
                MCIWndPlay(g_hwndMCIWnd);

                MCIWndChangeStyles(         // resumes original styles
                    g_hwndMCIWnd,
                    MCIWNDF_NOERRORDLG,
                    NULL);
            }
        break;
    }
break;
.
. // Handle other messages here.
.
```

Limiting the Playback Scope

Controlling playback begins with the **MCIWndPlay** macro, which plays the content or file associated with an MCIWnd window from the current position to the end of the content. If you want to limit playback to a specific portion of the content or file, you can choose from the other playback MCIWnd macros: **MCIWndPlayFrom**, **MCIWndPlayTo**, and **MCIWndPlayFromTo**.

You also need to set an appropriate time format. The time format determines whether the content is measured in frames, milliseconds, tracks, or some other units.

The following example creates an MCIWnd window and provides menu commands to play the last third, first third, or middle third of the content. These menu commands use **MCIWndPlayFrom**, **MCIWndPlayTo**, and **MCIWndPlayFromTo** to play the content segments. The example also uses the **MCIWndGetStart** and **MCIWndGetEnd** macros to identify the beginning and end of the content, and the **MCIWndHome** macro to move the playback position to the beginning of the content.

The **MCIWndCreate** function uses the WS_CAPTION and MCIWNDF_SHOWALL styles in addition to the standard window styles to display the filename, mode, and current position in the title bar of the MCIWnd window.

```
case WM_COMMAND:
    switch (LOWORD(wParam)) {
    case IDM_CREATEMCIWND:
        g_hwndMCIWnd = MCIWndCreate(hwnd,
            g_hinst,
            WS_CHILD | WS_VISIBLE | WS_CAPTION |
            MCIWNDF_SHOWALL,
            "sample.avi");
        break;
    case IDM_PLAYFROM:                    // plays last third of clip
        MCIWndUseTime(g_hwndMCIWnd);      // millisecond format

        // Get media start and end positions.
        lStart = MCIWndGetStart(g_hwndMCIWnd);
        lEnd = MCIWndGetEnd(g_hwndMCIWnd);

        // Determine playback end position.
        lPlayStart = 2 * (lEnd - lStart) / 3 + lStart;

        MCIWndPlayFrom(g_hwndMCIWnd, lPlayStart);
        break;
    case IDM_PLAYTO:                      // plays first third of clip
        MCIWndUseTime(g_hwndMCIWnd);      // millisecond format

        // Get media start and end positions.
        lStart = MCIWndGetStart(g_hwndMCIWnd);
        lEnd = MCIWndGetEnd(g_hwndMCIWnd);

        // Determine playback start position.
        lPlayEnd = (lEnd - lStart) / 3 + lStart;

        MCIWndHome(g_hwndMCIWnd);
        MCIWndPlayTo(g_hwndMCIWnd, lPlayEnd);
        break;
    case IDM_PLAYSOME:                    // plays middle third of clip
        MCIWndUseTime(g_hwndMCIWnd);      // millisecond format
```

```
    // Get media start and end positions.
    lStart = MCIWndGetStart(g_hwndMCIWnd);
    lEnd = MCIWndGetEnd(g_hwndMCIWnd);

    // Determine playback start and end positions.
    lPlayStart = (lEnd - lStart) / 3 + lStart;
    lPlayEnd = 2 * (lEnd - lStart) / 3 + lStart;

    MCIWndPlayFromTo(g_hwndMCIWnd, lPlayStart, lPlayEnd);
    break;
.
. // Handle other commands here.
.
}
```

Recording with MCIWnd Controls

The following example records waveform audio using the built-in controls of the MCIWnd window. The example creates an MCIWnd window by using the MCIWNDF_RECORD window style with the **MCIWndCreate** function to add a Record button to the toolbar. The **MCIWndNew** macro indicates a new file is associated with the MCIWnd window and that a waveform-audio device will provide its content. A second menu command, IDM_SAVEMCIWND, lets the user save the recording and select a filename by using the **MCIWndSaveDialog** macro.

```
case WM_COMMAND:
    switch (LOWORD(wParam)) {
    case IDM_CREATEMCIWND:
        g_hwndMCIWnd = MCIWndCreate(hwnd, g_hinst,
            WS_VISIBLE | MCIWNDF_RECORD, NULL);
        MCIWndNew(g_hwndMCIWnd, "waveaudio");
        break;
    case IDM_SAVEMCIWND:
        MCIWndSaveDialog(g_hwndMCIWnd);
        break;
    }
```

Customizing the Recording Process

You can customize the recording process, taking complete control of most everything – from creating the MCIWnd window to saving the recorded information in a file. The following example provides a more general case than "Recording with MCIWnd Controls" earlier in this chapter in adding the recording feature to an application. It queries the MCI device for recording and saving capabilities, and includes menu commands to record at the beginning or end of the content.

The following example uses the **MCIWndCreate** function to create a new window and allows you to specify an existing file to store the recorded data, or it allows you to use an existing window and uses the **MCIWndNew**, **MCIWndOpen**, or **MCIWndOpenDialog** macros to specify a file. Then it uses the **MCIWndCanRecord** and **MCIWndCanSave** macros to verify that the device can record and save information. The example then sets the current position by using the **MCIWndHome**, **MCIWndEnd**, and **MCIWndSeek** macros. The **MCIWndRecord** macro is used to start recording, and the **MCIWndStop** or the **MCIWndPause** macro stops recording. After the information is recorded, it is saved by using the **MCIWndSave** or **MCIWndSaveDialog** macros.

```
case WM_COMMAND:
    switch (LOWORD(wParam)) {
    case IDM_CREATEMCIWND:
        g_hwndMCIWnd = MCIWndCreate(hwnd, g_hinst,
            WS_VISIBLE | WS_CHILD |
            MCIWNDF_RECORD,                    // adds Record button
            NULL);

        MCIWndNew(g_hwndMCIWnd, "waveaudio");   // new file

        if (MCIWndCanRecord(g_hwndMCIWnd)) {
            MessageBox(hwnd,
            "Press the red button on the toolbar to start recording.",
            "MCIWnd Record",
            MB_OK);
        }
        else {
            MessageBox(hwnd,
                "This device doesn't record.",
                "MCIWnd Record",
                MB_OK);
        }
            break;
    case IDM_RECORDATSTART:
        if (MCIWndCanRecord(g_hwndMCIWnd)) {
            MCIWndHome(g_hwndMCIWnd);
            MCIWndRecord(g_hwndMCIWnd);
        }
        else {
            MessageBox(hwnd,
                "This device doesn't record.",
                "MCIWnd Record",
                MB_OK);
        }
        break;
    case IDM_RECORDATEND:
        if (MCIWndCanRecord(g_hwndMCIWnd)) {
            MCIWndEnd(g_hwndMCIWnd);
```

```
        MCIWndRecord(g_hwndMCIWnd);
    }
    else {
        MessageBox(hwnd,
            "This device doesn't record.",
            "MCIWnd Record",
            MB_OK);
    }
    break;
case IDM_SAVEMCIWND:
    if (MCIWndCanSave(g_hwndMCIWnd))
        MCIWndSaveDialog(g_hwndMCIWnd);
}
break;
.
. // Handle other messages here.
.
```

Cropping an Image

When you crop an image, you trim one or more edges of the video content from view during playback. Cropping allows you to choose which portions of the content to view without changing the content of the clip.

You can crop one or more edges from a video clip by redefining the dimensions of the source rectangle. The source rectangle overlays each frame in a video clip, encompassing the portion of each image used during playback. Initially, the source rectangle is as large as the video frame size.

The following example creates an MCIWnd window and loads an AVI file. The window includes a crop command in the menu, which crops one-quarter of the height or width off each of the four sides of the frame. The example retrieves the current (initial) dimensions of the source rectangle by using the **MCIWndGetSource** macro. The modified source rectangle is one-half the original height and width and centered in the original frame. The call to the **MCIWndPutSource** macro redefines the coordinates of the source rectangle.

```
// extern RECT rSource, rDest;

case WM_COMMAND:
    switch (LOWORD(wParam)) {
    case IDM_CREATEMCIWND:
        g_hwndMCIWnd = MCIWndCreate(hwnd,
            g_hinst,
            WS_CHILD | WS_VISIBLE,
            "sample.avi");
        break;
    case IDM_CROPIMAGE:                          // crops image
        MCIWndGetSource(g_hwndMCIWnd, &rSource); // source rectangle
        rDest.left = rSource.left +              // new boundaries
            ((rSource.right - rSource.left) / 4);
        rDest.right = rSource.right -
            ((rSource.right - rSource.left) / 4);
        rDest.top = rSource.top +
            ((rSource.bottom - rSource.top) / 4);
        rDest.bottom = rSource.bottom -
            ((rSource.bottom - rSource.top) / 4);

        MCIWndPutSource(g_hwndMCIWnd, &rDest);   // new source rectangle
    }
break;
.
. // Handle other messages here.
.
```

Stretching an Image

The following example stretches the images of a video clip. It increases the dimensions of the destination rectangle by using the **MCIWndPutDest** macro. The size of the playback area remains unchanged, so the result is a distorted, zoomed image. **MCIWndPutDest** is also used to reposition the destination rectangle with respect to the playback area, providing a way to view different portions of the stretched image.

```
// extern RECT rCurrent, rDest;

case WM_COMMAND:
    switch (LOWORD(wParam)) {
    case IDM_CREATEMCIWND:
        g_hwndMCIWnd = MCIWndCreate(hwnd,
             g_hinst,
             WS_CHILD | WS_VISIBLE,
             "sample.avi");
        break;

    case  IDM_STRETCHIMAGE:          // stretches dest. rectangle 3:2
                                     // leaves playback area same size
        MCIWndGetDest(g_hwndMCIWnd, &rCurrent);  // dest. rectangle
        rDest.top = rCurrent.top;                // new boundaries
        rDest.right = rCurrent.right;
        rDest.left = rCurrent.left +
            ((rCurrent.left - rCurrent.right) * 3);
        rDest.bottom = rCurrent.top +
            ((rCurrent.bottom - rCurrent.top) * 2);
        MCIWndPutDest(g_hwndMCIWnd, &rDest); // new dest. rectangle
        break;
    case IDM_MOVEDOWN:               // moves toward bottom of image
        MCIWndGetDest(g_hwndMCIWnd, &rCurrent); // dest. rectangle
        rCurrent.top -= 100;                    // new boundaries
        rCurrent.bottom -= 100;
        MCIWndPutDest(g_hwndMCIWnd, &rCurrent); // new dest. rectangle
        break;
    case IDM_MOVEUP:                 // moves toward top of image
        MCIWndGetDest(g_hwndMCIWnd, &rCurrent); // dest. rectangle
        rCurrent.top += 100;                    // new boundaries
        rCurrent.bottom += 100;
        MCIWndPutDest(g_hwndMCIWnd, &rCurrent); // new dest. rectangle
        break;
    case IDM_MOVELEFT:               // moves toward image left edge
        MCIWndGetDest(g_hwndMCIWnd, &rCurrent); // dest. rectangle
        rCurrent.right += 100;                  // new boundaries
        rCurrent.left += 100;
        MCIWndPutDest(g_hwndMCIWnd, &rCurrent); // new dest. rectangle
        break;
    case  IDM_MOVERIGHT:             // moves toward image right edge
        MCIWndGetDest(g_hwndMCIWnd, &rCurrent); // dest. rectangle
        rCurrent.right -= 100;                  // new boundaries
        rCurrent.left -= 100;
        MCIWndPutDest(g_hwndMCIWnd, &rCurrent); // new dest. rectangle
        break;
    }
```

```
break;
.
. // Handle other messages here.
.
```

Stretching an Image and Window

The following example stretches the images of a video clip and changes the aspect ratio of the displayed frames. The frames displayed in the MCIWnd window are twice the height and three times the width of the original frame. The **MCIWndGetDest** and **MCIWndPutDest** macros retrieve and redefine the destination rectangle coordinates. The **GetWindowRect** and **SetWindowPos** functions manage changes to the MCIWnd window dimensions.

```
// extern RECT rCurrent, rDest;

case WM_COMMAND:
    switch (LOWORD(wParam)) {
           case IDM_CREATEMCIWND:
            g_hwndMCIWnd = MCIWndCreate(hwnd,
                   g_hinst,
                   WS_CHILD | WS_VISIBLE,
                   "sample.avi");
            break;

        case IDM_RESIZEWINDOW:           // stretches dest. rectangle
                                         // and playback area 3:2
            GetWindowRect(g_hwndMCIWnd, &rWin);     // window size
            MCIWndGetDest(g_hwndMCIWnd, &rCurrent); // dest. rectangle
            rDest.top = rCurrent.top;               // new boundaries
            rDest.right = rCurrent.right;
            rDest.left = rCurrent.left +
                ((rCurrent.left - rCurrent.right) * 3);
            rDest.bottom = rCurrent.top +
                ((rCurrent.bottom - rCurrent.top) * 2);
            MCIWndPutDest(g_hwndMCIWnd, &rDest); // new dest. rectangle
            SetWindowPos(g_hwndMCIWnd,           // window being resized
                NULL,                            // z-order: don't care
                0, 0,                            // position: don't care
                rDest.right - rDest.left,        // width
                (rWin.bottom - rWin.top +          // height (window -
                (rCurrent.bottom - rCurrent.top) + // orig dest. rect. +
                (rDest.bottom - rDest.top)),       // new dest. rect.)
                SWP_NOMOVE | SWP_NOZORDER | SWP_NOACTIVATE);
            break;
        }
    break;
    .
    . // Handle other messages here.
    .
```

MCIWnd Reference

This section describes the functions, messages, macros, and notifications associated with the MCIWnd window class. These elements are grouped as follows.

Window Management

MCIWndChangeStyles
MCIWndCreate
MCIWndGetStyles
MCIWndRegisterClass

File and Device Management

MCIWndClose
MCIWndDestroy
MCIWndEject
MCIWndNew
MCIWndOpen
MCIWndOpenDialog
MCIWndSave
MCIWndSaveDialog

Playback Options

MCIWndGetRepeat
MCIWndPlay
MCIWndPlayFrom
MCIWndPlayFromTo
MCIWndPlayReverse
MCIWndPlayTo
MCIWndSetRepeat

Recording

MCIWndRecord

Positioning

MCIWndEnd
MCIWndGetEnd
MCIWndGetLength
MCIWndGetPosition
MCIWndGetPositionString
MCIWndGetStart
MCIWndHome
MCIWndSeek
MCIWndStep

Pause and Resume Playback

MCIWndGetRepeat
MCIWndPlay
MCIWndPlayFrom
MCIWndPlayFromTo
MCIWndPlayReverse
MCIWndPlayTo
MCIWndSetRepeat

Performance Tuning

MCIWndGetSpeed
MCIWndGetVolume

MCIWndGetZoom
MCIWndSetSpeed
MCIWndSetVolume
MCIWndSetZoom

Image and Palette Adjustments

MCIWndGetDest
MCIWndGetPalette
MCIWndGetSource
MCIWndPutDest
MCIWndPutSource
MCIWndRealize
MCIWndSetPalette

Event and Error Notification

MCIWndGetError
MCIWNDM_NOTIFYERROR
MCIWNDM_NOTIFYMEDIA
MCIWNDM_NOTIFYMODE
MCIWNDM_NOTIFYPOS
MCIWNDM_NOTIFYSIZE

Time Formats

MCIWndGetTimeFormat
MCIWndSetTimeFormat
MCIWndUseFrames
MCIWndUseTime
MCIWndValidateMedia

Status Updates

MCIWndGetActiveTimer
MCIWndGetInactiveTimer
MCIWndSetActiveTimer
MCIWndSetInactiveTimer
MCIWndSetTimers

Device Capabilities

MCIWndCanConfig
MCIWndCanEject
MCIWndCanPlay
MCIWndCanRecord
MCIWndCanSave
MCIWndCanWindow

MCI Device Settings

MCIWndGetAlias
MCIWndGetDevice
MCIWndGetDeviceID
MCIWndGetFileName
MCIWndGetMode

MCI Command-String Interface

MCIWndReturnString
MCIWndSendString

MCIWnd Functions

An application uses the MCIWnd functions to register the MCIWnd window class or to register and create an MCIWnd window. An MCIWnd window can use standard window styles as well as a set of MCIWnd-specific styles.

GetOpenFileNamePreview

```
BOOL GetOpenFileNamePreview(LPOPENFILENAME lpofn);
```

Selects a file by using the Open dialog box. The dialog box also allows the user to preview the currently specified AVI file. This function augments the capability found in the **GetOpenFileName** function.

- Returns a handle of the selected file.

lpofn
Address of an **OPENFILENAME** structure used to initialize the dialog box. On return, the structure contains information about the user's file selection.

GetSaveFileNamePreview

```
BOOL GetSaveFileNamePreview(LPOPENFILENAME lpofn);
```

Selects a file by using the SaveAs dialog box. The dialog box also allows the user to preview the currently specified file. This function augments the capability found in the **GetSaveFileName** function.

- Returns a handle of the selected file.

lpofn
Address of an **OPENFILENAME** structure used to initialize the dialog box. On return, the structure contains information about the user's file selection.

MCIWndCreate

```
HWND MCIWndCreate(HWND hwndParent, HINSTANCE hInstance, DWORD dwStyle,
    LPSTR szFile);
```

Registers the MCIWnd window class and creates an MCIWnd window for using MCI services. **MCIWndCreate** can also open an MCI device or file (such as an AVI file) and associate it with the MCIWnd window.

- Returns a handle of an MCIWnd window if successful or zero otherwise.

hwndParent
Handle of the parent window.

hInstance
Handle of the module instance to associate with the MCIWnd window.

dwStyle
Flags defining the window style. In addition to specifying the window styles used with the **CreateWindowEx** function, you can specify the following styles to use with MCIWnd windows:

MCIWNDF_NOAUTOSIZEWINDOW
Will not change the dimensions of an MCIWnd window when the image size changes.

MCIWNDF_NOAUTOSIZEMOVIE
Will not change the dimensions of the destination rectangle when an MCIWnd window size changes.

MCIWNDF_NOERRORDLG
Inhibits display of MCI errors to users.

MCIWNDF_NOMENU
Hides the Menu button from view on the toolbar and prohibits users from accessing its pop-up menu.

MCIWNDF_NOOPEN
Hides the open and close commands from the MCIWnd menu and prohibits users from accessing these choices in the pop-up menu.

MCIWNDF_NOPLAYBAR
Hides the toolbar from view and prohibits users from accessing it.

MCIWNDF_NOTIFYANSI
Causes MCIWnd to use an ANSI string instead of a Unicode string when notifying the parent window of device mode changes. This flag is used in combination with MCIWNDF_NOTIFYMODE and is exclusive to Windows NT.

MCIWNDF_NOTIFYMODE
Causes MCIWnd to notify the parent window with an MCIWNDM_NOTIFYMODE message whenever the device changes operating modes. The *lParam* parameter of this message identifies the new mode, such as MCI_MODE_STOP.

MCIWNDF_NOTIFYPOS
Causes MCIWnd to notify the parent window with an MCIWNDM_NOTIFYPOS message whenever a change in the playback or record position within the content occurs. The *lParam* parameter of this message contains the new position in the content.

MCIWNDF_NOTIFYMEDIA
Causes MCIWnd to notify the parent window with an MCIWNDM_NOTIFYMEDIA message whenever a new device is used or a data file is opened or closed. The *lParam* parameter of this message contains a pointer to the new filename.

MCIWNDF_NOTIFYSIZE
Causes MCIWnd to notify the parent window when the MCIWnd window size changes.

MCIWNDF_NOTIFYERROR

Causes MCIWnd to notify the parent window when an MCI error occurs.

MCIWNDF_NOTIFYALL
Causes all MCIWNDF window notification styles to be used.

MCIWNDF_RECORD
Adds a Record button to the toolbar and adds a new file command to the menu if the MCI device has recording capability.

MCIWNDF_SHOWALL
Causes all MCIWNDF_SHOW styles to be used.

MCIWNDF_SHOWMODE
Displays the current mode of the MCI device in the window title bar. For a list of device modes, see the **MCIWndGetMode** macro.

MCIWNDF_SHOWNAME
Displays the name of the open MCI device or data file in the MCIWnd window title bar.

MCIWNDF_SHOWPOS
Displays the current position within the content of the MCI device in the window title bar.

szFile
Null-terminated string indicating the name of an MCI device or data file to open.

Default window styles for a child window are WS_CHILD, WS_BORDER, and WS_VISIBLE. **MCIWndCreate** assumes a child window when a non-NULL handle of a parent window is specified.

Default window styles for a parent window are WS_OVERLAPPEDWINDOW and WS_VISIBLE. **MCIWndCreate** assumes a parent window when a NULL handle of a parent window is specified.

Use the window handle returned by this function for the window handle in the MCIWnd macros. If your application uses this function, it does not need to use the **MCIWndRegisterClass** function.

MCIWndRegisterClass

```
BOOL MCIWndRegisterClass(HINSTANCE hInstance);
```

Registers the MCI window class MCIWND_WINDOW_CLASS.

- Returns zero if successful.

hInstance
Handle of the device instance.

After registering the MCI window class, use the **CreateWindow** or **CreateWindowEx** functions to create an MCIWnd window. If your application uses this function, it does not need to use the **MCIWndCreate** function.

MCIWnd Macros and Messages

Applications use messages to communicate with MCIWnd windows and MCI devices associated with these windows. MCIWnd macros provide a shorthand method of sending these messages. The macros are based on the **SendMessage** function. Definitions of the macros identify the corresponding messages that are sent to MCIWnd windows.

You can control properties and behavior of MCIWnd windows by using the following macros and messages.

MCIWndCanConfig

```
BOOL MCIWndCanConfig(hwnd)

// Corresponding message
MCIWNDM_CAN_CONFIG
wParam = 0;
lParam = 0;
```

Determines if an MCI device can display a configuration dialog box.

- Returns TRUE if the device supports configuration or FALSE otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndCanEject

```
BOOL MCIWndCanEject(hwnd)

// Corresponding message
MCIWNDM_CAN_EJECT
wParam = 0;
lParam = 0;
```

Determines if an MCI device can eject its media.

- Returns TRUE if the device can eject its media or FALSE otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndCanPlay

```
BOOL MCIWndCanPlay(hwnd)

// Corresponding message
MCIWNDM_CAN_PLAY
wParam = 0;
lParam = 0;
```

Determines if an MCI device can play a data file or content of some other kind.

- Returns TRUE if the device supports playing the data or FALSE otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndCanRecord

```
BOOL MCIWndCanRecord(hwnd)

// Corresponding message
MCIWNDM_CAN_RECORD
wParam = 0;
lParam = 0;
```

Determines if an MCI device supports recording.

- Returns TRUE if the device supports recording or FALSE otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndCanSave

```
BOOL MCIWndCanSave(hwnd)

// Corresponding message
MCIWNDM_CAN_SAVE
wParam = 0;
lParam = 0;
```

Determines if an MCI device can save data.

- Returns TRUE if the device supports saving data or FALSE otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndCanWindow

```
BOOL MCIWndCanWindow(hwnd)

// Corresponding message
MCIWNDM_CAN_WINDOW
wParam = 0;
lParam = 0;
```

Determines if an MCI device supports window-oriented MCI commands.

- Returns TRUE if the device supports window-oriented MCI commands or FALSE otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndChangeStyles

```
LONG MCIWndChangeStyles(hwnd, mask, value)

// Corresponding message
MCIWNDM_CHANGESTYLES
wParam = (WPARAM) (UINT) mask;
lParam = (LPARAM) (LONG) value;
```

Changes the styles used by the MCIWnd window.

- Returns zero.

hwnd
Handle of the MCIWnd window.

mask
Mask that identifies the styles that can change. This mask is the bitwise OR operator of all styles that will be permitted to change.

value
New style settings for the window. Specify zero for this parameter to turn off all styles identified in the mask. For a list of the available styles, see the **MCIWndCreate** function.

For an example of using **MCIWndChangeStyles**, see "Pausing and Resuming Playback" earlier in this chapter.

MCIWndClose

```
LONG MCIWndClose(hwnd)

// Corresponding command
MCI_CLOSE
wParam = 0;
lParam = 0;
```

Closes an MCI device or file associated with an MCIWnd window. Although the MCI device closes, the MCIWnd window is still open and can be associated with another MCI device.

- Returns zero.

hwnd
Handle of the MCIWnd window.

MCIWndDestroy

```
VOID MCIWndDestroy(hwnd)

// Corresponding message
WM_CLOSE
wParam = 0;
lParam = 0;
```

Closes an MCI device or file associated with an MCIWnd window and destroys the window.

- No return value.

hwnd
Handle of the MCIWnd window.

MCIWndEject

```
LONG MCIWndEject(hwnd)

// Corresponding message
MCIWNDM_EJECT
wParam = 0;
lParam = 0;
```

Sends a command to an MCI device to eject its media.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndEnd

```
LONG MCIWndEnd(hwnd)

// Corresponding command
MCI_SEEK
wParam = 0;
lParam = (LPARAM) (LONG) MCIWND_END;
```

Moves the current position to the end of the content.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndGetActiveTimer

```
UINT MCIWndGetActiveTimer(hwnd)

// Corresponding message
MCIWNDM_GETACTIVETIMER
wParam = 0;
lParam = 0L;
```

Retrieves the update period used when the MCIWnd window is the active window.

- Returns the update period in milliseconds. The default is 500 milliseconds.

hwnd
Handle of the MCIWnd window.

MCIWndGetAlias

```
UINT MCIWndGetAlias(hwnd)

// Corresponding message
MCIWNDM_GETALIAS
wParam = 0;
lParam = 0;
```

Retrieves the alias used to open an MCI device or file with the **mciSendString** function.

- Returns the device alias.

hwnd
Handle of the MCIWnd window.

MCIWndGetDest

```
LONG MCIWndGetDest(hwnd, prc)

// Corresponding message
MCIWNDM_GET_DEST
wParam = 0;
lParam = (LPARAM) (LPRECT) prc;
```

Retrieves the coordinates of the destination rectangle used for zooming or stretching the images of an AVI file during playback.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

prc
Address of a **RECT** structure to return the coordinates of the destination rectangle.

MCIWndGetDevice

```
LONG MCIWndGetDevice(hwnd, lp, len)

// Corresponding message
MCIWNDM_GETDEVICE
wParam = (WPARAM) (UINT) len;
lParam = (LPARAM) (LPVOID) lp;
```

Retrieves the name of the currently open MCI device.

- Returns zero if successful or a nonzero value otherwise.

hwnd
Handle of the MCIWnd window.

lp
Address of an application-defined buffer to return the device name.

len
Size, in bytes, of the buffer.

If the null-terminated string containing the device name is longer than the buffer, MCIWnd truncates it.

MCIWndGetDeviceID

```
UINT MCIWndGetDeviceID(hwnd)

// Corresponding message
MCIWNDM_GETDEVICEID
wParam = 0;
lParam = 0;
```

Retrieves the identifier of the currently open MCI device to use with the **mciSendCommand** function.

- Returns the device identifier.

hwnd
Handle of the MCIWnd window.

MCIWndGetEnd

```
LONG MCIWndGetEnd(hwnd)

// Corresponding message
MCIWNDM_GETEND
wParam = 0;
lParam = 0;
```

Retrieves the location of the end of the content of an MCI device or file.

- Returns the location in the current time format.

hwnd
Handle of the MCIWnd window.

MCIWndGetError

```
LONG MCIWndGetError(hwnd, lp, len)

// Corresponding message
MCIWNDM_GETERROR
wParam = (WPARAM) (UINT) len;
lParam = (LPARAM) (LPVOID) lp;
```

Retrieves the last MCI error encountered.

- Returns the integer error value if successful.

hwnd
Handle of the MCIWnd window.

lp
Address of an application-defined buffer used to return the error string.

len
Size, in bytes, of the error buffer.

If *lp* is a valid pointer, a null-terminated string corresponding to the error is returned in its buffer. If the error string is longer than the buffer, MCIWnd truncates it.

MCIWndGetFileName

```
LONG MCIWndGetFileName(hwnd, lp, len)

// Corresponding message
MCIWNDM_GETFILENAME
wParam = (WPARAM) (UINT) len;
lParam = (LPARAM) (LPVOID) lp;
```

Retrieves the filename currently used by an MCI device.

- Returns 0 if successful or 1 otherwise.

hwnd
Handle of the MCIWnd window.

lp
Address of an application-defined buffer to return the filename.

len
Size, in bytes, of the buffer.

If the null-terminated string containing the filename is longer than the buffer, MCIWnd truncates the filename.

MCIWndGetInactiveTimer

```
UINT MCIWndGetInactiveTimer(hwnd)

// Corresponding message
MCIWNDM_GETINACTIVETIMER
wParam = 0;
lParam = 0L;
```

Retrieves the update period used when the MCIWnd window is the inactive window.

- Returns the update period, in milliseconds. The default value is 2000 milliseconds.

hwnd
Handle of the MCIWnd window.

MCIWndGetLength

```
LONG MCIWndGetLength(hwnd)

// Corresponding message
MCIWNDM_GETLENGTH
wParam = 0;
lParam = 0;
```

Retrieves the length of the content or file currently used by an MCI device.

- Returns the length. The units for the length depend on the current time format.

hwnd
Handle of the MCIWnd window.

This value added to the value returned for the **MCIWndGetStart** macro equals the end of the content.

MCIWndGetMode

```
LONG MCIWndGetMode(hwnd, lp, len)

// Corresponding message
MCIWNDM_GETMODE
wParam = (WPARAM) (UINT) len;
lParam = (LPARAM) (LPSTR) lp;
```

Retrieves the current operating mode of an MCI device. MCI devices have several operating modes, which are designated by constants.

- Returns an integer corresponding to the MCI constant defining the mode.

hwnd
Handle of the MCIWnd window.

lp
Address of the application-defined buffer used to return the mode.

len
Size, in bytes, of the buffer.

If the null-terminated string describing the mode is longer than the buffer, it is truncated.

Not all devices can operate in every mode. For example, the MCIAVI device is a playback device; it doesn't support the recording mode. The following modes can be retrieved by using MCIWNDM_GETMODE:

Operating mode	**MCI constant**
not ready	MCI_MODE_NOT_READY
open	MCI_MODE_OPEN
paused	MCI_MODE_PAUSE
playing	MCI_MODE_PLAY
recording	MCI_MODE_RECORD
seeking	MCI_MODE_SEEK
stopped	MCI_MODE_STOP

MCIWndGetPalette

```
HPALETTE MCIWndGetPalette(hwnd)

// Corresponding message
MCIWNDM_GETPALETTE
wParam = 0;
lParam = 0;
```

Retrieves a handle of the palette used by an MCI device.

- Returns the handle of the palette if successful.

hwnd
Handle of the MCIWnd window.

MCIWndGetPosition

```
LONG MCIWndGetPosition(hwnd)

// Corresponding messages
MCIWNDM_GETPOSITION
wParam = 0;
lParam = 0;
```

Retrieves the numerical value of the current position within the content of the MCI device.

- Returns an integer corresponding to the current position. The units for the position value depend on the current time format.

hwnd
Handle of the MCIWnd window.

MCIWndGetPositionString

```
LONG MCIWndGetPositionString(hwnd, lp, len)

// Corresponding messages
MCIWNDM_GETPOSITION
wParam = (WPARAM) (UINT) len;
lParam = (LPARAM) (LPTSTR) lp;
```

Retrieves the numerical value of the current position within the content of the MCI device. This macro also provides the current position in string form in an application-defined buffer.

- Returns an integer corresponding to the current position. The units for the position value depend on the current time format.

hwnd
Handle of the MCIWnd window.

lp
Address of an application-defined buffer used to return the position. Use zero to inhibit retrieval of the position as a string.

If the device supports tracks, the string position information is returned in the form TT:MM:SS:FF where TT corresponds to tracks, MM and SS correspond to minutes and seconds, and FF corresponds to frames.

len
Size, in bytes, of the buffer. If the null-terminated string is longer than the buffer, it is truncated. Use zero to inhibit retrieval of the position as a string.

MCIWndGetRepeat

```
BOOL MCIWndGetRepeat(hwnd)

// Corresponding message
MCIWNDM_GETREPEAT
wParam = 0;
lParam = 0;
```

Determines if continuous playback has been activated.

- Returns TRUE if continuous playback is activated or FALSE otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndGetSource

```
LONG MCIWndGetSource(hwnd, prc)

// Corresponding message
MCIWNDM_GET_SOURCE
wParam = 0;
lParam = (LPARAM) (LPRECT) prc;
```

Retrieves the coordinates of the source rectangle used for cropping the images of an AVI file during playback.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

prc
Address of a **RECT** structure to contain the coordinates of the source rectangle.

MCIWndGetSpeed

```
LONG MCIWndGetSpeed(hwnd)

// Corresponding message
MCIWNDM_GETSPEED
wParam = 0;
lParam = 0;
```

Retrieves the playback speed of an MCI device.

- Returns the playback speed if successful. The value for normal speed is 1000. Larger values indicate faster speeds; smaller values indicate slower speeds.

hwnd
Handle of the MCIWnd window.

MCIWndGetStart

```
LONG MCIWndGetStart(hwnd)

// Corresponding message
MCIWNDM_GETSTART
wParam = 0;
lParam = 0;
```

Retrieves the location of the beginning of the content of an MCI device or file.

- Returns the location in the current time format.

hwnd
Handle of the MCIWnd window.

Typically, the return value is zero; but some devices use a nonzero starting location. Seeking to this location sets the device to the start of the media.

MCIWndGetStyles

```
UINT MCIWndGetStyles(hwnd)

// Corresponding message
MCIWNDM_GETSTYLES
wParam = 0;
lParam = 0;
```

Retrieves the flags specifying the current MCIWnd window styles used by a window.

- Returns a value representing the current styles of the MCIWnd window. The return value is the bitwise OR operator of the MCIWnd window styles (MCIWNDF flags).

hwnd
Handle of the MCIWnd window.

MCIWndGetTimeFormat

```
LONG MCIWndGetTimeFormat(hwnd, lp, len)

// Corresponding message
MCIWNDM_GETTIMEFORMAT
wParam = (WPARAM) (UINT) len;
lParam = (LPARAM) (LPSTR) lp;
```

Retrieves the current time format of an MCI device in two forms: as a numerical value and as a string.

- Returns an integer corresponding to the MCI constant defining the time format.

hwnd
Handle of the MCIWnd window.

lp
Address of a buffer to contain the null-terminated string form of the time format.

len
Size, in bytes, of the buffer.

If the time format string is longer than the return buffer, MCIWnd truncates the string.

An MCI device can support one or more of the following time formats:

Time format	**MCI constant**
Bytes	MCI_FORMAT_BYTES
Frames	MCI_FORMAT_FRAMES
Hours, minutes, seconds	MCI_FORMAT_HMS
Milliseconds	MCI_FORMAT_MILLISECONDS
Minutes, seconds, frames	MCI_FORMAT_MSF
Samples	MCI_FORMAT_SAMPLES
SMPTE 24	MCI_FORMAT_SMPTE_24
SMPTE 25	MCI_FORMAT_SMPTE_25
SMPTE 30 drop	MCI_FORMAT_SMPTE_30DROP
SMPTE 30 (non-drop)	MCI_FORMAT_SMPTE_30
Tracks, minutes, seconds, frames	MCI_FORMAT_TMSF

MCIWndGetVolume

```
LONG MCIWndGetVolume(hwnd)

// Corresponding message
MCIWNDM_GETVOLUME
wParam = 0;
lParam = 0;
```

Retrieves the current volume setting of an MCI device.

- Returns the current volume setting. The default value is 1000. Higher values indicate louder volumes; lower values indicate quieter volumes.

hwnd
Handle of the MCIWnd window.

MCIWndGetZoom

```
UINT MCIWndGetZoom(hwnd)

// Corresponding message
MCIWNDM_GETZOOM
wParam = 0;
lParam = 0;
```

Retrieves the current zoom value used by an MCI device.

- Returns the most recent values used with **MCIWndSetZoom**.

hwnd
Handle of the MCIWnd window.

A return value of 100 indicates the image is not zoomed. A value of 200 indicates the image is enlarged to twice its original size. A value of 50 indicates the image is reduced to half its original size.

MCIWndHome

```
LONG MCIWndHome(hwnd)

// Corresponding command
MCI_SEEK
wParam = 0;
lParam = (LPARAM) (LONG) MCIWND_START;
```

Moves the current position to the beginning of the content.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndNew

```
LONG MCIWndNew(hwnd, lp)

// Corresponding message
MCIWNDM_NEW
wParam = 0;
lParam = (LPARAM) (LPVOID) lp;
```

Creates a new file for the current MCI device.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

lp
Address of a buffer containing the name of the MCI device that will use the file.

MCIWndOpen

```
LONG MCIWndOpen(hwnd, szFile, wFlags)

// Corresponding message
MCIWNDM_OPEN
wParam = (WPARAM) (UINT) wFlags;
lParam = (LPARAM) (LPVOID) szFile;
```

Opens an MCI device and associates it with an MCIWnd window. For MCI devices that use data files, this macro can also open a specified data file, name a new file to be created, or display a dialog box to let the user select a file to open.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

szFile
Address of a null-terminated string identifying the filename or MCI device name to open. Specify -1 for this parameter to display the Open dialog box.

wFlags
Flags associated with the device or file to open. The MCIWNDOPENF_NEW flag specifies a new file is to be created with the name specified in *szFile*.

MCIWndOpenDialog

```
LONG MCIWndOpenDialog(hwnd)

// Corresponding command
MCI_OPEN
wParam = -1;
lParam = 0;
```

Opens a user-specified data file and corresponding type of MCI device, and associates them with an MCIWnd window. This macro displays the Open dialog box for the user to select the data file to associate with an MCI window.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndOpenInterface

```
MCIWndOpenInterface(hwnd, pUnk)

// Corresponding message
MCIWNDM_OPENINTERFACE
wParam = 0;
lParam = (LPARAM) (LPUNKNOWN) pUnk;
```

Attaches the data stream or file associated with the specified interface to an MCIWnd window.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

pUnk
Address of an **IAVI** interface that points to a file or a data stream in a file.

For information about **IAVI** interfaces, see Chapter 6, "AVIFile Functions and Macros ."

MCIWndPause

```
LONG MCIWndPause(hwnd)

// Corresponding command
MCI_PAUSE
wParam = 0;
lParam = 0;
```

Sends a command to an MCI device to pause playing or recording.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndPlay

```
LONG MCIWndPlay(hwnd)

// Corresponding command
MCI_PLAY
wParam = 0;
lParam = 0;
```

Sends a command to an MCI device to start playing from the current position in the content.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndPlayFrom

```
LONG MCIWndPlayFrom(hwnd, lPos)

// Corresponding message
MCIWNDM_PLAYFROM
wParam = 0;
lParam = (LPARAM) (LONG) lPos;
```

Plays the content of an MCI device from the specified location to the end of the content or until another command stops playback.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

lPos
Starting location. The units for the starting location depend on the current time format.

You can also specify both a starting and ending location for playback by using the **MCIWndPlayFromTo** macro.

MCIWndPlayFromTo

```
LONG MCIWndPlayFromTo(hwnd, lStart, lEnd)

// Corresponding command and message
MCI_SEEK
wParam = 0;
lParam = (LPARAM) (LONG) lStart;

MCIWNDM_PLAYTO
wParam = 0;
lParam = (LPARAM) (LONG) lEnd;
```

Plays a portion of content between specified starting and ending locations. This macro seeks to the specified point to begin playback, then plays the content to the specified ending location. This macro is defined using the **MCIWndSeek** and **MCIWndPlayTo** macros, which in turn use the **MCI_SEEK** command and the MCIWNDM_PLAYTO message.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

lStart
Position to seek; it is also the starting location.

lEnd
Ending location.

The units for the seek position depend on the current time format.

MCIWndPlayReverse

```
LONG MCIWndPlayReverse(hwnd)

// Corresponding message
MCIWNDM_PLAYREVERSE
wParam = 0;
lParam = 0;
```

Plays the current content in the reverse direction, beginning at the current position and ending at the beginning of the content or until another command stops playback.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndPlayTo

```
LONG MCIWndPlayTo(hwnd, lPos)

// Corresponding message
MCIWNDM_PLAYTO
wParam = 0;
lParam = (LPARAM) (LONG) lPos;
```

Plays the content of an MCI device from the current position to the specified ending location or until another command stops playback. If the specified ending location is beyond the end of the content, playback stops at the end of the content.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

lPos
Ending location. The units for the ending location depend on the current time format.

You can also specify both a starting and ending location for playback by using the **MCIWndPlayFromTo** macro.

MCIWndPutDest

```
LONG MCIWndPutDest(hwnd, prc)

// Corresponding message
MCIWNDM_PUT_DEST
wParam = 0;
lParam = (LPARAM) (LPRECT) prc;
```

Redefines the coordinates of the destination rectangle used for zooming or stretching the images of an AVI file during playback.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

prc
Address of a **RECT** structure containing the coordinates of the destination rectangle.

MCIWndPutSource

```
LONG MCIWndPutSource(hwnd, prc)

// Corresponding message
MCIWNDM_PUT_SOURCE
wParam = 0;
lParam = (LPARAM) (LPRECT) prc;
```

Redefines the coordinates of the source rectangle used for cropping the images of an AVI file during playback.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

prc
Address of a **RECT** structure containing the coordinates of the source rectangle.

MCIWndRealize

```
LONG MCIWndRealize(hwnd, fBkgnd)

// Corresponding message
MCIWNDM_REALIZE
wParam = (WPARAM) (BOOL) fBkgnd;
lParam = 0;
```

Realizes the palette currently used by the MCI device in an MCIWnd window. This macro is defined with the MCIWNDM_REALIZE message.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

fBkgnd
Background flag. Specify TRUE for this parameter if the window is a background application.

MCIWNDM_REALIZE uses the palette of the MCI device and calls the **RealizePalette** function. If your application explicitly handles the WM_PALETTECHANGED and WM_QUERYNEWPALETTE messages, you should use this message in your application instead of using **RealizePalette**. If the body of one of these message handlers contains only **RealizePalette**, forward the message to the MCIWnd window, which will automatically realize the palette.

MCIWndRecord

```
LONG MCIWndRecord(hwnd)

// Corresponding command
MCI_RECORD
wParam = 0;
lParam = 0;
```

Begins recording content using the MCI device. The recording process begins at the current position in the content and will overwrite existing data for the duration of the recording.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

The function that an MCI device performs during recording depends on the characteristics of the device. An MCI device that uses files, such as a waveform-audio device, sends data to the file during recording. An MCI device that does not use files, such as a video-cassette recorder, receives and externally records data on another medium.

MCIWndResume

```
LONG MCIWndResume(hwnd)

// Corresponding command
MCI_RESUME
wParam = 0;
lParam = 0;
```

Resumes playback or recording content from the paused mode. This macro restarts playback or recording from the current position in the content.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndReturnString

```
LONG MCIWndReturnString(hwnd, lp, len)

// Corresponding message
MCIWNDM_RETURNSTRING
wParam = (WPARAM) (UINT) len;
lParam = (LPARAM) (LPVOID) lp;
```

Retrieves the reply to the most recent MCI string command sent to an MCI device. Information in the reply is supplied as a null-terminated string.

- Returns an integer corresponding to the MCI string.

hwnd
Handle of the MCIWnd window.

lp
Address of an application-defined buffer to contain the null-terminated string.

len
Size, in bytes, of the buffer.

If the null-terminated string is longer than the buffer, the string is truncated.

MCIWndSave

```
LONG MCIWndSave(hwnd, szFile)

// Corresponding command
MCI_SAVE
wParam = 0;
lParam = (LPARAM) (LPVOID) szFile;
```

Saves the content currently used by an MCI device. This macro can save the content to a specified data file or display the Save dialog box to let the user select a filename to store the content.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

szFile
Null-terminated string containing the name and path of the destination file. Specify - 1 for this parameter to display the Save dialog box.

MCIWndSaveDialog

```
LONG MCIWndSaveDialog(hwnd)

// Corresponding command
MCI_SAVE
wParam = 0;
lParam = -1;
```

Saves the content currently used by an MCI device. This macro displays the Save dialog box to let the user select a filename to store the content.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndSeek

```
LONG MCIWndSeek(hwnd, lPos)

// Corresponding command
MCI_SEEK
wParam = 0;
lParam = (LPARAM) (LONG) lPos;
```

Moves the playback position to the specified location in the content.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

lPos
Position to seek. You can specify a position using the current time format, the MCIWND_START constant to designate the beginning of the content, or the MCIWND_END constant to designate the end of the content.

MCIWndSendString

```
LONG MCIWndSendString(hwnd, sz)

// Corresponding message
MCIWNDM_SENDSTRING
wParam = 0;
lParam = (LPARAM) (LPSTR) sz;
```

Sends an MCI command in string form to the device associated with the MCIWnd window.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

sz
String command to send to the MCI device.

The message handler for **MCIWndSendString** (and MCIWNDM_SENDSTRING) appends a device alias to the MCI command you send to the device. Therefore, you should not use any alias in an MCI command that you issue with **MCIWndSendString**.

For more information about MCI string commands, see Chapter 4, "MCI Command Strings."

MCIWndSetActiveTimer

```
VOID MCIWndSetActiveTimer(hwnd, active)

// Corresponding message
MCIWNDM_SETACTIVETIMER
wParam = (WPARAM) (UINT) active;
lParam = 0L;
```

Sets the update period used by MCIWnd to update the trackbar in the MCIWnd window, update position information displayed in the window title bar, and send notification messages to the parent window when the MCIWnd window is active.

- No return value.

hwnd
Handle of the MCIWnd window.

active
Update period, in milliseconds. The default is 500 milliseconds.

MCIWndSetInactiveTimer

```
VOID MCIWndSetInactiveTimer(hwnd, inactive)

// Corresponding message
MCIWNDM_SETINACTIVETIMER
wParam = (WPARAM) (UINT) inactive;
lParam = 0;
```

Sets the update period used by MCIWnd to update the trackbar in the MCIWnd window, update position information displayed in the window title bar, and send notification messages to the parent window when the MCIWnd window is inactive.

- No return value.

hwnd
Handle of the MCIWnd window.

inactive
Update period, in milliseconds. The default is 2000 milliseconds.

MCIWndSetOwner

```
LONG MCIWndSetOwner(hwnd, hwndP)

// Corresponding message
MCIWNDM_SETOWNER
wParam = (WPARAM) hwndP;
lParam = 0;
```

Sets the window to receive notification messages associated with the MCIWnd window.

- Returns zero.

hwnd
Handle of the MCIWnd window.

hwndP
Handle of the window to receive the notification messages.

MCIWndSetPalette

```
MCIWndSetPalette(hwnd, hpal)

// Corresponding message
MCIWNDM_SETPALETTE
wParam = (WPARAM) (HPALETTE) hpal;
lParam = 0;
```

Sends a palette handle to the MCI device associated with the MCIWnd window.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

hpal
Palette handle.

MCIWndSetRepeat

```
VOID MCIWndSetRepeat(hwnd, f)

// Corresponding message
MCIWNDM_SETREPEAT
wParam = 0;
lParam = (LPARAM) (BOOL) f;
```

Sets the repeat flag associated with continuous playback. The MCIWNDM_SETREPEAT message works in conjunction with the **MCI_PLAY** command to provide a continuous playback loop.

- Returns zero.

hwnd
Handle of the MCIWnd window.

f
New state of the repeat flag. Specify TRUE to turn on continuous playback.

Currently, MCIAVI is the only device that supports continuous playback.

MCIWndSetSpeed

```
LONG MCIWndSetSpeed(hwnd,iSpeed)

// Corresponding message
MCIWNDM_SETSPEED
wParam = 0;
lParam = (LPARAM) (UINT) iSpeed;
```

Sets the playback speed of an MCI device.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

iSpeed
Playback speed. Specify 1000 for normal speed, larger values for faster speeds, and smaller values for slower speeds.

MCIWndSetTimeFormat

```
LONG MCIWndSetTimeFormat(hwnd, lp)

// Corresponding message
MCIWNDM_SETTIMEFORMAT
wParam = 0;
lParam = (LPARAM) (LPSTR) lp;
```

Sets the time format of an MCI device.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

lp
Address of a buffer containing the null-terminated string indicating the time format. Specify "frames" to set the time format to frames, or "ms" to set the time format to milliseconds.

An application can specify time formats other than frames or milliseconds as long as the formats are supported by the MCI device. Noncontinuous formats, such as tracks and SMPTE, can cause the trackbar to behave erratically. For these time formats, you might want to turn off the toolbar by using the **MCIWndChangeStyles** macro and specifying the MCIWNDF_NOPLAYBAR window style.

If you want to set the time format to frames or milliseconds, you can also use the **MCIWndUseFrames** or **MCIWndUseTime** macro. For a list of time formats, see the **MCIWndGetTimeFormat** macro.

MCIWndSetTimers

```
VOID MCIWndSetTimers(hwnd, active, inactive)

// Corresponding message
MCIWNDM_SETTIMERS
wParam = (WPARAM) (UINT) active;
lParam = (LPARAM) (UINT) inactive;
```

Sets the update periods used by MCIWnd to update the trackbar in the MCIWnd window, update the position information displayed in the window title bar, and send notification messages to the parent window.

- No return value.

hwnd
Handle of the MCIWnd window.

active
Update period used by MCIWnd when it is the active window. The default value is 500 milliseconds. Storage for this value is limited to 16 bits.

inactive
Update period used by MCIWnd when it is the inactive window. The default value is 2000 milliseconds. Storage for this value is limited to 16 bits.

MCIWndSetVolume

```
LONG MCIWndSetVolume(hwnd, iVol)

// Corresponding message
MCIWNDM_SETVOLUME
wParam = 0;
lParam = (LPARAM) (UINT) iVol;
```

Sets the volume level of an MCI device.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

iVol
New volume level. Specify 1000 for normal volume level. Specify a higher value for a louder volume or a lower value for a quieter volume.

MCIWndSetZoom

```
VOID MCIWndSetZoom(hwnd, iZoom)

// Corresponding message
MCIWNDM_SETZOOM
wParam = 0;
lParam = (LPARAM) (UINT) iZoom;
```

Resizes a video image according to a zoom factor. This marco adjusts the size of an MCIWnd window while maintaining a constant aspect ratio.

- No return value.

hwnd
Handle of the MCIWnd window.

iZoom
Zoom factor expressed as a percentage of the original image. Specify 100 to display the image at its authored size, 200 to display the image at twice its normal size, or 50 to display the image at half its normal size.

MCIWndStep

```
LONG MCIWndStep(hwnd, n)

// Corresponding command
MCI_STEP
wParam = 0;
lParam = (LPARAM) (long) n;
```

Moves the current position in the content forward or backward by a specified increment.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

n
Step value. Negative values step the device through the content in reverse. The units for the step value depend on the current time format.

MCIWndStop

```
LONG MCIWndStop(hwnd)

// Corresponding command
MCI_STOP
wParam = 0;
lParam = 0;
```

Stops playing or recording the content of the MCI device associated with the MCIWnd window.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndUseFrames

```
LONG MCIWndUseFrames(hwnd)

// Corresponding message
MCIWNDM_SETTIMEFORMAT
wParam = 0;
lParam = TEXT ("frames");
```

Sets the time format of an MCI device to frames.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndUseTime

```
LONG MCIWndUseTime(hwnd)

// Corresponding message
MCIWNDM_SETTIMEFORMAT
wParam = 0;
lParam = TEXT ("ms");
```

Sets the time format of an MCI device to milliseconds.

- Returns zero if successful or an error otherwise.

hwnd
Handle of the MCIWnd window.

MCIWndValidateMedia

```
VOID MCIWndValidateMedia(hwnd)

// Corresponding message
MCIWNDM_VALIDATEMEDIA
wParam = 0;
lParam = 0;
```

Updates the starting and ending locations of the content, the current position in the content, and the trackbar according to the current time format.

- No return value.

hwnd
Handle of the MCIWnd window.

Typically, you should not need to use this macro; however, if your application changes the time format of a device without using MCIWnd; the starting and ending locations of the content, as well as the trackbar, continue to use the old format. You can use this macro to update these values.

MCIWnd Notifications

The MCIWnd message handler can send event notifications to the parent of an MCIWnd window in the form of MCIWNDM_NOTIFY messages. You can enable notification for an event type by specifying an appropriate style for an MCIWnd window when the window is created or its styles are updated. The following notifications are specific to MCIWnd windows.

MCIWNDM_NOTIFYERROR

```
MCIWNDM_NOTIFYERROR
wParam = (WPARAM) (HWND) hwnd;
lParam = (LPARAM) (LONG) errorCode;
```

Notifies the parent window of an application that an MCI error occurred.

hwnd
Handle of the MCIWnd window.

errorCode
Numerical code for the MCI error.

You can enable MCI error notification by specifying the MCIWNDF_NOTIFYERROR window style.

MCIWNDM_NOTIFYMEDIA

```
MCIWNDM_NOTIFYMEDIA
wParam = (WPARAM) (HWND) hwnd;
lParam = (LPARAM) (LPSTR) lp;
```

Notifies the parent window of an application that the media has changed.

hwnd
Handle of the MCIWnd window.

lp
Address of a null-terminated string containing the new filename. If the media is closing, it specifies a null string.

You can enable notification of media changes by specifying the MCIWNDF_NOTIFYMEDIA window style.

MCIWNDM_NOTIFYMODE

```
MCIWNDM_NOTIFYMODE
wParam = (WPARAM) (HWND) hwnd;
lParam = (LPARAM) (LONG) mode;
```

Notifies the parent window of an application that the operating mode of the MCI device has changed.

hwnd
Handle of the MCIWnd window.

mode
Integer corresponding to the MCI mode.

You can enable notification of mode changes of an MCI device by specifying the MCIWNDF_NOTIFYMODE window style.

MCIWNDM_NOTIFYPOS

```
MCIWNDM_NOTIFYPOS
wParam = (WPARAM) (HWND) hwnd;
lParam = (LPARAM) (LONG) pos;
```

Notifies the parent window of an application that the window position has changed.

hwnd
Handle of the MCIWnd window.

pos
Describes the new position.

You can enable notification of changes in the position of an MCIWnd window by specifying the MCIWNDF_NOTIFYPOS window style.

MCIWNDM_NOTIFYSIZE

```
MCIWNDM_NOTIFYSIZE
wParam = (WPARAM) (HWND) hwnd;
lParam = 0;
```

Notifies the parent window of an application that the window size has changed.

hwnd
Handle of the MCIWnd window.

You can enable notification of changes in the size of an MCIWnd window by specifying the MCIWNDF_NOTIFYSIZE window style.

Multimedia Possibilities

A multimedia application delivers information in ways that can be more powerful than printed material or standard video and sound. Unlike printed material, a multimedia application communicates using more than a series of static images or text. Unlike standard video or sound presentations, a multimedia application allows the user to navigate through media and interact with information quickly and easily. The new information media that typically define a multimedia application are sound and video. An application that incorporates sound, video, or both is a multimedia application.

For many years, computers were expected to arrange data, but not to deliver it. The result of the user's work was typically printed on paper. Today, however, inexpensive computers are capable of overcoming many of the inherent limitations of printed material. Practically any computer that uses the Microsoft® Windows® operating system and has a VGA monitor and a sound card can exploit many of the features of a multimedia application. Millions of computer users already own equipment like this, and many also have more powerful multimedia computers with monitors that can display 256 colors or more and compact disc (CD) players. More and more, these computers are becoming the final delivery system for information. People are sending electronic mail instead of letters. Instead of reaching for a bulky printed encyclopedia, they are enjoying the full-color graphics, sound, and animation of a CD-based encyclopedia.

A multimedia application written for the Microsoft Win32® application programming interface (API) delivers information in ways that a printed page cannot. Even when the focus of the application is to help a user produce a printed document or perform calculations, the application can use sound, video, or both to enrich the user's experience.

The definition of a multimedia computer has been established by an industry-wide group, the Multimedia PC Marketing Council. This council has defined two sets of minimum specifications for multimedia computers. For a description of these specifications, see Multimedia PC Specifications. An application does not need to take full advantage of all of this hardware to qualify as a multimedia application.

Developing multimedia applications can be as simple as adding an existing sound or video recording to an application or as complex as building an editing tool for customizing multimedia presentations. No matter how complex your goal, this volume will help you achieve it.

The intended audience for this book is programmers who already have some background in the opportunities presented by multimedia computing. It focuses strictly on helping readers implement the multimedia features of the Win32 API. This book will help programmers who want to incorporate multimedia capabilities into their applications, create multimedia-based applications, or create tools for writing or editing multimedia publications.

What You Can Do with Multimedia

If you think about it for a few minutes, you are sure to come up with a number of applications that could use sound and video in new and exciting ways. For example, real estate agents have long organized descriptions and photographs of homes in large catalogs. Because these catalogs are printed on paper, the presentation of the homes is limited to a small picture and a paragraph or two of text. If the catalog were produced as a multimedia application, on the other hand, it could display a guided audio and visual tour of the inside and outside of these homes. Having potential buyers view these listings could be a powerful sales tool and could prevent wasted trips to unsuitable locations.

This real estate application is just one example of what you can do with multimedia. You can use multimedia to create applications that play, edit, and capture sounds and images. You can also create applications that can control multimedia hardware, such as CD players, video-cassette recorders, and MIDI (Musical Instrument Digital Interface) devices. For entertainment applications, the Win32 API also supports the use of joysticks and provides a timer mechanism that is more accurate than the standard Win32 timers.

Many developers use multimedia to improve applications that did not use sound and video when they were first designed and written. For example, developers are adding voice-annotation capabilities to word-processing applications, or video clips to presentation-graphics applications. More generally, developers are adding sounds to all types of applications; for example, to request input from the user (such as a password) or to signify an action (such as opening or closing a file).

Some applications integrate multimedia features more completely. Software developers are creating hundreds of such applications, such as entertainment programs, computerized reference works, and educational programs. Because extensive use of sound or video requires a great deal of data-storage space, these applications are often distributed on CDs.

You can create multimedia applications for anyone who routinely needs fast access to large amounts of data. These applications are often written for niche markets; the multimedia real estate catalogue discussed earlier is a good example. The possible variety and uses of such applications are almost limitless:

- Doctors could consolidate their records about a case on a CD, including not only all the relevant reports and histories but also such items as videotapes of surgical procedures and consultations with the client.
- Telephone directories for businesses could include full-color graphics, sophisticated search capabilities, audio clips, and detailed maps.
- A voicemail system could be integrated into an electronic mail application. It is feasible that a combination of multimedia software and hardware could be used to integrate the functions currently performed by telephones, modems, faxes, voicemail systems, and electronic mail applications.

Multimedia Playback with One Function Call

You can play waveform-audio files, CDs, video clips, or MIDI files in your application with a call to a single function: **MCIWndCreate**. This function creates a button that the user can use to play or stop the playback, a trackbar that displays the current position in the file, and, in the case of a video clip, a window in which the video is displayed. The following call to **MCIWndCreate** plays the video clip SAMPLE.AVI:

```
MCIWndCreate(hwndParent,                        // parent window handle
    g_hinst,                                    // instance handle
    WS_VISIBLE | WS_CHILD | MCIWNDF_SHOWALL,    // window styles
    "sample.avi");                              // filename
```

Another function, **PlaySound**, also enables you to implement multimedia playback with a single function call. You can use this function to play a waveform-audio file. For example, the following line of code plays the sound stored in the file CHIMES.WAV:

```
PlaySound("chimes.wav", NULL, SND_SYNC);
```

Note **PlaySound** cannot play a waveform-audio file larger than will fit in available memory. To play larger files, you should use either the MCIWnd window class or the high-level audio interface. For more information about the MCIWnd window class than is presented in this section, see Getting Started Using MCIWnd. For more information about **PlaySound**, see Waveform Audio.

Multimedia Data Formats

Windows supports three distinct types of multimedia data: waveform audio, MIDI sound, and video.

Waveform audio is a digitized recording of a sound. You can typically edit waveform audio using insertions and deletions, or you can modify it using filters. This sound format can store voice, music, and sound effects exactly as they should be heard by the user. Compared to MIDI sound, however, editing waveform audio is difficult and the storage requirements are high. In Windows, waveform-audio files typically have a .WAV filename extension.

MIDI sounds are stored as a series of instructions, rather than as a waveform. A synthesizer (often part of the computer's sound card) interprets the instructions to produce the sound. Because different synthesizers interpret MIDI instructions with greatly varying quality, the sound heard by the user cannot be guaranteed. This sound format can store music, and sometimes sound effects, but voice is not a practical option. Compared to waveform audio, however, MIDI is easy to edit and the storage requirements are low. In Windows, MIDI sound files typically have a .MID filename extension.

Video is a multiple-track recording that includes waveform audio and moving images. The moving images are recorded as a series of still images. In Windows, video files typically have an .AVI filename extension.

Version Checking

You may need to check the installed version of the multimedia system, particularly if your application takes advantage of features that were not available in previous releases. Although the multimedia header files contain two version-checking functions, they are obsolete. These obsolete functions are **mmsystemGetVersion** and **VideoForWindowsVersion**. Your application should rely on the standard Windows functions, **GetVersion** or **GetVersionEx**, instead.

The Multimedia Documentation

Unless you are developing a very complex multimedia application, you need not read all of the multimedia documentation. This volume is divided into five parts; the parts you need to read depend on the type of application you are writing. Your application can interact with a multimedia device using a high-level, mid-level, or low-level interface. The parts of this volume mirror this hierarchy of implementation levels.

Part 1 discusses how to design applications that use high-level interfaces based on window classes. Read this part if you want to add sound or video to an application and you do not need to implement complicated editing or recording functionality.

Part 2 discusses how to design applications that use a mid-level interface − called the Media Control Interface (MCI) − which offers applications a standard set of commands to use when communicating with a multimedia device. Read this part if you want to implement a customized user interface for your video or sound files but you do not need to take full advantage of the capabilities of a particular device.

Parts 3, 4, and 5 discuss how to design applications that use a set of low-level multimedia interfaces. These interfaces allow applications to achieve nearly complete control over an audio or video presentation. Read these parts if your application needs to take full advantage of one or more multimedia devices, if you plan to implement recording or editing features, or if you need a custom format for your data.

High-Level Interfaces

An application can use the MCIWnd window class to play a video, MIDI, or waveform-audio file. Several functions that play only waveform audio are also available. Part 1 of this volume describes the most widely used parts of this high-level multimedia interface. In addition, Multimedia Playback with One Function Call earlier shows you how to use this high-level interface to play a video or sound file very easily.

Another window class, AVICap, makes it easy to develop an interface for capturing video clips. For more information about AVICap, see Video Capture.

Mid-Level Interface

MCI is a device-independent interface for controlling virtually any multimedia device. Although many MCI commands are appropriate for any multimedia device, some commands exploit the features of a particular device or class of devices. You can use this mid-level interface to implement a customized user interface and achieve greater control over a multimedia device while still developing applications simply and quickly. Part 2 of this topic describes MCI.

Low-Level Interfaces

Part 3 of this topic describes advanced video techniques, including how to work with video files, and how to work with the compression management component that provides compression and decompression services for these files.

Part 4 describes advanced audio techniques (including MIDI services that are not available through the MCI interface documented in Part 2), advanced waveform-audio techniques, audio mixers, and the component for managing audio compression.

Other parts of the low-level interface to multimedia are discussed in Part 5 and the appendixes. These subjects include joysticks and multimedia timers, the file input and output services for multimedia files, and file formats for multimedia data files.

MCI Command Messages

The Media Control Interface (MCI) is a high-level command interface to multimedia devices and resource files. MCI provides standard commands for playing multimedia devices and recording multimedia resource files. MCI commands are a generic interface to multimedia devices.

There are two forms of MCI commands: strings and messages. You can use either or both forms in your MCI application. This chapter documents the command-message interface to MCI. For information about the command-string interface, see Chapter 4, "MCI Command Strings." For an overview of MCI, including information about whether you should use the string interface or the message interface in your application, see Chapter 3, "MCI Overview."

The command-message interface is designed to be used by applications requiring a C-language interface to control multimedia devices. It uses a message-passing paradigm to communicate with MCI devices. You can send a command by using the **mciSendCommand** function.

Syntax of Command Messages

MCI command messages consist of the following three elements:

- A constant message value
- A structure containing parameters for the command
- A set of flags specifying options for the command and validating fields in the parameter block

The following example sends the **MCI_PLAY** command to the device identified by a device identifier:

```
mciSendCommand(wDeviceID,            // device identifier
    MCI_PLAY,                        // command message
    0,                               // flags
    (DWORD)(LPVOID) &mciPlayParms);  // parameter block
```

The device identifier given in the first parameter is retrieved when the device is opened using the **MCI_OPEN** command. The last parameter is the address of an **MCI_PLAY_PARMS** structure, which might contain information about where to begin and end playback. Many MCI command messages use a structure to contain parameters of this kind. The first member of each of these structures identifies the window that receives an MM_MCINOTIFY message when the operation finishes.

Sending Command Messages

The Microsoft Windows operating system provides two functions for sending command messages to devices and to query devices for error information: **mciSendCommand** and **mciGetErrorString**. The **mciSendCommand** function sends a command message to an MCI device. The **mciGetErrorString** function returns the error string corresponding to an error number.

The **mciSendCommand** function returns zero if successful. If the function fails, the low-order word of the return value contains an error code. You can pass this error code to **mciGetErrorString** to get a text description of it.

Using MCI Command Messages

This section contains examples demonstrating how to perform the following tasks:

- Close all MCI devices used by an application.
- Open a simple device by using the device name.
- Open a simple device by using the device-type constant.
- Open a compound device by using a filename.
- Verify the output device.
- Select the MIDI mapper as the output device.
- Handle MCI errors.
- Play a waveform-audio file.
- Play a MIDI file.
- Play a compact disc (CD) track.
- Play a movie.
- Use the MCI_NOTIFY flag.
- Retrieve information about a movie.
- Retrieve CD track-specific information.
- Record with a waveform-audio device.

Closing All MCI Devices Used by an Application

The following example closes all of the MCI devices that are opened by an application:

```
UINT wDeviceID;
DWORD dwReturn;

// Closes all MCI devices opened by this application.
// Waits until devices are closed before returning.
if (dwReturn = mciSendCommand(MCI_ALL_DEVICE_ID, MCI_CLOSE, MCI_WAIT,
    NULL))
    .
    . // Error, unable to close all devices.
    .
```

Opening a Simple Device by Using the Device Name

The following example opens a CD audio device by specifying the device name:

```
UINT wDeviceID;
DWORD dwReturn;
MCI_OPEN_PARMS mciOpenParms;

// Opens a CD audio device by specifying the device name.
mciOpenParms.lpstrDeviceType = "cdaudio";
if (dwReturn = mciSendCommand(NULL, MCI_OPEN, MCI_OPEN_TYPE,
    (DWORD)(LPVOID) &mciOpenParms))
    .
    . // Error, unable to open device.
    .
// The device opened successfully; get the device ID.
wDeviceID = mciOpenParms.wDeviceID;
```

Opening a Simple Device by Using the Device-Type Constant

The following example opens a CD audio device by specifying a device-type constant:

```
UINT wDeviceID;
DWORD dwReturn;
MCI_OPEN_PARMS mciOpenParms;

// Opens a CD audio device by specifying a device-type constant.
mciOpenParms.lpstrDeviceType = (LPCSTR) MCI_DEVTYPE_CD_AUDIO;
if (dwReturn = mciSendCommand(NULL, MCI_OPEN,
    MCI_OPEN_TYPE | MCI_OPEN_TYPE_ID, (DWORD)(LPVOID) &mciOpenParms))
    .
    . // Error, unable to open device.
    .
// The device opened successfully; get the device ID.
wDeviceID = mciOpenParms.wDeviceID;
```

Opening a Compound Device by Using the Filename

The following example opens the waveform-audio device by specifying a waveform-audio file named "TIMPANI.WAV":

```
UINT wDeviceID;
DWORD dwReturn;
MCI_OPEN_PARMS mciOpenParms;

// Opens a waveform-audio device by specifying the device and filename.
mciOpenParms.lpstrDeviceType = "waveaudio";
mciOpenParms.lpstrElementName = "timpani.wav";
if (dwReturn = mciSendCommand(NULL, MCI_OPEN,
    MCI_OPEN_TYPE | MCI_OPEN_ELEMENT, (DWORD)(LPVOID) &mciOpenParms))
    .
    . // Error, unable to open device.
    .
// The device opened successfully; get the device ID.
wDeviceID = mciOpenParms.wDeviceID;
```

Verifying the Output Device

After opening the sequencer, you should check whether the MIDI mapper was available and selected as the output device. The following example uses the **MCI_STATUS** command to verify that the MIDI mapper is the output device for the MCI sequencer:

```
UINT wDeviceID;      // valid MCI sequencer ID
DWORD dwReturn;
MCI_STATUS_PARMS mciStatusParms;
.
. // Make sure the opened device is the MIDI mapper.
.
mciStatusParms.dwItem = MCI_SEQ_STATUS_PORT;
if (dwReturn = mciSendCommand(wDeviceID, MCI_STATUS, MCI_STATUS_ITEM,
    (DWORD)(LPVOID) &mciStatusParms))
{
    .
    . // Error sending MCI_STATUS command.
    .
    return;
}
if (LOWORD(mciStatusParms.dwReturn) == MIDI_MAPPER)
    .
    . // The MIDI mapper is the output device.
    .
else
    .
    . // The MIDI mapper is not the output device.
    .
```

Handling MCI Errors

You should always check the return value of the **mciSendCommand** function. If it indicates an error, you can use **mciGetErrorString** to get a textual description of the error.

The following example passes the MCI error code specified by *dwError* to **mciGetErrorString**, and then displays the resulting textual error description using the **MessageBox** function.

```
// Uses mciGetErrorString to get a textual description of an MCI error.
// Displays the error description using MessageBox.
void showError(DWORD dwError)
{
    char szErrorBuf[MAXERRORLENGTH];
    MessageBeep(MB_ICONEXCLAMATION);
    if(mciGetErrorString(dwError, (LPSTR) szErrorBuf, MAXERRORLENGTH))
        MessageBox(hMainWnd, szErrorBuf, "MCI Error",
        MB_ICONEXCLAMATION);
    else
        MessageBox(hMainWnd, "Unknown Error", "MCI Error",
            MB_ICONEXCLAMATION);
}
```

Note To interpret an **mciSendCommand** error return value yourself, mask the high-order word (the low-order word contains the error code). If you pass the error return value to **mciGetErrorString**, however, you must pass the entire doubleword value.

Playing a Waveform-Audio File

The following example opens a waveform-audio device and plays the waveform-audio file specified by the *lpszWAVEFileName* parameter:

```
// Plays a given waveform-audio file using MCI_OPEN and MCI_PLAY.
// Returns when playback begins. Returns 0L on success, otherwise
// returns an MCI error code.
DWORD playWAVEFile(HWND hWndNotify, LPSTR lpszWAVEFileName)
{
    UINT wDeviceID;
    DWORD dwReturn;
    MCI_OPEN_PARMS mciOpenParms;
    MCI_PLAY_PARMS mciPlayParms;

    // Open the device by specifying the device and filename.
    // MCI will choose a device capable of playing the given file.
    mciOpenParms.lpstrDeviceType = "waveaudio";
    mciOpenParms.lpstrElementName = lpszWAVEFileName;
    if (dwReturn = mciSendCommand(0, MCI_OPEN,
       MCI_OPEN_TYPE | MCI_OPEN_ELEMENT, (DWORD)(LPVOID) &mciOpenParms))
    {
        // Failed to open device. Don't close it; just return error.
        return (dwReturn);
    }

    // The device opened successfully; get the device ID.
    wDeviceID = mciOpenParms.wDeviceID;

    // Begin playback. The window procedure function for the parent
    // window will be notified with an MM_MCINOTIFY message when
    // playback is complete. At this time, the window procedure closes
    // the device.
    mciPlayParms.dwCallback = (DWORD) hWndNotify;
    if (dwReturn = mciSendCommand(wDeviceID, MCI_PLAY, MCI_NOTIFY,
        (DWORD)(LPVOID) &mciPlayParms))
    {
        mciSendCommand(wDeviceID, MCI_CLOSE, 0, NULL);
        return (dwReturn);
    }

    return (0L);
}
```

Playing a MIDI File

The following example opens a MIDI sequencer device, verifies that the MIDI mapper was selected as the output port, plays the MIDI file specified by the *lpszMIDIFileName* parameter, and closes the device after playback is complete:

```
// Plays a specified MIDI file by using MCI_OPEN and MCI_PLAY. Returns
// as soon as playback begins. The window procedure function for the
// given window will be notified when playback is complete. Returns 0L
// on success; otherwise, it returns an MCI error code.
DWORD playMIDIFile(HWND hWndNotify, LPSTR lpszMIDIFileName)
{
    UINT wDeviceID;
    DWORD dwReturn;
    MCI_OPEN_PARMS mciOpenParms;
    MCI_PLAY_PARMS mciPlayParms;
    MCI_STATUS_PARMS mciStatusParms;
    MCI_SEQ_SET_PARMS mciSeqSetParms;

    // Open the device by specifying the device and filename.
    // MCI will attempt to choose the MIDI mapper as the output port.
    mciOpenParms.lpstrDeviceType = "sequencer";
    mciOpenParms.lpstrElementName = lpszMIDIFileName;
    if (dwReturn = mciSendCommand(NULL, MCI_OPEN,
        MCI_OPEN_TYPE | MCI_OPEN_ELEMENT,
        (DWORD)(LPVOID) &mciOpenParms))
    {
        // Failed to open device. Don't close it; just return error.
        return (dwReturn);
    }

    // The device opened successfully; get the device ID.
    wDeviceID = mciOpenParms.wDeviceID;

    // Check if the output port is the MIDI mapper.
    mciStatusParms.dwItem = MCI_SEQ_STATUS_PORT;
    if (dwReturn = mciSendCommand(wDeviceID, MCI_STATUS,
        MCI_STATUS_ITEM, (DWORD)(LPVOID) &mciStatusParms))
    {
        mciSendCommand(wDeviceID, MCI_CLOSE, 0, NULL);
        return (dwReturn);
    }

    // The output port is not the MIDI mapper.
    // Ask if the user wants to continue.
    if (LOWORD(mciStatusParms.dwReturn) != MIDI_MAPPER)
    {
        if (MessageBox(hMainWnd,
            "The MIDI mapper is not available. Continue?",
            "", MB_YESNO) == IDNO)
        {
            // User does not want to continue. Not an error;
            // just close the device and return.
            mciSendCommand(wDeviceID, MCI_CLOSE, 0, NULL);
            return (0L);
        }
    }

    // Begin playback. The window procedure function for the parent
    // window will be notified with an MM_MCINOTIFY message when
    // playback is complete. At this time, the window procedure closes
```

```
    // the device.
    mciPlayParms.dwCallback = (DWORD) hWndNotify;
    if (dwReturn = mciSendCommand(wDeviceID, MCI_PLAY, MCI_NOTIFY,
        (DWORD)(LPVOID) &mciPlayParms))
    {
        mciSendCommand(wDeviceID, MCI_CLOSE, 0, NULL);
        return (dwReturn);
    }

    return (0L);
}
```

Playing a Compact Disc Track

The following example opens a CD audio device, plays the track specified by the *bTrack* parameter, and closes the device after playback is complete:

```
// Plays a given CD audio track using MCI_OPEN, MCI_PLAY. Returns as
// soon as playback begins. The window procedure function for the given
// window will be notified when playback is complete. Returns 0L on
// success; otherwise, returns an MCI error code.
DWORD playCDTrack(HWND hWndNotify, BYTE bTrack)
{
    UINT wDeviceID;
    DWORD dwReturn;
    MCI_OPEN_PARMS mciOpenParms;
    MCI_SET_PARMS mciSetParms;
    MCI_PLAY_PARMS mciPlayParms;

    // Open the CD audio device by specifying the device name.
    mciOpenParms.lpstrDeviceType = "cdaudio";
    if (dwReturn = mciSendCommand(NULL, MCI_OPEN,
        MCI_OPEN_TYPE, (DWORD)(LPVOID) &mciOpenParms))
    {
        // Failed to open device. Don't close it; just return error.
        return (dwReturn);
    }

    // The device opened successfully; get the device ID.
    wDeviceID = mciOpenParms.wDeviceID;

    // Set the time format to track/minute/second/frame (TMSF).
    mciSetParms.dwTimeFormat = MCI_FORMAT_TMSF;
    if (dwReturn = mciSendCommand(wDeviceID, MCI_SET,
        MCI_SET_TIME_FORMAT, (DWORD)(LPVOID) &mciSetParms))
    {
        mciSendCommand(wDeviceID, MCI_CLOSE, 0, NULL);
        return (dwReturn);
    }
```

```
    // Begin playback from the given track and play until the beginning
    // of the next track. The window procedure function for the parent
    // window will be notified with an MM_MCINOTIFY message when
    // playback is complete. Unless the play command fails, the window
    // procedure closes the device.
    mciPlayParms.dwFrom = 0L;
    mciPlayParms.dwTo = 0L;
    mciPlayParms.dwFrom = MCI_MAKE_TMSF(bTrack, 0, 0, 0);
    mciPlayParms.dwTo = MCI_MAKE_TMSF(bTrack + 1, 0, 0, 0);
    mciPlayParms.dwCallback = (DWORD) hWndNotify;
    if (dwReturn = mciSendCommand(wDeviceID, MCI_PLAY,
        MCI_FROM | MCI_TO | MCI_NOTIFY, (DWORD)(LPVOID) &mciPlayParms))
    {
        mciSendCommand(wDeviceID, MCI_CLOSE, 0, NULL);
        return (dwReturn);
    }

    return (0L);
}
```

To specify a position relative to a track on a CD, you must use the track/minute/second/frame (TMSF) time format.

Playing a Movie

The following examples show how to set up and play an audio-video interleaved (AVI) file.

Opening the Playback Window

The following example shows how to use the **MCI_OPEN** command to set a parent window and create a child of that window:

```
MCI_DGV_OPEN_PARMS    mciOpen;

mciOpen.lpstrElementName = lpstrFile;  // set the filename
mciOpen.dwStyle = WS_CHILD;            // set the style
mciOpen.hWndParent = hWnd;             // give a window handle

if (mciSendCommand(0, MCI_OPEN,
   (DWORD)(MCI_OPEN_ELEMENT|MCI_DGV_OPEN_PARENT|MCI_DGV_OPEN),
   (DWORD)(LPSTR)&mciOpen) == 0){

    // Open operation is successful. Continue.
}
```

Setting Up the Playback Window

The following example finds the dimensions needed to play an AVI file, creates a window corresponding to that size, and plays the file in the window by using the MCIAVI driver:

```
HWND hwnd;
MCI_DGV_RECT_PARMS mciRect;

// Get the movie dimensions with MCI_WHERE.

mciSendCommand(wDeviceID, MCI_WHERE, MCI_DGV_WHERE_SOURCE,
    (DWORD)(LPSTR)&mciRect);

// Create the playback window. Make it bigger for the border.
// Note that the right and bottom members of RECT structures in MCI
// are unusual; rc.right is set to the rectangle's width, and rc.bottom
// is set to the rectangle's height.
hwndMovie = CreateWindow("mywindow", "Playback",
    WS_CHILD|WS_BORDER, 0,0,
    mciRect.rc.right+(2*GetSystemMetric(SM_CXBORDER)),
    mciRect.rc.bottom+(2*GetSystemMetric(SM_CYBORDER)),
    hwndParent, hInstApp, NULL);

if (hwndMovie){
    // Window created OK; make it the playback window.

    MCI_DGV_WINDOW_PARMS mciWindow;

    mciWindow.hWnd = hwndMovie;
    mciSendCommand(wDeviceID, MCI_WINDOW, MCI_DGV_WINDOW_HWND,
        (DWORD)(LPSTR)&mciWindow);

}
```

Playing the AVI File

Before using the **mciSendCommand** function to send the **MCI_PLAY** command, your application allocates the memory for the structure, initializes the members it will use, and sets the flags corresponding to the members used in the structure. (If your application does not set a flag for a structure member, MCI drivers ignore the member.) For example, the following example plays a movie from the starting location specified by *dwFrom* to the ending location specified by *dwTo.* (If either location is zero, the example assumes the location is not used.)

```
DWORD PlayMovie(WORD wDevID, DWORD dwFrom, DWORD dwTo)
{
    MCI_DGV_PLAY_PARMS mciPlay;    // play parameters
    DWORD dwFlags = 0;

    // Check dwFrom. If it is != 0 then set parameters and flags.
    if (dwFrom){
        mciPlay.dwFrom = dwFrom; // set parameter
        dwFlags |= MCI_FROM;     // set flag to validate member
    }

    // Check dwTo. If it is != 0 then set parameters and flags.
    if (dwTo){
        mciPlay.dwTo = dwTo;    // set parameter
        dwFlags |= MCI_TO;      // set flag to validate member
    }

    // Send the MCI_PLAY command and return the result.
    return mciSendCommand(wDevID, MCI_PLAY, dwFlags,
       (DWORD)(LPVOID)&mciPlay);
}
```

Using the MCI_NOTIFY Flag

The following example shows how the MCI_NOTIFY flag is used with the **MCI_PLAY** command. The handle to the window procedure that will process the MM_MCINOTIFY message is specified in *hwnd*.

```
MCI_DGV_PLAY_PARMS mciPlay;
DWORD dwFlags;

mciPlay.dwCallback = MAKELONG(hwnd, 0);
dwFlags = MCI_NOTIFY;

mciSendCommand(wMCIDeviceID, MCI_PLAY, dwFlags, (DWORD)(LPSTR)&mciPlay);
```

Retrieving Information About a Movie

The following example sets the time format to frames and obtains the current position if the device is playing:

```
MCI_DGV_SET_PARMS mciSet;
MCI_DGV_STATUS_PARMS mciStatus;

// Put in frame mode.
mciSet.dwTimeFormat = MCI_FORMAT_FRAMES;
mciSendCommand(wDeviceID, MCI_SET,
    MCI_SET_TIME_FORMAT,
    (DWORD)(LPSTR)&mciSet);

mciStatus.dwItem = MCI_STATUS_MODE;
mciSendCommand(wDeviceID, MCI_STATUS,
    MCI_STATUS_ITEM,
    (DWORD)(LPSTR)&mciStatus);

// If device is playing, get the position.
if (mciStatus.dwReturn == MCI_MODE_PLAY){
     mciStatus.dwItem = MCI_STATUS_POSITION;
     mciSendCommand(wDeviceID, MCI_STATUS, MCI_STATUS_ITEM,
           (DWORD)(LPSTR)&mciStatus);

     // Update the position from mciStatus.dwReturn.
}
```

Retrieving Compact Disc Track-Specific Information

For CD audio devices, you can get the starting location and length of a track by specifying the MCI_TRACK flag and setting the **dwTrack** member of **MCI_STATUS_PARMS** to the desired track number. To get the starting location of a track, set the **dwItem** member to MCI_STATUS_POSITION. To get the length of a track, set **dwItem** to MCI_STATUS_LENGTH. For example, the following example retrieves the total number of tracks on the CD and the starting location of each track. Then, it uses the **MessageBox** function to report the starting locations of the tracks.

```
// Uses the MCI_STATUS command to get and display the
// starting times for the tracks on a CD.
// Returns 0L if successful; otherwise, it returns an
// MCI error code.
DWORD getCDTrackStartTimes(VOID)
{
    UINT wDeviceID;
    int i, iNumTracks;
    DWORD dwReturn;
    DWORD dwPosition;
    DWORD *pMem;
    char szTempString[64];
    char szTimeString[512] = "\0";  // room for 20 tracks
    MCI_OPEN_PARMS mciOpenParms;
    MCI_SET_PARMS mciSetParms;
    MCI_STATUS_PARMS mciStatusParms;

    // Open the device by specifying the device name.

    mciOpenParms.lpstrDeviceType = "cdaudio";
    if (dwReturn = mciSendCommand(NULL, MCI_OPEN,
        MCI_OPEN_TYPE, (DWORD)(LPVOID) &mciOpenParms))
    {
        // Failed to open device;
        // don't have to close it, just return error.
        return (dwReturn);
    }

    // The device opened successfully; get the device ID.
    wDeviceID = mciOpenParms.wDeviceID;

// Set the time format to minute/second/frame (MSF) format.
    mciSetParms.dwTimeFormat = MCI_FORMAT_MSF;
    if (dwReturn = mciSendCommand(wDeviceID, MCI_SET,
        MCI_SET_TIME_FORMAT,
        (DWORD)(LPVOID) &mciSetParms)) {
        mciSendCommand(wDeviceID, MCI_CLOSE, 0, NULL);
        return (dwReturn);
    }

    // Get the number of tracks;
```

```
    // limit to number that can be displayed (20).
    mciStatusParms.dwItem = MCI_STATUS_NUMBER_OF_TRACKS;
    if (dwReturn = mciSendCommand(wDeviceID, MCI_STATUS,
        MCI_STATUS_ITEM, (DWORD)(LPVOID) &mciStatusParms)) {
        mciSendCommand(wDeviceID, MCI_CLOSE, 0, NULL);
        return (dwReturn);
    }
    iNumTracks = mciStatusParms.dwReturn;
    iNumTracks = min(iNumTracks, 20);

    // Allocate memory to hold starting positions.
    pMem = (DWORD *)LocalAlloc(LPTR,
        iNumTracks * sizeof(DWORD));
    if (pMem == NULL) {
        mciSendCommand(wDeviceID, MCI_CLOSE, 0, NULL);
        return (-1);
    }

// For each track, get and save the starting location and
// build a string containing starting locations.
    for(i=1; i<=iNumTracks; i++) {
        mciStatusParms.dwItem = MCI_STATUS_POSITION;
        mciStatusParms.dwTrack = i;
        if (dwReturn = mciSendCommand(wDeviceID,
            MCI_STATUS, MCI_STATUS_ITEM | MCI_TRACK,
            (DWORD)(LPVOID) &mciStatusParms)) {
            mciSendCommand(wDeviceID, MCI_CLOSE, 0, NULL);
            return (dwReturn);
        }

        pMem[i-1] = mciStatusParms.dwReturn;

        wsprintf(szTempString,
            "Track %2d - %02d:%02d:%02d\n", i,
            MCI_MSF_MINUTE(pMem[i-1]),
            MCI_MSF_SECOND(pMem[i-1]),
            MCI_MSF_FRAME(pMem[i-1]));

        lstrcat(szTimeString, szTempString);
    }

    // Use MessageBox to display starting times.
    MessageBox(hMainWnd, szTimeString,
        "Track Starting Position", MB_ICONINFORMATION);

    // Free memory and close the device.
    LocalFree((HANDLE) pMem);
    if (dwReturn = mciSendCommand(wDeviceID,
        MCI_CLOSE, 0, NULL)) {
        return (dwReturn);
    }

    return (0L);
}
```

Recording with a Waveform-Audio Device

The following example opens a waveform-audio device with a new file, records for the specified time, plays the recording, and prompts the user to save the recording if desired:

```
// Uses the MCI_OPEN, MCI_RECORD, and MCI_SAVE commands to record and
// save a waveform-audio file. Returns 0L if successful; otherwise,
// it returns an MCI error code.
DWORD recordWAVEFile(DWORD dwMilliSeconds)
{
    UINT wDeviceID;
    DWORD dwReturn;
    MCI_OPEN_PARMS mciOpenParms;
    MCI_RECORD_PARMS mciRecordParms;
    MCI_SAVE_PARMS mciSaveParms;
    MCI_PLAY_PARMS mciPlayParms;

    // Open a waveform-audio device with a new file for recording.
    mciOpenParms.lpstrDeviceType = "waveaudio";
    mciOpenParms.lpstrElementName = "";
    if (dwReturn = mciSendCommand(0, MCI_OPEN,
        MCI_OPEN_ELEMENT | MCI_OPEN_TYPE,
        (DWORD)(LPVOID) &mciOpenParms))
    {
        // Failed to open device; don't close it, just return error.
        return (dwReturn);
    }

    // The device opened successfully; get the device ID.
    wDeviceID = mciOpenParms.wDeviceID;

    // Begin recording and record for the specified number of
    // milliseconds. Wait for recording to complete before continuing.
    // Assume the default time format for the waveform-audio device
    // (milliseconds).
    mciRecordParms.dwTo = dwMilliSeconds;
    if (dwReturn = mciSendCommand(wDeviceID, MCI_RECORD,
        MCI_TO | MCI_WAIT, (DWORD)(LPVOID) &mciRecordParms))
    {
        mciSendCommand(wDeviceID, MCI_CLOSE, 0, NULL);
        return (dwReturn);
    }

    // Play the recording and query user to save the file.
    mciPlayParms.dwFrom = 0L;
    if (dwReturn = mciSendCommand(wDeviceID, MCI_PLAY,
        MCI_FROM | MCI_WAIT, (DWORD)(LPVOID) &mciPlayParms))
    {
        mciSendCommand(wDeviceID, MCI_CLOSE, 0, NULL);
        return (dwReturn);
    }
    if (MessageBox(hMainWnd, "Do you want to save this recording?",
        "", MB_YESNO) == IDNO)
    {
        mciSendCommand(wDeviceID, MCI_CLOSE, 0, NULL);
        return (0L);
    }

    // Save the recording to a file named TEMPFILE.WAV. Wait for
    // the operation to complete before continuing.
```

```
    mciSaveParms.lpfilename = "tempfile.wav";
    if (dwReturn = mciSendCommand(wDeviceID, MCI_SAVE,
        MCI_SAVE_FILE | MCI_WAIT, (DWORD)(LPVOID) &mciSaveParms))
    {
        mciSendCommand(wDeviceID, MCI_CLOSE, 0, NULL);
        return (dwReturn);
    }

    return (0L);
}
```

MCI Command Reference

This section describes MCI command messages and structures. These elements are grouped as follows.

Configuring a Device

MCI_BREAK
MCI_BREAK_PARMS
MCI_CONFIGURE
MCI_DGV_SET_PARMS
MCI_DGV_SETAUDIO_PARMS
MCI_DGV_SETVIDEO_PARMS
MCI_ESCAPE
MCI_INDEX
MCI_SEQ_SET_PARMS
MCI_SET
MCI_SET_PARMS
MCI_SETAUDIO
MCI_SETTIMECODE
MCI_SETTUNER
MCI_SETVIDEO
MCI_SPIN
MCI_VCR_SET_PARMS
MCI_VCR_SETAUDIO_PARMS
MCI_VCR_SETTUNER_PARMS
MCI_VCR_SETVIDEO_PARMS
MCI_VD_ESCAPE_PARMS
MCI_WAVE_SET_PARMS

Controlling Playback

MCI_ANIM_PLAY_PARMS
MCI_DGV_FREEZE_PARMS
MCI_DGV_LOAD_PARMS
MCI_DGV_PAUSE_PARMS
MCI_DGV_PLAY_PARMS
MCI_DGV_RESUME_PARMS
MCI_DGV_STOP_PARMS
MCI_FREEZE
MCI_LOAD
MCI_LOAD_PARMS
MCI_OVLY_LOAD_PARMS
MCI_PAUSE
MCI_PLAY
MCI_PLAY_PARMS
MCI_RESUME
MCI_STOP
MCI_UNFREEZE
MCI_VCR_PLAY_PARMS
MCI_VD_PLAY_PARMS

Controlling the Position

MCI_ANIM_STEP_PARMS
MCI_CUE
MCI_DGV_CUE_PARMS
MCI_DGV_SIGNAL_PARMS

MCI_DGV_STEP_PARMS
MCI_MARK
MCI_SEEK
MCI_SEEK_PARMS
MCI_SIGNAL
MCI_STEP
MCI_VCR_CUE_PARMS
MCI_VCR_SEEK_PARMS
MCI_VCR_STEP_PARMS
MCI_VD_STEP_PARMS

Editing

MCI_COPY
MCI_CUT
MCI_DELETE
MCI_DGV_COPY_PARMS
MCI_DGV_CUT_PARMS
MCI_DGV_DELETE_PARMS
MCI_DGV_PASTE_PARMS
MCI_PASTE
MCI_UNDO
MCI_WAVE_DELETE_PARMS

Miscellaneous

MCI_GENERIC_PARMS

Opening and Closing

MCI_ANIM_OPEN_PARMS
MCI_CLOSE
MCI_DGV_OPEN_PARMS
MCI_OPEN
MCI_OPEN_PARMS
MCI_OVLY_OPEN_PARMS
MCI_WAVE_OPEN_PARMS

Realizing a Palette

MCI_REALIZE

Repainting a Frame

MCI_ANIM_UPDATE_PARMS
MCI_DGV_UPDATE_PARMS
MCI_UPDATE

Retrieving Information

MCI_DGV_INFO_PARMS
MCI_DGV_LIST_PARMS
MCI_DGV_STATUS_PARMS
MCI_GETDEVCAPS
MCI_GETDEVCAPS_PARMS
MCI_INFO
MCI_INFO_PARMS
MCI_LIST
MCI_STATUS
MCI_STATUS_PARMS
MCI_SYSINFO
MCI_SYSINFO_PARMS
MCI_VCR_LIST_PARMS

MCI_VCR_STATUS_PARMS

Saving

MCI_DGV_RECORD_PARMS
MCI_DGV_SAVE_PARMS
MCI_OVLY_SAVE_PARMS
MCI_RECORD
MCI_RECORD_PARMS
MCI_SAVE
MCI_SAVE_PARMS
MCI_VCR_RECORD_PARMS

Video Control

MCI_CAPTURE
MCI_DGV_MONITOR_PARMS
MCI_DGV_QUALITY_PARMS
MCI_DGV_RESERVE_PARMS
MCI_DGV_RESTORE_PARMS
MCI_MONITOR
MCI_QUALITY
MCI_RESERVE
MCI_RESTORE

Window or Display Rectangles

MCI_ANIM_RECT_PARMS
MCI_ANIM_WINDOW_PARMS
MCI_DGV_PUT_PARMS
MCI_DGV_RECT_PARMS
MCI_DGV_WINDOW_PARMS
MCI_OVLY_RECT_PARMS
MCI_OVLY_WINDOW_PARMS
MCI_PUT
MCI_WHERE
MCI_WINDOW

MCI_BREAK

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_BREAK,
    DWORD dwFlags, (DWORD) (LPMCI_BREAK_PARMS) lpBreak);
```

Sets a break key for an MCI device. MCI supports this command directly rather than passing it to the device. Any MCI application can use this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and video-cassette recorder (VCR) devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpBreak
Address of an **MCI_BREAK_PARMS** structure.

You might have to press the break key multiple times to interrupt a wait operation. Pressing the break key after a device wait is canceled can send the break to an application. If an application has an action defined for the virtual-key code, then it can inadvertently 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.

Additional Flags

The following additional flags apply to all devices:

MCI_BREAK_HWND
The **hwndBreak** member of the structure identified by *lpBreak* contains a window handle that 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_KEY
The **nVirtKey** member of the 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_OFF
Disables any existing break key for the indicated device.

MCI_CAPTURE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_CAPTURE,
    DWORD dwFlags, (DWORD) (LPMCI_DGV_CAPTURE_PARMS) lpCapture);
```

Captures the contents of the frame buffer and stores it in a specified file. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpCapture
Address of an **MCI_DGV_CAPTURE_PARMS** structure.

Additional Flags

The following additional flags apply to digital-video devices:

MCI_DGV_CAPTURE_AS
The **lpstrFileName** member of the structure identified by *lpCapture* contains an address of a buffer specifying the destination path and filename. (This flag is required.)

MCI_DGV_CAPTURE_AT
The **rc** member of the structure identified by *lpCapture* contains a valid rectangle. The rectangle specifies the rectangular region within the frame buffer that is cropped and saved to disk. If omitted, the cropped region defaults to the rectangle specified or defaulted on a previous **MCI_PUT** command that specifies the source area for this instance of the device driver.

MCI_CLOSE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_CLOSE,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpClose);
```

Releases access to a device or file. All devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY or MCI_WAIT. For information about these flags, see Chapter 3, "MCI Overview."

lpClose
Address of an **MCI_GENERIC_PARMS** structure. (You can also use an **MCI_CLOSE_PARMS** structure, which is identical to **MCI_GENERIC_PARMS**. Devices with extended command sets might replace this structure with a device-specific structure.)

Exiting an application without closing any MCI devices it has opened can leave the device inaccessible. Your application should explicitly close each device or file when it is finished with it. MCI unloads the device when all instances of the device or all associated files are closed.

MCI_CONFIGURE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_CONFIGURE,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpConfigure);
```

Displays a dialog box for setting the operating options. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpConfigure
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

MCI_COPY

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_COPY,
    DWORD dwFlags, (DWORD) (LPMCI_DGV_COPY_PARMS) lpCopy);
```

Copies data to the clipboard. Digital-video devices recognize this command.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpCopy
Address of an **MCI_DGV_COPY_PARMS** structure.

Additional Flags

The following additional flags apply to digital-video devices:

MCI_DGV_COPY_AT
A rectangle is included in the **rc** member of the structure identified by *lpCopy*. The rectangle specifies the portion of each frame to copy. If the flag is omitted, **MCI_COPY** copies the entire frame.

MCI_DGV_COPY_AUDIO_STREAM
An audio-stream number is included in the **dwAudioStream** member of the structure identified by *lpCopy*. If you use this flag and also want to copy video, you must also use the MCI_DGV_COPY_VIDEO_STREAM flag. (If neither flag is specified, data from all audio and video streams is copied.)

MCI_DGV_COPY_VIDEO_STREAM
A video-stream number is included in the **dwVideoStream** member of the structure identified by *lpCopy*. If you use this flag and also want to copy audio, you must also use the MCI_DGV_COPY_AUDIO_STREAM flag. (If neither flag is specified, data from all audio and video streams is copied.)

MCI_FROM
A starting location is included in the **dwFrom** member of the structure identified by *lpCopy*. The units assigned to the position values are specified with the MCI_SET_TIME_FORMAT flag of the **MCI_SET** command.

MCI_TO
An ending location is included in the **dwTo** member of the structure identified by *lpCopy*. The units assigned to the position values are specified with the MCI_SET_TIME_FORMAT flag of the **MCI_SET** command.

MCI_CUE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_CUE,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpCue);
```

Cues a device so that playback or recording begins with minimum delay.Digital-video, VCR, and waveform-audio devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and VCR devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpCue
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Digital-Video Flags

The following additional flags are used with the **digitalvideo** device type:

MCI_DGV_CUE_INPUT
A digital-video instance should prepare for recording. If the application has not reserved disk space, the device reserves the disk space using its default parameters. The application can omit this flag if the current presentation source is already the external input. (This flag has no effect on selecting the presentation source.)

MCI_DGV_CUE_NOSHOW
A digital-video instance should prepare for playing the frame specified with the command without displaying it. When this flag is specified, the display continues to show the image in the frame buffer even though its corresponding frame is not the current position. For example, if the frame buffer contains the image from frame 7, the device continues to show frame 7 when this flag is used to cue the device to any other position. A subsequent cue command without this flag and without the MCI_TO flag displays the current frame.

MCI_DGV_CUE_OUTPUT
A digital-video instance should prepare for playing. If the workspace is paused, no positioning occurs. If the workspace is stopped, the position might change to a previous key-frame image. The application can omit this flag if the current presentation source is already the workspace.

MCI_TO
A workspace position is included in the **dwTo** member of the structure identified by *lpCue*. The units assigned to position values are specified using the MCI_SET_TIME_FORMAT flag of the **MCI_SET** command. This is equivalent to seeking to a position, except the device is paused after the command.

For **digitalvideo** devices, the *lpCue* parameter points to an **MCI_DGV_CUE_PARMS** structure.

VCR Flags

The following additional flags are used with the **vcr** device type:

MCI_FROM
The **dwFrom** member of the structure pointed to by *lpCue* contains the starting location specified in the current time format.

MCI_TO
The **dwTo** member of the structure pointed to by *lpCue* contains the ending (pausing) location specified in the current time format.

MCI_VCR_CUE_INPUT
Prepare for recording.

MCI_VCR_CUE_OUTPUT
Prepare for playing. If neither MCI_VCR_CUE_INPUT nor MCI_VCR_CUE_OUTPUT is specified, MCI_VCR_CUE_OUTPUT is assumed.

MCI_VCR_CUE_PREROLL
Cue the device to the current position, or the **dwFrom** position, minus the preroll duration. This will allow the device to prepare itself before entering record or playback mode.

MCI_VCR_CUE_REVERSE
The direction of the next play or record command is reverse.

When cueing for playback by using the **MCI_CUE** command with the MCI_VCR_CUE_OUTPUT flag, you can cancel **MCI_CUE** by issuing the **MCI_PLAY** command with MCI_FROM, MCI_TO, or MCI_VCR_PLAY_REVERSE.

When cueing for recording by using **MCI_CUE** with the MCI_VCR_CUE_INPUT flag, you can cancel **MCI_CUE** by issuing the **MCI_RECORD** command with MCI_FROM, MCI_TO, or MCI_VCR_RECORD_INITIALIZE.

For **vcr** devices, the *lpCue* parameter points to an **MCI_VCR_CUE_PARMS** structure.

Waveform-Audio Flags

The following additional flags are used with the **waveaudio** device type:

MCI_WAVE_INPUT
A waveform-audio input device should be cued.

MCI_WAVE_OUTPUT
A waveform-audio output device should be cued. This is the default flag if a flag is not specified.

MCI_CUT

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_CUT,
    DWORD dwFlags, (DWORD) (LPMCI_DGV_CUT_PARMS) lpCut);
```

Removes data from the file and copies it to the clipboard. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpCut
Address of an **MCI_DGV_CUT_PARMS** structure.

Additional Flags

The following additional flags apply to digital-video devices:

MCI_DGV_CUT_AT
A rectangle is included in the **rc** member of the structure identified by *lpCut*. The rectangle specifies the portion of each frame to cut. If the flag is omitted, **MCI_CUT** cuts the entire frame.

MCI_DGV_CUT_AUDIO_STREAM
An audio-stream number is included in the **dwAudioStream** member of the structure identified by *lpCut*. If you use this flag and also want to cut video, you must also use the MCI_DGV_CUT_VIDEO_STREAM flag. (If neither flag is specified, data from all audio and video streams is cut.)

MCI_DGV_CUT_VIDEO_STREAM
A video-stream number is included in the **dwVideoStream** member of the structure identified by *lpCut*. If you use this flag and also want to cut audio, you must also use the MCI_DGV_CUT_AUDIO_STREAM flag. (If neither flag is specified, data from all audio and video streams is cut.)

MCI_FROM
A starting location is included in the **dwFrom** member of the structure identified by *lpCut*. The units assigned to the position values are specified with the MCI_SET_TIME_FORMAT flag of the **MCI_SET** command.

MCI_TO
An ending location is included in the **dwTo** member of the structure identified by *lpCut*. The units assigned to the position values are specified with the MCI_SET_TIME_FORMAT flag of **MCI_SET**.

MCI_DELETE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_DELETE,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpDelete);
```

Removes data from the file. Digital-video and waveform-audio devices recognize this command.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpDelete
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Digital-Video Flags

The following flags apply to the **digitalvideo** device type:

MCI_DGV_DELETE_AT
A rectangle is included in the **rc** member of the structure identified by *lpDelete*. The rectangle specifies the portion of each frame to delete. When this flag is used, the frame is retained in the workspace and the area specified by the rectangle becomes black. If the flag is omitted, **MCI_DELETE** defaults to the entire frame and removes the frame from the workspace.

MCI_DGV_DELETE_AUDIO_STREAM
An audio-stream number is included in the **dwAudioStream** member of the structure identified by *lpDelete*. If you use this flag and also want to delete video, you must also use the MCI_DGV_DELETE_VIDEO_STREAM flag. (If neither flag is specified, data from all audio and video streams is deleted.)

MCI_DGV_DELETE_VIDEO_STREAM
A video-stream number is included in the **dwVideoStream** member of the structure identified by *lpDelete*. If you use this flag and also want to delete audio, you must also use the MCI_DGV_DELETE_AUDIO_STREAM flag. (If neither flag is specified, data from all audio and video streams is deleted.)

MCI_FROM
A starting location is included in the **dwFrom** member of the structure identified by *lpDelete*. The units assigned to the position values are specified with the MCI_SET_TIME_FORMAT flag of the **MCI_SET** command.

MCI_TO
An ending location is included in the **dwTo** member of the structure identified by *lpDelete*. The units assigned to the position values are specified with the MCI_SET_TIME_FORMAT flag of **MCI_SET**.

For digital-video devices, the *lpDelete* parameter points to an **MCI_DGV_DELETE_PARMS** structure.

Waveform-Audio Flags

The following flags apply to the **waveaudio** device type:

MCI_FROM
A starting location is included in the **dwFrom** member of the structure identified by *lpDelete*. The units assigned to the position values are specified with the MCI_SET_TIME_FORMAT flag of **MCI_SET**.

MCI_TO
An ending location is included in the **dwTo** member of the structure identified by *lpDelete*. The units assigned to the position values are specified with the MCI_SET_TIME_FORMAT flag of **MCI_SET**.

For waveform-audio devices, the *lpDelete* parameter points to an **MCI_WAVE_DELETE_PARMS**

structure.

MCI_ESCAPE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_ESCAPE,
    DWORD dwFlags, (DWORD) (LPMCI_VD_ESCAPE_PARMS) lpEscape);
```

Sends a string directly to the device. Videodisc devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY or MCI_WAIT. For information about these flags, see Chapter 3, "MCI Overview."

lpEscape
Address of an **MCI_VD_ESCAPE_PARMS** structure.

The data sent with **MCI_ESCAPE** is device-dependent and is usually passed directly to the hardware associated with the device.

Additional Flag

The following additional flag applies to videodisc devices:

MCI_VD_ESCAPE_STRING
A command string is specified in the **lpstrCommand** member of the structure identified by *lpEscape*. This flag is required.

MCI_FREEZE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_FREEZE,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpFreeze);
```

Freezes motion on the display. Digital-video, video-overlay, and VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and VCR devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpFreeze
Address of an **MCI_GENERIC_PARMS** structure. (Devices with additional parameters might replace this structure with a device-specific structure.)

Digital-Video Flags

The following additional flags are used by the **digitalvideo** device type:

MCI_DGV_FREEZE_AT
The **rc** member of the structure identified by *lpFreeze* contains a valid rectangle. The rectangle specifies a region within the frame buffer that will have the lock mask bit for each pixel turned on. The specified pixels will not be updated until their lock mask bit is turned off. If this flag is not specified, the rectangle defaults to the entire frame buffer. This flag is supported only if the **MCI_GETDEVCAPS** command returns TRUE for the MCI_DGV_GETDEVCAPS_CAN_LOCK flag.

MCI_DGV_FREEZE_OUTSIDE
The area outside the region specified for the MCI_DGV_FREEZE_AT flag is frozen.

For digital-video devices, the *lpFreeze* parameter points to an **MCI_DGV_FREEZE_PARMS** structure.

VCR Flags

The following additional flags are used by the **vcr** device type:

MCI_VCR_FREEZE_FIELD
Freeze only one member of the current frame.

MCI_VCR_FREEZE_FRAME
Freeze both fields of the current frame.

MCI_VCR_FREEZE_INPUT
Freeze the current frame on the screen (used for recording).

MCI_VCR_FREEZE_OUTPUT
Freeze the current frame from the VCR (used with frame capture).

For VCR devices, the *lpFreeze* parameter points to an **MCI_GENERIC_PARMS** structure.

Video-Overlay Flags

The following additional flag is used by the **overlay** device type:

MCI_OVLY_RECT
The **rc** member of the structure identified by *lpFreeze* contains a valid rectangle. If this flag is not specified, the device driver will freeze the entire frame.

For video-overlay devices, the *lpFreeze* parameter points to an **MCI_OVLY_RECT_PARMS** structure.

MCI_GETDEVCAPS

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_GETDEVCAPS,
    DWORD dwFlags, (DWORD) (LPMCI_GETDEVCAPS_PARMS) lpCapsParms);
```

Retrieves static information about a device. All devices recognize this command. The parameters and flags available for this command depend on the selected device. Information is returned in the **dwReturn** member of the structure identified by *lpCapsParms*.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and VCR devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpCapsParms
Address of an **MCI_GETDEVCAPS_PARMS** structure.

Additional Flags

The following additional standard and command-specific flags apply to all devices supporting **MCI_GETDEVCAPS**:

MCI_GETDEVCAPS_COMPOUND_DEVICE
The **dwReturn** member is set to TRUE if the device uses data storage that must be explicitly opened and closed; it is set to FALSE otherwise.

MCI_GETDEVCAPS_DEVICE_TYPE
The **dwReturn** member is set to one of the values listed in "Constants: Device Types" later in this chapter.

MCI_GETDEVCAPS_HAS_AUDIO
The **dwReturn** member is set to TRUE if the device has audio output; it is set to FALSE otherwise.

MCI_GETDEVCAPS_HAS_VIDEO
The **dwReturn** member is set to TRUE if the device has video output; it is set to FALSE otherwise. For example, the member is set to TRUE for devices that support the animation or videodisc command set.

MCI_GETDEVCAPS_ITEM
Specifies that the **dwItem** member of the **MCI_GETDEVCAPS_PARMS** structure contains one of the following constants:

MCI_GETDEVCAPS_CAN_EJECT
The **dwReturn** member is set to TRUE if the device can eject the media; otherwise, it is set to FALSE.

MCI_GETDEVCAPS_CAN_PLAY
The **dwReturn** member 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 the **MCI_PAUSE** and **MCI_STOP** commands as well as the **MCI_PLAY** command.

MCI_GETDEVCAPS_CAN_RECORD
The **dwReturn** member 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 the **MCI_PAUSE** and **MCI_STOP** commands as well as the **MCI_RECORD** command.

MCI_GETDEVCAPS_CAN_SAVE
The **dwReturn** member is set to TRUE if the device can save a file; otherwise, it is set to FALSE.

MCI_GETDEVCAPS_USES_FILES
The **dwReturn** member is set to TRUE if the device requires a filename; it is set to FALSE otherwise. Only compound devices use files.

Animation Flags

The following flags can be specified in the **dwItem** member of **MCI_GETDEVCAPS_PARMS** for the **animation** device type:

MCI_ANIM_GETDEVCAPS_CAN_REVERSE
The **dwReturn** member is set to TRUE if the device can play in reverse; otherwise, it is set to FALSE.

MCI_ANIM_GETDEVCAPS_CAN_STRETCH
The **dwReturn** member 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** member is set to the standard fast play rate in frames per second.

MCI_ANIM_GETDEVCAPS_MAX_WINDOWS
The **dwReturn** member is set to the maximum number of windows that the device can handle simultaneously.

MCI_ANIM_GETDEVCAPS_NORMAL_RATE
The **dwReturn** member is set to the normal play rate in frames per second.

MCI_ANIM_GETDEVCAPS_PALETTES
The **dwReturn** member 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** member is set to the standard slow play rate in frames per second.

Digital-Video Flags

The following flags can be specified in the **dwItem** member of **MCI_GETDEVCAPS_PARMS** for the **digitalvideo** device type:

MCI_DGV_GETDEVCAPS_CAN_FREEZE
The **dwReturn** member is set to TRUE if the device can freeze frames; otherwise, it is set to FALSE.

MCI_DGV_GETDEVCAPS_CAN_LOCK
The **dwReturn** member is set to TRUE if the device can lock; otherwise, it is set to FALSE.

MCI_DGV_GETDEVCAPS_CAN_REVERSE
The **dwReturn** member is set to TRUE if the device can play in reverse; otherwise, it is set to FALSE.

MCI_DGV_GETDEVCAPS_CAN_STR_IN
The **dwReturn** member is set to TRUE if the device can stretch input; otherwise, it is set to FALSE.

MCI_DGV_GETDEVCAPS_CAN_STRETCH
The **dwReturn** member is set to TRUE if the device can stretch an image; otherwise, it is set to FALSE.

MCI_DGV_GETDEVCAPS_CAN_TEST
The **dwReturn** member is set to TRUE if the device can perform tests; otherwise, it is set to FALSE.

MCI_DGV_GETDEVCAPS_HAS_STILL
The **dwReturn** member is set to TRUE if the device can display still images; otherwise, it is set to FALSE.

MCI_DGV_GETDEVCAPS_MAX_WINDOWS
The **dwReturn** member is set to the maximum number of windows that the device can handle simultaneously.

MCI_DGV_GETDEVCAPS_MAXIMUM_RATE
The **dwReturn** member is set to the maximum play rate for the device, in frames per second.

MCI_DGV_GETDEVCAPS_MINIMUM_RATE
The **dwReturn** member is set to the minimum play rate for the device, in frames per second.

MCI_DGV_GETDEVCAPS_PALETTES
The **dwReturn** member is set to TRUE if the device can return a palette handle; otherwise, it is set to FALSE.

VCR Flags

The following flags can be specified in the **dwItem** member of **MCI_GETDEVCAPS_PARMS** for the **vcr** device type:

MCI_GETDEVCAPS_CLOCK_INCREMENT_RATE
The **dwReturn** member is set to the number of increments per second.

MCI_VCR_GETDEVCAPS_CAN_DETECT_LENGTH
The **dwReturn** member is set to TRUE if the device is capable of detecting the length of the media; otherwise, it is set to FALSE.

MCI_VCR_GETDEVCAPS_CAN_FREEZE
The **dwReturn** member is set to TRUE if the device is capable of freezing the output image; otherwise, it is set to FALSE.

MCI_VCR_GETDEVCAPS_CAN_MONITOR_SOURCES
The **dwReturn** member is set to TRUE if the device is capable of monitoring sources; otherwise, it is set to FALSE.

MCI_VCR_GETDEVCAPS_CAN_PREROLL
The **dwReturn** member is set to TRUE if the device is capable of preroll; otherwise, it is set to FALSE.

MCI_VCR_GETDEVCAPS_CAN_PREVIEW
The **dwReturn** member is set to TRUE if the device is capable of previews; otherwise, it is set to FALSE.

MCI_VCR_GETDEVCAPS_CAN_REVERSE
The **dwReturn** member is set to TRUE if the device is capable of playing in reverse; otherwise, it is set to FALSE.

MCI_VCR_GETDEVCAPS_CAN_TEST
The **dwReturn** member is set to TRUE if the device is capable of testing; otherwise, it is set to FALSE.

MCI_VCR_GETDEVCAPS_HAS_CLOCK
The **dwReturn** member is set to TRUE if the device supports an external clock; otherwise, it is set to FALSE.

MCI_VCR_GETDEVCAPS_HAS_TIMECODE
The **dwReturn** member is set to TRUE if device has timecode capability or if this capability is unknown; otherwise, it is set to FALSE.

MCI_VCR_GETDEVCAPS_NUMBER_OF_MARKS
The **dwReturn** member is set to the number of marks (99).

MCI_VCR_GETDEVCAPS_SEEK_ACCURACY
The **dwReturn** member is set to the seek accuracy of the device.

Video-Overlay Flags

The following flags can be specified in the **dwItem** member of **MCI_GETDEVCAPS_PARMS** for the **overlay** device type:

MCI_OVLY_GETDEVCAPS_CAN_FREEZE
The **dwReturn** member is set to TRUE if the device can freeze the image; otherwise, it is set to FALSE.

MCI_OVLY_GETDEVCAPS_CAN_STRETCH
The **dwReturn** member 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** member is set to the maximum number of windows that the device can handle simultaneously.

Videodisc Flags

The following flags can be specified in the **dwItem** member of **MCI_GETDEVCAPS_PARMS** for the **videodisc** device type:

MCI_VD_GETDEVCAPS_CAN_REVERSE
The **dwReturn** member 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_CAV
When combined with other items, specifies that the return information applies to CAV format videodiscs. This is the default if no videodisc is inserted.

MCI_VD_GETDEVCAPS_CLV
When combined with other items, specifies that the return information applies to CLV format videodiscs.

MCI_VD_GETDEVCAPS_FAST_RATE
The **dwReturn** member is set to the standard fast play rate in frames per second.

MCI_VD_GETDEVCAPS_NORMAL_RATE
The **dwReturn** member is set to the normal play rate in frames per second.

MCI_VD_GETDEVCAPS_SLOW_RATE
The **dwReturn** member is set to the standard slow play rate in frames per second.

Waveform-Audio Flags

The following flags can be specified in the **dwItem** member of **MCI_GETDEVCAPS_PARMS** for the **waveaudio** device type:

MCI_WAVE_GETDEVCAPS_INPUT
The **dwReturn** member is set to the total number of waveform input (recording) devices.

MCI_WAVE_GETDEVCAPS_OUTPUT
The **dwReturn** member is set to the total number of waveform output (playback) devices.

MCI_INDEX

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_INDEX,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpIndex);
```

Turns the on-screen display on or off. VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpIndex
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

The information presented in the on-screen display is controlled by the MCI_VCR_SET_INDEX flag in the **MCI_SET** command.

Additional Flags

The following additional flags apply to VCR devices:

MCI_SET_OFF
Turns on-screen display off.

MCI_SET_ON
Turns on-screen display on.

MCI_INFO

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_INFO,
    DWORD dwFlags, (DWORD) (LPMCI_INFO_PARMS) lpInfo);
```

Retrieves string information from a device. All devices recognize this command. Information is returned in the **lpstrReturn** member of the structure identified by *lpInfo*. The **dwRetSize** member specifies the buffer length for the returned data.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and VCR devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpInfo
Address of an **MCI_INFO_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Additional Flag

The following additional standard and command-specific flag applies to all devices supporting **MCI_INFO**:

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.

Animation Flags

The following additional flags apply to the **animation** device type:

MCI_ANIM_INFO_TEXT
Obtains the window caption.

MCI_INFO_FILE
Obtains the filename of the current file. This flag is supported only by devices that return TRUE when you call the **MCI_GETDEVCAPS** command with the MCI_GETDEVCAPS_USES_FILES flag.

CD Audio Flags

The following additional flags apply to the **cdaudio** device type:

MCI_INFO_MEDIA_IDENTITY
Produces a unique identifier for the audio CD currently loaded in the player being queried. This flag returns a string of 16 hexadecimal digits.

MCI_INFO_MEDIA_UPC
Produces the Universal Product Code (UPC) that is encoded on an audio CD. The UPC is a string of digits. It might not be available for all CDs.

Digital-Video Flags

The following additional flags apply to the **digitalvideo** device type:

MCI_DGV_INFO_ITEM
A constant indicating the information desired is included in the **dwItem** member of the structure identified by *lpInfo*. The following constants are defined for digital-video devices:

MCI_DGV_INFO_AUDIO_ALG
Returns the name for the current audio compression algorithm.

MCI_DGV_INFO_AUDIO_QUALITY

Returns the name for the current audio quality descriptor.

MCI_DGV_INFO_STILL_ALG
Returns the name for the current still image compression algorithm.

MCI_DGV_INFO_STILL_QUALITY
Returns the name for the current still image quality descriptor.

MCI_DGV_INFO_USAGE
Returns a string describing usage restrictions that might be imposed by the owner of the visual or audible data in the workspace.

MCI_DGV_INFO_VIDEO_ALG
Returns the name for the current video compression algorithm.

MCI_DGV_INFO_VIDEO_QUALITY
Returns the name for the current video quality descriptor.

MCI_INFO_VERSION
Returns the release level of the device driver and hardware. Device driver developers must document the syntax of the returned string.

MCI_DGV_INFO_TEXT
Obtains the window caption.

MCI_INFO_FILE
Obtains the path and filename of the last file specified with the **MCI_OPEN** or **MCI_LOAD** command. If a file has not been specified, the device returns a null-terminated string. This flag is supported only by devices that return TRUE to the MCI_GETDEVCAPS_USES_FILES flag of the **MCI_GETDEVCAPS** command.

For digital-video devices, *lpInfo* points to an **MCI_DGV_INFO_PARMS** structure.

Sequencer Flags

The following additional flags apply to the **sequencer** device type:

MCI_INFO_COPYRIGHT
Obtains the MIDI file copyright notice from the copyright meta event.

MCI_INFO_FILE
Obtains the filename of the current file. This flag is supported only by devices that return TRUE when you call the **MCI_GETDEVCAPS** command with the MCI_GETDEVCAPS_USES_FILES flag.

MCI_INFO_NAME
Obtains the sequence name from the sequence/track name meta event.

VCR Flags

The following additional flag applies to the **vcr** device type:

MCI_VCR_INFO_VERSION
Sets **lpstrReturn** member of the **MCI_INFO_PARMS** structure to point to the version number. Also sets the **dwRetSize** member equal to the length of the string pointed to.

Video-Overlay Flags

The following additional flags apply to the **overlay** device type:

MCI_INFO_FILE
Obtains the filename of the current file. This flag is supported only by devices that return TRUE to the MCI_GETDEVCAPS_USES_FILES flag of the **MCI_GETDEVCAPS** command.

MCI_OVLY_INFO_TEXT
Obtains the caption of the window associated with the video-overlay device.

Waveform-Audio Flags

The following additional flags apply to the **waveaudio** device type:

MCI_INFO_FILE
Obtains the filename of the current file. This flag is supported by devices that return TRUE when you call the **MCI_GETDEVCAPS** command with the MCI_GETDEVCAPS_USES_FILES flag.

MCI_WAVE_INPUT
Obtains the product name of the current input.

MCI_WAVE_OUTPUT
Obtains the product name of the current output and its value is device specific.

MCI_LIST

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_LIST,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpList);
```

Obtains information about the number and types of inputs available to the device. Digital-video and VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpList
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Digital-Video Flags

The following additional flags apply to the **digitalvideo** device type:

MCI_DGV_LIST_ALG
The **lpstrAlgorithm** member of the structure identified by *lpList* contains an address of a buffer containing the name of an algorithm. The name is used to retrieve the types of quality descriptors associated with an algorithm.

MCI_DGV_LIST_COUNT
Returns the number of options of the specified type.

MCI_DGV_LIST_ITEM
A constant indicating the list type is included in the **dwItem** member of the structure identified by *lpList*. This flag is required. Use one of the following constants to indicate the list type:

MCI_DGV_LIST_AUDIO_ALG
The command should retrieve names of audio algorithms.

MCI_DGV_LIST_AUDIO_QUALITY
The command should retrieve audio quality levels. The levels returned are associated with the algorithm referenced by the **lpstrAlgorithm** member of the structure identified by *lpList*. If that member is specified using the string "current", then the qualities associated with the current algorithm are returned.

MCI_DGV_LIST_AUDIO_STREAM
The command should retrieve names of audio streams.

MCI_DGV_LIST_STILL_AL
The command should retrieve names of still algorithms.

MCI_DGV_LIST_STILL_QUALITY
The command should retrieve quality levels. The levels returned are associated with the algorithm referenced by the **lpstrAlgorithm** member of the structure identified by *lpList*. If that member is specified using the string "current", then the qualities associated with the current algorithm are returned.

MCI_DGV_LIST_VIDEO_ALG
The command should retrieve names of video algorithms.

MCI_DGV_LIST_VIDEO_QUALITY
The command should retrieve video quality levels. The levels returned are associated with the algorithm referenced by the **lpstrAlgorithm** member of the structure identified by *lpList*. If that member is specified using the string "current", then the qualities associated with the current algorithm are returned.

MCI_DGV_LIST_VIDEO_SOURCE

The command should return information about the video sources. When used with MCI_DGV_LIST_COUNT, the command returns the number of video sources. When used with MCI_DGV_LIST_NUMBER, the command returns the type of a video source. MCI defines the following types:

MCI_DGV_SETVIDEO_SRC_GENERIC

MCI_DGV_SETVIDEO_SRC_NTSC

MCI_DGV_SETVIDEO_SRC_PAL

MCI_DGV_SETVIDEO_SRC_RGB

MCI_DGV_SETVIDEO_SRC_SECAM

MCI_DGV_SETVIDEO_SRC_SVIDEO

There might be more than one source of each type returned. The generic source type is used when more then one type of signal is allowed for that connector.

MCI_DGV_LIST_VIDEO_STREAM
The command should retrieve names of video streams.

MCI_DGV_LIST_NUMBER
An index is specified in the **dwNumber** member of the structure identified by *lpList*. The index must be an integer between 1 and the value returned for the MCI_DGV_LIST_COUNT flag.

For digital-video devices, *lpList* points to an **MCI_DGV_LIST_PARMS** structure.

VCR Flags

The following additional flags apply to the **vcr** device type:

MCI_VCR_LIST_AUDIO_SOURCE
List audio inputs or types.

MCI_VCR_LIST_COUNT
Sets the **dwReturn** member of the structure identified by *lpList* to the total number of video or audio inputs.

MCI_VCR_LIST_NUMBER
Sets the **dwReturn** member of the structure identified by *lpList* to the type of the video or audio input specified by the **dwNumber** member.

MCI_VCR_LIST_VIDEO_SOURCE
List video inputs or types.

For VCR devices, *lpList* points to an **MCI_VCR_LIST_PARMS** structure.

MCI_LOAD

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_LOAD,
    DWORD dwFlags, (DWORD) (LPMCI_LOAD_PARMS) lpLoad);
```

Loads a file. Digital-video and video-overlay devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpLoad
Address of an **MCI_LOAD_PARMS** structure. (Devices with additional parameters might replace this structure with a device-specific structure. For digital-video devices, the *lpLoad* parameter points to an **MCI_DGV_LOAD_PARMS** structure.)

Additional Flag

The following additional flag applies to all devices supporting **MCI_LOAD**:

MCI_LOAD_FILE
The **lpfilename** member of the structure identified by *lpLoad* contains an address of a buffer containing the filename.

Video-Overlay Flags

The following additional flag is used with the **overlay** device type:

MCI_OVLY_RECT
The **rc** member of the structure identified by *lpLoad* contains a valid display rectangle that identifies the area of the video buffer to update.

For video-overlay devices, the *lpLoad* parameter points to an **MCI_OVLY_LOAD_PARMS** structure.

MCI_MARK

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_MARK,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpMark);
```

Records or erases marks that can be used with the **MCI_SEEK** command for high-speed searches. VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpMark
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Additional Flags

The following additional flags apply to VCR devices:

MCI_VCR_MARK_ERASE
Erases a mark at the current position if one exists.

MCI_VCR_MARK_WRITE
Writes a mark at the current position.

MCI_MONITOR

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_MONITOR,
    DWORD dwFlags, (DWORD) (LPMCI_DGV_MONITOR_PARMS) lpMonitor);
```

Specifies the presentation source. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpMonitor
Address of an **MCI_DGV_MONITOR_PARMS** structure.

Additional Flags

The following additional flags apply to digital-video devices:

MCI_DGV_MONITOR_METHOD
A constant indicating the method of monitoring is included in the **dwMethod** member of the structure identified by *lpMonitor*.

When the MCI_DGV_MONITOR_INPUT flag is used in the **dwSource** member, this selects the method of monitoring. Typically, different monitoring methods have different implications on how the hardware is used. The default monitoring method is selected by the device.

MCI_DGV_MONITOR_SOURCE
A constant indicating the monitor source is included in the **dwSource** member of the structure identified by *lpMonitor*.

MCI_OPEN

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_OPEN,
    DWORD dwFlags, (DWORD) (LPMCI_OPEN_PARMS) lpOpen);
```

Initializes a device or file. All devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY or MCI_WAIT. For information about these flags, see Chapter 3, "MCI Overview."

lpOpen
Address of an **MCI_OPEN_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

The MCI_OPEN_TYPE flag must be used whenever a device is specified in the **mciSendCommand** function. If you open a device by specifying a device-type constant, you must specify the MCI_OPEN_TYPE_ID flag in addition to MCI_OPEN_TYPE. For a list of device-type constants, see "Constants: Device Types" later in this chapter.

If the MCI_OPEN_SHAREABLE flag is not specified when a device or file is initially opened, all subsequent **MCI_OPEN** commands to the device or file will fail. If the device or file is already open and this flag is not specified, the call will fail even if the first open command specified MCI_OPEN_SHAREABLE. Files opened for the MCISEQ.DRV and MCIWAVE.DRV devices are nonshareable.

Case is ignored in the device name, but there cannot be leading or trailing blanks.

To use automatic type selection (via the entries in the registry), assign the filename and file extension to the **lpstrElementName** member of the structure identified by *lpOpen*, set the **lpstrDeviceType** member to NULL, and set the MCI_OPEN_ELEMENT flag.

Additional Flags

The following additional flags apply to all devices supporting **MCI_OPEN**:

MCI_OPEN_ALIAS
An alias is included in the **lpstrAlias** member of the structure identified by *lpOpen*.

MCI_OPEN_SHAREABLE
The device or file should be opened as shareable.

MCI_OPEN_TYPE
A device type name or constant is included in the **lpstrDeviceType** member of the structure identified by *lpOpen*.

MCI_OPEN_TYPE_ID
The low-order word of the **lpstrDeviceType** member of the structure identified by *lpOpen* contains a standard MCI device type identifier and the high-order word optionally contains the ordinal index for the device. Use this flag with the MCI_OPEN_TYPE flag.

The following additional flags apply to compound devices:

MCI_OPEN_ELEMENT
A filename is included in the **lpstrElementName** member of the structure identified by *lpOpen*.

MCI_OPEN_ELEMENT_ID
The **lpstrElementName** member of the structure identified by *lpOpen* is interpreted as a doubleword value and has meaning internal to the device. Use this flag with the MCI_OPEN_ELEMENT flag.

Animation Flags

The following additional flags are used with the **animation** device type:

MCI_ANIM_OPEN_NOSTATIC
The device should reduce the number of static (system) colors in the palette to two.

MCI_ANIM_OPEN_PARENT
The parent window handle is specified in the **hWndParent** member of the structure identified by *lpOpen*. The parent window handle is required for some window styles.

MCI_ANIM_OPEN_WS
A window style is specified in the **dwStyle** member of the structure identified by *lpOpen*. The **dwStyle** value 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 window style constants are the same as the ones that are used in the **CreateWindow** function (for example, WS_CHILD, WS_OVERLAPPEDWINDOW, and WS_POPUP).

For animation devices, the *lpOpen* parameter points to an **MCI_ANIM_OPEN_PARMS** structure.

Digital-Video Flags

The following additional flags are used with the **digitalvideo** device type:

MCI_DGV_OPEN_NOSTATIC
The device should reduce the number of static (system) colors in the palette. This increases the number of colors available for rendering the video stream. This flag applies only to devices that share a palette with Windows.

MCI_DGV_OPEN_PARENT
The parent window handle is specified in the **hWndParent** member of the structure identified by *lpOpen*.

MCI_DGV_OPEN_WS
A window style is specified in the **dwStyle** member of the structure identified by *lpOpen*.

For digital-video devices, the *lpOpen* parameter points to an **MCI_DGV_OPEN_PARMS** structure.

Video-Overlay Flags

The following additional flags are used with the **overlay** device type:

MCI_OVLY_OPEN_PARENT
The parent window handle is specified in the **hWndParent** member of the structure identified by *lpOpen*.

MCI_OVLY_OPEN_WS
A window style is specified in the **dwStyle** member of the structure identified by *lpOpen*. The **dwStyle** value 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 standard window styles (such as WS_CHILD, WS_OVERLAPPEDWINDOW, or WS_POPUP).

For video-overlay devices, the *lpOpen* parameter points to an **MCI_OVLY_OPEN_PARMS** structure.

Waveform-Audio Flag

The following additional flag is used with the **waveaudio** device type:

MCI_WAVE_OPEN_BUFFER
A buffer length is specified in the **dwBufferSeconds** member of the structure identified by *lpOpen*.

For waveform-audio devices, the *lpOpen* parameter points to an **MCI_WAVE_OPEN_PARMS** structure. The MCIWAVE driver requires an asychronous waveform-audio device.

MCI_PASTE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_PASTE,
    DWORD dwlags, (DWORD) (LPMCI_DGV_PASTE_PARMS) lpPaste);
```

Pastes data from the clipboard into a file. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpPaste
Address of an **MCI_DGV_PASTE_PARMS** structure.

Additional Flags

The following additional flags apply to digital-video devices:

MCI_DGV_PASTE_AT
A rectangle is included in the **rc** member of the structure identified by *lpPaste*. The first two values of the rectangle specify the point within the frame to place the clipboard information. If the rectangle height and width are nonzero, the clipboard contents are scaled to those dimensions when they are pasted in the frame. If the flag is omitted, **MCI_PASTE** defaults to the entire frame rectangle.

MCI_DGV_PASTE_AUDIO_STREAM
An audio-stream number is included in the **dwAudioStream** member of the structure identified by *lpPaste*. If only one audio stream exists on the clipboard, the audio data is pasted into the designated stream. If more than one audio stream exists on the clipboard, the stream indicates the starting number for the stream sequences. If you use this flag and also want to paste video, you must also use the MCI_DGV_PASTE_VIDEO_STREAM flag. (If neither flag is specified, all audio and video streams are pasted starting with the first audio and video stream. Each pasted stream retains its original stream number.)

MCI_DGV_PASTE_INSERT
Clipboard data should be inserted in the existing workspace at the position specified by the MCI_TO flag. Any existing data after the insertion point is moved in the workspace to make room. This is the default.

MCI_DGV_PASTE_OVERWRITE
Clipboard data should replace data already present in the workspace. The workspace data replaced follows the insertion point.

MCI_DGV_PASTE_VIDEO_STREAM
A video-stream number is included in the **dwVideoStream** member of the structure identified by *lpPaste*. If only one video stream exists on the clipboard, the video data is pasted into the designated stream. If more than one video stream exists on the clipboard, the stream indicates the starting number for the stream sequences. If you use this flag and also want to paste audio, you must also use the MCI_DGV_PASTE_AUDIO_STREAM flag. (If neither flag is specified, all audio and video streams are pasted starting with the first audio and video stream. Each pasted stream retains its original stream number.)

MCI_TO
A position value is included in the **dwTo** member of the structure identified by *lpPaste*. The position value specifies the position to begin pasting data into the workspace. If this flag is omitted, the position defaults to the current position.

MCI_PAUSE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_PAUSE,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpPause);
```

Pauses the current action. Animation, CD audio, digital-video, MIDI sequencer, VCR, videodisc, and waveform-audio devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and VCR devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpPause
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

The difference between the **MCI_STOP** and **MCI_PAUSE** commands depends on the device. If possible, **MCI_PAUSE** suspends device operation but leaves the device ready to resume play immediately. With the MCICDA, MCISEQ, and MCIPIONR drivers, the **MCI_PAUSE** command works the same as the **MCI_STOP** command.

For digital-video devices, the *lpPause* parameter points to an **MCI_DGV_PAUSE_PARMS** structure.

MCI_PLAY

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_PLAY,
    DWORD dwFlags, (DWORD) (LPMCI_PLAY_PARMS ) lpPlay);
```

Signals the device to begin transmitting output data. Animation, CD audio, digital-video, MIDI sequencer, videodisc, VCR, and waveform-audio devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and VCR devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpPlay
Address of an **MCI_PLAY_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Additional Flags

The following additional flags apply to all devices supporting **MCI_PLAY**:

MCI_FROM
A starting location is included in the **dwFrom** member of the structure identified by *lpPlay*. The units assigned to the position values are specified with the MCI_SET_TIME_FORMAT flag of the **MCI_SET** command. If MCI_FROM is not specified, the starting location defaults to the current position.

MCI_TO
An ending location is included in the **dwTo** member of the structure identified by *lpPlay*. The units assigned to the position values are specified with the MCI_SET_TIME_FORMAT flag of **MCI_SET**. If MCI_TO is not specified, the ending location defaults to the end of the media.

Animation Flags

The following additional flags are used with the **animation** device type:

MCI_ANIM_PLAY_FAST
Play fast.

MCI_ANIM_PLAY_REVERSE
Play in reverse.

MCI_ANIM_PLAY_SCAN
Play as quickly as possible.

MCI_ANIM_PLAY_SLOW
Play slowly.

MCI_ANIM_PLAY_SPEED
The play speed is included in the **dwSpeed** member of the structure identified by *lpPlay*.

For animation devices, *lpPlay* points to an **MCI_ANIM_PLAY_PARMS** structure.

Digital-Video Flags

The following additional flags are used with the **digitalvideo** device type:

MCI_DGV_PLAY_REPEAT
Playback should start again at the beginning when the end of the content is reached.

MCI_DGV_PLAY_REVERSE
Playback should occur in reverse.

MCI_MCIAVI_PLAY_WINDOW

Playback should occur in the window associated with a device instance (the default). (This flag is specific to MCIAVI.DRV.)

MCI_MCIAVI_PLAY_FULLSCREEN
Playback should use a full-screen display. Use this flag only when playing compressed or 8-bit files.

For digital-video devices, *lpPlay* points to an **MCI_DGV_PLAY_PARMS** structure.

VCR Flags

The following additional flags are used with the **vcr** device type:

MCI_VCR_PLAY_AT
The **dwAt** member of the structure identified by *lpPlay* contains a time when the entire command begins, or if the device is cued, when the device reaches the from position given by the **MCI_CUE** command.

MCI_VCR_PLAY_REVERSE
Playback should occur in reverse.

MCI_VCR_PLAY_SCAN
Playback should be as fast as possible while maintaining video output.

For VCR devices, *lpPlay* points to an **MCI_VCR_PLAY_PARMS** structure.

Videodisc Flags

The following additional flags are used with the **videodisc** device type:

MCI_VD_PLAY_FAST
Play fast.

MCI_VD_PLAY_REVERSE
Play in reverse.

MCI_VD_PLAY_SCAN
Scan quickly.

MCI_VD_PLAY_SLOW
Play slowly.

MCI_VD_PLAY_SPEED
The play speed is included in the **dwSpeed** member in the structure identified by *lpPlay*.

For animation devices, *lpPlay* points to an **MCI_VD_PLAY_PARMS** structure.

MCI_PUT

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_PUT,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpDest);
```

Sets the source, destination, and frame rectangles. Animation, digital-video, and video-overlay devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpDest
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Animation Flags

The following additional flags are used with the **animation** device type:

MCI_ANIM_PUT_DESTINATION
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
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.

MCI_ANIM_RECT
The **rc** member of the 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.

For animation devices, *lpDest* points to an **MCI_ANIM_RECT_PARMS** structure.

Digital-Video Flags

The following additional flags are used with the **digitalvideo** device type:

MCI_DGV_PUT_CLIENT
The rectangle defined for MCI_DGV_RECT applies to the position of the client window. The rectangle specified is relative to the parent window of the display window. MCI_DGV_PUT_WINDOW must be set concurrently with this flag.

MCI_DGV_PUT_DESTINATION
The rectangle defined for MCI_DGV_RECT specifies a destination rectangle. The destination rectangle specifies the portion of the client window associated with this device driver instance that shows the image or video.

MCI_DGV_PUT_FRAME
The rectangle defined for MCI_DGV_RECT applies to the frame rectangle. The frame rectangle specifies the portion of the frame buffer used as the destination of the video images obtained from the video rectangle. The video should be scaled to fit within the frame buffer rectangle.

The rectangle is specified in frame buffer coordinates. The default rectangle is the full frame buffer. Specifying this rectangle lets the device scale the image as it digitizes the data. Devices that cannot scale the image reject this command with MCIERR_UNSUPPORTED_FUNCTION. You can use the MCI_GETDEVCAPS_CAN_STRETCH flag with the **MCI_GETDEVCAPS** command to determine if a device scales the image. A device returns FALSE if it cannot scale the image.

MCI_DGV_PUT_SOURCE

The rectangle defined for MCI_DGV_RECT specifies a source rectangle. The source rectangle specifies which portion of the frame buffer is to be scaled to fit into the destination rectangle.

MCI_DGV_PUT_VIDEO
The rectangle defined for MCI_DGV_RECT applies to the video rectangle. The video rectangle specifies which portion of the current presentation source is stored in the frame buffer. The rectangle is specified using the natural coordinates of the presentation source. It allows the specification of cropping that occurs prior to storing images and video in the frame buffer. The default rectangle is the full active scan area or the full decompressed images and video.

MCI_DGV_PUT_WINDOW
The rectangle defined for MCI_DGV_RECT applies to the display window. This rectangle is relative to the parent window of the display window (usually the desktop). If the window is not specified, it defaults to the initial window size and position.

MCI_DGV_RECT
The **rc** member of the structure identified by *lpDest* contains a valid rectangle.

For digital-video devices, *lpDest* points to an **MCI_DGV_PUT_PARMS** structure.

Video-Overlay Flags

The following additional flags are used with the **overlay** device type:

MCI_OVLY_PUT_DESTINATION
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
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
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
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.

MCI_OVLY_RECT
The **rc** member of the 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.

For video-overlay devices, *lpDest* points to an **MCI_OVLY_RECT_PARMS** structure.

MCI_QUALITY

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_QUALITY,
    DWORD dwFlags, (DWORD) (LPMCI_DGV_QUALITY_PARMS) lpQuality);
```

Defines a custom quality level for audio, video, or still image data compression. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpQuality
Address of an **MCI_DGV_QUALITY_PARMS** structure.

The name defined for this quality level can be used when setting the audio, video, or still quality with the **MCI_SETAUDIO** and **MCI_SETVIDEO** commands.

Additional Flags

The following additional flags apply to digital-video devices:

MCI_QUALITY_ALG
The **lpstrAlgorithm** member of the structure identified by *lpQuality* contains an address of a buffer containing the name of the algorithm. This algorithm must be supported by the device driver, and must be compatible with the audio, still, or video descriptor that is used. If this flag is omitted, the current algorithm is used.

MCI_QUALITY_DIALOG
The device driver should display a dialog box for specifying the quality level. The dialog box has algorithm-specific fields used internally by the device driver to create a structure describing a specific quality level.

MCI_QUALITY_HANDLE
The **dwHandle** member of the structure identified by *lpQuality* contains a handle to a structure. The structure contains algorithmic-specific data describing the specific quality level. The format of the structures for the algorithms is device dependent.

MCI_QUALITY_ITEM
A constant indicating the type of algorithm is included in the **dwItem** member of the structure identified by *lpQuality*.

MCI_QUALITY_NAME
The **lpstrName** member of the structure identified by *lpQuality* contains an address of a buffer containing the quality descriptor.

MCI_REALIZE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_REALIZE,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpRealize);
```

Causes a graphic device to realize its palette into a device context (DC). Animation and digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpRealize
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

You should use this command when your application receives the WM_QUERYNEWPALETTE message.

Animation Flags

The following additional flags are used with the **animation** device type:

MCI_ANIM_REALIZE_BKGD
Realizes the palette as a background palette.

MCI_ANIM_REALIZE_NORM
Realizes the palette normally. This is the default.

Digital-Video Flags

The following additional flags are used with the **digitalvideo** device type:

MCI_DGV_REALIZE_BKGD
Realizes the palette as a background palette.

MCI_DGV_REALIZE_NORM
Realizes the palette normally. This is the default.

For digital-video devices, the *lpRealize* parameter points to an **MCI_REALIZE_PARMS** structure. The **MCI_REALIZE_PARMS** structure is identical to the **MCI_GENERIC_PARMS** structure.

MCI_RECORD

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_RECORD,
    DWORD dwFlags, (DWORD) (LPMCI_RECORD_PARMS) lpRecord);
```

Starts recording from the current position or from one specified location to another specified location. VCR and waveform-audio devices recognize this command. Although digital-video devices and MIDI sequencers also recognize this command, the MCIAVI and MCISEQ drivers do not implement it.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and VCR devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpRecord
Address of an **MCI_RECORD_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

This command is supported by devices that return TRUE when you call the **MCI_GETDEVCAPS** command with the MCI_GETDEVCAPS_CAN_RECORD flag. For the MCIWAVE driver, all data recorded after a file is opened is discarded if the file is closed without saving it.

Additional Flags

The following additional flags apply to all devices supporting **MCI_RECORD**:

MCI_FROM
A starting location is included in the **dwFrom** member of the structure identified by *lpRecord*. The units assigned to the position values are specified with the MCI_SET_TIME_FORMAT flag of the **MCI_SET** command. If MCI_FROM is not specified, the starting location defaults to the current position.

MCI_RECORD_INSERT
Newly recorded information should be inserted or pasted into the existing data. Some devices might not support this. If supported, this is the default.

MCI_RECORD_OVERWRITE
Data should overwrite existing data. The MCIWAVE.DRV device returns MCIERR_UNSUPPORTED_FUNCTION in response to this flag.

MCI_TO
An ending location is included in the **dwTo** member of the structure identified by *lpRecord*. The units assigned to the position values are specified with the MCI_SET_TIME_FORMAT flag of the **MCI_SET** command. If MCI_TO is not specified, the ending location defaults to the end of the content.

Digital-Video Flags

The following additional flags are used with the **digitalvideo** device type:

MCI_DGV_RECORD_AUDIO_STREAM
An audio-stream number is included in the **dwAudioStream** member of the structure identified by *lpRecord*. If you omit this flag, audio data is recorded into the first physical stream.

MCI_DGV_RECORD_HOLD
When recording stops, the screen will hold the last image and will not resume showing the video until an **MCI_MONITOR** command is issued.

MCI_DGV_RECORD_VIDEO_STREAM
A video-stream number is included in the **dwVideoStream** member of the structure identified by *lpRecord*. If you omit this flag, video data is recorded into the first physical stream.

MCI_DGV_RECT
A rectangle is specified in the **rc** member of the structure identified by *lpRecord*. The rectangle specifies the region of the external input used as the source for the pixels compressed and saved. This rectangle defaults to the rectangle specified (or defaulted) by the MCI_DGV_PUT_VIDEO flag for the **MCI_PUT** command. When it is set differently than the video rectangle, what is displayed is not what is recorded

For digital-video devices, *lpRecord* points to an **MCI_DGV_RECORD_PARMS** structure.

VCR Flags

The following additional flags are used with the **vcr** device type:

MCI_VCR_RECORD_AT
The **dwAt** member of the structure identified by *lpRecord* contains a time when the entire command begins, or if the device is cued, when the device reaches the from position given by the cue command.

MCI_VCR_RECORD_INITIALIZE
Seek the device to the start of the media, begin recording blank video and audio, and record timecode, if possible.

For VCR devices, *lpRecord* points to an **MCI_VCR_RECORD_PARMS** structure.

MCI_RESERVE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_RESERVE,
    DWORD dwFlags, (DWORD) (LPMCI_DGV_RESERVE_PARMS) lpReserve);
```

Allocates contiguous disk space for the workspace of the device driver instance for use with subsequent recording. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpReserve
Address of an **MCI_DGV_RESERVE_PARMS** structure.

If the workspace contains unsaved data, this data is lost. If disk space is not reserved prior to recording, the **MCI_RECORD** command performs an implied reserve with device-specific default parameters. On some implementations, reserve is not required and might be ignored by the device driver. Explicitly reserving space gives you better control over when the delay for disk allocation occurs, how much space is allocated, and where the disk space is allocated. The amount and location of disk space already reserved for this device instance can be changed by issuing **MCI_RESERVE** again. Any allocated and still unused disk space is not deallocated until any recorded data is saved or until the device driver instance is closed.

If video is turned off with the MCI_OFF flag of the **MCI_SETVIDEO** command, the space reserved does not include any video. If audio is turned off with the MCI_OFF flag of the **MCI_SETAUDIO** command, the space reserved does not include any audio. If both audio and video are turned off or if the requested size is zero, no space is reserved and any existing reserved space is deallocated.

Additional Flags

The following additional flags apply to digital-video devices:

MCI_DGV_RESERVE_IN
The **lpstrPath** member of the structure identified by *lpReserve* contains an address of a buffer containing the location of a temporary file. The buffer contains only the drive and directory path of the file used to hold recorded data; the filename is specified by the device driver. This temporary file is deleted when the device instance is closed unless it is explicitly saved. If this flag is omitted, the device driver specifies where disk space is allocated.

MCI_DGV_RESERVE_SIZE
The **dwSize** member of the structure identified by *lpReserve* specifies the approximate amount of disk space to reserve in the workspace for recording. The value is specified in the current time format. The amount of disk space is estimated from the requested time and from which file format and video and audio algorithm and quality values are in effect. If this flag is omitted, the device driver might use a default value it defines.

MCI_RESTORE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_RESTORE,
    DWORD dwFlags, (DWORD) (LPMCI_DGV_RESTORE_PARMS) lpRestore);
```

Copies a bitmap from a file to the frame buffer. Digital-video devices recognize this command. This command performs the opposite action of the **MCI_CAPTURE** command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpRestore
Address of an **MCI_DGV_RESTORE_PARMS** structure.

The implementation can recognize a variety of image formats, but a Windows device-independent bitmap (DIB) is always accepted.

Additional Flags

The following additional flags apply to digital-video devices:

MCI_DGV_RESTORE_FROM
The **lpstrFileName** member of the structure identified by *lpRestore* contains an address of a buffer containing the source filename. The filename is required.

MCI_DGV_RESTORE_AT
The **rc** member of the structure identified by *lpRestore* contains a valid rectangle. The rectangle specifies a region of the frame buffer relative to its origin. The first pair of coordinates specifies the upper left corner of the rectangle; the second pair specifies the width and height. If this flag is not specified, the image is copied to the upper left corner of the frame buffer.

MCI_RESUME

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_RESUME,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpResume);
```

Causes a paused device to resume the paused operation. Animation, digital-video, VCR, and waveform-audio devices recognize this command. Although CD audio, MIDI sequencer, and videodisc devices also recognize this command, the MCICDA, MCISEQ, and MCIPIONR device drivers do not support it.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and VCR devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpResume
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

This command resumes playing and recording without changing the current track position set with **MCI_PLAY** or **MCI_RECORD**.

MCI_SAVE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_SAVE,
    DWORD dwFlags, (DWORD) (LPMCI_SAVE_PARMS ) lpSave);
```

Saves the current file. Devices that modify files should not destroy the original copy until they receive the save message. Video-overlay and waveform-audio devices recognize this command. Although digital-video devices and MIDI sequencers also recognize this command, the MCIAVI and MCISEQ drivers do not implement it.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and VCR devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpSave
Address of an **MCI_SAVE_PARMS** structure. (Devices with additional parameters might replace this structure with a device-specific structure.)

This command is supported by devices that return TRUE when you call the **MCI_GETDEVCAPS** command with the MCI_GETDEVCAPS_CAN_SAVE flag.

Additional Flag

The following additional flag applies to all devices supporting **MCI_SAVE**:

MCI_SAVE_FILE
The **lpfilename** member of the structure identified by *lpSave* contains an address of a buffer containing the destination filename.

Digital-Video Flags

The following additional flags are used with the **digitalvideo** device type:

MCI_DGV_RECT
The **rc** member of the structure identified by *lpSave* contains a valid rectangle. The rectangle specifies a region of the frame buffer that will be saved to the specified file. The first pair of coordinates specifies the upper left corner of the rectangle; the second pair specifies the width and height. Digital-video devices must use the **MCI_CAPTURE** command to capture the contents of the frame buffer. (Video-overlay devices should also use **MCI_CAPTURE**.) This flag is for compatibility with the existing MCI video-overlay command set.

MCI_DGV_SAVE_ABORT
Stops a save operation in progress. This must be the only flag present.

MCI_DGV_SAVE_KEEPRESERVE
Unused disk space left over from the original **MCI_RESERVE** command is not deallocated.

For digital-video devices, the *lpSave* parameter points to an **MCI_DGV_SAVE_PARMS** structure.

Video-Overlay Flags

The following additional flag is used with the **overlay** device type:

MCI_OVLY_RECT
The **rc** member of the structure identified by *lpSave* contains a valid display rectangle indicating the area of the video buffer to save.

For video-overlay devices, the *lpSave* parameter points to an **MCI_OVLY_SAVE_PARMS** structure.

MCI_SEEK

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_SEEK,
    DWORD dwFlags, (DWORD) (LPMCI_SEEK_PARMS) lpSeek);
```

Changes the current position in the content as quickly as possible. Video and audio output are disabled during the seek. After the seek is complete, the device is stopped. Animation, CD audio, digital-video, MIDI sequencer, VCR, videodisc, and waveform-audio devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and VCR devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpSeek
Address of an **MCI_SEEK_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

If a data sample size for a device is larger than 1 byte (such as with waveform-audio stereo data), this command moves to the beginning of the nearest sample when a specified position does not coincide with the start of a sample.

Additional Flags

The following additional flags apply to all devices supporting **MCI_SEEK**:

MCI_SEEK_TO_END
Seek to the end of the content.

MCI_SEEK_TO_START
Seek to the beginning of the content.

MCI_TO
A position is included in the **dwTo** member of the structure identified by *lpSeek*. The units assigned to the position values are specified with the MCI_SET_TIME_FORMAT flag of the **MCI_SET** command. Do not use this flag with MCI_SEEK_TO_END or MCI_SEEK_TO_START.

VCR Flags

The following additional flags are used with the **vcr** device type:

MCI_VCR_SEEK_AT
The **dwAt** member of the structure identified by *lpSeek* contains a time when the entire command begins.

MCI_VCR_SEEK_MARK
The **dwMark** member of the structure identified by *lpSeek* contains the numbered mark to search for.

MCI_VCR_SEEK_REVERSE
Seek direction is reverse; this is used only with the MCI_VCR_SEEK_MARK flag.

For VCR devices, the *lpSeek* parameter points to an **MCI_VCR_SEEK_PARMS** structure.

Videodisc Flag

The following additional flag is used with the **videodisc** device type:

MCI_VD_SEEK_REVERSE
Seek direction is reverse.

MCI_SET

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_SET,
    DWORD dwFlags, (DWORD) (LPMCI_SET_PARMS) lpSet);
```

Sets device information. Animation, CD audio, digital-video, MIDI sequencer, VCR, videodisc, video-overlay, and waveform-audio devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and VCR devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpSet
Address of an **MCI_SET_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Additional Flags

The following additional flags apply to all devices supporting **MCI_SET**:

MCI_SET_AUDIO
An audio channel number is included in the **dwAudio** member of the 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
All audio channels.

MCI_SET_AUDIO_LEFT
Left channel.

MCI_SET_AUDIO_RIGHT
Right channel.

MCI_SET_DOOR_CLOSED
Closes the media cover (if any).

MCI_SET_DOOR_OPEN
Opens the media cover (if any).

MCI_SET_OFF
Disables the specified video or audio channel.

MCI_SET_ON
Enables the specified video or audio channel.

MCI_SET_TIME_FORMAT
A time format parameter is included in the **dwTimeFormat** member of the structure identified by *lpSet*. The following flags are used with this flag:

MCI_FORMAT_BYTES
Within a PCM (Pulse Code Modulation) data format, changes the time member description to bytes for input or output. Recognized by the **waveaudio** device type.

MCI_FORMAT_FRAMES
Subsequent commands will use frames. Recognized by the **animation, digitalvideo, vcr,** and **videodisc** device types.

MCI_FORMAT_HMS
Changes the time format to hours, minutes, and seconds. Recognized by the **vcr** and **videodisc** device types.

MCI_FORMAT_MILLISECONDS
Changes the time format to milliseconds. Recognized by all device types.

MCI_FORMAT_MSF
Changes the time format to minutes, seconds, and frames. Recognized by the **cdaudio** and **vcr** device types.

MCI_FORMAT_SAMPLES
Changes the time format to samples for input or output. Recognized by the **waveaudio** device type.

MCI_FORMAT_SMPTE_24, MCI_FORMAT_SMPTE_25, and MCI_FORMAT_SMPTE_30
Sets the time format to 24, 25, and 30 frame SMPTE (Society of Motion Picture and Television Engineers), respectively. Recognized by the **sequencer** and **vcr** device types.

MCI_FORMAT_SMPTE_30DROP
Sets the time format to 30 drop-frame SMPTE. Recognized by the **sequencer** and **vcr** device types.

MCI_FORMAT_TMSF
Changes the time format to tracks, minutes, seconds, and frames. (MCI uses continuous track numbers.) Recognized by the **cdaudio** and **vcr** device types.

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.

Digital-Video Flags

The following additional flags are used with the **digitalvideo** device type:

MCI_DGV_SET_FILEFORMAT
A file format parameter is included in the **dwFileFormat** member of the structure identified by *lpSet*. For digital-video devices, the file format is used for save or capture commands. If omitted, this might default to a device driver defined format. If the specified file format conflicts with the currently selected algorithm and quality, then they are changed to the defaults for the file format. The following file format constants are defined:

MCI_DGV_FF_AVI
AVI format.

MCI_DGV_FF_AVSS
AVSS format.

MCI_DGV_FF_DIB
DIB format.

MCI_DGV_FF_JFIF
JFIF format.

MCI_DGV_FF_JPEG
JPEG format.

MCI_DGV_FF_MPEG
MPEG format.

MCI_DGV_FF_RDIB
RLE DIB format.

MCI_DGV_FF_RJPEG
RJPEG format.

MCI_DGV_SET_SEEK_EXACTLY
Sets the format used for positioning. This flag must be used with MCI_SET_ON or MCI_SET_OFF. If MCI_SET_ON is specified, playing or recording precisely accesses the frame specified with the MCI_FROM flag. This might add some extra delay if the requested frame is not a key frame. If MCI_SET_OFF is specified, the device will seek to a key-frame image that precedes the requested frame. For some files and devices, this might be the first frame of the file. The default for this flag is device dependent.

MCI_DGV_SET_SPEED
A speed parameter is included in the **dwSpeed** member of the structure identified by *lpSet*. Speed is specified as a ratio between the nominal frame rate and the desired frame rate where the nominal frame rate is designated as 1000. Half speed is 500 and double speed is 2000. The allowable speed range is dependent on the device and possibly the file, too.

MCI_DGV_SET_STILL
When used with MCI_DGV_SET_FILEFORMAT, **MCI_SET** sets the file format used for capture commands.

For digital-video devices, the *lpSet* parameter points to an **MCI_DGV_SET_PARMS** structure.

Sequencer Flags

The following additional flags are used with the **sequencer** device type:

MCI_SEQ_FORMAT_SONGPTR
Sets the time format to song pointer units.

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** member of the 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** member of the structure identified by *lpSet*. This affects only 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 identifier in the **dwPort** member of the structure identified by *lpSet*. The device closes the previous port (if any), and attempts to open and use the new port. If it fails, it returns an error and reopens 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 behaves exactly the same as if a port were open, except no MIDI message is sent.

MIDI_MAPPER
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** member of the structure identified by *lpSet*. MCISEQ returns MCIERR_UNSUPPORTED_FUNCTION. 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_MIDI
Sets the sequencer to receive MIDI synchronization data.

MCI_SEQ_NONE
Sets the sequencer to ignore synchronization data in a MIDI stream.

MCI_SEQ_SMPTE

Sets the sequencer to receive SMPTE synchronization data.

MCI_SEQ_SET_TEMPO
Changes the tempo of the MIDI sequence to that specified by the **dwTempo** member 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.

For sequencer devices, the *lpSet* parameter points to an **MCI_SEQ_SET_PARMS** structure.

VCR Flags

The following additional flags are used with the **vcr** device type:

MCI_VCR_SET_ASSEMBLE_RECORD
Sets the device to record in assemble or insert modes (when assemble is off, insert is on, and vice-versa). Use with one of the following flag:

MCI_SET_ON
Sets assemble record on, and turns insert record off. Records all video, audio and timecode tracks.

MCI_SET_OFF
Sets assemble record off, and turns insert record on. When assemble record is off, individual tracks of video, audio, and timecode can be selected for recording.

MCI_VCR_SET_CLOCK
The **dwClock** member of the structure identified by *lpSet* contains the new clock time.

MCI_VCR_SET_COUNTER_FORMAT
The **dwCounterFormat** member of the structure identified by *lpSet* contains a constant specifying the new counter-time format to be used by the status counter. For a list of valid constants, see MCI_SET_TIME_FORMAT in the list of additional flags for this command.

MCI_VCR_SET_COUNTER_VALUE
The **dwCounterValue** member of the structure identified by *lpSet* contains the new counter value.

MCI_VCR_SET_INDEX
The **dwIndex** member of the structure identified by *lpSet* contains a constant indicating the contents of the on-screen display and must be one of the following:

MCI_VCR_INDEX_COUNTER
Displays counter.

MCI_VCR_INDEX_DATE
Displays date.

MCI_VCR_INDEX_TIME
Displays time.

MCI_VCR_INDEX_TIMECODE
Displays timecode.

For more information, see the **MCI_INDEX** command.

MCI_VCR_SET_PAUSE_TIMEOUT
The **dwPauseTimeout** member of the structure identified by *lpSet* contains the maximum duration, in milliseconds, of a pause command.

MCI_VCR_SET_POSTROLL_DURATION
The **dwPostrollDuration** member of the structure identified by *lpSet* contains the videotape length, in the current time format, needed to brake the VCR transport when a stop or pause command is issued.

MCI_VCR_SET_POWER
Sets the power on or off. Must be used with one of the following flags:

MCI_SET_OFF
Turns power off.

MCI_SET_ON
Turns power on.

MCI_VCR_SET_PREROLL_DURATION
The **dwPrerollDuration** member of the structure identified by *lpSet* contains the videotape length, in the current time format, needed to stabilize the VCR output.

MCI_VCR_SET_RECORD_FORMAT
The **dwRecordFormat** member of the structure identified by *lpSet* contains a constant describing the record speed, which must be one of the following:

MCI_VCR_FORMAT_EP
Records at slow speed.

MCI_VCR_FORMAT_LP
Records at medium-slow speed.

MCI_VCR_FORMAT_SP
Records at standard speed.

MCI_VCR_SET_SPEED
The **dwSpeed** member of the structure identified by *lpSet* contains the new speed setting, where 1000 is normal speed, 2000 is double speed, and 500 is half speed, and so on.

MCI_VCR_SET_TAPE_LENGTH
The **dwTapeLength** member of the structure identified by *lpSet* contains the new length of the tape, provided that the length of the tape is undetectable.

MCI_VCR_SET_TIME_MODE
The **dwTimeMode** member of the structure identified by *lpSet* contains a constant indicating the new positional time mode. The following constants are valid:

MCI_VCR_TIME_COUNTER
Forces the device to use counter exclusively.

MCI_VCR_TIME_DETECT
Each time a new videotape is inserted into the device, or the mode changes from not ready to ready, the device should attempt to determine if there is timecode available on the videotape. If timecode is available, use timecode in all subsequent commands that specify positions. Otherwise, use the counter.

MCI_VCR_TIME_TIMECODE
Forces the device to use timecode exclusively.

MCI_VCR_SET_TRACKING
Tunes the speed of the VCR tape transport with a fine adjustment, and must be used with one of the following flags:

MCI_VCR_PLUS
Increases the tape transport speed.

MCI_VCR_MINUS
Decreases the tape transport speed.

MCI_VCR_RESET
Returns the tracking adjustment to zero.

For VCR devices, the *lpSet* parameter points to an **MCI_VCR_SET_PARMS** structure.

Videodisc Flags

The following additional flag is used with the **videodisc** device type:

MCI_VD_FORMAT_TRACK
Changes the time format to tracks. MCI uses continuous track numbers.

For videodisc devices, the *lpSet* parameter points to an **MCI_VD_SET_PARMS** structure.

Waveform-Audio Flags

The following additional flags are used with the **waveaudio** device type:

MCI_WAVE_INPUT
Sets the input used for recording to the **wInput** member of the structure identified by *lpSet*.

MCI_WAVE_OUTPUT
Sets the output used for playing to the **wOutput** member of the structure identified by *lpSet*.

MCI_WAVE_SET_ANYINPUT
Any wave input compatible with the current format can be used for recording.

MCI_WAVE_SET_ANYOUTPUT
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** member of the structure identified by *lpSet*.

MCI_WAVE_SET_BITSPERSAMPLE
Sets the bits per sample used for playing, recording, and saving to the **nBitsPerSample** member of the PCM data format identified by *lpSet*.

MCI_WAVE_SET_BLOCKALIGN
Sets the block alignment used for playing, recording, and saving to the **nBlockAlign** member of the structure identified by *lpSet*.

MCI_WAVE_SET_CHANNELS
The number of channels is indicated in the **nChannels** member of the structure identified by *lpSet*.

MCI_WAVE_SET_FORMATTAG
Sets the format type used for playing, recording, and saving to the **wFormatTag** member of the 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** member of the structure identified by *lpSet*.

For waveform-audio devices, the *lpSet* parameter points to an **MCI_WAVE_SET_PARMS** structure.

Several properties of waveform-audio data are defined when the file to store the data is created. These properties describe how the data is structured within the file and cannot be changed once recording begins. The following list of flags identifies these properties:

- MCI_WAVE_SET_AVGBYTESPERSEC
- MCI_WAVE_SET_BITSPERSAMPLE
- MCI_WAVE_SET_BLOCKALIGN
- MCI_WAVE_SET_CHANNELS
- MCI_WAVE_SET_FORMATTAG
- MCI_WAVE_SET_SAMPLESPERSEC

MCI_SETAUDIO

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_SETAUDIO,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpSetAudio);
```

Sets values associated with audio playback and capture. Digital-video and VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpSetAudio
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Digital-Video Flags

The following flags apply to the **digitalvideo** device type:

MCI_DGV_SETAUDIO_ALG
The **lpstrAlgorithm** member of the structure identified by *lpSetAudio* contains an address of a buffer containing the name of an audio compression algorithm. The compression algorithm is used by subsequent **MCI_RESERVE** or **MCI_RECORD** commands. The available algorithms are device dependent. If the algorithm is incompatible with the current file format, the file format is changed to the default format for the algorithm.

MCI_DGV_SETAUDIO_CLOCKTIME
The time specified is in milliseconds and is absolute time when used with MCI_DGV_SETAUDIO_OVER. (This time is not in step with the playing of the workspace.)

MCI_DGV_SETAUDIO_INPUT
Modifies the bass, treble, or volume flag so that it affects the input signal and modifies what is recorded. If possible, this is the default when monitoring the input.

MCI_DGV_SETAUDIO_ITEM
An audio constant is specified in the **dwItem** member of the structure identified by *lpSetAudio*. The constant identifies the value that is being set. The following constants are defined:

MCI_DGV_SETAUDIO_AVGBYTESPERSEC
The average number of bytes is specified in the **dwValue** member of the structure identified by *lpSetAudio*. This value sets the average number of bytes per second for playing or recording in the PCM (Pulse Code Modulation) and ADPCM (Adaptive Differential Pulse Code Modulation) formats. The file is saved in this format.

MCI_DGV_SETAUDIO_BASS
The audio low frequency level is specified as a factor in the **dwValue** member of the structure identified by *lpSetAudio*.

MCI_DGV_SETAUDIO_BITSPERSAMPLE
The number of bits per sample is specified in the **dwValue** member of the structure identified by *lpSetAudio*. This value sets the number of bits per sample played or recorded in the PCM format. The file is saved in this format.

MCI_DGV_SETAUDIO_BLOCKALIGN
The data block alignment is specified in the **dwValue** member of the structure identified by *lpSetAudio*. This value sets the alignment of data blocks relative to the start of input waveform data.

MCI_DGV_SETAUDIO_SAMPLESPERSEC
The sample rate is specified in the **dwValue** member of the structure identified by *lpSetAudio*.

This value sets the sample rate for playing and recording with the PCM and ADPCM algorithms. The file is saved in this format.

MCI_DGV_SETAUDIO_SOURCE

A constant specifying the source of audio input is included in the **dwValue** member of the structure identified by *lpSetAudio*. The following constants are defined for the audio input sources:

MCI_DGV_SETAUDIO_SOURCE_AVERAGE

The average of the left and right audio channels.

MCI_DGV_SETAUDIO_SOURCE_LEFT

Left audio channel.

MCI_DGV_SETAUDIO_SOURCE_RIGHT

Right audio channel.

MCI_DGV_SETAUDIO_SOURCE_STEREO

Stereo.

MCI_DGV_SETAUDIO_STREAM

An audio-stream is specified in the **dwValue** member of the structure identified by *lpSetAudio*. The integer value specifies the audio stream played back from the workspace. If the stream is not specified, the first physically interleaved audio stream is played.

MCI_DGV_SETAUDIO_TREBLE

The audio high-frequency level is specified as a factor in the **dwValue** member of the structure identified by *lpSetAudio*.

MCI_DGV_SETAUDIO_VOLUME

The audio level for one or both audio channels is specified as a factor in the **dwValue** member of the structure identified by *lpSetAudio*. If the left and right volumes have been set to different values, then the ratio of left to right volume is approximately unchanged.

MCI_DGV_SETAUDIO_LEFT

Enables the left audio channel when used with MCI_SET_ON. Disables the left audio channel when used with MCI_SET_OFF. When this flag is used with the combination of MCI_DGV_SETAUDIO_VALUE and MCI_DGV_SETAUDIO_VOLUME, it sets the volume of the left audio channel. When this flag is used with MCI_DGV_SETAUDIO_SOURCE, it specifies the left audio channel as the source for the audio input digitizer.

MCI_DGV_SETAUDIO_OVER

A transition length parameter is included in the **dwOver** member of the structure identified by *lpSetAudio*. The length value specifies how long (in units of the current time format) it should take to make a change that uses a factor. If this flag is not used, changes occur immediately.

MCI_DGV_SETAUDIO_QUALITY

The **lpstrQuality** member of the structure identified by *lpSetAudio* contains an address of a buffer defining the audio quality. A text-string within the buffer specifies the characteristics of the audio compression algorithm.

The MCI_DGV_SETAUDIO_ALG flag can be used to select a quality descriptor for the specified algorithm. If this flag is omitted, then the current algorithm is used.

The algorithms and descriptor names available depend on the device. Each device supplies documentation for the available algorithms and a description of the applicable descriptor names. The **MCI_QUALITY** command can define additional descriptor names.

MCI_DGV_SETAUDIO_RECORD

Specifies whether recording includes or excludes audio data. When combined with MCI_SET_ON, audio data is recorded. When combined with MCI_SET_OFF, audio data is excluded. The default includes audio data.

MCI_DGV_SETAUDIO_RIGHT

Enables the right audio channel when used with MCI_SET_ON. Disables the right audio channel when used with MCI_SET_OFF. When this flag is used with the combination of

MCI_DGV_SETAUDIO_VALUE and MCI_DGV_SETAUDIO_VOLUME, it sets the volume of the right audio channel.

MCI_DGV_SETAUDIO_VALUE
A value is specified in the **dwValue** member of the structure identified by *lpSetAudio*. The meaning of the value is specified by the constant defined for the MCI_DGV_SETAUDIO_ITEM flag.

MCI_SET_OFF
Disables the specified audio channel.

MCI_SET_ON
Enables the specified audio channel.

MCI_SETAUDIO_OUTPUT
Modifies the bass, treble, or volume flag so that it modifies only the played signal and not what is recorded. If possible, this is the default when monitoring the input.

For digital-video devices, the *lpSetAudio* parameter points to an **MCI_DGV_SETAUDIO_PARMS** structure.

VCR Flags

The following additional flags are used with the **vcr** device type:

MCI_VCR_SETAUDIO_RECORD
Sets the audio recording to on or off, which is used in conjunction with one of following flags:

MCI_SET_ON
Audio recording on.

MCI_SET_OFF
Audio recording off. It might be necessary to first turn off the assemble recording (using the **MCI_SET** command with the MCI_VCR_SET_ASSEMBLE_RECORD flag set to off) before the audio recording can be turned off.

MCI_TRACK
The **dwTrack** member of the structure identified by *lpSetAudio* specifies which track is affected by the command.

MCI_VCR_SETAUDIO_SOURCE
Sets the audio source. This flag must be used with the MCI_VCR_SETAUDIO_TO flag.

MCI_VCR_SETAUDIO_MONITOR
Sets the audio source monitor. This flag must be used with the MCI_VCR_SETAUDIO_TO flag.

MCI_VCR_SETAUDIO_TO
The **dwTo** member of the structure identified by *lpSetAudio* contains a constant describing the type of input or monitored input. It must be one of the following:

MCI_VCR_SRC_TYPE_TUNER
Type is tuner.

MCI_VCR_SRC_TYPE_LINE
Type is line.

MCI_VCR_SRC_TYPE_AUX
Type is auxiliary.

MCI_VCR_SRC_TYPE_GENERIC
Type is generic.

MCI_VCR_SRC_TYPE_MUTE
Type is mute. This can be used only with the MCI_VCR_SETAUDIO_SOURCE flag.

MCI_VCR_SRC_TYPE_OUTPUT
Type is output.

MCI_VCR_SETAUDIO_NUMBER
The **dwNumber** member of the structure identified by *lpSetAudio* contains the audio input (of the

type specified in the **dwTo** member) to use.

For VCR devices, the *lpSetAudio* parameter points to an **MCI_VCR_SETAUDIO_PARMS** structure.

MCI_SETTIMECODE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_SETTIMECODE,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpSetTimeCode);
```

Enables or disables timecode recording for a VCR. VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpSetTimeCode
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Additional Flags

The following additional flag applies to VCR devices:

MCI_VCR_SETTIMECODE_RECORD
Sets the timecode track recording to on or off. This flag is used in combination with one of the following additional flags:

MCI_SET_ON
Timecode recording on.

MCI_SET_OFF
Timecode recording off.

MCI_SETTUNER

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_SETTUNER,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpSetTuner);
```

Sets the current channel on the tuner. VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpSetTuner
Address of an **MCI_VCR_SETTUNER_PARMS** structure.

Additional Flags

The following additional flags apply to VCR devices:

MCI_VCR_SETTUNER_CHANNEL
The **dwChannel** member of the structure identified by *lpSetTuner* contains the new channel number.

MCI_VCR_SETTUNER_CHANNEL_DOWN
Decrements the tuner channel.

MCI_VCR_SETTUNER_CHANNEL_SEEK_DOWN
Searches for a valid channel in the reverse direction.

MCI_VCR_SETTUNER_CHANNEL_SEEK_UP
Searches for a valid channel in the forward direction.

MCI_VCR_SETTUNER_CHANNEL_UP
Increments the tuner channel.

MCI_VCR_SETTUNER_NUMBER
The **dwNumber** member of the structure identified by *lpSetTuner* specifies which logical tuner to affect with this command.

MCI_SETVIDEO

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_SETVIDEO,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpSetVideo);
```

Sets values associated with video playback. Digital-video and VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpSetVideo
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Digital-Video Flags

The following additional flags are used with the **digitalvideo** device type:

MCI_DGV_SETVIDEO_ALG
The **lpstrAlgorithm** member of the structure identified by *lpSetVideo* contains an address of a buffer containing the name of a video compression algorithm. The compression algorithm is used by subsequent **MCI_RESERVE** or **MCI_RECORD** commands. The available algorithms are device dependent.

If the specified algorithm is incompatible with the current file format, the file format is changed to the default format for the algorithm.

MCI_DGV_SETVIDEO_CLOCKTIME
When used with MCI_DGV_SETVIDEO_OVER, indicates time is specified in milliseconds and is absolute time. (This time is not in step with the playing of the workspace.)

MCI_DGV_SETVIDEO_INPUT
Modifies the MCI_DGV_SETVIDEO_BRIGHTNESS, MCI_DGV_SETVIDEO_COLOR, MCI_DGV_SETVIDEO_CONTRAST, MCI_DGV_SETVIDEO_GAMMA, MCI_DGV_SETVIDEO_SHARPNESS, or MCI_DGV_SETVIDEO_TINT so that it affects the input signal and modifies what is recorded. If possible, this is the default when monitoring the input.

MCI_DGV_SETVIDEO_ITEM
A video constant is specified in the **dwItem** member of the structure identified by *lpSetVideo*. The constant identifies the value that is being set. You can specify the following constants with this flag:

MCI_AVI_SETVIDEO_DRAW_PROCEDURE
A new drawing procedure address is specified in the **dwValue** member of the structure identified by *lpSetVideo*. You can specify a new drawing procedure only when the device is idle. This flag is recognized only by the MCIAVI digital-video driver. There is no equivalent to this flag in the string command interface.

MCI_AVI_SETVIDEO_PALETTE_COLOR
A new palette color is specified in the **dwOver** and **dwValue** members of the structure identified by *lpSetVideo*. The **dwOver** member specifies the palette index of the color to be changed and the **dwValue** member specifies the new color, as an RGB value. You must also specify the MCI_DGV_SETVIDEO_OVER and MCI_DGV_SETVIDEO_VALUE flags with MCI_DGV_SETVIDEO_ITEM when you use this constant. This flag is recognized only by the MCIAVI digital-video driver.

MCI_AVI_SETVIDEO_PALETTE_HALFTONE
Indicates that the halftone palette should be used, instead of the default palette. This flag is recognized only by the MCIAVI digital-video driver.

MCI_DGV_SETVIDEO_BITSPERPEL

The number of bits per pixel is specified in the **dwValue** member of the structure identified by *lpSetVideo*. The number of bits per pixel is used for saving captured or recorded data

MCI_DGV_SETVIDEO_BRIGHTNESS
The video brightness level is specified as a factor in the **dwValue** member of the structure identified by *lpSetVideo*.

MCI_DGV_SETVIDEO_COLOR
The video color saturation level is specified as a factor in the **dwValue** member of the structure identified by *lpSetVideo*.

MCI_DGV_SETVIDEO_CONTRAST
The video contrast level is specified as a factor in the **dwValue** member of the structure identified by *lpSetVideo*.

MCI_DGV_SETVIDEO_FRAME_RATE
A frame rate is specified in the **dwValue** member of the structure identified by *lpSetVideo*. The rate is specified in units of frames per second times 1000. For example, 29.97 frames per second is specified as 29970.

MCI_DGV_SETVIDEO_GAMMA
A gamma correction exponent value is specified in the **dwValue** member of the structure identified by *lpSetVideo*. Gamma correction adjusts the mapping between the intensity encoded in the presentation source and the displayed brightness. The value is the exponent multiplied by 1000. For example, 2200 indicates an exponent of 2.2. A value of 1000 indicates an exponent of 1, which applies no gamma correction.

MCI_DGV_SETVIDEO_KEY_COLOR
A key color is specified in the **dwValue** member of the structure identified by *lpSetVideo*. The key color is a Windows RGB value.

MCI_DGV_SETVIDEO_KEY_INDEX
A key index value is specified in the **dwValue** member of the structure identified by *lpSetVideo*. The index parameter is a physical palette index.

MCI_DGV_SETVIDEO_PALHANDLE
A palette handle is specified in the **dwValue** member of the structure identified by *lpSetVideo*. The palette handle is contained in the low-order word. Digital-video devices should not free the palette passed with this command. Applications should free it after they close the device. This flag is supported only by devices that use palettes. If this specified palette handle is zero, then the default palette is used.

MCI_DGV_SETVIDEO_SHARPNESS
A video sharpness value is specified as a factor in the **dwValue** member of the structure identified by *lpSetVideo*.

MCI_DGV_SETVIDEO_SOURCE
A constant specifying the source of the video input is specified in the **dwValue** member of the structure identified by *lpSetVideo*. The following constants are defined:

MCI_DGV_SETVIDEO_SRC_NTSC

Specifies NTSC.

MCI_DGV_SETVIDEO_SRC_PAL

Specifies PAL.

MCI_DGV_SETVIDEO_SRC_RGB

Specifies RGB.

MCI_DGV_SETVIDEO_SRC_SECAM

Specifies SECAM.

MCI_DGV_SETVIDEO_SRC_SVIDEO

Specifies SVIDEO.

MCI_DGV_SETVIDEO_STREAM

A video stream is specified in the **dwValue** member of the structure identified by *lpSetVideo*. The integer value specifies the video stream played back from the workspace. If the stream is not specified and the file format does not define a default stream, the first physically interleaved video stream is played.

MCI_DGV_SETVIDEO_TINT
A video tint value is specified as a factor in the **dwValue** member of the structure identified by *lpSetVideo*. Typically, this adjustment is modeled after the tint control of many color television sets, with 250 defined as green, 750 defined as red, and 0 (or 1000) defined as blue. The nominal value is always 500.

MCI_DGV_SETVIDEO_OUTPUT
The MCI_DGV_SETVIDEO_BRIGHTNESS, MCI_DGV_SETVIDEO_COLOR, MCI_DGV_SETVIDEO_CONTRAST, MCI_DGV_SETVIDEO_GAMMA, MCI_DGV_SETVIDEO_SHARPNESS, or MCI_DGV_SETVIDEO_TINT flag is modified so that it affects only the displayed signal and not what is recorded. If possible, this is the default when monitoring a file.

MCI_DGV_SETVIDEO_OVER
A transition length parameter is included in the **dwOver** member of the structure identified by *lpSetVideo*. The transition length specifies how long (in the current time format) it should take to make a change. If this flag is not used, the change occurs immediately.

MCI_DGV_SETVIDEO_QUALITY
The **lpstrQuality** member of the structure identified by *lpSetVideo* contains an address of a buffer describing the video quality. A text-string in the buffer specifies the characteristics of the video compression algorithm.

The MCI_DGV_SETVIDEO_ALG flag can be used to select a quality descriptor for the specified algorithm. If this flag is omitted, then the current algorithm is used.

The algorithms and descriptor names available depend on the device. Each device supplies documentation for the available algorithms and a description of the applicable descriptor names. The **MCI_QUALITY** command can define additional descriptor names. All devices support the descriptors "low", "medium", and "high". The default is driver specific.

MCI_DGV_SETVIDEO_RECORD
Specifies whether recording includes or excludes video data. When combined with MCI_SET_ON, video data is recorded. When combined with MCI_SET_OFF, video data is excluded. The default includes video data.

MCI_DGV_SETVIDEO_SRC_NUMBER
A number for the video source is specified in the **dwSourceNumber** member of the structure identified by *lpSetVideo*. If there is more than one input of the type specified by MCI_DGV_SETVIDEO_VALUE, the value selects the input. This flag must always be used with MCI_DGV_SETVIDEO_SOURCE. If MCI_DGV_SETVIDEO_VALUE is omitted, however, the specified source number indicates the absolute source to use as specified in the **MCI_LIST** command.

MCI_DGV_SETVIDEO_STILL
The algorithm name or quality value specified applies to still images.

Every device driver must support an algorithm of "none", which means no compression. This is the default. In this case, digital-video devices save still images as RGB format device-independent bitmaps (DIBs).

MCI_DGV_SETVIDEO_VALUE
A value is included in the **dwValue** member of the structure identified by *lpSetVideo*. The meaning of the value is specified by the MCI_DGV_SETVIDEO_ITEM flag.

MCI_SET_OFF
Disables video output. For digital-video devices, disabling video sets the pixels in the destination rectangle defined by the **MCI_PUT** command (or its default, the client region of the current window) to a solid color, but it has no effect on the frame buffer. You can hide the window with the

MCI_WINDOW command if desired. The source of video, whether it's the workspace or an external input, might continue to store new images in the frame buffer, but they are not displayed until the video is enabled. While applications should use the **MCI_SETVIDEO** command to control this function, digital-video devices must still support this flag. The default value after an open is on.

MCI_SET_ON
Enables video output.

For digital-video devices, the *lpSetVideo* parameter points to an **MCI_DGV_SETVIDEO_PARMS** structure.

VCR Flags

The following additional flags are used with the **vcr** device type:

MCI_VCR_SETVIDEO_RECORD
Sets the video recording to on or off. Used in conjunction with one of following flags:

MCI_SET_ON
Video recording on.

MCI_SET_OFF
Video recording off. It might be necessary to first turn off the assemble recording (using the **MCI_SET** command with the MCI_VCR_SET_ASSEMBLE_RECORD flag set to off) before the video recording can be turned off.

MCI_TRACK
The **dwTrack** member of the structure identified by *lpSetVideo* specifies which track is affected by the command.

MCI_VCR_SETVIDEO_SOURCE
Sets the video source, and must be used with the MCI_VCR_SETVIDEO_TO flag.

MCI_VCR_SETVIDEO_MONITOR
Sets the video source monitor, and must be used with the MCI_VCR_SETVIDEO_TO flag.

MCI_VCR_SETVIDEO_TO
The **dwTo** member of the structure identified by *lpSetVideo* contains one of the following constants:

MCI_VCR_SRC_TYPE_TUNER
MCI_VCR_SRC_TYPE_LINE
MCI_VCR_SRC_TYPE_AUX
MCI_VCR_SRC_TYPE_GENERIC
MCI_VCR_SRC_TYPE_MUTE
MCI_VCR_SRC_TYPE_OUTPUT
MCI_VCR_SRC_TYPE_RGB

MCI_VCR_SETVIDEO_NUMBER
The **dwNumber** member of the structure identified by *lpSetVideo* contains the video input (of the type specified in the **dwTo** member) to use.

For VCR devices, the *lpSetVideo* parameter points to an **MCI_VCR_SETVIDEO_PARMS** structure.

MCI_SIGNAL

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_SIGNAL,
    DWORD dwFlags, (DWORD) (LPMCI_DGV_SIGNAL_PARMS) lpSignal);
```

Sets a specified position in the workspace. Digital-video devices recognize this command. MCIAVI supports only one active signal at a time.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpSignal
Address of an **MCI_DGV_SIGNAL_PARMS** structure.

Additional Flags

The following flags apply to digital-video devices:

MCI_DGV_SIGNAL_AT
A signal position is included in the **dwPosition** member of the structure identified by *lpSignal*.

MCI_DGV_SIGNAL_CANCEL
Removes the signal position specified by the value associated with MCI_DGV_SIGNAL_USERVAL.

MCI_DGV_SIGNAL_EVERY
A signal-period value is included in the **dwEvery** member of the structure identified by *lpSignal*.

MCI_DGV_SIGNAL_POSITION
The device will send the position value with the Windows message instead of the user-specified value.

MCI_DGV_SIGNAL_USERVAL
A data value is included in the **dwUserParm** member of the structure identified by *lpSignal*. The data value associated with this request is reported back with the Windows message.

MCI_SPIN

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_SPIN,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpSpin);
```

Starts the device spinning up or down. Videodisc devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY or MCI_WAIT. For information about these flags, see Chapter 3, "MCI Overview."

lpSpin
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Additional Flags

The following additional flags apply to videodisc devices:

MCI_VD_SPIN_DOWN
Stops the disc spinning.

MCI_VD_SPIN_UP
Starts the disc spinning.

MCI_STATUS

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_STATUS,
    DWORD dwFlags, (DWORD) (LPMCI_STATUS_PARMS) lpStatus);
```

Retrieves information about an MCI device. All devices recognize this command. Information is returned in the **dwReturn** member of the structure identified by the *lpStatus* parameter.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and VCR devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpStatus
Address of an **MCI_STATUS_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Additional Flags

The following additional standard and command-specific flags apply to all devices supporting **MCI_STATUS**:

MCI_STATUS_ITEM
Specifies that the **dwItem** member of the 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** member of the structure:

MCI_STATUS_CURRENT_TRACK
The **dwReturn** member is set to the current track number. MCI uses continuous track numbers.

MCI_STATUS_LENGTH
The **dwReturn** member is set to the total media length.

MCI_STATUS_MODE
The **dwReturn** member 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** member is set to the total number of playable tracks.

MCI_STATUS_POSITION
The **dwReturn** member is set to the current position.

MCI_STATUS_READY
The **dwReturn** member is set to TRUE if the device is ready; it is set to FALSE otherwise.

MCI_STATUS_TIME_FORMAT
The **dwReturn** member 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** member of the structure identified by *lpStatus* to MCI_STATUS_POSITION.

MCI_TRACK
Indicates a status track parameter is included in the **dwTrack** member of the 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.

Animation Flags

The following additional flags are used with the **animation** device type. These constants are used in the **dwItem** member of the structure pointed to by the *lpStatus* parameter when MCI_STATUS_ITEM is specified for the *dwFlags* parameter.

MCI_ANIM_STATUS_FORWARD
The **dwReturn** member is set to TRUE if playing forward; it is set to FALSE otherwise.

MCI_ANIM_STATUS_HPAL
The **dwReturn** member is set to the handle of the movie palette.

MCI_ANIM_STATUS_HWND
The **dwReturn** member is set to the handle of the playback window.

MCI_ANIM_STATUS_SPEED
The **dwReturn** member is set to the animation speed.

MCI_ANIM_STATUS_STRETCH
The **dwReturn** member is set to TRUE if stretching is enabled; it is set to FALSE otherwise.

MCI_STATUS_MEDIA_PRESENT
The **dwReturn** member is set to TRUE if the media is inserted in the device; it is set to FALSE otherwise.

CD Audio Flags

The following additional flags are used with the **cdaudio** device type. These constants are used in the **dwItem** member of the structure pointed to by the *lpStatus* parameter when MCI_STATUS_ITEM is specified for the *dwFlags* parameter.

MCI_CDA_STATUS_TYPE_TRACK
The **dwReturn** member is set to one of the following values:

MCI_CDA_TRACK_AUDIO
MCI_CDA_TRACK_OTHER

To use this flag, the MCI_TRACK flag must be set, and the **dwTrack** member of the structure identified by *lpStatus* must contain a valid track number.

MCI_STATUS_MEDIA_PRESENT
The **dwReturn** member is set to TRUE if the media is inserted in the device; it is set to FALSE otherwise.

Digital-Video Flags

The following additional flags are used with the **digitalvideo** device type:

MCI_DGV_STATUS_DISKSPACE
The **lpstrDrive** member of the structure identified by *lpStatus* specifies a disk drive or, in some

implementations, a path. The **MCI_STATUS** command returns the approximate amount of disk space that could be obtained by the **MCI_RESERVE** command in the **dwReturn** member of the structure identified by *lpStatus*. The disk space is measured in units of the current time format.

MCI_DGV_STATUS_INPUT
The constant specified by the **dwItem** member of the structure identified by *lpStatus* applies to the input.

MCI_DGV_STATUS_LEFT
The constant specified by the **dwItem** member of the structure identified by *lpStatus* applies to the left audio channel.

MCI_DGV_STATUS_NOMINAL
The constant specified by the **dwItem** member of the structure identified by *lpStatus* requests the nominal value rather than the current value.

MCI_DGV_STATUS_OUTPUT
The constant specified by the **dwItem** member of the structure identified by *lpStatus* applies to the output.

MCI_DGV_STATUS_RECORD
The frame rate returned for the MCI_DGV_STATUS_FRAME_RATE flag is the rate used for compression.

MCI_DGV_STATUS_REFERENCE
The **dwReturn** member of the structure identified by *lpStatus* returns the nearest key-frame image that precedes the frame specified in the **dwReference** member.

MCI_DGV_STATUS_RIGHT
The constant specified by the **dwItem** member of the structure identified by *lpStatus* applies to the right audio channel.

The following constants are used with the **digitalvideo** device type in the **dwItem** member of the structure pointed to by the *lpStatus* parameter when MCI_STATUS_ITEM is specified for the *dwFlags* parameter.

MCI_AVI_STATUS_AUDIO_BREAKS
The **dwReturn** member returns the number of times the audio portion of the last AVI sequence broke up. The system counts an audio break whenever it attempts to write audio data to the device driver and discovers that the driver has already played all of the available data. This flag is recognized only by the MCIAVI digital-video driver.

MCI_AVI_STATUS_FRAMES_SKIPPED
The **dwReturn** member returns the number of frames that were not drawn when the last AVI sequence was played. This flag is recognized only by the MCIAVI digital-video driver.

MCI_AVI_STATUS_LAST_PLAY_SPEED
The **dwReturn** member returns a value representing how closely the actual playing time of the last AVI sequence matched the target playing time. The value 1000 indicates that the target time and the actual time were the same. A value of 2000, for example, would indicate that the AVI sequence took twice as long to play as it should have. This flag is recognized only by the MCIAVI digital-video driver.

MCI_DGV_STATUS_AUDIO
The **dwReturn** member returns MCI_ON or MCI_OFF depending on the most recent MCI_SET_AUDIO option for the **MCI_SET** command. It returns MCI_ON if either or both speakers are enabled, and MCI_OFF otherwise.

MCI_DGV_STATUS_AUDIO_INPUT
The **dwReturn** member returns the approximate instantaneous audio level of the analog audio signal. A value greater than 1000 implies there is clipping distortion. Some devices can determine this value only while recording audio. This status value has no associated **MCI_SET** or **MCI_SETAUDIO** command. This value is related to, but normalized differently from, the waveform-audio command MCI_WAVE_STATUS_LEVEL.

MCI_DGV_STATUS_AUDIO_RECORD
The **dwReturn** member returns MCI_ON or MCI_OFF reflecting the state set by the MCI_DGV_SETAUDIO_RECORD flag of the **MCI_SETAUDIO** command.

MCI_DGV_STATUS_AUDIO_SOURCE
The **dwReturn** member returns the current audio digitizer source:

MCI_DGV_SETAUDIO_AVERAGE
Specifies the average of the left and right audio channels.

MCI_DGV_SETAUDIO_LEFT
Specifies the left audio channel.

MCI_DGV_SETAUDIO_RIGHT
Specifies the right audio channel.

MCI_DGV_SETAUDIO_STEREO
Specifies stereo.

MCI_DGV_STATUS_AUDIO_STREAM
The **dwReturn** member returns the current audio-stream number.

MCI_DGV_STATUS_AVGBYTESPERSEC
The **dwReturn** member returns the average number of bytes per second used for recording.

MCI_DGV_STATUS_BASS
The **dwReturn** member returns the current audio bass level. Use MCI_DGV_STATUS_NOMINAL with this flag to obtain the nominal level.

MCI_DGV_STATUS_BITSPERPEL
The **dwReturn** member returns the number of bits per pixel used for saving captured or recorded data.

MCI_DGV_STATUS_BITSPERSAMPLE
The **dwReturn** member returns the number of bits per sample the device uses for recording. This applies only to devices supporting the PCM format.

MCI_DGV_STATUS_BLOCKALIGN
The **dwReturn** member returns the alignment of data blocks relative to the start of the input waveform.

MCI_DGV_STATUS_BRIGHTNESS
The **dwReturn** member returns the current video brightness level. Use MCI_DGV_STATUS_NOMINAL with this flag to obtain the nominal level.

MCI_DGV_STATUS_COLOR
The **dwReturn** member returns the current color level. Use MCI_DGV_STATUS_NOMINAL with this flag to obtain the nominal level.

MCI_DGV_STATUS_CONTRAST
The **dwReturn** member returns the current contrast level. Use MCI_DGV_STATUS_NOMINAL with this flag to obtain the nominal level.

MCI_DGV_STATUS_FILEFORMAT
The **dwReturn** member returns the current file format for recording or saving.

MCI_DGV_STATUS_FILE_MODE
The **dwReturn** member returns the state of the file operation:

MCI_DGV_FILE_MODE_EDITING
Returned during cut, copy, delete, paste, and undo operations.

MCI_DGV_FILE_MODE_IDLE
Returned when the file is ready for the next operation.

MCI_DGV_FILE_MODE_LOADING
Returned while the file is being loaded.

MCI_DGV_FILE_MODE_SAVING
Returned while the file is being saved.

MCI_DGV_STATUS_FILE_COMPLETION
The **dwReturn** member returns the estimated percentage a load, save, capture, cut, copy, delete, paste, or undo operation has progressed. (Applications can use this to provide a visual indicator of progress.) This flag is not supported by all digital-video devices.

MCI_DGV_STATUS_FORWARD
The **dwReturn** member returns TRUE if the device direction is forward or the device is not playing.

MCI_DGV_STATUS_FRAME_RATE
The **dwReturn** member must be used with MCI_DGV_STATUS_NOMINAL, MCI_DGV_STATUS_RECORD, or both. When used with MCI_DGV_STATUS_RECORD, the current frame rate used for recording is returned. When used with both MCI_DGV_STATUS_RECORD and MCI_DGV_STATUS_NOMINAL, the nominal frame rate associated with the input video signal is returned. When used with MCI_DGV_STATUS_NOMINAL, the nominal frame rate associated with the file is returned. In all cases the units are in frames per second multiplied by 1000.

MCI_DGV_STATUS_GAMMA
The **dwReturn** member returns the current gamma value. Use MCI_DGV_STATUS_NOMINAL with this flag to obtain the nominal level.

MCI_DGV_STATUS_HPAL
The **dwReturn** member returns the ASCII decimal value for the current palette handle. The handle is contained in the low-order word of the returned value.

MCI_DGV_STATUS_HWND
The **dwReturn** member returns the ASCII decimal value for the current explicit or default window handle associated with this device driver instance. The handle is contained in the low-order word of the returned value.

MCI_DGV_STATUS_KEY_COLOR
The **dwReturn** member returns the current key-color value.

MCI_DGV_STATUS_KEY_INDEX
The **dwReturn** member returns the current key-index value.

MCI_DGV_STATUS_MONITOR
The **dwReturn** member returns a constant indicating the source of the current presentation. The following constants are defined:

MCI_DGV_MONITOR_FILE
A file is the source.

MCI_DGV_MONITOR_INPUT
The input is the source.

MCI_DGV_STATUS_MONITOR_METHOD
The **dwReturn** member returns a constant indicating the method used for input monitoring. The following constants are defined:

MCI_DGV_METHOD_DIRECT
Direct input monitoring.

MCI_DGV_METHOD_POST
Post-input monitoring.

MCI_DGV_METHOD_PRE
Pre-input monitoring.

MCI_DGV_STATUS_PAUSE_MODE
The **dwReturn** member returns MCI_MODE_PLAY if the device was paused while playing and returns MCI_MODE_RECORD if the device was paused while recording. The command returns MCIERR_NONAPPLICABLE_FUNCTION as an error return if the device is not paused.

MCI_DGV_STATUS_SAMPLESPERSECOND
The **dwReturn** member returns the number of samples per second recorded.

MCI_DGV_STATUS_SEEK_EXACTLY

The **dwReturn** member returns TRUE or FALSE indicating whether or not the seek exactly format is set. (Applications can set this format by using the **MCI_SET** command with the MCI_DGV_SET_SEEK_EXACTLY flag.)

MCI_DGV_STATUS_SHARPNESS
The **dwReturn** member returns the current sharpness level. Use MCI_DGV_STATUS_NOMINAL with this flag to obtain the nominal level.

MCI_DGV_STATUS_SIZE
The **dwReturn** member returns the approximate playback duration of compressed data that the reserved workspace will hold. The duration units are in the current time format. It returns zero if there is no reserved disk space. The size returned is approximate since the precise disk space for compressed data cannot, in general, be predicted until after the data has been compressed.

MCI_DGV_STATUS_SMPTE
The **dwReturn** member returns the SMPTE time code associated with the current position in the workspace.

MCI_DGV_STATUS_SPEED
The **dwReturn** member returns the current playback speed.

MCI_DGV_STATUS_STILL_FILEFORMAT
The **dwReturn** member returns the current file format for the **MCI_CAPTURE** command.

MCI_DGV_STATUS_TINT
The **dwReturn** member returns the current video tint level. Use MCI_DGV_STATUS_NOMINAL with this flag to obtain the nominal level.

MCI_DGV_STATUS_TREBLE
The **dwReturn** member returns the current audio treble level. Use MCI_DGV_STATUS_NOMINAL with this flag to obtain the nominal level.

MCI_DGV_STATUS_UNSAVED
The **dwReturn** member returns TRUE if there is recorded data in the workspace that might be lost as a result of a **MCI_CLOSE**, **MCI_LOAD**, **MCI_RECORD**, **MCI_RESERVE**, **MCI_CUT**, **MCI_DELETE**, or **MCI_PASTE** command. The member returns FALSE otherwise.

MCI_DGV_STATUS_VIDEO
The **dwReturn** member returns MCI_ON if video is enabled or MCI_OFF if it is disabled.

MCI_DGV_STATUS_VIDEO_RECORD
The **dwReturn** member returns MCI_ON or MCI_OFF, reflecting the state set by the MCI_DGV_SETVIDEO_RECORD flag of the **MCI_SETVIDEO** command.

MCI_DGV_STATUS_VIDEO_SOURCE
The **dwReturn** member returns a constant indicating the type of video source set by the MCI_DGV_SETVIDEO_SOURCE flag of the **MCI_SETVIDEO** command.

MCI_DGV_STATUS_VIDEO_SRC_NUM
The **dwReturn** member returns the number within its type of the video-input source currently active.

MCI_DGV_STATUS_VIDEO_STREAM
The **dwReturn** member returns the current video-stream number.

MCI_DGV_STATUS_VOLUME
The **dwReturn** member returns the average of the volume to the left and right speakers. Use MCI_DGV_STATUS_NOMINAL with this flag to obtain the nominal level.

MCI_DGV_STATUS_WINDOW_VISIBLE
The **dwReturn** member returns TRUE if the window is not hidden.

MCI_DGV_STATUS_WINDOW_MINIMIZED
The **dwReturn** member returns TRUE if the window is minimized.

MCI_DGV_STATUS_WINDOW_MAXIMIZED
The **dwReturn** member returns TRUE if the window is maximized.

MCI_STATUS_MEDIA_PRESENT

The **dwReturn** member returns TRUE.

For digital-video devices, the *lpStatus* parameter points to an **MCI_DGV_STATUS_PARMS** structure.

Sequencer Flags

The following additional flags are used with the **sequencer** device type. These constants are used in the **dwItem** member of the structure pointed to by the *lpStatus* parameter when MCI_STATUS_ITEM is specified for the *dwFlags* parameter.

MCI_SEQ_STATUS_DIVTYPE
The **dwReturn** member 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** member is set to the synchronization type used for master operation.

MCI_SEQ_STATUS_OFFSET
The **dwReturn** member is set to the current SMPTE offset of a sequence.

MCI_SEQ_STATUS_PORT
The **dwReturn** member is set to the MIDI device identifier for the current port used by the sequence.

MCI_SEQ_STATUS_SLAVE
The **dwReturn** member is set to the synchronization type used for slave operation.

MCI_SEQ_STATUS_TEMPO
The **dwReturn** member 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** member is set to TRUE if the media is inserted in the device; it is set to FALSE otherwise.

VCR Flags

The following additional flags are used with the **vcr** device type. These constants are used in the **dwItem** member of the structure pointed to by the *lpStatus* parameter when MCI_STATUS_ITEM is specified for the *dwFlags* parameter.

MCI_STATUS_MEDIA_PRESENT
The **dwReturn** member is set to TRUE if the media is inserted in the device; it is set to FALSE otherwise.

MCI_VCR_STATUS_ASSEMBLE_RECORD
The **dwReturn** member is set to TRUE if assemble mode is on; it is set to FALSE otherwise.

MCI_VCR_STATUS_AUDIO_MONITOR
The **dwReturn** member is set to a constant, indicating the currently selected audio-monitor type.

MCI_VCR_STATUS_AUDIO_MONITOR_NUMBER
The **dwReturn** member is set to the number of the currently selected audio-monitor type.

MCI_VCR_STATUS_AUDIO_RECORD
The **dwReturn** member is set to TRUE if audio will be recorded when the next record command is given; it is set to FALSE otherwise. If you specify MCI_TRACK in the *dwFlags* parameter of this command, **dwTrack** contains the track this inquiry applies to.

MCI_VCR_STATUS_AUDIO_SOURCE

The **dwReturn** member is set to a constant, indicating the current audio-source type.

MCI_VCR_STATUS_AUDIO_SOURCE_NUMBER
The **dwReturn** member is set to the number of the currently selected audio-source type.

MCI_VCR_STATUS_CLOCK
The **dwReturn** member is set to the current clock value, in total clock increments.

MCI_VCR_STATUS_CLOCK_ID
The **dwReturn** member is set to a number which uniquely describes the clock in use.

MCI_VCR_STATUS_COUNTER_FORMAT
The **dwReturn** member is set to a constant describing the current counter format. For more information, see the MCI_SET_TIME_FORMAT flag of the **MCI_SET** command.

MCI_VCR_STATUS_COUNTER_RESOLUTION
The **dwReturn** member is set to a constant describing the resolution of the counter, and is one of the following values:

MCI_VCR_COUNTER_RES_FRAMES
Counter has resolution of frames.

MCI_VCR_COUNTER_RES_SECONDS
Counter has resolution of seconds.

MCI_VCR_STATUS_COUNTER_VALUE
The **dwReturn** member is set to the current counter reading, in the current counter-time format.

MCI_VCR_STATUS_FRAME_RATE
The **dwReturn** member is set to the current native frame rate of the device.

MCI_VCR_STATUS_INDEX
The **dwReturn** member is set to a constant, describing the current contents of the on-screen display, and is one of the following:

MCI_VCR_INDEX_COUNTER

MCI_VCR_INDEX_DATE

MCI_VCR_INDEX_TIME

MCI_VCR_INDEX_TIMECODE

MCI_VCR_STATUS_INDEX_ON
The **dwReturn** member is set to TRUE if the on-screen display is on; it is set to FALSE otherwise.

MCI_VCR_STATUS_MEDIA_TYPE
The **dwReturn** member is set to one of the following:

MCI_VCR_MEDIA_8MM

MCI_VCR_MEDIA_HI8

MCI_VCR_MEDIA_VHS

MCI_VCR_MEDIA_SVHS

MCI_VCR_MEDIA_BETA

MCI_VCR_MEDIA_EDBETA

MCI_VCR_MEDIA_OTHER

MCI_VCR_STATUS_NUMBER
The **dwNumber** member is set to the logical-tuner number when you use this flag with the MCI_VCR_STATUS_TUNER_CHANNEL flag.

MCI_VCR_STATUS_NUMBER_OF_AUDIO_TRACKS
The **dwReturn** member is set to the number of audio tracks that are independently selectable.

MCI_VCR_STATUS_NUMBER_OF_VIDEO_TRACKS
The **dwReturn** member is set to the number of video tracks that are independently selectable.

MCI_VCR_STATUS_PAUSE_TIMEOUT
The **dwReturn** member is set to the maximum duration, in milliseconds, of a pause command. The

return value of zero indicates that no time-out will occur.

MCI_VCR_STATUS_PLAY_FORMAT
The **dwReturn** member is set to one of the following:
MCI_VCR_FORMAT_EP
MCI_VCR_FORMAT_LP
MCI_VCR_FORMAT_OTHER
MCI_VCR_FORMAT_SP

MCI_VCR_STATUS_POSTROLL_DURATION
The **dwReturn** member is set to the length of the videotape that will play after the spot at which it was stopped, in the current time format. This is needed to brake the VCR tape transport from a stop or pause command.

MCI_VCR_STATUS_POWER_ON
The **dwReturn** member is set to TRUE if the power is on; it is set to FALSE otherwise.

MCI_VCR_STATUS_PREROLL_DURATION
The **dwReturn** member is set to the length of the videotape that will play before the spot at which it was started, in the current time format. This is needed to stabilize the VCR output.

MCI_VCR_STATUS_RECORD_FORMAT
The **dwReturn** member is set to one of the following:
MCI_VCR_FORMAT_EP
MCI_VCR_FORMAT_LP
MCI_VCR_FORMAT_OTHER
MCI_VCR_FORMAT_SP

MCI_VCR_STATUS_SPEED
The **dwReturn** member is set to the current speed. For more information, see the MCI_VCR_SET_SPEED flag of the **MCI_SET** command.

MCI_VCR_STATUS_TIME_MODE
The **dwReturn** member is set to one of the following:
MCI_VCR_TIME_COUNTER
MCI_VCR_TIME_DETECT
MCI_VCR_TIME_TIMECODE
For more information, see the MCI_VCR_SET_TIME_MODE flag of the **MCI_SET** command.

MCI_VCR_STATUS_TIME_TYPE
The **dwReturn** member is set to a constant describing the current time type in use (used by **play**, **record**, **seek**, and so on), and is one of the following:
MCI_VCR_TIME_COUNTER
Counter is in use.
MCI_VCR_TIME_TIMECODE
Timecode is in use.

MCI_VCR_STATUS_TIMECODE_PRESENT
The **dwReturn** member is set to TRUE if timecode is present at the current position in the content; it is set to FALSE otherwise.

MCI_VCR_STATUS_TIMECODE_RECORD
The **dwReturn** member is set to TRUE if the timecode will be recorded when the next record command is given; it is set to FALSE otherwise.

MCI_VCR_STATUS_TIMECODE_TYPE
The **dwReturn** member is set to a constant, describing the type of timecode that is directly supported by the device, and is one of the following:
MCI_VCR_TIMECODE_TYPE_NONE
This device does not use a timecode.

MCI_VCR_TIMECODE_TYPE_OTHER
This device uses an unspecified timecode.

MCI_VCR_TIMECODE_TYPE_SMPTE
This device uses SMPTE timecode.

MCI_VCR_TIMECODE_TYPE_SMPTE_DROP
This device uses SMPTE drop timecode.

MCI_VCR_STATUS_TUNER_CHANNEL
The **dwReturn** member is set to the current channel number. If you specify MCI_VCR_STATUS_NUMBER in the *dwFlags* parameter of this command, **dwNumber** contains the logical-tuner number this command applies to.

MCI_VCR_STATUS_VIDEO_MONITOR
The **dwReturn** member is set to a constant, indicating the currently selected video-monitor type.

MCI_VCR_STATUS_VIDEO_MONITOR_NUMBER
The **dwReturn** member is set to the number of the currently selected video-monitor type.

MCI_VCR_STATUS_VIDEO_RECORD
The **dwReturn** member is set to TRUE if video will be recorded when the next record command is given; it is set to FALSE otherwise. If you specify MCI_TRACK in the *dwFlags* parameter of this command, **dwTrack** contains the track this inquiry applies to.

MCI_VCR_STATUS_VIDEO_SOURCE
The **dwReturn** member is set to a constant indicating the currently selected video-source type.

MCI_VCR_STATUS_VIDEO_SOURCE_NUMBER
The **dwReturn** member is set to the number of the currently selected video-source type.

MCI_VCR_STATUS_WRITE_PROTECTED
The **dwReturn** member is set to TRUE if the media is write-protected; it is set to FALSE otherwise.

For VCR devices, the *lpStatus* parameter points to an **MCI_VCR_STATUS_PARMS** structure.

Using the MCI_STATUS_LENGTH flag to determine the length of the media always returns 2 hours for VCR devices, unless the length has been explicitly changed using the **MCI_SET** command.

Video-Overlay Flags

The following additional flags are used with the **overlay** device type. These constants are used in the **dwItem** member of the structure pointed to by the *lpStatus* parameter when MCI_STATUS_ITEM is specified for the *dwFlags* parameter.

MCI_OVLY_STATUS_HWND
The **dwReturn** member is set to the handle of the window associated with the video-overlay device.

MCI_OVLY_STATUS_STRETCH
The **dwReturn** member is set to TRUE if stretching is enabled; it is set to FALSE otherwise.

MCI_STATUS_MEDIA_PRESENT
The **dwReturn** member is set to TRUE if the media is inserted in the device; it is set to FALSE otherwise.

Videodisc Flags

The following additional flags are used with the **videodisc** device type. These constants are used in the **dwItem** member of the structure pointed to by the *lpStatus* parameter when MCI_STATUS_ITEM is specified for the *dwFlags* parameter.

MCI_STATUS_MEDIA_PRESENT
The **dwReturn** member is set to TRUE if the media is inserted in the device; it is set to FALSE otherwise.

MCI_STATUS_MODE
The **dwReturn** member is set to the current mode of the device. Videodisc devices can return the

MCI_VD_MODE_PARK constant, in addition to the constants any device can return, as documented with the *dwFlags* parameter.

MCI_VD_STATUS_DISC_SIZE
The **dwReturn** member is set to the size of the loaded disc in inches (8 or 12).

MCI_VD_STATUS_FORWARD
The **dwReturn** member is set to TRUE if playing forward; it is set to FALSE otherwise.

The MCI videodisc device does not support this flag.

MCI_VD_STATUS_MEDIA_TYPE
The **dwReturn** member 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_VD_STATUS_SIDE
The **dwReturn** member 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** member is set to the play speed in frames per second. The MCIPIONR.DRV device driver returns MCIERR_UNSUPPORTED_FUNCTION.

Waveform-Audio Flags

The following additional flags are used with the **waveaudio** device type. These constants are used in the **dwItem** member of the structure pointed to by the *lpStatus* parameter when MCI_STATUS_ITEM is specified for the *dwFlags* parameter.

MCI_WAVE_FORMATTAG
The **dwReturn** member is set to the current format tag used for playing, recording, and saving.

MCI_WAVE_INPUT
The **dwReturn** member 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 MCIERR_WAVE_INPUTUNSPECIFIED.

MCI_WAVE_OUTPUT
The **dwReturn** member 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 MCIERR_WAVE_OUTPUTUNSPECIFIED.

MCI_WAVE_STATUS_AVGBYTESPERSEC
The **dwReturn** member is set to the current bytes per second used for playing, recording, and saving.

MCI_WAVE_STATUS_BITSPERSAMPLE
The **dwReturn** member is set to the current bits per sample used for playing, recording, and saving PCM formatted data.

MCI_WAVE_STATUS_BLOCKALIGN
The **dwReturn** member is set to the current block alignment used for playing, recording, and saving.

MCI_WAVE_STATUS_CHANNELS
The **dwReturn** member is set to the current channel count used for playing, recording, and saving.

MCI_WAVE_STATUS_LEVEL
The **dwReturn** member is set to the current record or playback level of PCM formatted data. 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** member is set to the current samples per second used for playing, recording, and saving.

MCI_STEP

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_STEP,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpStep);
```

Steps the player one or more frames. Animation, digital-video, VCR, and CAV-format videodisc devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and VCR devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpStep
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

This command supports devices that return TRUE to the MCI_GETDEVCAPS_HAS_VIDEO flag of the **MCI_GETDEVCAPS** command.

Animation Flags

The following additional flags are used with the **animation** device type:

MCI_ANIM_STEP_FRAMES
The **dwFrames** member of the structure identified by *lpStep* specifies the number of frames to step.

MCI_ANIM_STEP_REVERSE
Steps in reverse.

For animation devices, the *lpStep* parameter points to an **MCI_ANIM_STEP_PARMS** structure.

Digital-Video Flags

The following additional flags are used with the **digitalvideo** device type:

MCI_DGV_STEP_FRAMES
The **dwFrames** member of the structure identified by *lpStep* specifies the number of frames to advance before displaying another image.

MCI_DGV_STEP_REVERSE
Steps in reverse.

For digital-video devices, the *lpStep* parameter points to an **MCI_DGV_STEP_PARMS** structure.

VCR Flags

The following additional flags are used with the **vcr** device type:

MCI_VCR_STEP_FRAMES
The **dwFrames** member of the structure identified by *lpStep* specifies the number of frames to advance before displaying another image.

MCI_VCR_STEP_REVERSE
Steps in reverse.

For VCR devices, the *lpStep* parameter points to an **MCI_VCR_STEP_PARMS** structure.

Videodisc Flags

The following additional flags are used with the **videodisc** device type:

MCI_VD_STEP_FRAMES

The **dwFrames** member of the structure identified by *lpStep* specifies the number of frames to step.

MCI_VD_STEP_REVERSE

Steps in reverse.

For videodisc devices, the *lpStep* parameter points to an **MCI_VD_STEP_PARMS** structure.

MCI_STOP

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_STOP,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpStop);
```

Stops all play and record sequences, unloads all play buffers, and ceases display of video images. Animation, CD audio, digital-video, MIDI sequencer, videodisc, VCR, and waveform-audio devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and VCR devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpStop
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

The difference between the **MCI_STOP** and **MCI_PAUSE** commands depends on the device. If possible, **MCI_PAUSE** suspends device operation but leaves the device ready to resume play immediately.

For the CD audio device, **MCI_STOP** resets the current track position to zero; in contrast, **MCI_PAUSE** maintains the current track position, anticipating that the device will resume playing.

MCI_SYSINFO

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_SYSINFO,
    DWORD dwFlags, (DWORD) (LPMCI_SYSINFO_PARMS) lpSysInfo);
```

Retrieves information about MCI devices. MCI supports this command directly rather than passing it to the device. Any MCI application can use this command. String information is returned in the application-supplied buffer pointed to by the **lpstrReturn** member of the structure identified by *lpSysInfo*. Numeric information is returned as a doubleword value placed in the application-supplied buffer. The **dwRetSize** member specifies the buffer length.

- Returns zero if successful or an error otherwise.

dwFlags

One or more of the following standard and command-specific flags:

MCI_SYSINFO_INSTALLNAME
: Obtains the name (listed in the registry or 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** member of the 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 registry or the [mci] section of the SYSTEM.INI file. If the MCI_SYSINFO_OPEN flag is set, the number of open devices is returned.

lpSysInfo

Address of an **MCI_SYSINFO_PARMS** structure.

The **wDeviceType** member of the structure identified by *lpSysInfo* is used to indicate the device type of the query. If the *wDeviceID* parameter is set to MCI_ALL_DEVICE_ID, it overrides the value of **wDeviceType**. For a list of device types, see "Constants: Device Types" later in this chapter.

Integer return values are doubleword values returned in the buffer pointed to by the **lpstrReturn** member of the structure identified by *lpSysInfo*.

String return values are null-terminated strings returned in the buffer pointed to by the **lpstrReturn** member of the structure identified by *lpSysInfo*.

MCI_UNDO

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_UNDO,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpUndo);
```

Reverses the most recent successful **MCI_CUT**, **MCI_COPY**, **MCI_DELETE**, or **MCI_PASTE** command. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpUndo
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

MCI_UNFREEZE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_UNFREEZE,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpUnfreeze);
```

Restores motion to an area of the video buffer frozen with the **MCI_FREEZE** command. Digital-video, VCR, and video-overlay devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video and VCR devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpUnfreeze
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Digital-Video Flags

The following additional flag is used with the **digitalvideo** device type:

MCI_DGV_RECT
The **rc** member of the structure identified by *lpUnfreeze* contains a valid display rectangle. The rectangle specifies a region within the frame buffer whose pixels should have their lock mask bit turned off. Rectangular regions are specified as described for the **MCI_PUT** command. If omitted, the rectangle defaults to the entire frame buffer. By using a sequence of freeze and unfreeze commands with different rectangles, arbitrary patterns of lock mask bits can be described.

For digital-video devices, the *lpUnfreeze* parameter points to an **MCI_DGV_UNFREEZE_PARMS** structure. The **MCI_DGV_UNFREEZE_PARMS** structure is identical to the **MCI_DGV_RECT_PARMS** structure.

VCR Flags

The following additional flags are used with the **vcr** device type:

MCI_VCR_UNFREEZE_INPUT
Unfreeze the input.

MCI_VCR_UNFREEZE_OUTPUT
Unfreeze the output.

Video-Overlay Flags

The following additional flag is used with the **overlay** device type:

MCI_OVLY_RECT
The **rc** member of the structure identified by *lpUnfreeze* contains a valid display rectangle. This is a required parameter.

For video-overlay devices, the *lpUnfreeze* parameter points to an **MCI_OVLY_RECT_PARMS** structure.

MCI_UPDATE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_UPDATE,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpDest);
```

Updates the display rectangle. Animation and digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpDest
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Animation Flags

The following additional flags are used with the **animation** device type:

MCI_ANIM_RECT
The **rc** member of the structure identified by *lpDest* contains a valid rectangle. If this flag is not specified, the entire window is updated.

MCI_ANIM_UPDATE_HDC
The **hDC** member of the structure identified by *lpDest* contains a handle to the display context (DC). This flag is required.

For animation devices, the *lpDest* parameter points to an **MCI_ANIM_UPDATE_PARMS** structure.

Digital-Video Flags

The following additional flags are used with the **digitalvideo** device type:

MCI_DGV_UPDATE_HDC
The **hDC** member of the structure identified by *lpDest* contains a valid window of the DC to paint. This flag is required.

MCI_DGV_RECT
The **rc** member of the structure identified by *lpUnfreeze* contains a valid display rectangle. The rectangle specifies the clipping rectangle relative to the client rectangle.

MCI_DGV_UPDATE_PAINT
An application uses this flag when it receives a WM_PAINT message that is intended for a display DC. A frame-buffer device usually paints the key color. If the display device does not have a frame buffer, it might ignore the **MCI_UPDATE** command when the MCI_DGV_UPDATE_PAINT flag is used because the display will be repainted during the playback operation.

For digital-video devices, the *lpDest* parameter points to an **MCI_DGV_UPDATE_PARMS** structure.

MCI_WHERE

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_WHERE,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpQuery);
```

Obtains the clipping rectangle for the video device. Animation, digital-video, and video-overlay devices recognize this command. The **top** and **left** members of the returned **RECT** contain the origin of the clipping rectangle, and the **right** and **bottom** members contain the width and height of the clipping rectangle. (This is not the standard use of the **right** and **bottom** members.)

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpQuery
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Animation Flags

The following additional flags are used with the **animation** device type:

MCI_ANIM_WHERE_DESTINATION
Obtains the destination display rectangle. The rectangle coordinates are placed in the **rc** member of the structure identified by *lpQuery*.

MCI_ANIM_WHERE_SOURCE
Obtains the animation source rectangle. The rectangle coordinates are placed in the **rc** member of the structure identified by *lpQuery*.

For animation devices, the *lpQuery* parameter points to an **MCI_ANIM_RECT_PARMS** structure.

Digital-Video Flags

The following additional flags are used with the **digitalvideo** device type:

MCI_DGV_WHERE_DESTINATION
Obtains a description of the rectangular region used to display video and images in the client area of the current window.

MCI_DGV_WHERE_FRAME
Obtains a description of the rectangular region of the frame buffer into which images from the video rectangle are scaled. The rectangle coordinates are placed in the **rc** member of the structure identified by *lpQuery*.

MCI_DGV_WHERE_MAX
When used with MCI_DGV_WHERE_DESTINATION or MCI_DGV_WHERE_SOURCE, the rectangle returned indicates the maximum width and height of the specified region. When used with MCI_DGV_WHERE_WINDOW, the rectangle returned indicates the size of the entire display.

MCI_DGV_WHERE_SOURCE
Obtains a description of the rectangular region (cropped from the frame buffer) that is stretched to fit the destination rectangle on the display.

MCI_DGV_WHERE_VIDEO
Obtains a description of the rectangular region cropped from the presentation source to fill the frame rectangle in the frame buffer. The rectangle coordinates are placed in the **rc** member of the structure identified by *lpQuery*.

MCI_DGV_WHERE_WINDOW
Obtains a description of the display-window frame.

For digital-video devices, the *lpQuery* parameter points to an **MCI_DGV_WHERE_PARMS** structure. The **MCI_DGV_WHERE_PARMS** structure is identical to the **MCI_DGV_RECT_PARMS** structure.

Video-Overlay Flags

The following additional flags are used with the **overlay** device type:

MCI_OVLY_WHERE_DESTINATION
Obtains the destination display rectangle. The rectangle coordinates are placed in the **rc** member of the structure identified by *lpQuery*.

MCI_OVLY_WHERE_FRAME
Obtains the overlay frame rectangle. The rectangle coordinates are placed in the **rc** member of the structure identified by *lpQuery*.

MCI_OVLY_WHERE_SOURCE
Obtains the source rectangle. The rectangle coordinates are placed in the **rc** member of the structure identified by *lpQuery*.

MCI_OVLY_WHERE_VIDEO
Obtains the video rectangle. The rectangle coordinates are placed in the **rc** member of the structure identified by *lpQuery*.

For video-overlay devices, the *lpQuery* parameter points to an **MCI_OVLY_RECT_PARMS** structure.

MCI_WINDOW

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_WINDOW,
    DWORD dwFlags, (DWORD) (LPMCI_GENERIC_PARMS) lpWindow);
```

Specifies the window and the window characteristics for graphic devices. Animation, digital-video, and video-overlay devices recognize this command.

- Returns zero if successful or an error otherwise.

dwFlags
MCI_NOTIFY, MCI_WAIT, or, for digital-video devices, MCI_TEST. For information about these flags, see Chapter 3, "MCI Overview."

lpWindow
Address of an **MCI_GENERIC_PARMS** structure. (Devices with extended command sets might replace this structure with a device-specific structure.)

Graphic devices should create a default window when a device is opened but should not display it until they receive the **MCI_PLAY** command. The **MCI_WINDOW** command is used to supply an application-created window to the device and to change the display characteristics of an application-defined or default display window. If the application supplies the display window, it should be prepared to update an invalid rectangle on the window.

Animation Flags

The following additional flags are used with the **animation** device type:

MCI_ANIM_WINDOW_DISABLE_STRETCH
Disables stretching of the image.

MCI_ANIM_WINDOW_ENABLE_STRETCH
Enables stretching of the image.

MCI_ANIM_WINDOW_HWND
The handle of the window to use for the destination is included in the **hWnd** member of the structure identified by *lpWindow*. Set this to MCI_ANIM_WINDOW_DEFAULT to return to the default window.

MCI_ANIM_WINDOW_STATE
The **nCmdShow** member of the structure identified by *lpWindow* contains parameters for setting the window state. This flag is equivalent to calling the **ShowWindow** function with a state parameter such as SW_HIDE, SW_MINIMIZE, or SW_SHOWNORMAL.

MCI_ANIM_WINDOW_TEXT
The **lpstrText** member of the structure identified by *lpWindow* contains an address of a buffer containing the caption used for the window.

For animation devices, the *lpWindow* parameter points to an **MCI_ANIM_WINDOW_PARMS** structure.

Digital-Video Flags

The following additional flags are used with the **digitalvideo** device type:

MCI_DGV_WINDOW_HWND
The handle of the window needed for use as the destination is included in the **hWnd** member of the structure identified by *lpWindow*.

MCI_DGV_WINDOW_STATE
The **nCmdShow** member of the structure identified by *lpWindow* contains parameters for setting the window state.

MCI_DGV_WINDOW_TEXT
The **lpstrText** member of the structure identified by *lpWindow* contains an address of a buffer containing the caption used in the window title bar.

For digital-video devices, the *lpWindow* parameter points to an **MCI_DGV_WINDOW_PARMS** structure.

Video-Overlay Flags

The following additional flags are used with the **overlay** device type:

MCI_OVLY_WINDOW_DISABLE_STRETCH
Disables stretching of the image.

MCI_OVLY_WINDOW_ENABLE_STRETCH
Enables stretching of the image.

MCI_OVLY_WINDOW_HWND
The handle of the window used for the destination is included in the **hWnd** member of the structure identified by *lpWindow*. Set this flag to MCI_OVLY_WINDOW_DEFAULT to return to the default window.

MCI_OVLY_WINDOW_STATE
The **nCmdShow** member of the *lpWindow* 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 those defined in WINDOWS.H (such as SW_HIDE, SW_MINIMIZE, or SW_SHOWNORMAL).

MCI_OVLY_WINDOW_TEXT
The **lpstrText** member of the structure identified by *lpWindow* contains an address of a buffer containing the caption used for the window.

For video-overlay devices, the *lpWindow* parameter points to an **MCI_OVLY_WINDOW_PARMS** structure.

MCI_ANIM_OPEN_PARMS

```
typedef struct {
    DWORD  dwCallback;        // see below
    MCIDEVICEID wDeviceID;    // identifier returned to application
    LPCSTR lpstrDeviceType;   // see below
    LPCSTR lpstrElementName;  // device element name (usually a path)
    LPCSTR lpstrAlias;        // optional device alias
    DWORD  dwStyle;           // window style
    HWND   hWndParent;        // handle of parent window
} MCI_ANIM_OPEN_PARMS;
```

Contains window and device information for the **MCI_OPEN** command for animation devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpstrDeviceType
The name or constant identifier of the device type. If this member is a constant, it can be one of the values listed in "Constants: Device Types" later in this chapter.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

You can use the **MCI_OPEN_PARMS** structure instead of **MCI_ANIM_OPEN_PARMS** if you are not using the extended data members.

MCI_ANIM_PLAY_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    DWORD dwFrom;      // position to play from
    DWORD dwTo;        // position to play to
    DWORD dwSpeed;     // play rate, in frames per second
} MCI_ANIM_PLAY_PARMS;
```

Contains play information for the **MCI_PLAY** command for animation devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

You can use the **MCI_PLAY_PARMS** structure instead of **MCI_ANIM_PLAY_PARMS** if you are not using the extended data members.

MCI_ANIM_RECT_PARMS

```
typedef struct {
    DWORD dwCallback;
    RECT  rc;
} MCI_ANIM_RECT_PARMS;
```

Contains positioning information for the **MCI_PUT** and **MCI_WHERE** commands for animation devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Rectangle containing positioning information. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_ANIM_STEP_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    DWORD dwFrames;    // number of frames to step
} MCI_ANIM_STEP_PARMS;
```

Contains information for the **MCI_STEP** command for animation devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_ANIM_UPDATE_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    RECT  rc;          // see below
    HDC   hDC;         // handle to device context (DC)
} MCI_ANIM_UPDATE_PARMS;
```

Contains position and DC information for the **MCI_UPDATE** command for animation devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Rectangle containing positioning information. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_ANIM_WINDOW_PARMS

```
typedef struct {
    DWORD  dwCallback;  // see below
    HWND   hWnd;        // handle to display window
    UINT   nCmdShow;    // window-display command
    LPCSTR lpstrText;   // window caption
} MCI_ANIM_WINDOW_PARMS;
```

Contains display information for the **MCI_WINDOW** command for animation devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_BREAK_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    int   nVirtKey;    // virtual-key code for break key
    HWND  hwndBreak;   // see below
} MCI_BREAK_PARMS;
```

Contains virtual-key code and window information for the **MCI_BREAK** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

hwndBreak
Handle of the window that must be the current window for break detection.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members. The following flags are defined:

MCI_BREAK_HWND
Validates the **hwndBreak** member specifying the window that must have focus to enable break detection.

MCI_BREAK_KEY
Validates the **nVirtKey** member specifying the virtual-key code to be used for the break key.

MCI_BREAK_OFF
Disables any existing break key.

MCI_DGV_CAPTURE_PARMS

```
typedef struct  {
    DWORD dwCallback;
    LPSTR lpstrFileName;
    RECT  rc;
} MCI_DGV_CAPTURE_PARMS;
```

Contains parameters for the **MCI_CAPTURE** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpstrFileName
Address of a null-terminated string specifying the destination path and filename for the file that receives the captured data.

rc
Rectangle containing positioning information. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_CUE_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    DWORD dwTo;        // cue position
} MCI_DGV_CUE_PARMS;
```

Contains parameters for the **MCI_CUE** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_COPY_PARMS

```
typedef struct  {
    DWORD dwCallback;      // see below
    DWORD dwFrom;          // starting position for copy
    DWORD dwTo;            // ending position for copy
    RECT  rc;              // see below
    DWORD dwAudioStream;   // audio stream
    DWORD dwVideoStream;   // video stream
} MCI_DGV_COPY_PARMS;
```

Contains parameters for the **MCI_COPY** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Rectangle describing area to be copied. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_CUT_PARMS

```
typedef struct  {
    DWORD dwCallback;      // see below
    DWORD dwFrom;          // starting position for cut
    DWORD dwTo;            // ending position for cut
    RECT  rc;              // see below
    DWORD dwAudioStream;   // audio stream
    DWORD dwVideoStream;   // video stream
} MCI_DGV_CUT_PARMS;
```

Contains parameters for the **MCI_CUT** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Rectangle describing area to be cut. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_DELETE_PARMS

```
typedef struct  {
    DWORD dwCallback;      // see below
    DWORD dwFrom;          // starting position for delete
    DWORD dwTo;            // ending position for delete
    RECT  rc;              // see below
    DWORD dwAudioStream;   // audio stream
} MCI_DGV_DELETE_PARMS;
```

Contains parameters for the **MCI_DELETE** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Rectangle describing area to delete. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_FREEZE_PARMS

```
typedef struct  {
    DWORD dwCallback;
    RECT  rc;
} MCI_DGV_FREEZE_PARMS;
```

Contains parameters for the **MCI_FREEZE** and **MCI_UNFREEZE** commands for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Rectangle containing positioning information. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_INFO_PARMS

```
typedef struct {
    DWORD  dwCallback;   // see below
    LPSTR  lpstrReturn;  // address of buffer for return string
    DWORD  dwRetSize;    // size, in bytes, of return buffer
    DWORD  dwItem;       // constant describing information to return
} MCI_DGV_INFO_PARMS;
```

Contains parameters for the **MCI_INFO** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_LIST_PARMS

```
typedef struct  {
    DWORD dwCallback;       // see below
    LPSTR lpstrReturn;      // buffer for return string
    DWORD dwLength;         // length, in bytes, of buffer
    DWORD dwNumber;         // index of item in list
    DWORD dwItem;           // type of list item
    LPSTR lpstrAlgorithm;   // string containing algorithm name
} MCI_DGV_LIST_PARMS;
```

Contains the information for the **MCI_LIST** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_LOAD_PARMS

```
typedef struct  {
    DWORD dwCallback;  // see below
    LPSTR lpfilename;  // string naming file to load
} MCI_DGV_LOAD_PARMS;
```

Contains the information for the **MCI_LOAD** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_MONITOR_PARMS

```
typedef struct  {
    DWORD dwCallback;
    DWORD dwSource;
    DWORD dwMethod;
} MCI_DGV_MONITOR_PARMS;
```

Contains parameters for the **MCI_MONITOR** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwSource
One of the following flags for the monitor source:

MCI_DGV_MONITOR_FILE
The workspace is the presentation source. (This is the default source.) If this flag is used during recording, the recording pauses. If the **MCI_MONITOR** command changes the presentation source, recording or playing stops and the current position is the value returned by the **MCI_STATUS** command for the start position.

MCI_DGV_MONITOR_INPUT
The external input is the presentation source. Playback is paused before the input is selected. If the **MCI_SETVIDEO** command has been enabled using the MCI_SET_ON flag, this flag displays a default hidden window. Device drivers might limit what other device instances can do while monitoring input.

dwMethod
One of the following constants for the type of monitoring:

MCI_DGV_METHOD_DIRECT
The device should be configured for optimum display quality during monitoring. Direct monitoring might be incompatible with motion-video recording.

MCI_DGV_METHOD_POST
The device should show the external input after compression. Post monitoring supports motion-video recording.

MCI_DGV_METHOD_PRE
The device should show the external input prior to compression.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_OPEN_PARMS

```
typedef struct {
    DWORD dwCallback;        // see below
    UINT  wDeviceID;         // device ID returned to user
    LPSTR lpstrDeviceType;   // name or constant ID of device type
    LPSTR lpstrElementName;  // device element name (usually a path)
    LPSTR lpstrAlias;        // optional device alias
    DWORD dwStyle;           // window style
    HWND  hWndParent;        // handle of parent window
} MCI_DGV_OPEN_PARMS;
```

Contains information for the **MCI_OPEN** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_PASTE_PARMS

```
typedef struct  {
    DWORD dwCallback;      // see below
    DWORD dwTo;            // starting position for paste
    RECT  rc;              // see below
    DWORD dwAudioStream;   // audio stream
    DWORD dwVideoStream;   // video stream
} MCI_DGV_PASTE_PARMS;
```

Contains parameters for the **MCI_PASTE** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Rectangle containing positioning information. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_PAUSE_PARMS

```
typedef struct {
    DWORD dwCallback;
} MCI_DGV_PAUSE_PARMS;
```

Contains information for the **MCI_PAUSE** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_PLAY_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    DWORD dwFrom;      // position to play from
    DWORD dwTo;        // position to play to
} MCI_DGV_PLAY_PARMS;
```

Contains parameters for the **MCI_PLAY** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_PUT_PARMS

```
typedef struct {
    DWORD dwCallback;
    RECT  rc;
} MCI_DGV_PUT_PARMS;
```

Contains parameters for the **MCI_PUT** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Rectangle containing positioning information. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_QUALITY_PARMS

```
typedef struct  {
    DWORD dwCallback;      // see below
    DWORD dwItem;          // see below
    LPSTR lpstrName;       // string naming descriptor
    DWORD lpstrAlgorithm;  // string naming algorithm
    DWORD dwHandle;        // see below
} MCI_DGV_QUALITY_PARMS;
```

Contains parameters for the **MCI_QUALITY** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwItem
One of the following constants indicating the type of algorithm:

MCI_QUALITY_ITEM_AUDIO
Definitions are for an audio compression algorithm.

MCI_QUALITY_ITEM_STILL
Definitions are for a still video compression algorithm.

MCI_QUALITY_ITEM_VIDEO
Definitions are for a video compression algorithm.

dwHandle
Handle of a structure containing information describing the quality attributes.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_RECORD_PARMS

```
typedef struct  {
    DWORD dwCallback;      // see below
    DWORD dwFrom;          // position to record from
    DWORD dwTo;            // position to record to
    RECT  rc;              // see below
    DWORD dwAudioStream;   // audio stream
    DWORD dwVideoStream;   // video stream
} MCI_DGV_RECORD_PARMS;
```

Contains parameters for the **MCI_RECORD** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
The region of the frame buffer used as the source for the pixels compressed and saved. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_RECT_PARMS

```
typedef struct {
    DWORD dwCallback;
    RECT  rc;
} MCI_DGV_RECT_PARMS;
```

Contains parameters for the **MCI_FREEZE**, **MCI_PUT**, **MCI_UNFREEZE**, and **MCI_WHERE** commands for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Rectangle containing positioning information. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

The **MCI_DGV_UNFREEZE_PARMS** and **MCI_DGV_WHERE_PARMS** structures are identical to the **MCI_DGV_RECT_PARMS** structure.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_RESERVE_PARMS

```
typedef struct  {
    DWORD dwCallback;  // see below
    LPSTR lpstrPath;   // see below
    DWORD dwSize;      // size of reserved disk space
} MCI_DGV_RESERVE_PARMS;
```

Contains information for the **MCI_RESERVE** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpstrPath
Address of a null-terminated string containing the location of a temporary file. The buffer contains only the drive and directory path of the file used to hold recorded data; the filename is specified by the device driver.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_RESTORE_PARMS

```
typedef struct  {
    DWORD dwCallback;
    DWORD lpstrFileName;
    RECT  rc;
} MCI_DGV_RESTORE_PARMS;
```

Contains information for the **MCI_RESTORE** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpstrFileName
Address of a null-terminated string containing the filename from which the frame buffer information will be restored.

rc
Rectangle containing positioning information. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_RESUME_PARMS

```
typedef struct  {
    DWORD dwCallback;
} MCI_DGV_RESUME_PARMS;
```

Contains information for the **MCI_RESUME** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_SAVE_PARMS

```
typedef struct  {
    DWORD dwCallback;      // see below
    DWORD lpstrFileName;   // string for filename to save
    RECT  rc;              // see below
} MCI_DGV_SAVE_PARMS;
```

Contains information for the **MCI_SAVE** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Rectangle containing positioning information. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_SET_PARMS

```
typedef struct {
    DWORD dwCallback;     // see below
    DWORD dwTimeFormat;   // time format of device
    DWORD dwAudio;        // channel for audio output
    DWORD dwFileFormat;   // file format
    DWORD dwSpeed;        // playback speed
} MCI_DGV_SET_PARMS;
```

Contains parameters for the **MCI_SET** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_SETAUDIO_PARMS

```
typedef struct {
    DWORD dwCallback;      // see below
    DWORD dwItem;          // see below
    DWORD dwValue;         // adjustment level
    DWORD dwOver;          // transmission length
    LPSTR lpstrAlgorithm;  // see below
    LPSTR lpstrQuality;    // see below
} MCI_DGV_SETAUDIO_PARMS;
```

Contains parameters for the **MCI_SETAUDIO** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwItem
Constant indicating the target adjustment. For a list of possible values, see the **MCI_SETAUDIO** command.

lpstrAlgorithm
Address of a null-terminated string containing the name of the audio-compression algorithm.

lpstrQuality
Address of a null-terminated string containing a descriptor of the audio-compression algorithm.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_SETVIDEO_PARMS

```
typedef struct {
    DWORD  dwCallback;      // see below
    DWORD  dwItem;          // see below
    DWORD  dwValue;         // adjustment level
    DWORD  dwOver;          // transmission length
    LPSTR  lpstrQuality;    // see below
    LPSTR  lpstrAlgorithm;  // see below
    DWORD  dwSourceNumber;  // index of input source
} MCI_DGV_SETVIDEO_PARMS;
```

Contains parameters for the **MCI_SETVIDEO** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwItem
Constant indicating the target adjustment.

lpstrQuality
Address of a null-terminated string containing a descriptor of the video-compression algorithm.

lpstrAlgorithm
Address of a null-terminated string containing the name of the video-compression algorithm.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_SIGNAL_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    DWORD dwPosition;  // position to be marked
    DWORD dwPeriod;    // interval of the position marks
    DWORD dwUserParm;  // user value associated with signals
} MCI_DGV_SIGNAL_PARMS;
```

Contains parameters for the **MCI_SIGNAL** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_STATUS_PARMS

```
typedef struct {
    DWORD dwCallback;    // see below
    DWORD dwReturn;      // buffer for return information
    DWORD dwItem;        // identifies capability being queried
    DWORD dwTrack;       // length or number of tracks
    LPSTR lpstrDrive;    // see below
    DWORD dwReference;   // see below
} MCI_DGV_STATUS_PARMS;
```

Contains parameters for the **MCI_STATUS** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpstrDrive
Specifies the approximate amount of disk space that can be obtained by the **MCI_RESERVE** command.

dwReference
Specifies the approximate location of the nearest previous intraframe-encoded image.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_STEP_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    DWORD dwFrames;    // number of frames to step
} MCI_DGV_STEP_PARMS;
```

Contains parameters for the **MCI_STEP** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_STOP_PARMS

```
typedef struct {
    DWORD dwCallback;
} MCI_DGV_STOP_PARMS;
```

Contains information for the **MCI_STOP** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_UPDATE_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    RECT  rc;          // see below
    HDC   hDC;         // handle to display context
} MCI_DGV_UPDATE_PARMS;
```

Contains parameters for the **MCI_UPDATE** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Rectangle containing positioning information. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_DGV_WINDOW_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    WORD  hWnd;        // see below
    WORD  nCmdShow;    // window-display command
    LPSTR lpstrText;   // window caption
} MCI_DGV_WINDOW_PARMS;
```

Contains parameters for **MCI_WINDOW** command for digital-video devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

hWnd
Handle to the display window. If this member is MCI_DGV_WINDOW_HWND, the system uses a default window.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members

MCI_GENERIC_PARMS

```
typedef struct {
    DWORD dwCallback;
} MCI_GENERIC_PARMS;
```

Contains the handle of the window that receives notification messages. This structure is used for MCI command messages that have empty parameter lists.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

The **MCI_CLOSE_PARMS** and **MCI_REALIZE_PARMS** structures are identical to the **MCI_GENERIC_PARMS** structure.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_GETDEVCAPS_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    DWORD dwReturn;    // contains information on exit
    DWORD dwItem;      // see below
} MCI_GETDEVCAPS_PARMS;
```

Contains device-capability information for the **MCI_GETDEVCAPS** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwItem
Capability being queried. This member can be one of the constants listed in the reference material for the **MCI_GETDEVCAPS** command.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_INFO_PARMS

```
typedef struct {
    DWORD dwCallback;    // see below
    LPSTR lpstrReturn;   // buffer for return string
    DWORD dwRetSize;     // size, in bytes, of return string
} MCI_INFO_PARMS;
```

Contains information for the **MCI_INFO** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_LOAD_PARMS

```
typedef struct {
    DWORD   dwCallback;  // see below
    LPCSTR  lpfilename;  // filename to load
} MCI_LOAD_PARMS;
```

Contains the filename to load for the **MCI_LOAD** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_OPEN_PARMS

```
typedef struct {
    DWORD        dwCallback;        // see below
    MCIDEVICEID  wDeviceID;  // identifier returned to application
    LPCSTR       lpstrDeviceType;   // see below
    LPCSTR       lpstrElementName;  // device element (often a path)
    LPCSTR       lpstrAlias;        // optional device alias
} MCI_OPEN_PARMS;
```

Contains information for the **MCI_OPEN** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpstrDeviceType
Name or constant identifier of the device type. (The name of the device is typically obtained from the registry or SYSTEM.INI file.) If this member is a constant, it can be one of the values listed in "Constants: Device Types" later in this chapter.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_OVLY_LOAD_PARMS

```
typedef struct {
    DWORD  dwCallback;  // see below
    LPCSTR lpfilename;  // name of file to load
    RECT   rc;          // see below
} MCI_OVLY_LOAD_PARMS;
```

Contains information for the **MCI_LOAD** command for video-overlay devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Identifies the area of the video buffer to update. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_OVLY_OPEN_PARMS

```
typedef struct {
    DWORD  dwCallback;        // see below
    MCIDEVICEID wDeviceID;    // identifier returned to application
    LPCSTR lpstrDeviceType;   // see below
    LPCSTR lpstrElementName;  // device element name (usually a path)
    LPCSTR lpstrAlias;        // optional device alias
    DWORD  dwStyle;           // window style
    DWORD  hWndParent;        // handle of parent window
} MCI_OVLY_OPEN_PARMS;
```

Contains information for the **MCI_OPEN** command for video-overlay devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpstrDeviceType
Name or constant identifier of the device type. (The name of the device is typically obtained from the registry or SYSTEM.INI file.) If this member is a constant, it can be one of the values listed in "Constants: Device Types" later in this chapter.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

You can use the **MCI_OPEN_PARMS** structure in place of **MCI_OVLY_OPEN_PARMS** if you are not using the extended data members.

MCI_OVLY_RECT_PARMS

```
typedef struct {
    DWORD dwCallback;
    RECT  rc;
} MCI_OVLY_RECT_PARMS;
```

Contains positioning information for the **MCI_PUT** and **MCI_WHERE** commands for video-overlay devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Rectangle containing positioning information. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_OVLY_SAVE_PARMS

```
typedef struct {
    DWORD  dwCallback;  // see below
    LPCSTR lpfilename;  // name of file to save
    RECT   rc;          // see below
} MCI_OVLY_SAVE_PARMS;
```

Contains information for the **MCI_SAVE** command for video-overlay devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Rectangle indicating the area of the video buffer to save. **RECT** structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_OVLY_WINDOW_PARMS

```
typedef struct {
    DWORD  dwCallback;  // see below
    HWND   hWnd;        // handle of display window
    UINT   nCmdShow;    // window-display command
    LPCSTR lpstrText;   // window caption
} MCI_OVLY_WINDOW_PARMS;
```

Contains window-display information for the **MCI_WINDOW** command for video-overlay devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_PLAY_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    DWORD dwFrom;      // position to play from
    DWORD dwTo;        // position to play to
} MCI_PLAY_PARMS;
```

Contains positioning information for the **MCI_PLAY** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_RECORD_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    DWORD dwFrom;      // position to record from
    DWORD dwTo;        // position to record to
} MCI_RECORD_PARMS;
```

Contains positioning information for the **MCI_RECORD** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_SAVE_PARMS

```
typedef struct {
    DWORD  dwCallback;  // see below
    LPCSTR lpfilename;  // name of file to save
} MCI_SAVE_PARMS;
```

Contains the filename information for the **MCI_SAVE** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_SEEK_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    DWORD dwTo;        // position to seek to
} MCI_SEEK_PARMS;
```

Contains positioning information for the **MCI_SEEK** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_SEQ_SET_PARMS

```
typedef struct {
    DWORD dwCallback;     // see below
    DWORD dwTimeFormat;   // sequencer's time format
    DWORD dwAudio;        // audio output channel
    DWORD dwTempo;        // tempo
    DWORD dwPort;         // output port
    DWORD dwSlave;        // see below
    DWORD dwMaster;       // see below
    DWORD dwOffset;       // data offset
} MCI_SEQ_SET_PARMS;
```

Contains information for the **MCI_SET** command for MIDI sequencer devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwSlave
Type of synchronization used by the sequencer for slave operation.

dwMaster
Type of synchronization used by the sequencer for master operation.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_SET_PARMS

```
typedef struct {
    DWORD dwCallback;     // see below
    DWORD dwTimeFormat;   // time format for device
    DWORD dwAudio;        // audio output channel
} MCI_SET_PARMS;
```

Contains information for the **MCI_SET** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_STATUS_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    DWORD dwReturn;    // contains information on return
    DWORD dwItem;      // capability being queried
    DWORD dwTrack;     // length or number of tracks
} MCI_STATUS_PARMS;
```

Contains information for the **MCI_STATUS** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

The MCI_STATUS_ITEM flag must be set in the *fdwCommand* parameter of the **mciSendCommand** function to validate the **dwItem** member, which should contain one of the constants indicating what status information is being requested.

MCI_SYSINFO_PARMS

```
typedef struct {
    DWORD dwCallback;    // see below
    LPSTR lpstrReturn;   // see below
    DWORD dwRetSize;     // size, in bytes, of return buffer
    DWORD dwNumber;      // see below
    UINT  wDeviceType;   // see below
} MCI_SYSINFO_PARMS;
```

Contains information for the **MCI_SYSINFO** command.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpstrReturn
Address of a user-supplied buffer for the return string. It is also used to return a doubleword value when the MCI_SYSINFO_QUANTITY flag is used.

dwNumber
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
Type of device. This member can be one of the values listed in "Constants: Device Types" later in this chapter.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_VCR_CUE_PARMS

```
typedef struct tagMCI_VCR_CUE_PARMS {
    DWORD dwCallback;  // see below
    DWORD dwFrom;      // position to cue from
    DWORD dwTo;        // position to cue to
} MCI_VCR_CUE_PARMS;
```

Contains parameters for the **MCI_CUE** command for video-cassette recorders.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_VCR_LIST_PARMS

```
typedef struct tagMCI_VCR_LIST_PARMS {
    DWORD dwCallback;  // see below
    DWORD dwReturn;    // buffer for returned information
    DWORD dwNumber;    // number of VCR's video or audio inputs
} MCI_VCR_LIST_PARMS;
```

Contains parameters for the **MCI_LIST** command for video-cassette recorders.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_VCR_PLAY_PARMS

```
typedef struct tagMCI_VCR_PLAY_PARMS {
    DWORD dwCallback;  // see below
    DWORD dwFrom;      // position to play from
    DWORD dwTo;        // position to play to
    DWORD dwAt;        // see below
} MCI_VCR_PLAY_PARMS;
```

Contains parameters for the **MCI_PLAY** command for video-cassette recorders.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwAt
Time value that affects the **MCI_PLAY** or **MCI_CUE** command. For **MCI_PLAY**, this is the time when playback begins. For **MCI_CUE**, this is the time when the cued device reaches the position given in **dwFrom**.

Positions are specified in the current time format.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_VCR_RECORD_PARMS

```
typedef struct tagMCI_VCR_RECORD_PARMS {
    DWORD dwCallback;  // see below
    DWORD dwFrom;      // position to record from
    DWORD dwTo;        // position to record to
    DWORD dwAt;        // see below
} MCI_VCR_RECORD_PARMS;
```

Contains parameters for the **MCI_RECORD** command for video-cassette recorders.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwAt
Time value that affects the **MCI_RECORD** or **MCI_CUE** command. For **MCI_RECORD**, this is the time when recording begins. For **MCI_CUE**, this is the time when the cued device reaches the position given in **dwFrom**.

Positions are specified in the current time format.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_VCR_SEEK_PARMS

```
typedef struct tagMCI_VCR_SEEK_PARMS {
    DWORD dwCallback;  // see below
    DWORD dwTo;        // position to seek to
    DWORD dwMark;      // numbered mark to seek for
    DWORD dwAt;        // time when seek begins
} MCI_VCR_SEEK_PARMS;
```

Contains parameters for the **MCI_SEEK** command for video-cassette recorders.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

Positions are specified in the current time format.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_VCR_SET_PARMS

```
typedef struct tagMCI_VCR_SET_PARMS {
    DWORD dwCallback;          // see below
    DWORD dwTimeFormat;        // current time format
    DWORD dwAudio;             // not used
    DWORD dwTimeMode;          // see below
    DWORD dwRecordFormat;      // recording rate
    DWORD dwCounterFormat;     // format of a new counter time value
    DWORD dwIndex;             // contents of on-screen display
    DWORD dwTracking;          // see below
    DWORD dwSpeed;             // see below
    DWORD dwLength;            // see below
    DWORD dwCounter;           // new counter value
    DWORD dwClock;             // new clock time
    DWORD dwPauseTimeout;      // new timeout value for pause command
    DWORD dwPrerollDuration;   // see below
    DWORD dwPostrollDuration;  // see below
} MCI_VCR_SET_PARMS;
```

Contains parameters for the **MCI_SET** command for video-cassette recorders.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwTimeMode
Constant that specifies the timing source used by the device. The timing source is either a timecode recorded on videotape, or the counters in the device that sense videotape movement.

dwTracking
Speed adjustment used when tracking the VCR playback rate.

dwSpeed
Playback speed used by the device as an integer. Normal playback speed is 1000, double speed is 2000, and half speed is 500.

dwLength
Videotape length when the length is undetectable by the device.

dwPrerollDuration
Videotape length needed to stabilize the VCR output.

dwPostrollDuration
Videotape length needed to brake the VCR transport when a **MCI_STOP** or **MCI_PAUSE** command is issued.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_VCR_SETAUDIO_PARMS

```
typedef struct tagMCI_VCR_SETAUDIO_PARMS {
    DWORD dwCallback;  // see below
    DWORD dwTrack;     // audio track
    DWORD dwTo;        // type of input or monitored input
    DWORD dwNumber;    // see below
} MCI_VCR_SETAUDIO_PARMS;
```

Contains parameters for the **MCI_SETAUDIO** command for video-cassette recorders.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwNumber
Audio input (of the type specified in the **dwTo** member) to use.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_VCR_SETTUNER_PARMS

```
typedef struct tagMCI_VCR_SETTUNER_PARMS {
    DWORD dwCallback;  // see below
    DWORD dwChannel;   // new channel number
    DWORD dwNumber;    // see below
} MCI_VCR_SETTUNER_PARMS;
```

Contains parameters for the **MCI_SETTUNER** command for video-cassette recorders.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwNumber
Logical tuner that the **MCI_SETTUNER** command affects.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_VCR_SETVIDEO_PARMS

```
typedef struct tagMCI_VCR_SETVIDEO_PARMS {
    DWORD dwCallback;  // see below
    DWORD dwTrack;     // affected track
    DWORD dwTo;        // type of input or monitored input
    DWORD dwNumber;    // see below
} MCI_VCR_SETVIDEO_PARMS;
```

Contains parameters for the **MCI_SETVIDEO** command for video-cassette recorders.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwNumber
Video input (of the type specified in the **dwTo** member) to use.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_VCR_STATUS_PARMS

```
typedef struct tagMCI_VCR_STATUS_PARMS {
    DWORD dwCallback;  // see below
    DWORD dwReturn;    // see below
    DWORD dwItem;      // type of information requested
    DWORD dwTrack;     // see below
    DWORD dwNumber;    // see below
} MCI_VCR_STATUS_PARMS;
```

Contains parameters for the **MCI_STATUS** command for video-cassette recorders.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwReturn
Value returned by the **MCI_STATUS** command. The return value varies according to the inquiry of the command. For more information, see the description of the **MCI_STATUS** command.

dwTrack
Audio or video track that will store information during the next recording. This member is used to return information when the **MCI_STATUS** command inquires about the video or audio recording status.

dwNumber
Logical tuner that the current channel is associated with. This member is used to return information when the **MCI_STATUS** command inquires about the current channel number.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_VCR_STEP_PARMS

```
typedef struct tagMCI_VCR_STEP_PARMS {
    DWORD dwCallback;
    DWORD dwFrames;
} MCI_VCR_STEP_PARMS;
```

Contains parameters for the **MCI_STEP** command for video-cassette recorders.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwFrames
Number of frames to jump (the length of a single step) as the **MCI_STEP** command steps forward or backward through the content.

When assigning data to the members in this structure, set the corresponding flags in the *fdwCommand* parameter of **mciSendCommand** to validate the members.

MCI_VD_ESCAPE_PARMS

```
typedef struct {
    DWORD  dwCallback;    // see below
    LPCSTR lpstrCommand;  // command to send to device
} MCI_VD_ESCAPE_PARMS;
```

Contains the command sent to a device for the **MCI_ESCAPE** command for videodisc devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_VD_PLAY_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    DWORD dwFrom;      // position to play from
    DWORD dwTo;        // position to play to
    DWORD dwSpeed;     // playback speed in frames per second
} MCI_VD_PLAY_PARMS;
```

Contains position and speed information for the **MCI_PLAY** command for videodisc devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

You can use the **MCI_PLAY_PARMS** structure instead of **MCI_VD_PLAY_PARMS** if you are not using the extended data members.

MCI_VD_STEP_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    DWORD dwFrames;    // number of frames to step
} MCI_VD_STEP_PARMS;
```

Contains information for the **MCI_STEP** command for videodisc devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_WAVE_DELETE_PARMS

```
typedef struct {
    DWORD dwCallback;  // see below
    DWORD dwFrom;      // position to delete from
    DWORD dwTo;        // position to delete to
} MCI_WAVE_DELETE_PARMS;
```

Contains position information for the **MCI_DELETE** command for waveform-audio devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

MCI_WAVE_OPEN_PARMS

```
typedef struct {
    DWORD  dwCallback;        // see below
    MCIDEVICEID   wDeviceID;  // identifier returned to application
    LPCSTR lpstrDeviceType;   // see below
    LPCSTR lpstrElementName;  // device element name (usually a path)
    LPCSTR lpstrAlias;        // optional device alias
    DWORD  dwBufferSeconds;   // buffer length, in seconds
} MCI_WAVE_OPEN_PARMS;
```

Contains information for **MCI_OPEN** command for waveform-audio devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpstrDeviceType
Name or constant identifier of the device type. (The name of the device is typically obtained from the registry or SYSTEM.INI file.) This member can be one of the values listed in "Constants: Device Types" later in this chapter.

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

You can use the **MCI_OPEN_PARMS** structure instead of **MCI_WAVE_OPEN_PARMS** if you are not using the extended data members.

MCI_WAVE_SET_PARMS

```
typedef struct {
    DWORD dwCallback;       // see below
    DWORD dwTimeFormat;     // device's time format
    DWORD dwAudio;          // see below
    UINT  wInput;           // audio input channel
    UINT  wOutput;          // see below
    WORD  wFormatTag;       // see below
    WORD  wReserved2;       // reserved
    WORD  nChannels;        // mono (1) or stereo (2)
    WORD  wReserved3;       // reserved
    DWORD nSamplesPerSec;   // samples per second
    DWORD nAvgBytesPerSec;  // sample rate in bytes per second
    WORD  nBlockAlign;      // block alignment of the data
    WORD  wReserved4;       // reserved
    WORD  wBitsPerSample;   // bits per sample
    WORD  wReserved5;       // reserved
} MCI_WAVE_SET_PARMS;
```

Contains information for the **MCI_SET** command for waveform-audio devices.

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwAudio
Channel number for audio output. Typically used when turning a channel on or off.

wOutput
Output device to use. For example, this value could be 2 if a system had two installed sound cards.

wFormatTag
Format of the waveform-audio data. This member can be one of the following:

WAVE_FORMAT_ADPCM
Microsoft Corporation

WAVE_FORMAT_ALAW
Microsoft Corporation

WAVE_FORMAT_ANTEX_ADPCME
Antex Electronics Corporation

WAVE_FORMAT_APTX
Audio Processing Technology

WAVE_FORMAT_AUDIOFILE_AF10
Audiofile

WAVE_FORMAT_AUDIOFILE_AF36
Audiofile

WAVE_FORMAT_CONTROL_RES_CR10
Control Resources Corporation

WAVE_FORMAT_CONTROL_RES_VQLPC
Control Resources Corporation

WAVE_FORMAT_CREATIVE_ADPCM
Creative Labs, Inc.

WAVE_FORMAT_CREATIVE_FASTSPEECH10
Creative Labs, Inc.

WAVE_FORMAT__CREATIVE__FASTSPEECH8

Creative Labs, Inc.

WAVE_FORMAT_DIALOGIC_OKI_ADPCM
Dialogic Corporation

WAVE_FORMAT_DIGIADPCM
DSP Solutions, Inc.

WAVE_FORMAT_DIGIFIX
DSP Solutions, Inc.

WAVE_FORMAT_DIGIREAL
DSP Solutions, Inc.

WAVE_FORMAT_DIGISTD
DSP Solutions, Inc.

WAVE_FORMAT_DOLBY_AC2
Dolby Laboratories, Inc.

WAVE_FORMAT_DSPGROUP_TRUESPEECH
DSP Group, Inc.

WAVE_FORMAT_DVI_ADPCM
Intel Corporation

WAVE_FORMAT_ECHOSC1
Echo Speech Corporation

WAVE_FORMAT_FM_TOWNS_SND
Fujitsu, Ltd.

WAVE_FORMAT_G721_ADPCM
Antex Electronics Corporation

WAVE_FORMAT_G723_ADPCM
Antex Electronics Corporation

WAVE_FORMAT_GSM610
Microsoft Corporation

WAVE_FORMAT_IBM_CVSD
International Business Machines

WAVE_FORMAT_IMA_ADPCM
Intel Corporation

WAVE_FORMAT_MEDIASPACE_ADPCM
VideoLogic, Inc.

WAVE_FORMAT_MPEG
Microsoft Corporation

WAVE_FORMAT_MULAW
Microsoft Corporation

WAVE_FORMAT_NMS_VBXADPCM
Natural MicroSystems Corporation

WAVE_FORMAT_OKI_ADPCM
OKI

WAVE_FORMAT_OLIADPCM
Ing C. Olivetti & C., S.p.A.

WAVE_FORMAT_OLICELP
Ing C. Olivetti & C., S.p.A.

WAVE_FORMAT_OLIGSM
Ing C. Olivetti & C., S.p.A.

WAVE_FORMAT_OLIOPR
Ing C. Olivetti & C., S.p.A.

WAVE_FORMAT_OLISBC
Ing C. Olivetti & C., S.p.A.

WAVE_FORMAT_SIERRA_ADPCM
Sierra Semiconductor Corporation

WAVE_FORMAT_SONARC
Speech Compression

WAVE_FORMAT_UNKNOWN
Microsoft Corporation

WAVE_FORMAT_YAMAHA_ADPCM
Yamaha Corporation of America

When assigning data to the members of this structure, set the corresponding flags in the *fdwCommand* parameter of the **mciSendCommand** function to validate the members.

Constants: Device Types

The following values identify devices in MCI messages and structures:

Value	Meaning
MCI_ALL_DEVICE_ID	Any device
MCI_DEVTYPE_ANIMATION	Animation-playback device
MCI_DEVTYPE_CD_AUDIO	CD audio device
MCI_DEVTYPE_DAT	Digital-audio tape device
MCI_DEVTYPE_DIGITAL_VIDEO	Digital-video playback device
MCI_DEVTYPE_OTHER	Undefined device
MCI_DEVTYPE_OVERLAY	Video-overlay device
MCI_DEVTYPE_SCANNER	Scanner device
MCI_DEVTYPE_SEQUENCER	MIDI sequencer device
MCI_DEVTYPE_VCR	Video-cassette recorder
MCI_DEVTYPE_VIDEODISC	Videodisc players
MCI_DEVTYPE_WAVEFORM_AUDIO	Waveform-audio device

MCI Overview

The Media Control Interface (MCI) is a high-level command interface to multimedia devices and resource files. MCI provides applications with device-independent capabilities for controlling audio and visual peripherals. Your application can use MCI to control any supported multimedia device, including waveform-audio devices, MIDI sequencers, CD audio devices, and digital-video (video playback) devices.

MCI provides standard commands for playing multimedia devices and recording multimedia resource files. These commands are a generic interface to virtually every kind of multimedia device.

MCI Command Strings and Messages

There are two forms of MCI commands: strings and messages. You can use either or both forms in your MCI application.

- The *command-message interface* consists of constants and structures of the C programming language. You use the **mciSendCommand** function to send a message to an MCI device.
- The *command-string interface* provides a textual version of the command messages. You use the **mciSendString** function to send a string to an MCI device. Command strings duplicate the functionality of the command messages. The Microsoft Windows operating system converts the command strings to command messages before sending them to the MCI driver for processing.

The command messages that retrieve information do so in the form of structures, which are easy to interpret in a C application. These structures can contain information on many different aspects of a device. The command strings that retrieve information do so in the form of strings, and can only retrieve one string at a time. Your application must parse or test each string to interpret it. You might find that the command messages are easier to use than the command strings in some cases, but the command strings are easy to remember and implement. Some MCI applications use command strings when the return value is unimportant or simply "true" or "false" and command messages when retrieving information from the device.

This chapter documents MCI without going into great detail on either the command-string or command-message interfaces. When commands or flags are discussed, this chapter uses the string form of the command or flag followed by the message form in parentheses. For information about the command-string interface, see Chapter 4, "MCI Command Strings." For information about the command-message interface, see Chapter 5, "MCI Command Messages."

MCI Devices

Every MCI multimedia device reacts to a core set of MCI commands as you would probably expect it to. For example, the **play** (**MCI_PLAY**) command causes the open device to play a file or track, no matter what kind of data the device works with. This section discusses MCI devices and how they respond to standard MCI commands.

Device Control

To control an MCI device, you open the device, send the necessary commands to it, and then close the device. The commands can be very similar, even for completely different MCI devices. For example, the following series of MCI commands plays the sixth track of an audio CD by using the command-string interface:

```
mciSendString("open cdaudio", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
mciSendString("set cdaudio time format tmsf", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
mciSendString("play cdaudio from 6 to 7", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
mciSendString("close cdaudio", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
```

The next example shows a similar series of MCI commands that plays the first 10,000 samples of a waveform-audio file:

```
mciSendString(
    "open c:\mmdata\purplefi.wav type waveaudio alias finch",
    lpszReturnString, lstrlen(lpszReturnString), NULL);
mciSendString("set finch time format samples", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
mciSendString("play finch from 1 to 10000", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
mciSendString("close finch", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
```

These examples illustrate some interesting facts about MCI commands:

- The same basic commands (**open**, **set**, **play**, and **close**) are used with CD audio and waveform-audio devices. The same MCI commands are used with all MCI devices.
- The **open** command for the waveform-audio device includes a filename specification. The waveform-audio device is a *compound device* (one associated with a data file), while the CD audio device is a *simple device* (one without an associated data file).
- The **set** command specifies time formats in each case, but the time-format flag for the CD audio device specifies tracks/minutes/seconds/frames (TMSF) format, while the time format used with the waveform-audio device specifies "samples".
- The variables used with the "from" and "to" flags are appropriate to the respective time format. For example, for the CD audio device, the variables specify a range of tracks, but for the waveform-audio device, the variables specify a range of samples.

Playback and Positioning

A number of MCI commands, such as **play** (**MCI_PLAY**), **stop** (**MCI_STOP**), **pause** (**MCI_PAUSE**), **resume** (**MCI_RESUME**), and **seek** (**MCI_SEEK**), affect the playback or positioning of a multimedia file. If an MCI device receives a playback command while another playback command is in progress, it accepts the command and either stops or supersedes the previous command.

Many MCI commands, such as **set** (**MCI_SET**), do not affect playback. A notification from one of these commands does not interfere with pending playback or position commands as long as the notifications are not performed from the same instance of the driver. For example, you can issue a **set** or **status** (**MCI_STATUS**) command while a device is performing a **seek** command without stopping or superseding the **seek** command.

However, there can only be one pending notification. For example, if an application requests a notification for **play** and follows that request with **status** "start position notify," the **play** notification will return "superseded" and the notification for the status command will return when it is finished. In this case, however, the **play** command will still succeed, even though the application did not receive the notification.

Device Types

MCI recognizes a basic set of *device types*. A device type is a set of MCI drivers that share a common command set and are used to control similar multimedia devices or data files. Many MCI commands, such as **open** (**MCI_OPEN**), require you to specify a device type.

The following table lists the defined device types. The current implementation of MCI includes command sets for a subset of these devices.

Device type	Constant	Description
animation	MCI_DEVTYPE_ANIMATION	Animation device
cdaudio	MCI_DEVTYPE_CD_AUDIO	CD audio player
dat	MCI_DEVTYPE_DAT	Digital-audio tape player
digitalvideo	MCI_DEVTYPE_DIGITAL_VIDEO	Digital video in a window (not GDI based)
other	MCI_DEVTYPE_OTHER	Undefined MCI device
overlay	MCI_DEVTYPE_OVERLAY	Overlay device (analog video in a window)
scanner	MCI_DEVTYPE_SCANNER	Image scanner
sequencer	MCI_DEVTYPE_SEQUENCER	MIDI sequencer
vcr	MCI_DEVTYPE_VCR	Video-cassette recorder or player
videodisc	MCI_DEVTYPE_VIDEODISC	Videodisc player
waveaudio	MCI_DEVTYPE_WAVEFORM_AUDIO	Audio device that plays digitized waveform files

In this document, the names of device types are bold. Device-type names are used with the command-string interface. Device-type constants are used with the command-message interface.

Device Names

For any given device type, there might be several MCI drivers that share the command set but operate on different data formats. For example, the **animation** device type might include several MCI drivers that use the same command set but play different types of animation files. To uniquely identify an MCI driver, MCI uses *device names*.

Device names are identified either in the [mci] section of the SYSTEM.INI file or in the appropriate part of the registry. This information identifies all MCI drivers to Windows. The entries in the [mci] section use the following form:

device_name = *driver_filename.extension*

The following example shows a typical [mci] section from SYSTEM.INI:

```
[mci]
cdaudio=mcicda.drv
sequencer=mciseq.drv
waveaudio=mciwave.drv
avivideo=mciavi.drv
```

If an MCI driver is installed using a device name that already exists in SYSTEM.INI or the registry, the system appends an integer to the device name of the new driver, creating a unique device name. In the preceding example, an additional driver installed using the "cdaudio" device name would be assigned the device name "cdaudio1".

Driver Support for MCI Commands

MCI drivers provide the functionality for MCI commands. The system software performs some basic data-management tasks, but all the multimedia playback, presentation, and recording is handled by the individual MCI drivers.

Drivers vary in their support for MCI commands and command flags. Because multimedia devices can have widely different capabilities, MCI is designed to let individual drivers extend or reduce the command sets to match the capabilities of the device. For example, the **record** (**MCI_RECORD**) command is part of the command set for MIDI sequencers, but the MCISEQ driver included with Windows does not support this command. The reference material for the **record** command explains that devices of the **sequencer** device type recognize the command; this does not mean that all devices of this type support the command. Applications should use the **capability** (**MCI_GETDEVCAPS**) command to determine the capabilities of a particular device.

Default Behavior of Drivers

In many situations, the MCI command specifications define the default values and behavior for drivers of a particular device type. Since multimedia devices can have a wide range of features (and limitations), there can be undefined areas of behavior. Also, drivers might handle exceptions differently based on the device capabilities and the design goals of the programmer who developed the driver.

For example, consider the following commands sent to a waveform-audio driver:

```
mciSendString("open sound.wav alias sound", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
mciSendString("play sound notify", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
mciSendString("record sound from 0 notify", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
```

The **record** command returns a "Parameter out of range" value and stops the playback started by the previous **play** command. One might expect the driver to validate the **record** command before stopping playback, but the driver stops the playback first.

Classifications of MCI Commands

MCI defines four command classifications: system, required, basic, and extended. The following list describes these command classifications:

- *System commands* are handled by MCI directly, rather than by the driver.
- *Required commands* are handled by the driver. All MCI drivers must support the required commands and flags.
- *Basic commands* (or optional commands) are used by some devices. If a device supports a basic command, it must support a defined set of flags for that command.
- *Extended commands* are specific to a device type or driver. Extended commands include new commands (like the **put** (**MCI_PUT**) and **where** (**MCI_WHERE**) commands for the **animation**, **digitalvideo**, and **overlay** device types) and extensions to existing commands (like the "stretch" flag of the **status** (**MCI_STATUS**) command for the **animation** and **overlay** device types).

While system and required commands are the minimum command set for any MCI driver, basic and extended commands are not supported by all drivers. Your application can always use system and required commands and their flags, but if it needs to use a basic or extended command or flag, it should first query the driver by using the **capability** (**MCI_GETDEVCAPS**) command. The following sections summarize the specific commands in each category.

System Commands

MCI processes the following system commands directly, rather than passing them to MCI devices:

String	Message	Description
break	**MCI_BREAK**	Sets a break key for an MCI device.
sysinfo	**MCI_SYSINFO**	Returns information about MCI devices.

Required Commands

All MCI devices support the following required commands:

String	Message	Description
capability	**MCI_GETDEVCAPS**	Obtains the capabilities of a device.
close	**MCI_CLOSE**	Closes the device.
info	**MCI_INFO**	Obtains textual information from a device.
open	**MCI_OPEN**	Initializes the device.
status	**MCI_STATUS**	Obtains status information from the device. Some of this command's flags are not required, so it is also a basic command.

Devices must also support a standard set of command flags for the required commands.

Basic Commands

The following list summarizes the basic commands. The use of these commands by an MCI device is optional.

String	Message	Description
load	**MCI_LOAD**	Loads data from a file.

pause	MCI_PAUSE	Stops playing. Playback or recording can be resumed at the current position.
play	MCI_PLAY	Starts transmitting output data.
record	MCI_RECORD	Starts recording input data.
resume	MCI_RESUME	Resumes playing or recording on a paused device.
save	MCI_SAVE	Saves data to a disk file.
seek	MCI_SEEK	Seeks forward or backward.
set	MCI_SET	Sets the operating state of the device.
status	MCI_STATUS	Obtains status information about the device. This is also a required command; since some of its flags are not required, it is also listed here. (The optional items support devices that use linear media with identifiable positions.)
stop	MCI_STOP	Stops playing.

If a driver supports a basic command, it must also support a standard set of flags for the command.

Extended Commands

Some MCI devices have additional commands, or they add flags to existing commands. While some extended commands apply only to a specific device driver, most of them apply to all drivers of a particular device type. For example, the command set for the **sequencer** device type extends the set (MCI_SET) command to add time formats that are needed by MIDI sequencers.

You should not assume that the device supports the extended commands or flags. You can use the capability (MCI_GETDEVCAPS) command to determine whether a specific feature is supported, and your application should be ready to deal with "unsupported command" or "unsupported function" return values.

The following extended commands are available with the listed device types:

String	Message	Device types	Description
configure	MCI_CONFIGURE	**digitalvideo**	Displays a configuration dialog box.
cue	MCI_CUE	**digitalvideo, waveaudio**	Prepares for playing or recording.
delete	MCI_DELETE	**waveaudio**	Deletes a data segment from the media file.
escape	MCI_ESCAPE	**videodisc**	Sends custom information to a device.
freeze	MCI_FREEZE	**overlay**	Disables video acquisition to the frame buffer.
put	MCI_PUT	**animation, digitalvideo, overlay**	Defines the source, destination, and frame windows.
realize	MCI_REALIZE	**animation, digitalvideo**	Tells the device to select and realize its palette into a device context (DC) of the displayed window.

setaudio	**MCI_SETAUDIO**	**digitalvideo**	Sets audio parameters for video.
setvideo	**MCI_SETVIDEO**	**digitalvideo**	Sets video parameters.
signal	**MCI_SIGNAL**	**digitalvideo**	Identifies a specified position with a signal.
spin	**MCI_SPIN**	**videodisc**	Starts the disc spinning or stops the disc from spinning.
step	**MCI_STEP**	**animation, digitalvideo, videodisc**	Steps the play one or more frames forward or reverse.
unfreeze	**MCI_UNFREEZE**	**overlay**	Enables the frame buffer to acquire video data.
update	**MCI_UPDATE**	**animation, digitalvideo**	Repaints the current frame into the DC.
where	**MCI_WHERE**	**animation, digitalvideo, overlay**	Obtains the rectangle specifying the source, destination, or frame area.
window	**MCI_WINDOW**	**animation, digitalvideo, overlay**	Controls the display window.

Working with MCI Devices

This section describes how to perform the following tasks:

- Open a device.
- Retrieve information about a device.
- Obtain MCI system information.
- Play a device.
- Record.
- Stop, pause, and resume a device.
- Close a device.

In addition, this section provides you with shortcuts you can use with MCI commands.

Opening a Device

Before using a device, you must initialize it by using the **open** (**MCI_OPEN**) command. This command loads the driver into memory (if it isn't already loaded) and retrieves the device identifier you will use to identify the device in subsequent MCI commands. You should check the return value of the **mciSendString** or **mciSendCommand** function before using a new device identifier to ensure that the identifier is valid. (You can also retrieve a device identifier by using the **mciGetDeviceID** function.)

Like all MCI command messages, **MCI_OPEN** has an associated structure. These structures are sometimes called *parameter blocks*. The default structure for **MCI_OPEN** is **MCI_OPEN_PARMS**. Certain devices (such as **waveform**, **animation**, and **overlay**) have extended structures to accommodate additional optional parameters. Unless you need to use these additional parameters, you can use the **MCI_OPEN_PARMS** structure with any MCI device.

The number of devices you can have open is limited only by the amount of available memory.

Using an Alias

When you open a device, you can use the "alias" flag to specify a device identifier for the device. This flag lets you assign a short device identifier for compound devices with lengthy filenames, and it lets you open multiple instances of the same file or device.

For example, the following command assigns the device identifier "birdcall" to the lengthy filename C:\NABIRDS\SOUNDS\MOCKMTNG.WAV:

```
mciSendString(
    "open c:\nabirds\sounds\mockmtng.wav type waveaudio alias birdcall",
    lpszReturnString, lstrlen(lpszReturnString), NULL);
```

In the command-message interface, you specify an alias by using the **lpstrAlias** member of the **MCI_OPEN_PARMS** structure.

Specifying a Device Type

When you open a device, you can use the "type" flag to refer to a device type, rather than to a specific device driver. The following example opens the waveform-audio file C:\WINDOWS\CHIMES.WAV (using the "type" flag to specify the **waveaudio** device type) and assigns the alias "chimes":

```
mciSendString(
    "open c:\windows\chimes.wav type waveaudio alias chimes",
    lpszReturnString, lstrlen(lpszReturnString), NULL);
```

In the command-message interface, the functionality of the "type" flag is supplied by the **lpstrDeviceType** member of the **MCI_OPEN_PARMS** structure.

Simple and Compound Devices

MCI classifies device drivers as *compound* or *simple*. Drivers for compound devices require the name of a data file for playback; drivers for simple devices do not.

Simple devices include **cdaudio** and **videodisc** devices. There are three ways to open simple devices:

- Specify a pointer to a null-terminated string containing the device name from the registry or the SYSTEM.INI file.

  For example, you can open a **videodisc** device by using the following command:

  ```
  mciSendString("open videodisc", lpszReturnString,
      lstrlen(lpszReturnString), NULL);
  ```

  In this case, "videodisc" is the device name from the registry or the [mci] section of SYSTEM.INI.

- Specify the actual name of the device driver. Opening a device using the device-driver filename, however, makes the application device-dependent and can prevent the application from running if the system configuration changes. If you use a filename, you do not need to specify the complete path or the filename extension; MCI assumes drivers are located in a system directory and have the .DRV filename extension.

Compound devices include **waveaudio** and **sequencer** devices. The data for a compound device is sometimes called a *device element*. This document, however, generally refers to this data as a file, even though in some cases the data might not be stored as a file.

There are three ways to open a compound device:

- Specify only the device name. This lets you open a compound device without associating a filename. Most compound devices process only the **capability** (**MCI_GETDEVCAPS**) and **close** (**MCI_CLOSE**) commands when they are opened this way.
- Specify only the filename. The device name is determined from the associations entered in the registry.
- Specify the filename and the device name. MCI ignores the entries in the registry and opens the specified device name.

To associate a data file with a particular device, you can specify the filename and device name. For example, the following command opens the **waveaudio** device with the filename MYVOICE.SND:

```
mciSendString("open myvoice.snd type waveaudio", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
```

In the command-string interface, you can also abbreviate the device name specification by using the alternative exclamation-point format, as documented with the **open** command.

Opening a Device Using the Filename Extension

If the **open** (**MCI_OPEN**) command specifies only the filename, MCI uses the filename extension to select the appropriate device from the list in the registry or the [mci extensions] section of the SYSTEM.INI file. The entries in the [mci extensions] section use the following form:

filename_extension=device_name

MCI implicitly uses *device_name* if the extension is found and if a device name has not been specified in the **open** command.

The following example shows a typical [mci extensions] section:

```
[mci extensions]
wav=waveaudio
mid=sequencer
rmi=sequencer
```

Using these definitions, MCI opens the **waveaudio** device if the following command is issued:

```
mciSendString("open train.wav", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
```

New Data Files

To create a new data file, simply specify a blank filename. MCI does not save a new file until you save it by using the **save** (**MCI_SAVE**) command. When creating a new file, you must include a device alias with the **open** (**MCI_OPEN**) command.

The following example opens a new **waveaudio** file, starts and stops recording, then saves and closes the file:

```
mciSendString("open new type waveaudio alias capture", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
mciSendString("record capture", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
mciSendString("stop capture", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
mciSendString("save capture orca.wav", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
mciSendString("close capture", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
```

Shareable Devices

The "shareable" (MCI_OPEN_SHAREABLE) flag of the **open** (**MCI_OPEN**) command lets multiple applications access the same device (or file) and device instance simultaneously. If your application opens a device or file as shareable, other applications can also access it by opening it as shareable. The shared device or file gives each application the ability to change the parameters governing its operating state. Each time a device or file is opened as shareable, MCI returns a unique device identifier, even though the identifiers refer to the same instance.

If your application opens a device or file without specifying that it is shareable, no other application can access it until your application closes it. Also, if a device supports only one open instance, the **open** command will fail if you specify the shareable flag.

If your application opens a device and specifies that it is shareable, your application should not make any assumptions about the state of this device. Your application might need to compensate for changes made by other applications accessing the device.

Most compound files are not shareable; however, you can open multiple files (where each is unique), or you can open a single file multiple times. If you open a single file multiple times, MCI creates an independent instance for each, with each instance having a unique operating status.

If you open multiple instances of a file, you must assign a unique device identifier to each. You can use an alias, as described in the following section, to assign a unique name for each file.

Retrieving Information About a Device

Every device responds to the **capability** (**MCI_GETDEVCAPS**), **status** (**MCI_STATUS**), and **info** (**MCI_INFO**) commands. These commands obtain information about the device. For example, the following command returns "true" if a **cdaudio** device can eject the disc:

```
mciSendString(
    "capability cdaudio can eject",
    lpszReturnString, lstrlen(lpszReturnString), NULL);
```

The flags listed for the required and basic commands provide a minimum amount of information about a device. Many devices supplement the required and basic commands with extended flags to provide additional information about the device.

Obtaining MCI System Information

The **sysinfo** (**MCI_SYSINFO**) command obtains system information about MCI devices. MCI handles this command without relaying it to any MCI device. For the command-message interface, MCI returns the system information in the **MCI_SYSINFO_PARMS** structure.

You can use the **sysinfo** (**MCI_SYSINFO**) command to retrieve information such as the number of MCI devices on a system, the number of MCI devices of a particular type, the number of open MCI devices, and the names of the devices. This command is often called more than once to retrieve a particular piece of information. For example, you might retrieve the number of devices of a particular type in the first call and then enumerate the names of the devices in the next.

Playing a Device

The **play** (**MCI_PLAY**) command starts playing a device. Without any flags, this command starts playing from the current position and plays until the command is interrupted or until the end of the media or file is reached. After playback, the current position is at the end of the media. You can also use the **seek** (**MCI_SEEK**) command to change the current position.

Most devices that support the **play** command also support the "from" (MCI_FROM) and "to" (MCI_TO) flags. These flags indicate the position at which the device should start and stop playing. For example, the following command plays a CD audio disc from the beginning of the first track:

```
mciSendString("play cdaudio from 0", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
```

Some device types extend this command to exploit the capabilities of a particular device. For example, the **play** command for the **videodisc** device type includes the "fast" (MCI_VD_PLAY_FAST) , "slow" (MCI_VD_PLAY_SLOW), and "scan" (MCI_VD_PLAY_SCAN) flags.

Note The units assigned to the position value depend on the time format used by the device. Each device has a default time format, but you should specify the time format by using the **set** (**MCI_SET**) command before issuing any commands that use position values.

Playing an AVI File

Video files in Windows are made up of at least two interleaved data streams: a video (pictorial) stream and an audio stream. You can easily play these audio-video interleaved (AVI) files by using MCI commands. The following sections discuss playing AVI files.

Setting Up an MCIAVI Playback Window

Your application can specify the following options to define the playback window for playing an AVI file:

- Use the MCIAVI driver's default pop-up window.
- Specify a parent window and window style that the MCIAVI driver can use to create the playback window.
- Specify a playback window for the MCIAVI driver to use for playback.
- Play the AVI file on a full-screen display.

If your application does not specify any window options, the MCIAVI driver creates a default window for playing the sequence. The driver creates this playback window for the **open** (**MCI_OPEN**) command, but it does not display the window until your application sends a command to either display the window or play the file. This default playback window is a pop-up window with a sizing border, title bar, a thick frame, a system menu, and a minimize button.

Your application can also specify a parent window handle and a window style when it issues the **open** command. In this case, the MCIAVI driver creates a window based on these specifications instead of the default pop-up window. Your application can specify any window style available for the **CreateWindow** function. Styles that require a parent window, such as WS_CHILD, should include a parent window handle.

Your application can also create its own window and supply the handle to the MCIAVI driver by using the **window** (**MCI_WINDOW**) command. The MCIAVI driver uses this window instead of creating one of its own.

When the MCIAVI driver creates the playback window or obtains a window handle from your application, it does not display the window until your application either plays the sequence or sends a command to display the window. Your application can use the **window** command to display the window without playing the sequence. For example, the following command displays the window using the

command-string interface:

```
mciSendString("window movie state show", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
```

In this example, "movie" is an alias for the digital-video device.

Your application can also play an AVI file full-screen. To play full-screen, modify the **play** (**MCI_PLAY**) command with the "fullscreen" (MCI_MCIAVI_PLAY_FULLSCREEN) flag. When your application uses this flag, the MCIAVI driver uses a 320- by 240-pixel full-screen format for playing the sequence. For example, the following command plays the opened file full-screen (using "movie" as an alias):

```
mciSendString("play movie fullscreen", lpszReturnString,
    lstrlen(lpszReturnString), NULL);
```

Changing the Playback State for an AVI file

Your application can use the **seek** (**MCI_SEEK**) command to move the current position to the beginning, the end, or an arbitrary position in an AVI file. There are two seek modes for the MCIAVI driver: exact and inexact. Your application can change the seek mode by using the **set** (**MCI_SET**) command. When seek exactly is enabled (**set** "seek exactly on"), the MCIAVI driver seeks exactly to the frame your application specifies. This might cause a delay if the file is temporally encoded and your application does not specify a key frame. With seek exactly disabled (**set** "seek exactly off"), the MCIAVI driver seeks to the nearest key frame in a temporally encoded file.

Some MCI commands let your application alter the playback of an AVI file in other ways. For example, an AVI file, by default, plays at its normal speed, but your application can increase or decrease this speed by using the "speed" flag for the **set** command. For AVI files, a speed value of 1000 is normal. Thus, to play a movie at half its normal speed, your application can use the command **set** "movie speed 500"; alternatively, it can use **set** "movie speed 2000" to play the sequence at twice its normal speed.

The **setaudio** (**MCI_SETAUDIO**) command lets your application control the audio portion of an AVI file. Your application can mute audio during playback or, in the case of multiple audio stream files, select the audio stream that is played.

The MCIAVI driver has a dialog box to control some of its playback options. Some of the options available to the user include selecting window-oriented or full-screen playback, selecting the seek mode, and zooming the image. Your application can have MCIAVI display this dialog box by using the **configure** (**MCI_CONFIGURE**) command.

Stream Handlers

The data in an AVI file is treated as a series of streams. An AVI file typically contains an audio and video stream, and there might be a custom stream as well containing text or some other custom data. The MCIAVI driver can use different handlers for these data streams. For more information about custom AVI files, see Chapter 9, "Custom File and Stream Handlers."

Recording

The general MCI specification supports recording with digital-video, MIDI sequencer, video-cassette recorder (VCR), and waveform-audio devices; however, only waveform-audio and VCR devices currently implement recording capabilities. You can insert or overwrite recorded information into an existing file or record into a new file. To record to an existing file, open a waveform-audio device and file as you would normally. To record into a new file, when you open the device specify "new" as the device name if you are using the command-string interface, or a zero-length filename if you are using the command-message interface.

When MCI creates a new file for recording, the data format is set to a default format specified by the device driver. To use a format other than the default format, you can use the **set** (**MCI_SET**) command.

To begin recording, use the **record** command (or **MCI_RECORD** and the **MCI_RECORD_PARMS** structure).

If you record in insert mode to an existing file, you can use the "from" (MCI_FROM) and "to" (MCI_TO) flags of the **record** command to specify starting and ending locations for recording. For example, if you record to a file that is 20 seconds long, and you begin recording at 5 seconds and end recording at 10 seconds, the resulting file will be 25 seconds long. The file will have a 5-second segment inserted 5 seconds into the original recording.

If you record with overwrite mode to an existing file, you can use the "from" and "to" flags to specify starting and ending locations of the section that is overwritten. For example, if you record to a file that is 20 seconds long, and you begin recording at 5 seconds and end recording at 10 seconds, you still have a recording 20 seconds long, but the section beginning at 5 seconds and ending at 10 seconds will have been replaced.

If you do not specify an ending location, recording continues until you send a **stop** (**MCI_STOP**) command, or until the driver runs out of free disk space. If you record to a new file, you can omit the "from" flag or set it to zero to start recording at the beginning of a new file. You can specify an ending location to terminate recording when recording to a new file.

The **record** command is sometimes accurate to within only 1 second of the starting location, such as with VCR devices. To record more accurately, you should use the **cue** (**MCI_CUE**) command. This command is recognized by digital-video, VCR, and waveform-audio devices. For more information about recording with VCR devices, see "VCR Services" later in this chapter.

Saving a Recorded File

When recording is complete, use the **save** command (or **MCI_SAVE** and the **MCI_SAVE_PARMS** structure) to save the recording before closing the device.

Note If you close the device without saving, the recorded data is lost.

Checking Input Levels (PCM only)

To get the level of the input signal before recording on a PCM (Pulse Code Modulation) waveform-audio input device, use the **status** (**MCI_STATUS**) command. Specify the "level" flag (or the MCI_STATUS_ITEM flag and set the **dwItem** member of the **MCI_STATUS_PARMS** structure to MCI_WAVE_STATUS_LEVEL). The average input signal level is returned. The left-channel value is in the high-order word and the right- or mono-channel value is in the low-order word

The input level is represented as an unsigned value. For 8-bit samples, this value is in the range 0 through 127 (0x7F). For 16-bit samples, it is in the range 0 through 32,767 (0x7FFF).

Stopping, Pausing, and Resuming a Device

The **stop** (**MCI_STOP**) command suspends the playing or recording of a device. Many devices also support the **pause** (**MCI_PAUSE**) command. The difference between **stop** and **pause** depends on the device. Usually **pause** suspends operation but leaves the device ready to resume playing or recording immediately.

Using the **play** (**MCI_PLAY**) or **record** (**MCI_RECORD**) command to restart a device resets the locations specified with the "to" (MCI_TO) and "from" (MCI_FROM) flags before the device was paused or stopped. Without the "from" flag, these commands reset the starting location to the current position. Without the "to" flag, they reset the ending location to the end of the media. To continue playing or recording without resetting a previously specified stop position, use the **play** or **record** command's "to" flag to specify an ending location.

Some devices support the **resume** (**MCI_RESUME**) command to restart a paused device. This command does not change the "to" and "from" locations specified with the **play** or **record** command that preceded the **pause** command.

Closing a Device

The **close** (**MCI_CLOSE**) command releases access to a device or file. MCI frees a device when all tasks using a device have closed it. To help MCI manage the devices, your application must close each device or file when it is finished using it.

When you close an external MCI device that uses its own media instead of files (such as CD audio), the driver leaves the device in its current mode of operation. Thus, if you close a CD audio device that is playing, even though the device driver is released from memory, the CD audio device will continue to play until it reaches the end of its content.

Note Closing an application with open MCI devices can prevent other applications from using those devices until Windows is restarted.

MCI Implementations for Specific Devices

This section discusses using MCI commands with specific MCI devices.

MCIAVI

An AVI file can contain more than two streams – for example, a video sequence, an English soundtrack, and a French soundtrack. Your application can use a stream independently of the other streams in the file.

The **digitalvideo** device type controls video files. For a list of the MCI commands recognized by digital-video devices, see "Digital-Video Command Set" later in this chapter.

The MCIAVI driver plays video sequences and other data streams under the control of MCI commands. Data streams can contain images, audio, and palettes. The image data can consist of images with either color palettes or true-color information.

Audio is synchronized with the video within one-thirtieth of a second. If audio hardware is not available, however, the driver plays only the video stream. The MCIAVI driver can drop video frames, if necessary, to play a stream without audio interruption.

Your application can use the MCIWnd window class services instead of the MCI command interface to control any MCI driver. This window class handles many of the details of managing the window supporting the MCI device and simplifies the programming required to send the MCI commands. Your application can use the MCIWnd library services directly to control the MCI device, or it can have MCIWnd display a toolbar, scroll bar, and menus that let the user control the device. For more information about the MCIWnd window class, see Chapter 2, "Getting Started Using MCIWnd."

VCR Services

Windows provides VCR services through a device driver that is based on the MCI command set for VCRs. This section describes the MCI Video System Control Architecture (VISCA) driver and explains how to use it to control a VCR.

The **vcr** device type controls VCRs. For a list of the MCI commands recognized by VCR devices, see "VCR Command Set" later in this chapter.

The MCI VISCA Driver

The MCI VISCA driver controls Sony® VISCA-compatible VCRs, such as the CVD-1000 VDeck. The VISCA driver controls the tape transport, channel tuners, and VCR input and output channels.

Searching and Positioning with a VCR

The VISCA driver uses two methods to track videotape movement within the VCR tape transport: *timecode information* and *tape counters*. Timecode information is timing information that has been recorded on the videotape. Most VCRs allow timecodes to be recorded without destroying audio and video tracks. Tape counters estimate the amount of videotape that travels past the videotape head to obtain a position.

Both timecode information and tape counters increase as the videotape moves from beginning to end. Because of its accuracy, using timecode information to position a videotape is almost always preferable to using tape counters.

The MCI command flags for specifying positioning information are expressed as time dependencies: "time format", "duration", "from", "to", and "seek". (Also, the **status** "position" command returns its time value in the current time format.)

The VISCA driver uses the **set** "time mode" command to select the type of positioning to use with a videotape. When the time mode is set to "timecode", the **status** "position" and **set** "time format" commands use the timecode on the videotape. When the time mode is set to "counter", the **status** "position" and **set** "time format" commands use counters.

An application can set the time mode to "detect" if it doesn't matter that there might be two sources of position information. When in detect mode, the VISCA driver uses timecode information for positioning when any of the following conditions occur:

- The timecode information is present when the driver is opened.
- You change a videotape with the **set** "door open" command and timecode information is present on the videotape.
- The **set** "time mode" command is reissued.

If timecode information cannot be found, the driver uses the tape counters.

To determine the current positioning method, issue the **status** "time type" command, which returns either "timecode" or "counter". You can also identify the current positioning mode by using the **status** "time mode" command, which returns "timecode", "counter", or "detect".

The **status** "counter" command retrieves the current tape counter value, regardless of the current positioning method; however, you can use this counter reading only with the **set** "counter" command.

The VISCA driver can retrieve the native timecode format recorded on a videotape by using the **status** "timecode type" and **status** "frame rate" commands together. For example, if timecode type is "smpte" and frame rate is 25, the native timecode format recorded on the videotape is SMPTE 25.

The VISCA driver can also retrieve the counter resolution by using the **status** "counter resolution" command, which returns "seconds" or "frames". The counter format might still be set to SMPTE 30, but the return value returns only a frame of 0. If the current time type is counter, then this resolution applies also to the value returned by **status** "position".

Capturing Frames

Frame-capturing commands provide still images for a *frame-capture device*. A frame-capture device is a separate piece of hardware capable of reading and storing the video image. The VISCA driver supports the **freeze** (**MCI_FREEZE**) command to stabilize a still image for capturing. Also, the **unfreeze** (**MCI_UNFREEZE**) command can be used to restart the tape transport following a **freeze** command.

The **freeze** command provides a high-quality, stabilized, time-base - corrected image for a frame-capture device. This command exists because a device might not always deliver its maximum-quality output image during playback or while paused; such a video image is not suitable for capturing.

The **unfreeze** command unlocks the tape transport and resumes the transport mode in effect prior to the **freeze** command.

When your application needs to record a video image on the VCR, as is common with software-animation applications, use the **freeze** "input" command or the **cue** (**MCI_CUE**) command to record the image.

Selecting Inputs

The VISCA driver supports three input types: video, audio, and timecode. The video inputs include two standard channels (lines 1 and 2), an SVideo channel, an auxiliary channel, and a channel from an internal tuner. The audio inputs include two standard channels (lines 1 and 2) and a channel from an internal tuner. The timecode input is internal to the VCR.

The normal outputs carry the currently selected inputs when the VCR is recording or when the tape transport is stopped, and they carry the contents of the videotape when the tape transport is playing or paused. The monitored outputs carry the same information as the normal outputs plus current timecode and channel information.

Assuming the appropriate external inputs are connected to your VCR and you have decided what you want to record, you can select the inputs to be recorded. For example, to record or view from the "svideo" video and the "line 1" audio inputs, you would use the **setvideo** (**MCI_SETVIDEO**) and **setaudio** (**MCI_SETAUDIO**) commands to select these input sources. You can verify these selections by using the **status** (**MCI_STATUS**) command.

By default, the monitor shows exactly what appears as the output. Sometimes, however, you might want to view one source while recording from another. This is a common practice using the tuner. For example, you might want to watch channel 4 while you record channel 7. In this case, you have two logical tuner inputs. You could set up the VCR by using the following commands:

1. Use the **settuner** (**MCI_SETTUNER**) command to select the channels to watch and record.
2. Use the **setvideo** command to select the video-recording source.
3. Use the **setaudio** command to select the audio-recording source.
4. Use the **setvideo** command to route the channel 4 video input to the monitored output to display it on-screen.
5. Use the **setaudio** command to route the channel 4 audio input to the monitored output to play the audio.
6. Verify your selections by using the **status** command.

The VISCA driver also supports a special input type for audio and video called *mute*. Mute allows the selection of "no input," which is useful when recording a blank signal.

Selecting Recording Tracks

There are three types of recording tracks on a videotape: video, audio, and timecode. You have only one video or timecode track, but you can use more than one audio track. When you do so, make track 1 the main audio track.

The VISCA driver supports two operating modes: assemble and insert. In *assemble mode*, all tracks are selected to be recorded. In *insert mode*, tracks can be independently selected for recording. Most VCRs are in assemble mode by default. Use the **set** (**MCI_SET**) command to change these modes.

Recording and Editing

The **record** (**MCI_RECORD**) command provides simple recording and is accurate to approximately 1 second of the starting position. To record more accurately or if you expect to edit the video content while simultaneously operating multiple decks, you should use the **cue** (**MCI_CUE**) command.

The **cue** command prepares the device for recording or playing. Use the **cue** "input" command to prepare the device for recording. The **cue** command is required because an application must know when the device is ready to perform the command (and because it can take several minutes to prepare for a **play** (**MCI_PLAY**) or **record** command).

The VCR prepares itself for recording or playing by seeking to the *in-point*, which is the current position or the position specified by using the **cue** "from" command. If the "preroll" flag is specified with the **cue** command, however, the VCR positions itself the preroll distance from the in-point. The "preroll" flag also indicates that the VCR uses any applicable editing mode, so it's important that you use "preroll", especially when you want the most accurate recording. (Use the **capability** (**MCI_GETDEVCAPS**) command with the "can preroll" flag to check whether preroll is supported.)

Note When you record using "from" and "to" positions, the "from" position is included in the edit, and the "to" position is not.

For more information about recording, see "Recording" earlier in this chapter.

Using the Clock While Editing

When editing, you might want to record segments from one VCR to another. You can begin recording at a specific time and position on one VCR while another begins playing at the same time and position by specifying an action (play or record), a position, and a time for each VCR.

Both VCRs must use the same clock for this type of editing; the clock helps synchronize both devices. You can determine if two VCRs share the same clock by using the **status** (**MCI_STATUS**) command with the "clock id" flag to query each VCR. If the identification numbers returned by the **status** command are the same, the devices use the same clock. As a shared resource, the clock can be connected to multiple VCRs. The VISCA driver supports only one shared clock.

You can also determine clock resolution by using the **status** "clock increment rate" command. This command returns the number of increments the clock supports per second. For example, if the clock is updated every millisecond, the command returns 1000 as the clock increment rate. The advantage of using the increment rate is that the rate is expressed as an integer; otherwise, the increment would be a (single- or double-precision) floating-point value. As an integer, manipulating the increment rate is a simple operation and is not susceptible to rounding errors. You can reset the clock by using the **set** (**MCI_SET**) command with the "clock 0" (zero) flag.

When issuing a **play** (**MCI_PLAY**), **record** (**MCI_RECORD**), or **seek** (**MCI_SEEK**) command, you can specify when the command is to be executed. The characteristics of the VCRs being used determine when to start each VCR. The timing must account for the amount of preroll each device requires and the amount of time needed to complete the MCI commands used to set up the edit session. To do this, retrieve the clock time and add a waiting interval of 5 to 10 seconds. (The waiting interval must be long enough to let the preroll and any outstanding MCI commands complete.)

To ensure that the waiting period is long enough, place the **record** command last in your application and check the time immediately before it. If the interval is too short, restart the **play** command. Alternatively, you could check the time immediately after the last command of the script to verify that there is enough time to send and complete all of the commands.

Using the "Wait", "Notify", and "Test" Flags

Most MCI commands include flags that modify the command. The "wait" (MCI_WAIT) and "notify" (MCI_NOTIFY) flags are common to every command. The "test" (MCI_TEST) flag is available to digital-video and VCR devices. This section describes the use of these flags.

The "Wait" Flag

MCI commands usually return to the user immediately, even if it takes several minutes to complete the action initiated by the command. You can use the "wait" (MCI_WAIT) flag to direct the device to wait until the requested action is completed before returning control to the application.

For example, the following **play** command will not return control to the application until the playback completes:

```
mciSendString("play mydevice from 0 to 100 wait",
    lpszReturnString, lstrlen(lpszReturnString), NULL);
```

Note The user can cancel a wait operation by pressing a break key. By default, this key is CTRL+BREAK. Applications can redefine this key by using the **break** (**MCI_BREAK**) command. (**MCI_BREAK** uses the **MCI_BREAK_PARMS** structure.) When a wait operation is canceled, MCI attempts to return control to the application without interrupting the command associated with the "wait" flag.

The "Notify" Flag

The "notify" (MCI_NOTIFY) flag directs the device to post an MM_MCINOTIFY message when the device completes an action. Your application must have a window procedure to process the MM_MCINOTIFY message for notification to have any effect. An MM_MCINOTIFY message indicates that the processing of a command has completed, but it does not indicate whether the command was completed successfully, failed, or was superseded or aborted.

The application specifies the handle to the destination window for the message when it issues a command. In the command-string interface, this handle is the last parameter of the **mciSendString** function. In the command-message interface, the handle is specified in the low-order word of the **dwCallBack** member of the structure sent with the command message. (Every structure associated with a command message contains this member.)

The "Test" Flag

The "test" (MCI_TEST) flag queries the device to determine if it can execute the command. The device returns an error if it cannot execute the command. It returns no error if it can handle the command. When you specify this flag, MCI returns control to the application without executing the command.

This flag is supported by digital-video and VCR devices for all commands except **open** (**MCI_OPEN**) and **close** (**MCI_CLOSE**).

Using Command Shortcuts and Variations

You can use several shortcuts when working with MCI commands. These shortcuts enable you to use a single identifier to refer to all the devices your application has opened, or to open a device without explicitly issuing an **open** (**MCI_OPEN**) command.

Using "All" as a Device Identifier

You can specify "all" (MCI_ALL_DEVICE_ID) as a device identifier for any command that does not return information. When you specify "all", MCI sends the command sequentially to all devices opened by the current application.

For example, the **close** "all" command closes all open devices and the **play** "all" command starts playing all devices opened by the application. Because MCI sequentially sends the commands to the MCI devices, there is an interval between when the first and last devices receive the command.

Using "all" is a convenient way to broadcast a command to all your devices, but you should not rely on it to synchronize devices; the timing between messages can vary.

Automatically Opening a Device

When you issue a command and specify a device that is not open, MCI tries to open the device before implementing the command. The following rules apply to automatically opening devices :

- Automatic open works only with the command-string interface.
- Automatic open fails for commands that are specific to custom device drivers.
- Automatically opened devices do not respond to commands that use "all" as a device name.
- Automatic open does not let your application specify the "type" flag. Without the device name, MCI determines the device name from the entries in the registry. To use a specific device, you can combine the device name with the filename by using the exclamation point, as described in the reference material for the **open** command.

If an application uses automatic open to open a device, the application should check the return value of every subsequent **open** command to verify that the device is still open. MCI also automatically closes any device that it automatically opens. MCI typically closes a device in the following situations:

- The command is completed.
- You abort the command.
- You request notification in a subsequent command.
- MCI detects a failure.

MCI Functions, Macros, and Messages

Most MCI applications use the **mciSendString** and **mciSendCommand** functions dozens of times. MCI provides some other useful functions that your application will use less frequently.

The device identifier that is required by most MCI commands is typically retrieved in a call to the **open** (**MCI_OPEN**) command. If you need a device identifier but do not want to open the device – for example, if you want to query the capabilities of the device before taking any other action – you can call the **mciGetDeviceID** function.

The **mciGetCreatorTask** function allows your application to use a device identifier to retrieve a handle to the task that created that identifier.

You can use the **mciGetYieldProc** and **mciSetYieldProc** functions to assign and retrieve the address of the callback function associated with the "wait" (MCI_WAIT) flag.

The **mciGetErrorString** function retrieves a string that describes an MCI error value. Each string that MCI returns, whether data or an error description, is a maximum of 128 characters. Dialog box fields that are smaller than 128 characters will truncate the longer strings returned by MCI. For more information about these strings, see "Constants: MCIERR Return Values" later in this chapter.

The MCI macros are tools you can use to create and disassemble values that specify time formats. These time formats are used in many MCI commands. The formats acted on by the macros are hours/minutes/seconds (HMS), minutes/seconds/frames (MSF), and tracks/minutes/seconds/frames (TMSF). The following table lists the macros and their descriptions:

Macro	Description
MCI_HMS_HOUR	Retrieves the hours component from an HMS value.
MCI_HMS_MINUTE	Retrieves the minutes component from an HMS value.
MCI_HMS_SECOND	Retrieves the seconds component from an HMS value.
MCI_MAKE_HMS	Creates an HMS value.
MCI_MAKE_MSF	Creates an MSF value.
MCI_MAKE_TMSF	Creates a TMSF value.
MCI_MSF_FRAME	Retrieves the frames component from an MSF value.
MCI_MSF_MINUTE	Retrieves the minutes component from an MSF value.
MCI_MSF_SECOND	Retrieves the seconds component from an MSF value.
MCI_TMSF_FRAME	Retrieves the frames component from a TMSF value.
MCI_TMSF_MINUTE	Retrieves the minutes component from a TMSF value.
MCI_TMSF_SECOND	Retrieves the seconds component from a TMSF value.
MCI_TMSF_TRACK	Retrieves the tracks component from a TMSF value.

MCI also provides two messages: MM_MCINOTIFY and MM_MCISIGNAL. MM_MCINOTIFY notifies an application of the outcome of an MCI command whenever that command specifies the "notify" (MCI_NOTIFY) flag. MM_MCISIGNAL is specific to digital-video devices; it notifies the application

when a specified position is reached.

Device-Specific Command Sets

This section lists the commands supported by each device type.

Animation Command Set

Animation devices support the following set of commands:

String form	Message form
break	MCI_BREAK
capability	MCI_GETDEVCAPS
close	MCI_CLOSE
info	MCI_INFO
open	MCI_OPEN
pause	MCI_PAUSE
play	MCI_PLAY
put	MCI_PUT
realize	MCI_REALIZE
resume	MCI_RESUME
seek	MCI_SEEK
set	MCI_SET
status	MCI_STATUS
step	MCI_STEP
stop	MCI_STOP
sysinfo	MCI_SYSINFO
update	MCI_UPDATE
where	MCI_WHERE
window	MCI_WINDOW

CD Audio Command Set

CD audio devices support the following set of commands:

String form	Message form
break	MCI_BREAK
capability	MCI_GETDEVCAPS
close	MCI_CLOSE
info	MCI_INFO
open	MCI_OPEN
pause	MCI_PAUSE
play	MCI_PLAY
resume	MCI_RESUME
seek	MCI_SEEK
set	MCI_SET
status	MCI_STATUS
stop	MCI_STOP
sysinfo	MCI_SYSINFO

Digital-Video Command Set

Digital-video devices (for example, the MCIAVI driver) support the following set of commands:

String form	Message form
break	MCI_BREAK
capability	MCI_GETDEVCAPS
capture	MCI_CAPTURE
close	MCI_CLOSE
configure	MCI_CONFIGURE
copy	MCI_COPY
cue	MCI_CUE
cut	MCI_CUT
delete	MCI_DELETE
freeze	MCI_FREEZE
info	MCI_INFO
list	MCI_LIST
load	MCI_LOAD
monitor	MCI_MONITOR
open	MCI_OPEN
paste	MCI_PASTE
pause	MCI_PAUSE
play	MCI_PLAY
put	MCI_PUT
quality	MCI_QUALITY
realize	MCI_REALIZE
record	MCI_RECORD
reserve	MCI_RESERVE
restore	MCI_RESTORE
resume	MCI_RESUME
save	MCI_SAVE
seek	MCI_SEEK
set	MCI_SET
setaudio	MCI_SETAUDIO
setvideo	MCI_SETVIDEO
signal	MCI_SIGNAL
status	MCI_STATUS
step	MCI_STEP
stop	MCI_STOP
sysinfo	MCI_SYSINFO
undo	MCI_UNDO
unfreeze	MCI_UNFREEZE
update	MCI_UPDATE
where	MCI_WHERE
window	MCI_WINDOW

MIDI Sequencer Command Set

The MIDI sequencer supports the following set of commands:

String form	Message form
break	MCI_BREAK
capability	MCI_GETDEVCAPS
close	MCI_CLOSE
info	MCI_INFO
open	MCI_OPEN
pause	MCI_PAUSE
play	MCI_PLAY
record	MCI_RECORD
resume	MCI_RESUME
save	MCI_SAVE
seek	MCI_SEEK
set	MCI_SET
status	MCI_STATUS
stop	MCI_STOP
sysinfo	MCI_SYSINFO

VCR Command Set

VCRs support the following set of commands:

String form	Message form
break	MCI_BREAK
capability	MCI_GETDEVCAPS
cue	MCI_CUE
freeze	MCI_FREEZE
index	MCI_INDEX
info	MCI_INFO
list	MCI_LIST
mark	MCI_MARK
pause	MCI_PAUSE
play	MCI_PLAY
record	MCI_RECORD
resume	MCI_RESUME
seek	MCI_SEEK
set	MCI_SET
setaudio	MCI_SETAUDIO
settimecode	MCI_SETTIMECODE
settuner	MCI_SETTUNER
setvideo	MCI_SETVIDEO
status	MCI_STATUS
step	MCI_STEP
stop	MCI_STOP
sysinfo	MCI_SYSINFO
unfreeze	MCI_UNFREEZE

Videodisc Command Set

Videodisc devices support the following set of commands:

String form	Message form
break	MCI_BREAK
capability	MCI_GETDEVCAPS
close	MCI_CLOSE
escape	MCI_ESCAPE
info	MCI_INFO
open	MCI_OPEN
pause	MCI_PAUSE
play	MCI_PLAY
resume	MCI_RESUME
seek	MCI_SEEK
set	MCI_SET
spin	MCI_SPIN
status	MCI_STATUS
step	MCI_STEP
stop	MCI_STOP
sysinfo	MCI_SYSINFO

Video-Overlay Command Set

Video-overlay devices support the following set of commands:

String form	Message form
break	MCI_BREAK
capability	MCI_GETDEVCAPS
close	MCI_CLOSE
freeze	MCI_FREEZE
info	MCI_INFO
load	MCI_LOAD
open	MCI_OPEN
put	MCI_PUT
save	MCI_SAVE
set	MCI_SET
status	MCI_STATUS
sysinfo	MCI_SYSINFO
unfreeze	MCI_UNFREEZE
where	MCI_WHERE
window	MCI_WINDOW

Waveform-Audio Command Set

Waveform-audio devices support the following set of commands:

String form	Message form
break	MCI_BREAK
capability	MCI_GETDEVCAPS
close	MCI_CLOSE
cue	MCI_CUE
delete	MCI_DELETE
info	MCI_INFO
open	MCI_OPEN
pause	MCI_PAUSE
play	MCI_PLAY
record	MCI_RECORD
resume	MCI_RESUME
save	MCI_SAVE
seek	MCI_SEEK
set	MCI_SET
status	MCI_STATUS
stop	MCI_STOP
sysinfo	MCI_SYSINFO

MCI Reference

This section describes the MCI functions, macros, messages, and error values. These elements are grouped as follows.

Notifications

MM_MCINOTIFY
MM_MCISIGNAL

Retrieving Information

mciGetCreatorTask
mciGetDeviceID
mciGetErrorString

Sending Commands

mciSendCommand
mciSendString

Time Formats

MCI_HMS_HOUR
MCI_HMS_MINUTE
MCI_HMS_SECOND
MCI_MAKE_HMS
MCI_MAKE_MSF
MCI_MAKE_TMSF
MCI_MSF_FRAME
MCI_MSF_MINUTE
MCI_MSF_SECOND
MCI_TMSF_FRAME
MCI_TMSF_MINUTE
MCI_TMSF_SECOND
MCI_TMSF_TRACK

Yield Procedures

mciGetYieldProc
mciSetYieldProc

mciGetCreatorTask

```
HANDLE mciGetCreatorTask(MCIDEVICEID IDDevice);
```

Retrieves a handle to the creator task for the specified device.

- Returns the handle of the creator task responsible for opening the device if successful. If the device identifier is invalid, the return value is NULL.

IDDevice
Device for which the creator task is returned.

mciGetDeviceID

```
MCIDEVICEID mciGetDeviceID(LPCTSTR lpszDevice);
```

Retrieves the device identifier corresponding to the name of an open device.

- Returns the device identifier assigned to the device when it was opened if successful. The identifier is used in the **mciSendCommand** function. If the device name is not known, if the device is not open, or if there was not enough memory to complete the operation, the return value is zero.

lpszDevice
Address of a null-terminated string that specifies the device name or the alias name by which the device is known.

Each file for a compound device has a unique device identifier. The identifier of the "all" device is MCI_ALL_DEVICE_ID.

mciGetErrorString

```
BOOL mciGetErrorString(DWORD fdwError, LPTSTR lpszErrorText,
    UINT cchErrorText);
```

Retrieves a string that describes the specified MCI error code.

- Returns TRUE if successful or FALSE if the error code is not known.

fdwError
Error code returned by the **mciSendCommand** or **mciSendString** function.

lpszErrorText
Address of a buffer that receives a null-terminated string describing the specified error.

cchErrorText
Length of the buffer, in characters, pointed to by the *lpszErrorText* parameter.

Each string that MCI returns, whether data or an error description, can be a maximum of 128 characters.

mciGetYieldProc

```
YIELDPROC mciGetYieldProc(MCIDEVICEID IDDevice, LPDWORD lpdwYieldData);
```

Retrieves the address of the callback function associated with the "wait" (MCI_WAIT) flag. The callback function is called periodically while an MCI device waits for a command specified with the "wait" flag to finish.

- Returns the address of the current yield callback function if successful or NULL if the device identifier is invalid.

IDDevice
MCI device being monitored (the device performing an MCI command).

lpdwYieldData
Address of a buffer containing yield data to be passed to the callback function. This parameter can be NULL if there is no yield data.

mciSendCommand

```
MCIERROR mciSendCommand(MCIDEVICEID IDDevice, UINT uMsg,
    DWORD fdwCommand, DWORD dwParam);
```

Sends a command message to the specified MCI device.

- Returns zero if successful or an error otherwise. The low-order word of the returned doubleword value contains the error return value. If the error is device-specific, the high-order word of the return value is the driver identifier; otherwise, the high-order word is zero. For a list of possible return values, see "Constants: MCIERR Return Values" later in this chapter.

  To retrieve a text description of **mciSendCommand** return values, pass the return value to the **mciGetErrorString** function.

IDDevice
Device identifier of the MCI device that is to receive the command message. This parameter is not used with the **MCI_OPEN** command message.

uMsg
Command message. For information about command messages, see Chapter 5, "MCI Command Messages."

fdwCommand
Flags for the command message.

dwParam
Address of a structure that contains parameters for the command message.

Error values that are returned when a device is being opened are listed with the **MCI_OPEN** command message. In addition to the **MCI_OPEN** error return values, this function can return the values listed in "Constants: MCIERR Return Values" later in this chapter.

Use **MCI_OPEN** to obtain the device identifier specified by the *IDDevice* parameter.

mciSendString

```
MCIERROR mciSendString(LPCTSTR lpszCommand, LPTSTR lpszReturnString,
    UINT cchReturn, HANDLE hwndCallback);
```

Sends a command string to an MCI device. The device that the command is sent to is specified in the command string.

- Returns zero if successful or an error otherwise. The low-order word of the returned doubleword value contains the error return value. If the error is device-specific, the high-order word of the return value is the driver identifier; otherwise, the high-order word is zero. For a list of possible error values, see "Constants: MCIERR Return Values" later in this chapter.

  To retrieve a text description of **mciSendString** return values, pass the return value to the **mciGetErrorString** function.

lpszCommand
Address of a null-terminated string that specifies an MCI command string. For more information about the command strings, see Chapter 4, "MCI Command Strings."

lpszReturnString
Address of a buffer that receives return information. If no return information is needed, this parameter can be NULL.

cchReturn
Size, in characters, of the return buffer specified by the *lpszReturnString* parameter.

hwndCallback
Handle of a callback window if the "notify" flag was specified in the command string.

mciSetYieldProc

```
UINT mciSetYieldProc(MCIDEVICEID IDDevice, YIELDPROC yp,
    DWORD dwYieldData);
```

Sets the address of a procedure to be called periodically when an MCI device is waiting for a command to finish because the "wait" (MCI_WAIT) flag was specified.

- Returns TRUE if successful or FALSE otherwise.

IDDevice
Identifier of the device to assign a procedure to.

yp
Address of the procedure to call when yielding for the specified device. If this parameter is NULL, the function disables any existing yield procedure.

dwYieldData
Data to be sent to the yield procedure when it is called for the specified device.

This function overrides any previous yield procedure for this device.

MCI_HMS_HOUR

```
BYTE MCI_HMS_HOUR(DWORD dwHMS)
```

Retrieves the hours component from a parameter containing packed hours/minutes/seconds (HMS) information.

- Returns the hours component of the specified HMS information.

dwHMS
Time in HMS format.

Time in HMS format is expressed as a doubleword value 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.

The **MCI_HMS_HOUR** macro is defined as follows:

```
#define MCI_HMS_HOUR(hms) ((BYTE)(hms))
```

MCI_HMS_MINUTE

```
BYTE MCI_HMS_MINUTE(DWORD dwHMS)
```

Retrieves the minutes component from a parameter containing packed hours/minutes/seconds (HMS) information.

- Returns the minutes component of the specified HMS information.

dwHMS
Time in HMS format.

Time in HMS format is expressed as a doubleword value 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.

The **MCI_HMS_MINUTE** macro is defined as follows:

```
#define MCI_HMS_MINUTE(hms)  ((BYTE)(((WORD)(hms)) >> 8))
```

MCI_HMS_SECOND

```
BYTE MCI_HMS_SECOND(DWORD dwHMS)
```

Retrieves the seconds component from a parameter containing packed hours/minutes/seconds (HMS) information.

- Returns the seconds component of the specified HMS information.

dwHMS
Time in HMS format.

Time in HMS format is expressed as a doubleword value 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.

The **MCI_HMS_SECOND** macro is defined as follows:

```
#define MCI_HMS_SECOND(hms) ((BYTE)((hms) >> 16))
```

MCI_MAKE_HMS

```
DWORD MCI_MAKE_HMS(BYTE hours, BYTE minutes, BYTE seconds)
```

Creates a time value in packed hours/minutes/seconds (HMS) format from the given hours, minutes, and seconds values.

- Returns the time in packed HMS format.

hours, minutes, and *seconds*
Number of hours, minutes, and seconds.

Time in HMS format is expressed as a doubleword value 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.

The **MCI_MAKE_HMS** macro is defined as follows:

```
#define MCI_MAKE_HMS(h, m, s)  ((DWORD)(((BYTE)(h)  | \
                                ((WORD)(m)  << 8))  | \
                                (((DWORD)(BYTE)(s))  << 16)))
```

MCI_MAKE_MSF

```
DWORD MCI_MAKE_MSF(BYTE minutes, BYTE seconds, BYTE frames)
```

Creates a time value in packed minutes/seconds/frames (MSF) format from the given minutes, seconds, and frame values.

- Returns the time in packed MSF format.

minutes, seconds, and *frames*
Number of minutes, seconds, and frames.

Time in MSF format is expressed as a doubleword value 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.

The **MCI_MAKE_MSF** macro is defined as follows:

```
#define MCI_MAKE_MSF(m, s, f)  ((DWORD)(((BYTE)(m)  | \
                                ((WORD)(s)  << 8))  | \
                                (((DWORD)(BYTE)(f))  << 16)))
```

MCI_MAKE_TMSF

```
DWORD MCI_MAKE_TMSF(BYTE tracks, BYTE minutes, BYTE seconds,
    BYTE frames)
```

Creates a time value in packed tracks/minutes/seconds/frames (TMSF) format from the given tracks, minutes, seconds, and frames values.

- Returns the time in packed TMSF format.

tracks, minutes, seconds, and *frames*
Number of tracks, minutes, seconds, and frames.

Time in TMSF format is expressed as a doubleword value 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.

The **MCI_MAKE_TMSF** macro is defined as follows:

```
#define MCI_MAKE_TMSF(t, m, s, f) ((DWORD)(((BYTE)(t) | \
                                   ((WORD)(m) << 8)) | \
                                   (((DWORD)(BYTE)(s) | \
                                   ((WORD)(f) << 8)) << 16)))
```

MCI_MSF_FRAME

```
BYTE MCI_MSF_FRAME(DWORD dwMSF)
```

Creates the frames component from a parameter containing packed minutes/seconds/frames (MSF) information.

- Returns the frames component of the specified MSF information.

dwMSF
Time in MSF format.

Time in MSF format is expressed as a doubleword value 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.

The **MCI_MSF_FRAME** macro is defined as follows:

```
#define MCI_MSF_FRAME(msf) ((BYTE)((msf) >> 16))
```

MCI_MSF_MINUTE

```
BYTE MCI_MSF_MINUTE(DWORD dwMSF)
```

Retrieves the minutes component from a parameter containing packed minutes/seconds/frames (MSF) information.

- Returns the minutes component of the specified MSF information.

dwMSF
Time in MSF format.

Time in MSF format is expressed as a doubleword value 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.

The **MCI_MSF_MINUTE** macro is defined as follows:

```
#define MCI_MSF_MINUTE(msf) ((BYTE)(msf))
```

MCI_MSF_SECOND

```
BYTE MCI_MSF_SECOND(DWORD dwMSF)
```

Retrieves the seconds component from a parameter containing packed minutes/seconds/frames (MSF) information.

- Returns the seconds component of the specified MSF information.

dwMSF
Time in MSF format.

Time in MSF format is expressed as a doubleword value 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.

The **MCI_MSF_SECOND** macro is defined as follows:

```
#define MCI_MSF_SECOND(msf) ((BYTE)(((WORD)(msf)) >> 8))
```

MCI_TMSF_FRAME

```
BYTE MCI_TMSF_FRAME(DWORD dwTMSF)
```

Retrieves the frames component from a parameter containing packed tracks/minutes/seconds/frames (TMSF) information.

- Returns the frames component of the specified TMSF information.

dwTMSF
Time in TMSF format.

Time in TMSF format is expressed as a doubleword value 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.

The **MCI_TMSF_FRAME** macro is defined as follows:

```
#define MCI_TMSF_FRAME(tmsf) ((BYTE)((tmsf) >> 24))
```

MCI_TMSF_MINUTE

```
BYTE MCI_TMSF_MINUTE(DWORD dwTMSF)
```

Retrieves the minutes component from a parameter containing packed tracks/minutes/seconds/frames (TMSF) information.

- Returns the minutes component of the specified TMSF information.

dwTMSF
Time in TMSF format.

Time in TMSF format is expressed as a doubleword value 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.

The **MCI_TMSF_MINUTE** macro is defined as follows:

```
#define MCI_TMSF_MINUTE(tmsf) ((BYTE)(((WORD)(tmsf)) >> 8))
```

MCI_TMSF_SECOND

```
BYTE MCI_TMSF_SECOND(DWORD dwTMSF)
```

Retrieves the seconds component from a parameter containing packed tracks/minutes/seconds/frames (TMSF) information.

- Returns the seconds component of the specified TMSF information.

dwTMSF
Time in TMSF format.

Time in TMSF format is expressed as a doubleword value 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.

The **MCI_TMSF_SECOND** macro is defined as follows:

```
#define MCI_TMSF_SECOND(tmsf) ((BYTE)((tmsf) >> 16))
```

MCI_TMSF_TRACK

```
BYTE MCI_TMSF_TRACK(DWORD dwTMSF)
```

Retrieves the tracks component from a parameter containing packed tracks/minutes/seconds/frames (TMSF) information.

- Returns the tracks component of the specified TMSF information.

dwTMSF
Time in TMSF format.

Time in TMSF format is expressed as a doubleword value 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.

The **MCI_TMSF_TRACK** macro is defined as follows:

```
#define MCI_TMSF_TRACK(tmsf) ((BYTE)(tmsf))
```

MM_MCINOTIFY

```
MM_MCINOTIFY
wParam = (WPARAM) wFlags
lParam = (LONG) lDevID
```

Notifies an application that an MCI device has completed an operation. MCI devices send this message only when the "notify" (MCI_NOTIFY) flag is used.

- Returns zero if successful or an error otherwise.

wFlags

Reason for the notification. The following values are defined:

MCI_NOTIFY_ABORTED

The device received a command that prevented the current conditions for initiating the callback function from being met. If a new command interrupts the current command and it also requests notification, the device sends this message only and not MCI_NOTIFY_SUPERCEDED.

MCI_NOTIFY_FAILURE

A device error occurred while the device was executing the command.

MCI_NOTIFY_SUCCESSFUL

The conditions initiating the callback function have been met.

MCI_NOTIFY_SUPERSEDED

The device received another command with the "notify" flag set and the current conditions for initiating the callback function have been superseded.

lDevID

Identifier of the device initiating the callback function.

For more information about the "notify" (MCI_NOTIFY) flag, see "The "Notify" Flag" earlier in this chapter.

A device returns the MCI_NOTIFY_SUCCESSFUL flag with MM_MCINOTIFY when the action for a command finishes. For example, a CD audio device uses this flag for notification for the **play** (**MCI_PLAY**) command when the device finishes playing. The **play** command is successful only when it reaches the specified end position or reaches the end of the media. Similarly, the **seek** (**MCI_SEEK**) and **record** (**MCI_RECORD**) commands do not return MCI_NOTIFY_SUCCESSFUL until they reach the specified end position or reach the end of the media.

A device returns the MCI_NOTIFY_ABORTED flag with MM_MCINOTIFY only when it receives a command that prevents it from meeting the notification conditions. For example, the **play** command 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. The **seek** and **record** commands behave similarly. MCI also does not send MCI_NOTIFY_ABORTED when playback or recording is paused with the **pause** (**MCI_PAUSE**) command. Sending the **resume** (**MCI_RESUME**) command allows them to continue to meet the callback conditions.

When your application requests notification for a command, check the error return of the **mciSendString** or **mciSendCommand** functions. If these functions encounter an error and return a nonzero value, MCI will not set notification for the command.

MM_MCISIGNAL

```
MM_MCISIGNAL
wParam = (WPARAM) wID
lParam = (LONG) lUserParm
```

Sent to a window to notify an application that an MCI device has reached a position defined in a previous **signal** (**MCI_SIGNAL**) command.

wID
Identifier of the device initiating the signal message.

lUserParm
Value passed in the **dwUserParm** member of the **MCI_DGV_SIGNAL_PARAMS** structure when the **signal** command has defined this callback function. Alternatively, it might contain the position value.

Constants: MCIERR Return Values

The **mciSendCommand** and **mciSendString** functions return zero if they are successful; otherwise, they return a doubleword value that contains one of the following error values in the low word. You can obtain a description of individual return values by passing the return values to the **mciGetErrorString** function.

General Error Values

The following error values can be returned by either the **mciSendCommand** or **mciSendString** function:

Value	Meaning
MCIERR_BAD_TIME_FORMAT	The specified value for the time format is invalid.
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. Wait 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_FILENAME_REQUIRED	The filename is invalid. Make sure the filename is no longer than eight characters, followed by a period and an extension.
MCIERR_FLAGS_NOT_COMPATIBLE	The specified parameters cannot be used together.
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 given 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_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 (structure) 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 the specified 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 the system is using does not support the specified command.

mciSendString Errors

The following errors are returned by the **mciSendString** function but not by **mciSendCommand**:

Value	Meaning
MCIERR_BAD_CONSTANT	The value specified for a parameter is unknown.
MCIERR_BAD_INTEGER	An integer in the command was invalid or missing.
MCIERR_DUPLICATE_FLAGS	A flag or value was specified twice.
MCIERR_MISSING_COMMAND_STRING	No command was specified.
MCIERR_MISSING_DEVICE_NAME	No device name was specified.
MCIERR_MISSING_STRING_ARGUMENT	A string value was missing from the command.
MCIERR_NEW_REQUIRES_ALIAS	An alias must be used with the "new" device name.
MCIERR_NO_CLOSING_QUOTE	A closing quotation mark is missing.
MCIERR_NOTIFY_ON_AUTO_OPEN	The "notify" flag is illegal with auto-open.
MCIERR_PARAM_OVERFLOW	The output string was not long enough.
MCIERR_PARSER_INTERNAL	An internal parser error occurred.
MCIERR_UNRECOGNIZED_KEYWORD	An unknown command parameter was specified.

Digital-Video Errors

The following additional return values are defined for digital-video devices:

Value	Meaning
MCIAVI_PRODUCTNAME	Video
MCIERR_AVI_AUDIOERROR	Unknown error while attempting to play audio.
MCIERR_AVI_BADPALETTE	Unable to switch to new palette.
MCIERR_AVI_CANTPLAYFULLSCREEN	This AVI file cannot be played in full screen mode.
MCIERR_AVI_DISPLAYERROR	Unknown error while attempting to display video.
MCIERR_AVI_NOCOMPRESSOR	Can't locate installable compressor needed to play this file.
MCIERR_AVI_NODISPDIB	256 color VGA mode not available.
MCIERR_AVI_NOTINTERLEAVED	This AVI file is not interleaved.
MCIERR_AVI_OLDAVIFORMAT	This AVI file is of an obsolete format.
MCIERR_AVI_TOOBIGFORVGA	This AVI file is too big to be played in the selected VGA mode.

Sequencer Errors

The following additional return values are defined for MCI sequencers:

Value	Meaning
MCIERR_SEQ_DIV_INCOMPATIBLE	The time formats of the "song pointer" and SMPTE are singular. 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 from the Control Panel to edit the setup.
MCIERR_SEQ_PORT_MISCERROR	An error occurred with 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.

Waveform-Audio Errors

The following additional return values are defined for MCI waveform-audio devices:

Value	Meaning
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 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; then, try again to set the device for playback.
MCIERR_WAVE_SETOUTPUTUNSUITABLE	The device you are using to playback a waveform cannot recognize the data format.

MCI Command Strings

The Media Control Interface (MCI) is a high-level command interface to multimedia devices and resource files. MCI provides standard commands for playing multimedia devices and recording multimedia resource files. MCI commands are a generic interface to multimedia devices.

There are two forms of MCI commands: strings and messages. You can use either or both forms in your MCI application. This chapter documents the command-string interface to MCI. For information about the command-message interface, see Chapter 5, "MCI Command Messages." For an overview of MCI, including information about whether you should use the string interface or the message interface in your application, see Chapter 3, "MCI Overview."

You can send a string command by using the **mciSendString** function, which includes parameters for the string command and a buffer for any returned information. For more information about **mciSendString**, see "Sending Command Strings" later in this chapter.

Syntax of Command Strings

MCI command strings use a consistent verb-object-modifier syntax. Each command string includes a command, a device identifier, and command arguments. Arguments are optional for some commands and required for others.

A command string has the following form:

command device_id arguments

These components contain the following information:

- The *command* specifies an MCI command, such as **open**, **close**, or **play**.
- The *device_id* identifies an instance of an MCI driver. The *device_id* is created when the device is opened.
- The *arguments* specify the flags and variables used by the command. Flags are keywords recognized with the MCI command. Variables are numbers or strings that apply to the MCI command or flag.

  For example, the **play** command uses the arguments "from *position*" and "to *position*" to indicate the positions to start and end playing. You can list the flags used with a command in any order. When you use a flag that has a variable associated with it, you must supply a value for the variable.

  Unspecified (and optional) command arguments assume a default value.

Data Types for Command Variables

You can use the following data types for the variables in a command string:

Data type	Description
Strings	String data types are delimited by leading and trailing white spaces and quotation marks ("). MCI removes single quotation marks from a string. To put a quotation mark in a string, use a set of two quotation marks where you want to embed your quotation mark. To use an empty string, use two quotation marks delimited by leading and trailing white spaces.
Signed long integers	Signed long integer data types are delimited by leading and trailing white spaces. Unless otherwise specified, integers can be positive or negative. If you use negative integers, you should not separate the minus sign and the first digit with a space.
Rectangles	Rectangle data types are an ordered list of four signed short values. White space delimits this data type and separates each integer in the list.

Sending Command Strings

Windows provides two functions to send command strings to devices and to query devices for error information: The **mciSendString** function sends a command string to an MCI device. The **mciGetErrorString** function returns the error string corresponding to an error number.

The **mciSendString** function returns zero if successful. If the function fails, the low-order word of the return value contains an error code. You can pass this error code to **mciGetErrorString** to get a text description of it.

Using MCI Command Strings

This section contains examples demonstrating how to use the MCI command-string interface to perform the following tasks:

- Send a command.
- Open an audio-video interleaved (AVI) file.
- Change the playback state.
- Convert strings.

Sending a Command

The following example function sends the **play** command with the "from" and "to" flags.

```
DWORD PlayFromTo(LPSTR lpstrAlias, DWORD dwFrom, DWORD dwTo)
{
    char    achCommandBuff[128];
    wsprintf(achCommandBuff, "play %s from %u to %u",
        lpstrAlias, dwFrom, dwTo);
    return mciSendString(achCommandBuff, NULL, 0, NULL);
}
```

Opening Multiple AVI Files

If your application opens multiple files, it should include routines such as the following simple functions. The application would use the "initAVI" function during its initialization and the "termAVI" function during its termination.

```
// Initialize the MCIAVI driver. This returns TRUE if OK,
// FALSE on error.

BOOL initAVI(VOID)
{
    // Perform additional initialization before loading first file.
    return mciSendString("open digitalvideo", NULL, 0, NULL) == 0;
}

// Close the MCIAVI driver.
void termAVI(VOID)
{
    mciSendString("close digitalvideo", NULL, 0, NULL);
}
```

Changing the Playback State

The following examples show how to use the **pause**, **resume**, **stop**, and **seek** commands.

```
// Assume the file was opened with the alias 'movie'.

// Pause play.
mciSendString("pause movie", NULL, 0, NULL);

// Resume play.
mciSendString("resume movie", NULL, 0, NULL);

// Stop play.
mciSendString("stop movie", NULL, 0, NULL);

// Seek to the beginning.
mciSendString("seek movie to start", NULL, 0, NULL);
```

The following example shows how to change the seek mode:

```
// Set seek mode with the string interface.
// Assume the file was opened with the alias 'movie'.

void SetSeekMode(BOOL fExact)
{
    if (fExact)
        mciSendString("set movie seek exactly on", NULL, 0, NULL);
    else
        mciSendString("set movie seek exactly off", NULL, 0, NULL);
}
```

Converting Strings

When you use the string interface, all values passed with the command and all return values are text strings, so your application needs conversion routines to translate from variables to strings or back again. The following example retrieves the source rectangle and converts the returned string into rectangle coordinates.

```
void GetSourceRect(LPSTR lpstrAlias, LPRECT lprc)
{
    char achRetBuff[128];
    char achCommandBuff[128];

    // Build the command string.
    wsprintf(achCommandBuff, "where %s source", lpstrAlias);
    SetRectEmpty(lprc);    // clears the RECT

    // Send the command.

    if (mciSendString(achCommandBuff, achRetBuff,
        sizeof(achRetBuff), NULL) == 0){

        // The rectangle is returned as "x y dx dy".
        // Both x and y are 0 because this is the source
        // rectangle. Translate the string into the RECT
        // structure.
        char *p;
        p = achRetBuff;           // point to the return string
        while (*p != ' ') p++;    // go past the x (0)
        while (*p == ' ') p++;    // go past spaces
        while (*p != ' ') p++;    // go past the y (0)
        while (*p == ' ') p++;    // go past spaces

        // Retrieve the width.
        for ( ; *p != ' '; p++)
            lprc->right = (10 * lprc->right) + (*p - '0');

        while (*p == ' ') p++;    // go past spaces

        // Retrieve the height.
        for ( ; *p != ' '; p++)
            lprc->bottom = (10 * lprc->bottom) + (*p - '0');
    }
}
```

Note RECT structures are handled differently in MCI than in other parts of Windows; in MCI, **rc.right** contains the width of the rectangle and **rc.bottom** contains its height. In the string interface, a rectangle is specified as *X1 Y1 X2 Y2*. The coordinates *X1 Y1* specify the upper left corner of the rectangle, and the coordinates *X2 Y2* specify the width and height.

MCI Command String Reference

This section describes the MCI command strings. These elements are grouped as follows.

Configuring a Device

break
configure
escape
index
set
setaudio
settimecode
settuner
setvideo
spin

Controlling Playback

freeze
load
pause
play
resume
stop
unfreeze

Controlling the Position

cue
mark
seek
signal
step

Editing

copy
cut
delete
paste
undo

Opening and Closing

close
open

Realizing a Palette

realize

Repainting a Frame

update

Retrieving Information

capability
info
list
status
sysinfo

Saving

record
save

Video Control

capture
monitor
quality
reserve
restore

Window or Display Rectangles

put
where
window

break

```
wsprintf(lpstrCommand, "break %s %s %s", lpszDeviceID, lpszVirtKey,
    lpszFlags);
```

Specifies a key to abort a command that was invoked using the "wait" flag. This command is an MCI system command; it is interpreted directly by MCI.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszVirtKey
One of the following flags:

on *virtual key code*	Specifies the key that aborts a command that was started using the "wait" flag.
off	Disables the current break key.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

When the break key is enabled and the user presses the key identified by the virtual-key code specified in the *lpsVirtKey* parameter, the device returns control to the application. If possible, the command continues execution.

The following command sets `F2` as the break key for the "mysound" device:

```
break mysound on 113
```

capability

```
wsprintf(lpstrCommand, "capability %s %s %s", lpszDeviceID, lpszRequest,
    lpszFlags);
```

Requests information about a particular capability of a device. All MCI devices recognize this command.

- Returns information in the *lpstrReturnString* parameter of the **mciSendString** function. The information is dependent on the request type.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszRequest
Flag that identifies a device capability. The following table lists device types that recognize the **capability** command and the flags used by each type:

animation	can eject	fast play rate
	can play	has audio
	can record	has video
	can reverse	normal play rate
	can save	slow play rate
	can stretch	uses files
	compound device	uses palettes
	device type	windows
cdaudio	can eject	device type
	can play	has audio
	can record	has video
	can save	uses files
	compound device	
digitalvideo	can eject	compound device
	can freeze	device type
	can lock	has audio
	can play	has still
	can record	has video
	can reverse	maximum play rate
	can save	minimum play rate
	can stretch	uses files
	can stretch input	uses palettes
	can test	windows
overlay	can eject	compound device
	can freeze	device type
	can play	has audio
	can record	has video
	can save	uses files
	can stretch	windows
sequencer	can eject	device type
	can play	has audio
	can record	has video
	can save	uses files
	compound device	
vcr	can detect length	clock increment rate
	can eject	compound device
	can freeze	device type
	can monitor sources	has audio

	can play	has clock
	can preroll	has timecode
	can preview	has video
	can record	number of marks
	can reverse	seek accuracy
	can save	uses files
	can test	
videodisc	can eject	device type
	can play	fast play rate
	can record	has audio
	can reverse	has video
	can save	normal play rate
	CAV	slow play rate
	CLV	uses files
	compound device	
waveaudio	can eject	has audio
	can play	has video
	can record	inputs
	can save	outputs
	compound device	uses files
	device type	

The following table lists the flags that can be specified in the *lpszRequest* parameter and their meanings:

can detect length	Returns TRUE if the device can detect the length of the media.
can eject	Returns TRUE if the device can eject the media.
can freeze	Returns TRUE if the device can freeze data in the frame buffer.
can lock	Returns TRUE if the device can lock data.
can monitor sources	Returns TRUE if the device can pass an input (source) to the monitored output, independent of the current input selection.
can play	Returns TRUE if the device can play.
can preroll	Returns TRUE if the device supports the "preroll" flag with the **cue** command.
can preview	Returns TRUE if the device supports previews.
can record	Returns TRUE if the device supports recording.
can reverse	Returns TRUE if the device can play in reverse.
can save	Returns TRUE if the device can save data.
can stretch	Returns TRUE if the device can stretch frames to fill a given display rectangle.
can stretch input	Returns TRUE if the device can resize an image in the process of digitizing it into the frame buffer.
can test	Returns TRUE if the device recognizes the test keyword.
cav	When combined with other items, this flag specifies that the return information applies to CAV format videodiscs. This is the default if no videodisc is inserted.
clock increment	Returns the number of subdivisions the external clock

rate	supports per second. If the external clock is a millisecond clock, the return value is 1000. If the return value is 0, no clock is supported.
clv	When combined with other items, this flag specifies that the return information applies to CLV format videodiscs.
compound device	Returns TRUE if the device supports an element name (filename).
device type	Returns a device type name, which can be one of the following: **animation** **cdaudio** **dat** **digitalvideo** **other** **overlay** **scanner** **sequencer** **vcr** **videodisc** **waveaudio**
fast play rate	Returns the fast play rate in frames per second, or zero if the device cannot play fast.
has audio	Returns TRUE if the device supports audio playback.
has clock	Returns TRUE if the device has a clock.
has still	Returns TRUE if the device treats files with a single image more efficiently than motion video files.
has timecode	Returns TRUE if the device is capable of supporting timecode, or if it is unknown.
has video	Returns TRUE if the device supports video.
inputs	Returns the total number of input devices.
maximum play rate	Returns the maximum play rate, in frames per second, for the device.
minimum play rate	Returns the minimum play rate, in frames per second, for the device.
normal play rate	Returns the normal play rate, in frames per second, for the device.
number of marks	Returns the maximum number of marks that can be used; zero indicates that marks are unsupported.
outputs	Returns the total number of output devices.
seek accuracy	Returns the expected accuracy of a search in frames; 0 indicates that the device is frame accurate, 1 indicates that the device expects to be within one frame of the indicated seek position, and so on.
slow play rate	Returns the slow play rate in frames per second, or zero if the device cannot play slowly.
uses files	Returns TRUE if the data storage used by a compound device is a file.
uses palettes	Returns TRUE if the device uses palettes.
windows	Returns the number of simultaneous display windows

the device can support.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

The following command returns the device type of the "mysound" device:

```
capability mysound device type
```

capture

```
wsprintf(lpstrCommand, "capture %s %s %s", lpszDeviceID, lpszCapture,
    lpszFlags);
```

Copies the contents of the frame buffer and stores it in the specified file. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszCapture
One or more of the following flags:

as *pathname*	Specifies the destination path and filename for the captured image. This flag is required.
at *rectangle*	Specifies the rectangular region within the frame buffer that the device crops and saves to disk. If omitted, the cropped region defaults to the rectangle specified or defaulted on a previous **put** "source" command for this device instance.

lpszFlags
Can be "wait", "notify", "test", or a combination of these. For more information about these flags, see Chapter 3, "MCI Overview."

This command might fail if the device is currently playing motion video or executing some other resource-intensive operation. If the frame buffer is being updated in real time, the updating momentarily pauses so that a complete image is captured. If the device pauses the updating, there might be a visual or audible effect. If the file format, compression algorithm, and quality levels have not been set, their defaults are used.

close

```
wsprintf(lpstrCommand, "close %s %s", lpszDeviceID, lpszFlags);
```

Closes the device or file and any associated resources. MCI unloads a device when all instances of the device or all files are closed. All MCI devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszFlags
Can be "wait", "notify", or both. For more information about these flags, see Chapter 3, "MCI Overview."

To close all devices opened by your application, specify the "all" device identifier for the *lpszDeviceID* parameter.

The following command closes the "mysound" device:

```
close mysound
```

configure

```
wsprintf(lpstrCommand, "configure %s %s", lpszDeviceID, lpszFlags);
```

Displays a dialog box used to configure the device. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszFlags
Can be "wait", "notify", or "test". For more information about these flags, see Chapter 3, "MCI Overview."

copy

```
wsprintf(lpstrCommand, "copy %s %s %s", lpszDeviceID, lpszItem,
    lpszFlags);
```

Copies data to the clipboard. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszItem
One of the following flags identifying the item to copy:

at *rectangle*	Specifies the portion of each frame that will be copied. If omitted, the default setting is the entire frame.
audio stream *stream*	Specifies the audio stream in the workspace affected by the command. If you use this flag and also want to copy video, you must also use the "video stream" flag. (If neither flag is specified, all audio and video streams are copied.)
from *position*	Specifies the start of the range copied. If omitted, the default setting is the current position.
to *position*	Specifies the end of the range copied. The audio and video data copied are exclusive of this position. If omitted, the default setting is the end of the workspace.
video stream *stream*	Specifies the video stream in the workspace affected by the command. If you use this flag and also want to copy audio, you must also use the "audio stream" flag. (If neither flag is specified, all audio and video streams are copied.)

lpszFlags
Can be "wait", "notify", "test", or a combination of these. For more information about these flags, see Chapter 3, "MCI Overview."

cue

```
wsprintf(lpstrCommand, "cue %s %s %s", lpszDeviceID, lpszInOutTo,
    lpszFlags);
```

Prepares for playing or recording. Digital-video, VCR, and waveform-audio devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszInOutTo
Flag that prepares a device for playing or recording. The following table lists device types that recognize the **cue** command and the flags used by each type:

digitalvideo	input noshow	output to *position*
vcr	from *position* input output	preroll reverse to *position*
waveaudio	input	output

The following table lists the flags that can be specified in the *lpszInOutTo* parameter and their meanings:

from *position*	Indicates where to start.
input	Prepares for recording. For digital-video devices, this flag can be omitted if the current presentation source is already the external input.
noshow	Prepares for playing a frame without displaying it. When this flag is specified, the display continues to show the image in the frame buffer even though its corresponding frame is not the current position. A subsequent **cue** command without this flag and without the "to" flag displays the current frame.
output	Prepares for playing. If neither "input" nor "output" is specified, the default setting is "output".
preroll	Moves the preroll distance from the *in-point*. The in-point is the current position, or the position specified by the "from" flag.
reverse	Indicates play direction is in reverse (backward).
to *position*	Moves the workspace to the specified position. For VCR devices, this flag indicates where to stop.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

Although it is not necessary, issuing the **cue** command before playing or recording on some devices might reduce the delay before the device starts the action.

This command fails if playing or recording is in progress or if the device is paused.

When cueing for playback (using **cue** "output"), issuing the **play** command with the "from", "to", or "reverse" flag cancels the **cue** command.

When cueing for recording (using **cue** "input"), issuing the **record** command with the "from", "to", or "initialize" flag cancels the **cue** command.

The following command prepares the "mysound" device for recording:

```
cue mysound input
```

cut

```
wsprintf(lpstrCommand, "cut %s %s %s", lpszDeviceID, lpszItem,
    lpszFlags);
```

Removes data from the workspace and copies it to the clipboard. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszItem
One of the following flags identifying the item to cut:

at *rectangle*	Specifies the portion of each frame cut. If omitted, it defaults to the entire frame. When this item is specified, frames are not deleted. Instead the area inside the rectangle becomes black.
audio stream *stream*	Specifies the audio stream in the workspace affected by the command. If you use this flag and also want to cut video, you must also use the "video stream" flag. (If neither flag is specified, all audio and video streams are cut.)
from *position*	Specifies the start of the range cut. If omitted, it defaults to the current position.
to *position*	Specifies the end of the range cut. The audio and video data cut are exclusive of this position. If omitted it defaults to the end of the workspace.
video stream *stream*	Specifies the video stream in the workspace affected by the command. If you use this flag and also want to cut audio, you must also use the "audio stream" flag. (If neither flag is specified, all audio and video streams are cut.)

lpszFlags
Can be "wait", "notify", "test", or a combination of these. For more information about these flags, see Chapter 3, "MCI Overview."

The change becomes permanent only when the data is explicitly saved; however, playback acts as if the data has been removed.

delete

```
wsprintf(lpstrCommand, "delete %s %s %s", lpszDeviceID, lpszPosition,
    lpszFlags);
```

Deletes a data segment from a file. Digital-video and waveform-audio devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszPosition
Flag that identifies a data segment to delete. The following table lists device types that recognize the **delete** command and the flags used by each type:

digitalvideo	at *rectangle* audio stream *stream* from *position*	to *position* video stream *stream*
waveaudio	from *position*	to *position*

The following table lists the flags that can be specified in the *lpszPosition* parameter and their meanings:

at *rectangle*	Specifies the portion of each frame deleted. If omitted, it defaults to the entire frame. When this item is specified, frames are not deleted. Instead the area inside the rectangle becomes black.
audio stream *stream*	Specifies the audio stream in the workspace affected by the command. If you use this flag and also want to delete video, you must also use the "video stream" flag. (If neither flag is specified, all audio and video streams are deleted.)
from *position*	Specifies the position at which deletion begins. If this flag is omitted, the deletion begins at the current position.
to *position*	Specifies the position at which deletion ends. If this flag is omitted, the deletion continues to the end of the content or workspace.
video stream *stream*	Specifies the video stream in the workspace affected by the command. If you use this flag and also want to delete audio, you must also use "audio stream" flag. (If neither flag is specified, all audio and video streams are deleted.)

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

Before issuing any commands that use position values, you should set the desired time format by using the **set** command.

The following command deletes the waveform-audio data from 1 millisecond through 900 milliseconds (assuming the time format is set to milliseconds):

```
delete mysound from 1 to 900
```

escape

```
wsprintf(lpstrCommand, "escape %s %s %s", lpszDeviceID, lpszEscape,
    lpszFlags);
```

Sends device-specific information to a device. Videodisc devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszEscape
Custom information to send to the device.

lpszFlags
Can be "wait", "notify", or both. For more information about these flags, see Chapter 3, "MCI Overview."

The following command sends the escape string "SA" to the videodisc device:

```
escape videodisc SA
```

freeze

```
wsprintf(lpstrCommand, "freeze %s %s %s", lpszDeviceID, lpszFreezeFlags,
    lpszFlags);
```

Freezes video input or video output on a VCR or disables video acquisition to the frame buffer. Digital-video, video-overlay, and VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszFreezeFlags
Flag that identifies what to freeze. The following table lists device types that recognize the **freeze** command and the flags used by each type:

digitalvideo	at *rectangle*	outside
overlay	at *rectangle*	
vcr	field frame	input output

The following table lists the flags that can be specified in the *lpszFreezeFlags* parameter and their meanings:

at *rectangle*	Specifies the region that will be frozen. For video-overlay devices, this region will have video acquisition disabled. For digital-video devices, the pixels within the rectangle will have their lock mask bit turned on (unless the "outside" flag is specified). The rectangle is relative to the video buffer origin and is specified as *X1 Y1 X2 Y2*. The coordinates *X1 Y1* specify the upper left corner of the rectangle, and the coordinates *X2 Y2* specify the width and height.
field	Freezes the first field. Field is assumed by default (if neither frame nor field is specified).
frame	Freezes the entire frame, displaying both fields.
input	Freezes the current frame of the input image, whether it is paused or running.
output	Freezes the current frame of the output from the VCR. If the VCR is playing when **freeze** is issued, the current frame is frozen and the VCR is paused. If the VCR is paused when this command is issued, the current frame is frozen. The frozen image remains on the output device until an **unfreeze** command is issued. If neither "input" nor "output" is specified, "output" is assumed.
outside	Indicates that the area outside the region specified using the "at" flag is frozen.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

When used with VCR devices, this command is intended for frame-grabbing cards.

To specify irregular acquisition regions with the "at" flag, use a series of **freeze** and **unfreeze**

commands. Some video-overlay devices limit the complexity of the acquisition region.

This command is supported only if a call to the **capability** command with the "can freeze" flag returns TRUE.

The following command disables video acquisition in a 100-pixel square at the upper left corner of the video buffer:

```
freeze vboard at 0 0 100 100
```

index

```
wsprintf(lpstrCommand, "index %s %s %s", lpszDeviceID, lpszIndex,
    lpszFlags);
```

Controls a VCR's on-screen display. VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszIndex
One of the following flags:

off	Turns off the on-screen display.
on	Turns on the on-screen display. The item to be displayed is specified by the "index" flag of the **set** command.

lpszFlags
Can be "wait", "notify", or "test". For more information about these flags, see Chapter 3, "MCI Overview."

info

```
wsprintf(lpstrCommand, "info %s %s %s", lpszDeviceID, lpszInfoType,
    lpszFlags);
```

Retrieves a hardware description from a device. All MCI devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszInfoType
Flag that identifies the type of information required. The following table lists device types that recognize the **info** command and the flags used by each type:

animation	file product	window text
cdaudio	info identity info upc	product
digitalvideo	audio algorithm audio quality file product still algorithm still quality	usage version video algorithm video quality window text
overlay	file product	window text
sequencer	copyright file	name product
vcr	product	version
videodisc	product	
waveaudio	file input	output product

The following table lists the flags that can be specified in the *lpszInfoType* parameter and their meanings:

audio algorithm	Returns the name of the current audio compression algorithm.
audio quality	Returns the name for the current audio quality descriptor. This might return "unknown" if the application has set parameters to specific values that do not correspond to defined qualities.
copyright	Retrieves the MIDI file copyright notice from the copyright meta event.
file	Retrieves the name of the file used by the compound device. If the device is opened without a file and the **load** command has not been used, a null string is returned.
info identity	Produces a unique identifier for the audio CD currently loaded in the player being queried.
info upc	Produces the Universal Product Code (UPC) that is encoded on an audio CD. The UPC is a string of digits.

	It might not be available for all CDs.
input	Retrieves the description of the current input device. Returns "none" if an input device is not set.
name	Retrieves the sequence name from the sequence/track name meta event.
output	Retrieves the description of the current output device. Returns "none" if an output device is not set.
product	Retrieves a description of the device. This information often includes the product name and model. The string length will be 31 characters or fewer.
still algorithm	Returns the name of the current still image compression algorithm.
still quality	Returns the name for the current still image quality descriptor. This might return "unknown" if the application has set parameters to specific values that do not correspond to defined qualities.
usage	Returns a string describing usage restrictions that might be imposed by the owner of the visual or audio data in the workspace.
version	Returns the release level of the device driver and hardware.
video algorithm	Returns the name of the current video compression algorithm.
video quality	Returns the name for the current video quality descriptor. This might return "unknown" if the application has set parameters to specific values that do not correspond to defined qualities.
window text	Retrieves the caption of the window used by the device.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

The following command retrieves a description of the hardware associated with the "mysound" device:

```
info mysound product
```

list

```
wsprintf(lpstrCommand, "list %s %s %s", lpszDeviceID, lpszList,
    lpszFlags);
```

Determines the number and types of video and audio inputs. Digital-video and VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszList
Flag that identifies the number and types of video and audio inputs. The following table lists device types that recognize the **list** command and the flags used by each type:

digitalvideo	audio algorithm audio quality algorithm *algorithm* audio stream count number *index*	still algorithm still quality algorithm *algorithm* video algorithm video quality algorithm *algorithm* video source video stream
vcr	audio source count audio source number *index*	video source count video source number *index*

The following table lists the flags that can be specified in the *lpszList* parameter and their meanings:

audio algorithm	Specifies the command should retrieve audio algorithm names.
audio quality algorithm *algorithm*	Specifies the command should retrieve quality levels associated with the specified *algorithm*. If *algorithm* is "current", the quality level of the current algorithm is returned.
audio source count	Returns the total number of audio inputs.
audio source number *index*	Returns the type of audio input of source *index*.
audio stream	Specifies the command should retrieve the names of the audio streams associated with the workspace. These strings (such as "English" or "German") are embedded in the file and identify the stream.
count	Returns the number of options of the specified type.
number *index*	Returns a string describing a specific option (as identified by *index*) of the specified option type. *Index* must be an integer between 1 and the value returned by "count".
still algorithm	Specifies the command should retrieve still algorithm names.
still quality algorithm	Specifies the command should retrieve

algorithm	quality levels associated with the specified still *algorithm*. If *algorithm* is "current", the quality level of the current algorithm is returned.
video algorithm	Specifies the command should retrieve video algorithm names.
video quality algorithm *algorithm*	Specifies the command should retrieve quality levels associated with the specified video *algorithm*. If *algorithm* is "current", the quality level of the current algorithm is returned.
video source	Specifies the command should return information about the video sources. When used with the "count" flag, it returns the number of video sources. When used with the "number" flag, it returns the type of a video source. MCI defines the following constants for type: "ntsc", "rgb", "pal", "secam", "svideo", and "generic". There might be more than one source of each type returned. The "generic" source type is used when more than one signal is allowed for that connector.
video source count	Returns total number of video inputs.
video source number *index*	Returns the type of video input of source *index*.
video stream	Specifies the command should retrieve the names of video streams associated with the workspace. These strings (such as "funny ending" or "sad ending") are embedded in the file and identify the stream.

lpszFlags
Can be "wait", "notify", or "test". For more information about these flags, see Chapter 3, "MCI Overview."

For VCR devices, either "video source" or "audio source" must be specified with either the "count" or "number" flags. If "count" is specified, the total number of inputs of video or audio is returned. If "number" is specified, the driver returns a type corresponding to the input. The type can be any one of the following: "tuner", "line", "svideo", "aux", or "generic". Typically, you should first query the VCR for the "count" and then use the count as the range for the "number" flag. The "source" numbers start from 1.

load

```
wsprintf(lpstrCommand, "load %s %s %s", lpszDeviceID, lpszFilePos,
    lpszFlags);
```

Loads a file in a device-specific format. Digital-video and video-overlay devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszFilePos
Path and filename to load. For video-overlay devices, this can also include a target rectangle for the data. To specify a target rectangle, specify "at" followed by *X1 Y1 X2 Y2*, where *X1 Y1* specify the upper left corner of the rectangle, and *X2 Y2* specify the width and height. The rectangle is relative to the video buffer origin.

lpszFlags
Can be "wait", "notify", or both. For digital-video devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

The following command loads a file into the "vidboard" device:

```
load vidboard c:\vid\fish.vid notify
```

The "vidboard" device sends a notification message when the loading is completed.

mark

```
wsprintf(lpstrCommand, "mark %s %s %s", lpszDeviceID, lpszMark,
    lpszFlags);
```

Controls recording and erasing of marks on the videotape. VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszMark
One of the following flags:

erase	Erases a mark at the current position, if one exists. To erase a mark, first seek to the mark and then issue the **mark** "erase" command.
write	Writes a mark at the current position. The VCR might need to be in play or record mode for this command to succeed.

lpszFlags
Can be "wait", "notify", or "test". For more information about these flags, see Chapter 3, "MCI Overview."

Marks are special signals written to the content that can be detected by the VCR during high-speed searches. Marks are VCR specific.

monitor

```
wsprintf(lpstrCommand, "monitor %s %s %s", lpszDeviceID, lpszMonitor,
    lpszFlags);
```

Specifies the presentation source. (The default presentation source is the workspace.) Switching the presentation source switches all audio and video streams in the source. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID

Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszMonitor

One or more of the following flags:

file	Specifies that the workspace is the presentation source. This is the default source.
input	Specifies that the external input is the presentation source. If a **play** command is in progress, it is first paused. If **setvideo** is "on", this flag displays a default hidden window. Devices might limit what other device instances can do while monitoring input.
method *method*	When used with **monitor** "input", this flag selects the method of monitoring. The *method* is either "pre", "post", or "direct". Direct monitoring requests that the device be configured for optimum display quality during monitoring. The direct monitoring method might be incompatible with motion video recording. Pre- and post-monitoring allow motion video recording. Pre-monitoring shows the external input prior to compression, while post-monitoring shows the external input after compression. Typically, different monitoring methods have different hardware implications. The default monitoring method is selected by the device.

lpszFlags

Can be "wait", "notify", "test", or a combination of these. For more information about these flags, see Chapter 3, "MCI Overview."

The presentation source automatically switches to the workspace after a **play**, **step**, **pause**, **cue** "output", or **seek** command. The **record** command does not automatically switch presentation sources, which gives your application the option of not showing video while it is being recorded.

open

```
wsprintf(lpstrCommand, "open %s %s %s", lpszDevice, lpszOpenFlags,
    lpszFlags);
```

Initializes a device. All MCI devices recognize this command.

- Returns zero if successful or an error otherwise. If an error occurs, it returns one of the values listed in the reference section of Chapter 3, "MCI Overview."

lpszDevice
Identifier of an MCI device or device driver. This can be either a device name (as given in the registry or the SYSTEM.INI file) or the filename of the device driver. If you specify the filename of the device driver, you can optionally include the .DRV extension, but you should not include the path to the file.

lpszOpenFlags
Flag that identifies what to initialize. The following table lists device types that recognize the **open** command and the flags used by each type:

animation	alias *device_alias* nostatic parent *hwnd* shareable style child	style overlapped style popup style *style_type* type *device_type*
cdaudio	alias *device_alias* shareable	type *device_type*
digitalvideo	alias *device_alias* *elementname* nostatic parent *hwnd* shareable	style child style overlapped style popup style *style_type* type *device_type*
overlay	alias *device_alias* parent *hwnd* shareable style child	style overlapped style popup style *style_type* type *device_type*
sequencer	alias *device_alias* shareable	type *device_type*
vcr	alias *device_alias* shareable	type *device_type*
videodisc	alias *device_alias* shareable	type *device_type*
waveaudio	alias *device_alias* buffer *buffer_size*	shareable type *device_type*

The following table lists the flags that can be specified in the *lpszOpenFlags* parameter and their meanings:

alias *device_alias*	Specifies an alternate name for the given device. If specified, it must be used as the *device_id* in subsequent commands.
elementname	Specifies the name of the device element (file) loaded when the device opens.
buffer *buffer_size*	Sets the size, in seconds, of the buffer used by the waveform-audio device. 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.
nostatic	Indicates that the animation device should reduce the number of static (system) colors in the palette. This increases the number of colors controlled by the animation.
parent *hwnd*	Specifies the window handle of the parent window.
shareable	Initializes the device or file as shareable. Subsequent attempts to open the device or file fail unless you specify "shareable" in both the original and subsequent **open** commands. MCI returns an invalid device error if the device is already open and not shareable. The MCISEQ sequencer and MCIWAVE devices do not support shared files.
style child	Opens a window with a child window style.
style overlapped	Opens a window with an overlapped window style.
style popup	Opens a window with a pop-up window style.
style *style_type*	Indicates a window style.
type *device_type*	Specifies the device type of a file.

lpszFlags
Can be "wait", "notify", or both. For more information about these flags, see Chapter 3, "MCI Overview."

MCI reserves "cdaudio" for the CD audio device type, "videodisc" for the videodisc device type, "sequencer" for the MIDI sequencer device type, "AVIVideo" for the digital-video device type, and "waveaudio" for the waveform-audio device type.

As an alternative to the "type" flag, MCI can select the device based on the extension used by the file, as recorded in the registry or the [mci extension] section of the SYSTEM.INI file.

MCI can open AVI files by using a file-interface pointer or a stream-interface pointer. To open a file by using either type of interface pointer, specify an at sign (@) followed by the interface pointer in place of the file or device name for the *lpszDevice* parameter. For more information about the file and stream interfaces, see Chapter 6, "AVIFile Functions and Macros ."

The following command opens the "mysound" device:

```
open new type waveaudio alias mysound buffer 6
```

With device name "new", the waveform driver prepares a new waveform resource. The command assigns the device alias "mysound" and specifies a 6-second buffer.

You can eliminate the "type" flag if you combine the device name with the filename. MCI recognizes this combination when you use the following syntax:

device_name!*element_name*

The exclamation point separates the device name from the filename. The exclamation point should not be delimited by white spaces.

The following example opens the RIGHT.WAV file using the "waveaudio" device:

```
open waveaudio!right.wav
```

The MCIWAVE driver requires an asynchronous waveform-audio device.

paste

```
wsprintf(lpstrCommand, "paste %s %s %s", lpszDeviceID, lpszItem,
    lpszFlags);
```

Copies the contents of the clipboard into the workspace. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszItem
One or more of the following flags:

at *rectangle*	Specifies the location within the frame where the data is pasted. The upper left corner of the *rectangle* corresponds to the upper left corner of the added data. If the rectangle has a nonzero size in X or Y, the contents of the clipboard are scaled in those dimensions when they are pasted into the frame. If omitted, the *rectangle* defaults to the entire frame. If this flag is specified in "insert" mode (the default), any region outside the rectangle is painted a solid color.
audio stream *stream*	Specifies the audio stream in the workspace affected by the command. If only one audio stream exists on the clipboard, the audio data is pasted into the designated *stream*. If more than one audio stream exists on the clipboard, the *stream* indicates the starting number for the stream sequences. If you use this flag and also want to paste video, you must also use the "video stream" flag. (If neither flag is specified, all audio and video streams are pasted and retain their original stream numbers.)
insert	Specifies that the data is inserted into the workspace. Any data after the insertion point is moved forward in the workspace to make room. This is the default value.
overwrite	Specifies that the data is copied into the workspace by writing over any existing data after the insertion point. The "insert" and "overwrite" flags affect whether frames are destroyed or moved during the paste operation, not how the data is pasted within each frame.
to *position*	Specifies the position in the workspace at which the data is pasted. If omitted, it defaults to the current position.
video stream *stream*	Specifies the video stream in the workspace affected by the command. If only one video stream exists on the clipboard, the video data is pasted into the designated *stream*. If more than one video stream exists on the clipboard, the *stream* indicates the starting number for the stream sequences. If you use this flag and also want to paste audio, you

must also use the "audio stream" flag. (If neither flag is specified, all audio and video streams are pasted and retain their original stream numbers.)

lpszFlags
Can be "wait", "notify", "test", or a combination of these. For more information about these flags, see Chapter 3, "MCI Overview."

No signals are present in the data copied from the clipboard. The change becomes permanent only when the data is explicitly saved; however, playback acts as if the data has been added.

pause

```
wsprintf(lpstrCommand, "pause %s %s", lpszDeviceID, lpszFlags);
```

Pauses playing or recording. Most drivers retain the current position and eventually resume playback or recording at this position. Animation, CD audio, digital-video, MIDI sequencer, VCR, videodisc, and waveform-audio devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

With the MCICDA, MCISEQ, and MCIPIONR drivers, the **pause** command works the same as the **stop** command.

The following command pauses the "mysound" device:

```
pause mysound
```

play

```
wsprintf(lpstrCommand, "play %s %s %s", lpszDeviceID, lpszPlayFlags,
    lpszFlags);
```

Starts playing a device. Animation, CD audio, digital-video, MIDI sequencer, videodisc, VCR, and waveform-audio devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszPlayFlags
Flag for playing a device. The following table lists device types that recognize the **play** command and the flags used by each type:

animation	fast from *position* reverse scan	slow speed *fps* to *position*
cdaudio	from *position*	to *position*
digitalvideo	from *position* fullscreen repeat	reverse to *position* window
sequencer	from *position*	to *position*
vcr	at *time* from *position* reverse	scan to *position*
videodisc	fast from *position* reverse scan	slow speed *integer* to *position*
waveaudio	from *position*	to *position*

The following table lists the flags that can be specified in the *lpszPlayFlags* parameter and their meanings:

at *time*	Indicates when the device should begin performing this command, or, if the device has been cued, when the cued command begins. For more information, see the **cue** command.
fast	Indicates that the device should play faster than normal. To determine the exact speed on a videodisc player, use the "speed" flag of the **status** command. To specify the speed more precisely, use the "speed" flag of this command.
from *position*	Specifies a starting position for the playback. If the "from" flag is not specified, playback begins at the current position. For **cdaudio** devices, if the "from" position is greater than the end position of the disc, or if the "from" position is greater than the "to" position, the driver returns an error. For **videodisc** devices, the default positions are in frames for CAV discs and in hours, minutes, and seconds for CLV discs.
fullscreen	Specifies that a full-screen display should be used. Use

	this flag only when playing compressed files. (Uncompressed files won't play full-screen.)
repeat	Specifies that playback should restart when the end of the content is reached.
reverse	Specifies that the play direction is backward. You cannot specify an ending location with the "reverse" flag. For videodiscs, "scan" applies only to CAV format.
scan	Plays as fast as possible without disabling video (although audio might be disabled). For videodiscs, "scan" applies only to CAV format.
slow	Plays slowly. To determine the exact speed on a videodisc player, use the "speed" flag of the **status** command. To specify the speed more precisely, use the "speed" flag of this command. For videodiscs, "slow" applies only to CAV format.
speed *fps*	Plays an animation sequence at the specified speed, in frames per second.
speed *integer*	Plays a videodisc at the specified speed, in frames per second. This flag applies only to CAV discs.
to *position*	Specifies an ending position for the playback. If the "to" flag is not specified, playback stops at the end of the content. For **cdaudio** devices, if the "to" position is greater than the end position of the disc, the driver returns an error. For **videodisc** devices, the default positions are in frames for CAV discs and in hours, minutes, and seconds for CLV discs.
window	Specifies that playing should use the window associated with the device instance. This is the default setting.

lpszFlags

Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

Before issuing commands that use position values, you should set the desired time format by using the **set** command. This command begins playing at the current speed, as set with the **set** "speed" command. The direction is reverse if the "reverse" flag is specified, or if the "to" flag is specified as a value less than the "from" flag. If the "from" flag is not specified, playback begins at the current position. The "to" and "reverse" flags cannot be used together.

The following command plays the "mysound" device from position 1000 through position 2000, sending a notification message when the playback completes:

```
play mysound from 1000 to 2000 notify
```

put

```
wsprintf(lpstrCommand, "put %s %s %s", lpszDeviceID, lpszRegions,
    lpszFlags);
```

Defines the area of the source image and destination window used for display. Animation, digital-video, and video-overlay devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszRegions
Flag for defining the area. The following table lists device types that recognize the **put** command and the flags used by each type:

animation	destination	source
	destination at *rectangle*	source at *rectangle*
digitalvid eo	destination	video
	destination at *rectangle*	video at *rectangle*
	frame	window
	frame at *rectangle*	window at *rectangle*
	source	window client
	source at *rectangle*	window client at *rectangle*
overlay	destination	source
	destination at *rectangle*	source at *rectangle*
	frame	video
	frame at *rectangle*	video at *rectangle*

The following table lists the flags that can be specified in the *lpszRegions* parameter and their meanings:

destination	Selects the entire client area of the destination window to display the data.
destination at *rectangle*	Selects a portion of the client area of the destination window used to display the image. When an area of the display window is specified and the device supports stretching, the source image is stretched to the destination offset and extent.
frame	Selects the entire frame buffer to receive the incoming video images.
frame at *rectangle*	Selects a portion of the frame buffer to receive the incoming video images.
source	Selects the entire image for display in the destination window.
source at *rectangle*	Selects a portion of the image to display in the destination window. When an area of the source image is specified, and the device supports stretching, the source image is stretched to the destination offset and extent.
video	Selects the entire incoming video image to capture in the frame buffer.
video at *rectangle*	Selects a portion of the incoming video image to capture in the frame buffer.

window	Restores the initial window size on the display. This command also displays the window.
window at *rectangle*	Changes the size and location of the display window. The specified rectangle is relative to the parent window of the display window (usually the desktop) if the "style child" flag has been used for the **open** command. To change the location of the window without changing its height or width, specify zero for the height and width.
window client	Restores the client area of the window.
window client at *rectangle*	Changes the size and location of the client area of the window. The specified rectangle is relative to the parent window of the client window. To change the location of the window without changing its height or width, specify zero for the height and width.

When a flag includes a rectangle, the rectangle coordinates are relative to the window origin or the image origin, as appropriate, and are specified as *X1 Y1 X2 Y2*. The coordinates *X1 Y1* specify the upper left corner, and the coordinates *X2 Y2* specify the width and height of the rectangle.

lpszFlags
Can be "wait", "notify", or both. For digital-video devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

The **put** command defines one or more of the following rectangles when working with video-overlay devices:

- The video rectangle defines the region of the incoming video image to capture.
- The frame rectangle defines the region of the frame buffer that receives the incoming video image.
- The source rectangle defines which region of the frame buffer is copied to the destination rectangle.
- The destination rectangle defines the region of the display window client area that receives the video image.

The video rectangle is related to the frame rectangle in the same way the source rectangle is related to the destination rectangle. Stretching can occur from the video rectangle to the frame rectangle and from the source rectangle to the destination rectangle. Not all devices support stretching, and stretching must be enabled (by using the **set** command).

The following command defines three regions for the video, frame, and source:

```
put vboard video 120 120 200 200 frame 0 0 200 200 source 0 0 200 200
```

The regions in this example are defined as follows:

- A 200- by 200-pixel region of the incoming video data, starting at an origin 120 pixels from the upper left corner, will be captured to the frame buffer.
- The video data will be placed in a 200- by 200-pixel region at the upper left corner of the frame buffer.
- Transfers are made from the 200- by 200-pixel region at the upper left corner of the frame buffer to the destination window.

quality

```
wsprintf(lpstrCommand, "quality %s %s %s", lpszDeviceID, lpszQuality,
    lpszFlags);
```

Defines a custom quality level for either audio, video or still image data compression. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszQuality
One or more of the following flags. (One of the three flags "audio", "still", and "video" must be present.)

algorithm *algorithm*	Associates the quality level with the specified *algorithm*. This *algorithm* must be supported by the device and be compatible with the "audio", "still", or "video" flag that is used. If omitted, the current algorithm is used.
audio *name*	Indicates this command specifies an "audio" quality level identified with *name*.
dialog	Requests that the device display a dialog box. This dialog box has algorithm-specific fields that are used internally by the device to create the structure describing a specific quality level.
handle *handle*	Specifies a *handle* to a structure that contains algorithmic-specific data describing a specific quality level. The structures for the data referenced by this handle are device specific.
still *name*	Indicates the command specifies a "still" quality level identified with *name.*
video *name*	Indicates the command specifies a "video" quality level identified with *name*.

lpszFlags
Can be "wait", "notify", "test", or a combination of these. For more information about these flags, see Chapter 3, "MCI Overview."

This command defines a string name for the quality level, which can then be used in a **setvideo** "quality", **setvideo** "still quality", or **setaudio** "quality" command to establish it as the current video, still, or audio-compression quality level.

realize

```
wsprintf(lpstrCommand, "realize %s %s %s", lpszDeviceID, lpszPalette,
    lpszFlags);
```

Instructs a device to select and realize its palette into the display context of the displayed window. Animation and digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszPalette
One of the following flags:

background	Realizes the palette as a background palette.
normal	Realizes the palette for a top-level window. This is the default setting.

lpszFlags
Can be "wait", "notify", or both. For digital-video devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

Use this command only if your application uses a window handle and receives a WM_QUERYNEWPALLETTE or WM_PALETTECHANGED message.

The following command tells the "myvideo" device to realize its palette:

```
realize myvideo normal
```

record

```
wsprintf(lpstrCommand, "record %s %s %s", lpszDeviceID, lpszRecordFlags,
    lpszFlags);
```

Starts recording data. VCR and waveform-audio devices recognize this command. Although digital-video devices and MIDI sequencers also recognize this command, the MCIAVI and MCISEQ drivers do not implement it.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszRecordFlags
Flag for recording data. The following table lists device types that recognize the **record** command and the flags used by each type:

digitalvideo	at *rectangle* audio stream *stream* from *position* hold	insert overwrite to *position* video stream *stream*
sequencer	from *position* insert	overwrite to *position*
vcr	at *time* from *position* initialize	insert overwrite to *position*
waveaudio	from *position* insert	overwrite to *position*

The following table lists the flags that can be specified in the *lpszRecordFlags* parameter and their meanings:

at *rectangle*	Specifies a rectangular region of the external input used as the source for the pixels compressed and saved. If not specified, the rectangle defaults to the rectangle specified for **put** "video". When it is set differently from the "video" rectangle, the displayed image is not what is recorded.
at *time*	Indicates when the device should begin performing this command, or, if the device has been cued, when the cued command begins. For more information, see the **cue** command.
audio stream *stream*	Specifies the audio stream used for recording. If this flag is not specified and the file format does not define a default, it is recorded into the stream that is physically first.
from *position*	Specifies a starting position for the recording. If the "from" flag is not specified, the device starts recording at the current position.
hold	Freezes the image when recording has finished instead of showing live video. When recording stops, an automatic **monitor** "file" command is performed. To return to live video, issue the **monitor** "input" command.
initialize	Initialize the tape (media), which involves recording

	timecode (if possible) for blank video and audio. This command might take several hours if the entire tape must be initialized.
insert	Specifies that new data is added to the file at the current position.
overwrite	Specifies that new data will replace data in the file.
to *position*	Specifies an ending position for the recording. If the "to" flag is not specified, the device records until it receives a **stop** or **pause** command.
video stream *stream*	Specifies the video stream used for recording. If this is not specified and the file format does not define a default, then it is recorded into the stream that is physically first.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

The recording stops when a **stop** or **pause** command is issued. For the MCIWAVE driver, all data recorded after a file is opened is discarded if the file is closed without saving it.

Before issuing any commands that use position values, you should set the desired time format by using the **set** command. The tracks to be recorded are specified by the **settimecode** "record", **set** "assemble record", **setvideo** "record", and **setaudio** "record" commands.

The following command starts recording at the current position:

```
record mysound
```

reserve

```
wsprintf(lpstrCommand, "reserve %s %s %s", lpszDeviceID, lpszReserve,
    lpszFlags);
```

Allocates contiguous disk space for the device instance's workspace. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszReserve
One or more of the following flags:

in *path*	Specifies the drive and directory path (but not the name) of a temporary file used to hold recorded data. The name of this file is specified by the device. The temporary file is deleted when the device is closed. If this flag is omitted, the device specifies the location of the disk space.
size *duration*	Specifies the approximate amount of disk space to reserve in the workspace. The *duration* value is specified in the current time format. The device bases its estimate of the required disk space on the following parameters: the requested time, the file format, the video and audio compression algorithm, and the compression quality values in effect. If **setvideo** "record" is "off", then space is reserved only for audio. If **setaudio** "record" is "off", then space is reserved only for video. If both are "off", or if *duration* is zero, then no space is reserved and any existing reserved space is deallocated. If this flag is omitted, the device will use a device-defined default.

lpszFlags
Can be "wait", "notify", "test", or a combination of these. For more information about these flags, see Chapter 3, "MCI Overview."

If needed, subsequent **record** or **save** commands use the space reserved by this command. If the workspace contains unsaved data, the data is lost. Some devices do not require **reserve** and ignore it. If disk space is not reserved prior to recording, the **record** command performs an implied **reserve** with device-specific default flags. Use an explicit **reserve** command if you want better control of when the delay for disk allocation occurs, control of how much space is allocated, and control of where the disk space is allocated. Your application can change the amount and location of previously reserved disk space with subsequent **reserve** commands. Any allocated and still unused disk space is not deallocated until any recorded data is saved, or until the device instance is closed.

restore

```
wsprintf(lpstrCommand, "restore %s %s %s", lpszDeviceID, lpszRestore,
    lpszFlags);
```

Copies a still image from a file to the frame buffer. This is the reverse of the **capture** command. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszRestore
One or more of the following flags:

at *rectangle*	Specifies a rectangle relative to the frame buffer origin. The *rectangle* is specified as *X1 Y1 X2 Y2*. The coordinates *X1 Y1* specify the upper left corner and the coordinates *X2 Y2* specify the width and height. If this flag is not used, the image is copied to the upper left corner of the frame buffer.
from *filename*	Specifies the image filename to recall. This flag is required.

lpszFlags
Can be "wait", "notify", "test", or a combination of these. For more information about these flags, see Chapter 3, "MCI Overview."

Devices can recognize a variety of image formats; a Windows device-independent bitmap is always recognized.

resume

```
wsprintf(lpstrCommand, "resume %s %s", lpszDeviceID, lpszFlags);
```

Continues playing or recording on a device that has been paused using the **pause** command. Animation, digital-video, VCR, and waveform-audio devices recognize this command. Although CD audio, MIDI sequencer, and videodisc devices also recognize this command, the MCICDA, MCISEQ, and MCIPIONR device drivers do not support it.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

The following command continues playing or recording the "newsound" device:

```
resume newsound
```

save

```
wsprintf(lpstrCommand, "save %s %s %s", lpszDeviceID, lpszFilename,
    lpszFlags);
```

Saves an MCI file. Video-overlay and waveform-audio devices recognize this command. Although digital-video devices and MIDI sequencers also recognize this command, the MCIAVI and MCISEQ drivers do not support it.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszFilename
Flag specifying the name of the file being saved and, optionally, additional flags modifying the save operation. The following table lists device types that recognize the **save** command and the flags used by each type:

digitalvideo	abort at *rectangle*	*filename* keepreserve
overlay	at *rectangle*	*filename*
sequencer	*filename*	
waveaudio	*filename*	

The following table lists the flags that can be specified in the *lpszFilename* parameter and their meanings:

abort	Stops a **save** operation in progress. If used, this must be the only item present.
at *rectangle*	Specifies a rectangle relative to the frame buffer origin. The *rectangle* is specified as *X1 Y1 X2 Y2*. The coordinates *X1 Y1* specify the upper left corner and the coordinates *X2 Y2* specify the width and height. For digital-video devices, the **capture** command is used to capture the contents of the frame buffer.
filename	Specifies the filename to assign to the data file. If a path is not specified, the file will be placed on the disk and in the directory previously specified on the explicit or implicit **reserve** command. If **reserve** has not been issued, the default drive and directory are those associated with the application's task. If a path is specified, the device might require it to be on the disk drive specified by the explicit or implicit **reserve**. This flag is required.
keepreserve	Specifies that unused disk space left over from the original **reserve** command is not deallocated.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

The *filename* variable is required if the device was opened using the "new" device identifier.

The following command saves the entire video buffer to a file named VCAPFILE.TGA:

```
save vboard c:\vcap\vcapfile.tga
```

seek

```
wsprintf(lpstrCommand, "seek %s %s %s", lpszDeviceID, lpszSeekFlags,
    lpszFlags);
```

Moves to the specified position and stops. Animation, CD audio, digital-video, MIDI sequencer, VCR, videodisc, and waveform-audio devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszSeekFlags
Flag for moving to a specified position. The following table lists device types that recognize the **seek** command and the flags used by each type:

animation	to end to *position*	to start
cdaudio	to end to *position*	to start
digitalvideo	to end to *position*	to start
sequencer	to end to *position*	to start
vcr	at *time* mark *mark_num* reverse	to end to *position* to start
videodisc	reverse to end	to *position* to start
waveaudio	to end to *position*	to start

The following table lists the flags that can be specified in the *lpszSeekFlags* parameter and their meanings:

at *time*	Indicates when the device should begin performing this command, or, if the device has been cued, when the cued command begins. For more information, see the **cue** command.
mark *mark_num*	Seeks to the relative mark indicated by *mark_num*, which must be a positive integer value. Marks are signals written to the VCR tape using the **mark** command and are used for high-speed searching.
reverse	Indicates that the seek direction on VCRs and CAV videodiscs is backward. This flag is invalid if the "to" flag is specified. For VCRs, this flag must be used with the "mark" flag.
to end	Seeks to the end of the content.
to *position*	Specifies the position to stop the seek. For **cdaudio** devices, MCI returns an out-of-range error if the specified position is greater than the length of the disc.
to start	Seeks to the start of the content.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For

more information about these flags, see Chapter 3, "MCI Overview."

Before issuing any commands that use position values, you should set the desired time format by using the **set** command.

Digital-video devices support two seek modes, which you can change by using the **set** command. The "seek exactly on" mode causes the seek command to move to the specified frame. The "seek exactly off" mode causes the seek command to move to the closest key frame prior to the specified frame.

If a CD audio device is playing when the **seek** command is issued, playback is stopped. When the **seek** command is issued with a videodisc device, the device searches using fast forward or fast reverse with video and audio off.

When the **seek** command is issued with a waveform-audio device, the behavior depends on the sample size. If the sample size is 16 bits or greater, **seek** moves to the beginning of the nearest sample when a specified position does not coincide with the start of a sample.

The following command seeks to the start of the media file associated with the "mysound" device:

```
seek mysound to start
```

set

```
wsprintf(lpstrCommand, "set %s %s %s", lpszDeviceID, lpszSetting,
    lpszFlags);
```

Establishes control settings for the device. Animation, CD audio, digital-video, MIDI sequencer, VCR, videodisc, video-overlay, and waveform-audio devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszSetting
Flag for establishing control settings. The following table lists device types that recognize the **set** command and the flags used by each type:

animation	audio all off	door closed
	audio all on	door open
	audio left off	time format frames
	audio left on	time format milliseconds
	audio right off	video off
	audio right on	video on
cdaudio	audio all off	door closed
	audio all on	door open
	audio left off	time format milliseconds
	audio left on	time format msf
	audio right off	time format tmsf
	audio right on	
digitalvideo	audio all off	file format *format*
	audio all on	seek exactly on
	audio left off	seek exactly off
	audio left on	speed *factor*
	audio right off	still file format *format*
	audio right on	time format frames
	door closed	time format milliseconds
	door open	video off
		video on
overlay	audio all off	audio right on
	audio all on	door closed
	audio left off	door open
	audio left on	video off
	audio right off	video on
sequencer	audio all off	port mapper
	audio all on	port none
	audio left off	port *port_number*
	audio left on	slave file
	audio right off	slave MIDI
	audio right on	slave none
	door closed	slave SMPTE
	door open	tempo *tempo_value*
	master MIDI	time format milliseconds
	master none	time format SMPTE *fps*
	master SMPTE	time format SMPTE 30 drop
	offset *time*	time format song pointer
vcr	assemble record on	power on

	assemble record off	power off
	audio all off	preroll duration *duration*
	audio all on	record format SP
	audio left off	record format LP
	audio left on	record format EP
	audio right off	speed *factor*
	audio right on	time format frames
	clock *time*	time format hms
	counter format	time format milliseconds
	counter *value*	time format msf
	door closed	time format SMPTE *fps*
	door open	time format SMPTE 30 drop
	index counter	time format tmsf
	index date	time mode counter
	index time	time mode detect
	index timecode	time mode timecode
	length *duration*	tracking plus
	pause *timeout*	tracking minus
	postroll duration - *duration*	tracking reset
videodisc	audio all off	door open
	audio all on	time format frames
	audio left off	time format hms
	audio left on	time format milliseconds
	audio right off	time format track
	audio right on	video off
	door closed	video on
waveaudio	alignment *integer*	channels *channel_count*
	any input	door closed
	any output	door open
	audio all off	format tag pcm
	audio all on	format tag *tag*
	audio left off	input *integer*
	audio left on	output *integer*
	audio right off	samplespersec *integer*
	audio right on	time format bytes
	bitspersample *bit_count*	time format milliseconds
	bytespersec *byte_rate*	time format samples

The following table lists the flags that can be specified in the *lpszSetting* parameter and their meanings:

alignment *integer*	Sets the alignment of data blocks relative to the start of data passed to the waveform-audio device. The file is saved in this format.
any input	Use any input that supports the current format when recording. This is the default setting.
any output	Use any output that supports the current format when playing. This is the default.
assemble record on assemble record off	In assemble mode, all tracks are recorded as defined by the device. Most VCRs are in

	assemble mode by default.
audio all off audio all on	Disables or enables audio output. Video-overlay devices, the MCISEQ sequencer, and the MCIWAVE waveform-audio device do not support this flag.
audio left off audio left on audio right off audio right on	Disables or enables output to either the left or the right audio channel. Video-overlay devices, the MCISEQ sequencer, and the MCIWAVE waveform-audio device do not support this flag.
bitspersample *bit_count*	Sets the number of bits per PCM (Pulse Code Modulation) sample played or recorded. The file is saved in this format.
bytespersec *byte_rate*	Sets the average number of bytes per second played or recorded. The file is saved in this format.
channels *channel_count*	Sets the channels for playing and recording. The file is saved in this format.
clock *time*	Sets time on the external clock to *time*. The format is specified as a long unsigned integer.
counter format	Set the time format for the counter, as returned by **status** "counter". For information about applicable types, see the **set** "time format" command.
counter *value*	Sets the VCR counter to the specified value. The value must be specified in the current counter format. For more information, see the **set** "counter format" command.
door closed	Retracts the tray and closes the door, if possible. For VCRs, loads the tape automatically.
door open	Opens the door and ejects the tray or tape, if possible.
file format *format*	Specifies a file format that is used for **save** or **capture** commands. If omitted, this might default to a device driver defined format. If the specified file format conflicts with the currently selected algorithm and quality, then they are changed to the defaults for the file format. The following file formats are defined: avi Specifies AVI format. avss Specifies AVSS format. dib Specifies DIB format. jfif Specifies JFIF format. jpeg Specifies JPEG format.

	mpeg Specifies MPEG format. rdib Specifies RLE DIB format. rjpeg Specifies RJPEG format.
format tag pcm	Sets the format type to PCM for playing and recording. The file is saved in this format.
format tag *tag*	Sets the format type for playing and recording. The file is saved in this format.
index timecode index counter index date index time	Sets the current display screen on the VCR.
input *integer*	Sets the audio channel used as the input.
length *duration*	Sets the user-specified length of the tape in the VCR. This length is returned by the **status** "length" command and is provided for compatibility with applications that require this command to return a valid length.
master midi	Sets the MIDI sequencer as the synchronization source. Synchronization data is sent in MIDI format. The MCISEQ sequencer does not support this flag.
master none	Inhibits the MIDI sequencer from sending synchronization data. The MCISEQ sequencer does not support this flag.
master smpte	Sets the MIDI sequencer as the synchronization source. Synchronization data is sent in SMPTE (Society of Motion Picture and Television Engineers) format. The MCISEQ sequencer does not support this flag.
offset *time*	Sets the SMPTE offset *time*. The offset is the beginning time of a SMPTE based sequence. The *time* is expressed as *hh*:*mm*:*ss*:*ff*, where *hh* is hours, *mm* is minutes, *ss* is seconds, and *ff* is frames.
output *integer*	Sets the audio channel used as the output.
pause *timeout*	Sets the maximum duration, in milliseconds, of a **pause** command. A *timeout* value of zero indicates that no time-out will occur.
postroll duration *duration*	Sets the length, in the current time format, needed to brake the VCR transport when a **stop** or **pause** command is issued.
port mapper	Sets the MIDI mapper as the port receiving the MIDI messages. This command fails if the MIDI mapper or a port it needs is being used by another application.
port none	Disables the sending of MIDI messages. This command also closes a MIDI port.

port *port_number*	Sets the MIDI port receiving the MIDI messages. This command fails if the port you are trying to open is being used by another application.
power on power off	Sets the device power to on or off.
preroll duration *duration*	Sets the length, in the current time format, needed to stabilize the VCR output.
record format SP record format LP record format EP	Sets the recording mode for the VCR to SP for standard play, EP for extended play, or LP for long play. These values are not intended to be VHS specific. They map to any three appropriate modes with other tape formats. For example, SP maps to the fastest, highest quality recording.
samplespersec *integer*	Sets the sample rate for playing and recording. The file is saved in this format.
seek exactly on seek exactly off	Selects one of two seek modes. With "seek exactly on", seek will always move to the frame specified. With "seek exactly off", seek will move to the closest key frame prior to the frame specified.
slave file	Sets the MIDI sequencer to use file data as the synchronization source. This is the default setting.
slave midi	Sets the MIDI sequencer to use incoming MIDI data for the synchronization source. The sequencer recognizes synchronization data with the MIDI format. The MCISEQ sequencer does not support this flag.
slave none	Sets the MIDI sequencer to ignore synchronization data.
slave smpte	Sets the MIDI sequencer to use incoming MIDI data for the synchronization source. The sequencer recognizes synchronization data with the SMPTE format. The MCISEQ sequencer does not support this flag.
speed *factor*	Sets the relative speed of video and audio playback from the workspace. *Factor* is the ratio between the nominal frame rate and the desired frame rate, where the nominal frame rate is designated as 1000. (A rate of 500 is half normal speed, 2000 is twice normal speed, and so on.) Setting the speed to zero plays the video as fast as possible without dropping frames and without audio.
still file format *format*	Specifies the file format used for **capture** commands.
tempo *tempo_value*	Sets the tempo of the sequence according to the current time format. For a PPQN-based file, the *tempo_value* is interpreted as beats per minute. For a SMPTE-based file, the

	tempo_value is interpreted as frames per second.
time format bytes	In a PCM file format, sets the time format to bytes. All position information is specified as bytes following this command.
time format frames	Sets the time format to frames. All commands that use position values will assume frames. When the device is opened, frames is the default mode. Supported by videodiscs using CAV format.
time format hms	Sets the time format to hours, minutes, and seconds. All commands that use position values will assume HMS. HMS is the default format for CLV videodiscs. Specify an HMS value as *hh*:*mm*:*ss*, where *hh* is hours, *mm* is minutes, and *ss* is seconds. You can omit a field if it and all following fields are zero. For example, 3, 3:0, and 3:0:0 are all valid ways to express 3 hours.
time format milliseconds	Sets the time format to milliseconds. All commands that use position values will assume milliseconds. You can abbreviate milliseconds as "ms". For sequencer devices, the sequence file sets the default format to PPQN or SMPTE. Video-overlay devices do not support this flag.
time format msf	Sets the time format to minutes, seconds, and frames. All commands that use position values will assume MSF (the default format for CD audio). Specify an MSF value as *mm*:*ss*:*ff*, where *mm* is minutes, *ss* is seconds, and *ff* is frames. You can omit a field if it and all following fields are zero. For example, 3, 3:0, and 3:0:0 are valid ways to express 3 minutes. The MSF fields have the following maximum values: Minutes 99 Seconds 59 Frames 74
time format samples	Sets the time format to samples. All position information is specified as samples following this command.
time format smpte 24 time format smpte 25 time format smpte 30	Sets the time format to an SMPTE frame rate. For VCRs, sets the time format to hh:mm:ss:ff, where the legal values are 00:00:00:00 through 23:59:59:*xx*, and *xx* is one less than the frames per second as

	specified by the number 24, 25, or 30 as specified in the flag. On input, colons (:) are required to separate the components. The least significant units can be omitted if they are 00; for example, 02:00 is the same as 02:00:00:00. All commands that use position values will assume SMPTE format. The sequence file sets the default format to PPQN or SMPTE.
time format smpte 30 drop	Sets the time format to SMPTE 30 drop frame rate. For VCRs, same as SMPTE 30, except that certain timecode positions are dropped from the format to have the recorded timecode positions for each frame (at the NTSC frame rate of 29.97 fps) correspond to real time (at 30 fps). Timecode positions that are dropped are as follows: two every minute, on the minute, for the first nine of every ten minutes of recorded content. For example, at 01:04:59:29, the next timecode position would be 01:05:00:02, not 01:05:00:00. All commands that use position values will assume SMPTE format. The sequence file sets the default format to PPQN or SMPTE.
time format song pointer	Sets the time format to song pointer (sixteenth notes). All commands that use position values will assume song pointer units. This flag is valid only for a sequence of division type PPQN.
time format tmsf	Sets the time format to tracks, minutes, seconds, and frames. All commands that use position values will assume TMSF. Specify a TMSF value as *tt*:*mm*:*ss*:*ff*, where *tt* is tracks, *mm* is minutes, *ss* is seconds, and *ff* is frames. You can omit a field if it and all following fields are zero. For example, 3, 3:0, 3:0:0, and 3:0:0:0 are all valid ways to express track 3. The TMSF fields have the following maximum values: Tracks 99 Minutes 99 Seconds 59 Frames 74
time format track	Sets the position format to tracks. All commands that use position values will assume tracks.
time mode counter	Sets the position-information mode to use the

	VCR counters.
time mode detect	Sets the position information mode based on detection of timecode information on the tape. If timecode information is detected, the time type is set to "timecode"; otherwise, the time type is set to "counter". "Detect" is a special mode. Whenever the driver is opened, a new tape is inserted, or the "time mode" command is issued, the driver checks the current time mode available on the tape and sets "time type" to either "timecode" or "counter". Once "time type" is set, the driver doesn't change it until one of the above conditions occurs again.
time mode timecode	Sets the position information mode to use "timecode" information on the tape.
tracking plus tracking minus tracking reset	Adjusts the speed of the videotape transport in fine increments. Use the "tracking" flags when a noisy picture is obtained from a VCR. "Tracking plus" increases the transport speed. "Tracking minus" decreases the transport speed. "Tracking reset" returns the tracking adjustment to zero.
video off	Disables video output.
video on	Enables video output.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

Several properties of waveform-audio data are defined when the file to store the data is created. These properties describe how the data is structured within the file and cannot be changed once recording begins. The following list identifies these properties:

- alignment
- bitspersample
- bytespersec
- channels
- format tag
- samplespersec

The following command sets the time format to milliseconds and sets the waveform-audio format to 8 bit, mono, 11 kHz:

```
set mysound time format ms bitspersample 8 channels 1 samplespersec 11025
```

setaudio

```
wsprintf(lpstrCommand, "setaudio %s %s %s", lpszDeviceID, lpszAudio,
    lpszFlags);
```

Sets values associated with audio playback and capture. Digital-video and VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszAudio
Flag for audio playback and capture. The following table lists device types that recognize the **setaudio** command and the flags used by each type:

digitalvideo	algorithm *algorithm*	over *duration*
	alignment to *integer*	quality *descriptor*
	bass to *factor*	record off
	bitspersample to *bit_count*	record on
	bytespersec to *integer*	right off
	clocktime	right on
	input	right volume to *factor*
	left off	samplespersec to *integer*
	left on	source to *sourcename*
	left volume to *factor*	stream to *number*
	off	treble to *factor*
	on	volume to *factor*
	output	
vcr	off	record on
	on	record track *track_number* on
	monitor to *type* number *number*	source to *type* number *number*
	record off	track *track_number* off
	record track *track_number* off	track *track_number* on

The following table lists the flags that can be specified in the *lpszAudio* parameter and their meanings:

algorithm *algorithm*	Selects a specific audio compression algorithm for use by a subsequent **reserve** or **record** command. The algorithms supported are device specific. MCI defines the values "g711", "g721", "g722", "g728", "pcm", "cdxa", "adpcm", and "adpcm4e" for *algorithm*. If a device supports the algorithm names "pcm", "cdxa", and "adpcm4e", they adhere to standard definitions. The "cdxa" algorithm has been defined by Sony Corporation. The "adpcm4e" algorithm has been defined by Intel Corporation. The "g711", "g721", "g722", and "g728" values represent audio algorithms recommended by the International Telegraph and Telephone Consultative

	Committee (CCITT).
	If the specified algorithm conflicts with the current file format, the file format is changed to the default format for the algorithm.
alignment to *integer*	Sets the alignment of data blocks relative to the start of input waveform-audio data.
bass to *factor*	Sets the audio low frequency level.
bitspersample to *bit_count*	Sets the number of bits per sample recorded. The file is saved in this format. This flag applies only to devices supporting the "pcm" algorithm.
bytespersec to *integer*	Sets the average number of bytes per second for recording in the "pcm" and "adpcm" algorithms. The file is saved in this format.
clocktime	Indicates the time specified in the "over" flag is in milliseconds. This time is absolute and not in step with the playing of the workspace.
input	Modifies the "bass", "treble", or "volume" flag so that it affects the input signal and modifies what is recorded. If possible, this is the default when monitoring the input.
left off left on	Enables or disables audio output on the left channel. The audio presentation source can be the external input or the workspace. The default is "left on". If there is only one channel, that channel is set on or off.
left volume to *factor*	Sets the audio volume of the left audio channel. If there is only one channel it sets its volume.
monitor to *type* number *number*	Controls which source input will be passed to the VCR output without changing the recording source input selection. *Type* can be "output," or one of the valid input sources. If *number* is not specified, then the first input of that type will be chosen.
off on	Enables or disables audio. The audio presentation source can either be the external input or the workspace. This command affects the left and right audio channels simultaneously. The default is **setaudio** "on".
output	Modifies the "bass", "treble", or "volume" flag so that it modifies only the played signal and not what is recorded. If possible, this is the default when monitoring a file.

over *duration*	Specifies how long it should take to make a change that uses a *factor* variable. The units for *duration* are in the current time format. Changes occur in step with the playing of the workspace. When playing is suspended, the change is also suspended until the play continues. If "over" is not specified or not supported, the change occurs immediately.
quality *descriptor*	Specifies the characteristics of the audio compression performed when audio is recorded to a file. All devices support the three descriptors "low", "medium", and "high". The default is device specific. If the "algorithm" flag is not specified, the "quality" adjustment applies to the current algorithm. The **quality** command can be used to define additional descriptor names.
record off	Clears the audio-source selection so that no audio will be recorded with the next **record** command.
record on	Enables recording of audio data. The default is to record audio data.
record track *track_number* off	Clears the audio-source selection so that no audio will be recorded with the next **record** command. "Track" allows independent track selection. Track 2 corresponds to the PCM track in Hi8. If "track" is not specified, a default value of 1 is assumed.
record track *track_number* on	Selects the audio source to be recorded with the next **record** command. "Track" allows independent track selection. Track 2 corresponds to the PCM track in Hi8. If "track" is not specified, a default value of 1 is assumed.
right off right on	Enables or disables audio output on the right channel. The audio presentation source can be the external input or the workspace. The default is "right on". If there is only one channel, this flag has no effect.
right volume to *factor*	Sets the audio volume to the right audio channel. If there is only one channel, it has no effect.
samplespersec to *integer*	Sets the sample rate for recording with the "pcm" and "adpcm" algorithms. The file is saved in this format.
source to *sourcename*	Specifies the source for the audio input digitizer. The constants defined for *sourcename* include: "left", "right",

	"average", and "stereo". The first three specify monophonic recording using the left input only, the right input only, and the average of the two inputs.
source to *type* number *number*	Selects the audio source to be recorded on the tape. *Type* must be "tuner", "line", "svideo", "aux", "generic", or "mute".
stream to *number*	Specifies the audio stream played back from the workspace. If the stream is not specified and the file format does not define a default, then the interleaved audio stream that is physically first will be played.
track *track_number* off	Disables an individual track.
track *track_number* on	Enables an individual track.
treble to *factor*	Sets the audio high-frequency level.
volume to *factor*	Sets the average audio volume for both audio channels. If the left and right volumes have been set to different values, then the ratio of left-to-right volume is approximately unchanged.

lpszFlags

Can be "wait", "notify", "test", or a combination of these. For more information about these flags, see Chapter 3, "MCI Overview."

For VCR devices, using **setaudio** with a flag that turns off an individual track ("track *track_number* off") might cause your application to receive a status message indicating that the command could not be carried out. Some VCRs can turn off only combinations of tracks, not individual tracks; for example, the first audio track and a video track of a video cassette. In this case, simply use **setaudio** and **setvideo** to continue to turn off the other tracks that make up the combination. The driver will turn off the tracks when it receives the command to turn off the last track in the combination.

settimecode

```
wsprintf(lpstrCommand, "settimecode %s %s %s", lpszDeviceID,
    lpszTimecode, lpszFlags);
```

Enables or disables timecode recording for a VCR. VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszTimecode
One of the following flags:

record on	Sets the VCR to record timecode.
record off	Disables timecode recording.

lpszFlags
Can be "wait", "notify", "test", or a combination of these. For more information about these flags, see Chapter 3, "MCI Overview."

settuner

```
wsprintf(lpstrCommand, "settuner %s %s %s", lpszDeviceID, lpszTuner,
    lpszFlags);
```

Changes the current tuner or the channel setting of the current tuner. VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszTuner
One of the following flags:

channel *channel*	Sets the tuner to *channel*. You might not be able to change the channel while recording, depending on the VCR. A channel larger than the maximum sets the tuner to the maximum channel.
channel seek up channel seek down	Seeks the next channel with a valid signal. "Seek up" increments the channel number in its search. "Seek down" decrements the channel number in its search. The tuner wraps to channel 1 when the maximum channel number is exceeded. Similarly, when seeking down, the tuner wraps to the maximum channel.
channel up channel down	Increments or decrements the tuner channel. You might not be able to change the channel while recording, depending on the VCR. The tuner wraps to channel 1 when the maximum channel number is exceeded. Similarly, when seeking down, the tuner wraps to the maximum channel.
number *number*	Specifies the tuner to be affected by the **settuner** command. If *number* is not given, the default value of 1 is assumed.

lpszFlags
Can be "wait", "notify", "test", or a combination of these. For more information about these flags, see Chapter 3, "MCI Overview."

setvideo

```
wsprintf(lpstrCommand, "setvideo %s %s %s", lpszDeviceID, lpszVideo,
    lpszFlags);
```

Sets values associated with video playback and capture. Digital-video and VCR devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszVideo
Flag for video playback and capture. The following table lists device types that recognize the **setvideo** command and the flags used by each type:

digitalvideo	algorithm *algorithm* bitsperpel to *count* brightness to *factor* clocktime color to *factor* contrast to *factor* gamma to *value* halftone input key color to *r:g:b* key index to *index* off on output	over *duration* palette color *color* over *index* to *newrgb* palette handle to *handle* quality *descriptor* record frame rate to *rate* record on record off sharpness to *factor* source to *source* number *value* still algorithm *algorithm* still quality *descriptor* stream to *number* tint to *factor*
vcr	off on monitor to *type* number *number* record off record track *track_number* off	record on record track *track_number* on source to *type* number *number* track *track_number* off track *track_number* on

The following table lists the flags that can be specified in the *lpszVideo* parameter and their meanings:

algorithm *algorithm*	Specifies a video compression algorithm for use by a subsequent **reserve** or **record** command. Algorithms supported by a device are device specific. MCI defines the constants "mpeg" and "h261" for *algorithm*. If the specified algorithm conflicts with the current file format, the file format is changed to the default format for the algorithm.
bitsperpel to *count*	Sets the number of bits per pixel for saving data with the **capture** or **record** command.
brightness to *factor*	Sets the video brightness level.
clocktime	Indicates that the time specified in the

	"over" flag is in milliseconds. The time is absolute and not in step with the playing of the workspace.
color to *factor*	Sets the color-saturation level.
contrast to *factor*	Sets the video-contrast level.
gamma to *value*	Specifies the gamma correction exponent multiplied by 1000. For example, to specify an exponent of 2.2, use 2200 for *value*. A gamma value of 1.0 (1000) indicates no gamma correction is applied. Gamma correction adjusts the mapping between the intensity encoded in the presentation source and the displayed brightness.
halftone	Causes the halftone palette to be used instead of the default palette. This flag is recognized only by the MCIAVI digital-video driver.
input	Modifies the "brightness", "color", "contrast", "gamma", "sharpness", or "tint" flag so that it affects the input signal and modifies what is recorded. If possible, this is the default when monitoring the input.
key color to *r:g:b*	Sets the key color. The *r:g:b* variable is a Windows RGB value. Colons (:) separate the individual red, green, and blue values.
key index to *index*	Sets the key index. The *index* variable is a physical palette index.
monitor to *type* number *number*	Controls which source input will be passed to the VCR output, without changing the recording source input selection. Type can be "output", or one of the valid input sources. If "number" is not specified, then the first input of that type is chosen.
off on	Enables or disables display of video. Disabling video sets the pixels in the **put** "destination" rectangle (or its default, the client region of the current window) to a solid color. It has no effect on the frame buffer. The video source, whether the workspace or an external input, might continue to store new images in the frame buffer. They are not displayed until video is enabled. You can use the **window** "state" command to hide the window. The default is **setvideo** "on".
output	Modifies the "brightness", "color", "contrast", "gamma", "sharpness", or "tint" flag so that it modifies only the displayed signal and not what is recorded. If possible, this is the default when monitoring a file.
over *duration*	Specifies how long it should take to make a

	change that uses a *factor* variable. The units for *duration* are in the current time format. Changes occur in step with the playing of the workspace. When playing is suspended, the change is also suspended until the play continues. If "over" is not used or not supported, the change occurs immediately.
palette color *color* over *index* to *newrgb*	Sets a new palette color. The color and palette index to be changed are specified by the *color* and *index* parameters; the new color is specified by *newrgb*. This flag is recognized only by the MCIAVI digital-video driver.
palette handle to *handle*	Specifies the handle to a palette the device must use for rendering. This item is supported only by devices that use palettes. If *handle* is zero, the default palette is used. Digital-video devices should not free the palette passed with this command. Applications should free it after they close the device.
quality *descriptor*	Specifies the characteristics of the video compression performed when video is recorded to a file. All devices support the three descriptors: "low", "medium", and "high". The default is device specific. The significance of these names depends on the algorithm and the device. Devices might define additional descriptor names. The **quality** command can be used to define additional descriptor names. If the "algorithm" flag is not used, the *descriptor* applies to the current algorithm.
record frame rate to *rate*	Sets the recording for motion video. The recording *rate* is specified in units of frames per second multiplied by 1000. For example, the NTSC frame rate of 29.97 frames per second is represented as 29970.
record on record off	Enables or disables recording of video data. Recording video data is the default.
record track *track_number* off	Clears the video-source selection so that no video will be recorded with the next **record** command. "Track" allows independent track selection. If "track" is not specified, a default value of 1 is assumed. It might be necessary to first issue a **set** "assemble record off" command before the video recording can be turned off.
record track *track_number* on	Selects the video source to be recorded with the next **record** command. "Track"

	allows independent track selection. Track 2 corresponds to the PCM track in Hi8. If "track" is not specified, a default of 1 is assumed.
sharpness to *factor*	Sets the video sharpness level.
source to *source* number *value*	Sets the source of the video input. This usually corresponds to external connectors. The constants defined for *source* include "rgb", "pal", "ntsc", "svideo", and "secam". If more than one input of the specified type exists, the optional "number" *value* indicates the desired input. For example, **setvideo** "source to ntsc number 2" specifies the second NTSC input. If "to" *source* is omitted, then the absolute source is used as defined by the **list** "video source" command.
source to *type* number *number*	Selects the video source to be recorded on the tape. *Type* must be "tuner", "line", "svideo", "aux", "generic", "mute", or "rgb".
still algorithm *algorithm*	Specifies the still image compression algorithm used by the **capture** command. Every device must support an *algorithm* of "none", which means no compression. This is the default. In this case, digital-video devices save still images as RGB format device-independent bitmaps. Devices might also support a device-specific list of additional algorithms.
still quality *descriptor*	Specifies the characteristics of the still-image compression performed while capturing a still image. All devices support the descriptors "low", "medium", and "high". The default is device specific. If the "algorithm" flag is not used, the *descriptor* applies to the current algorithm. The **quality** command can be used to define other descriptor names.
stream to *number*	Specifies the video stream played back from the workspace. If the stream is not specified and a default stream is not defined by the file format, then the physically first interleaved video stream is played.
tint to *factor*	Sets the image tint. Typically, this adjustment is modeled after the tint control of many color television sets, with 250 meaning green, 750 meaning red, and 0 (or 1000) meaning blue. The nominal value is always 500.
track *track_number* off	Disables an individual video track.
track *track_number* on	Enables an individual video track.

lpszFlags

Can be "wait", "notify", "test", or a combination of these. For more information about these flags, see Chapter 3, "MCI Overview."

For VCR devices, using **setvideo** with a flag that turns off an individual track ("track *track_number* off") might cause your application to receive a status message indicating that the command could not be carried out. Some VCRs can turn off only combinations of tracks, not individual tracks; for example, the first audio track and a video track of a video cassette. In this case, simply use **setaudio** and **setvideo** to continue to turn off the other tracks that make up the combination. The driver will turn off the tracks when it receives the command to turn off the last track in the combination.

signal

```
wsprintf(lpstrCommand, "signal %s %s %s", lpszDeviceID, lpszSignalFlags,
    lpszFlags);
```

Identifies a specified position in the workspace by sending the application an MM_MCISIGNAL message. Digital-video devices recognize this command. MCIAVI supports only one active signal at a time.

- Returns zero if successful or an error otherwise.

lpszDeviceID

Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszSignalFlags

One of the following flags:

at *position*	Specifies the frame to invoke a signal.
cancel	Removes signals from the workspace. An individual signal is specified by using the "uservalue" flag. If the "uservalue" flag is not specified by using "cancel", the device cancels all signals. The "cancel" flag is incompatible with the "at", "every", and "return position" flags.
every *interval*	Specifies the period of the signals. The *interval* value is specified in the current time format. If used with "at" *position*, signals are placed throughout the workspace with one signal mark placed at *position*. Without the "at" flag, signals are placed throughout the workspace with one signal at the current position. If this flag is omitted, only the position indicated by the "at" flag is marked. If the *interval* value is less than the minimum frequency supported by a device, it will use its minimum value.
return position	Indicates the device should send the position value instead of the "uservalue" identifier in the signaling message. The "uservalue" identifier can still be used to cancel or to redefine the signal marks.
uservalue *id*	Specifies an identifier that is reported back with the signaling message. This identifier acts as an identifier that can be used with other **signal** commands to reference this **signal** setting. If omitted, the default value is zero.

lpszFlags

Can be "wait", "notify", "test", or a combination of these. For more information about these flags, see Chapter 3, "MCI Overview."

spin

```
wsprintf(lpstrCommand, "spin %s %s %s", lpszDeviceID, lpszUpDown,
    lpszFlags);
```

Starts spinning a disc or stops the disc from spinning. Videodisc devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszUpDown
One of the following flags:

down	Stops the disc from spinning.
up	Starts spinning the disc.

lpszFlags
Can be "wait", "notify", or both. For more information about these flags, see Chapter 3, "MCI Overview."

The following command starts spinning a videodisc device:

```
spin videodisc up
```

status

```
wsprintf(lpstrCommand, "status %s %s %s", lpszDeviceID, lpszRequest,
    lpszFlags);
```

Requests status information from a device. All devices recognize this command.

- Returns information in the *lpstrReturnString* parameter of **mciSendString**. The information is dependent on the request type.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszRequest
Flag for requesting status information. The following table lists device types that recognize the **status** command and the flags used by each type:

animation	current track	position
	forward	position track *number*
	length	ready
	length track *number*	speed
	media present	start position
	mode	stretch
	number of tracks	time format
	palette handle	window handle
cdaudio	cdaudio type track *number*	number of tracks
	current track	position
	length	position track *number*
	length track *number*	ready
	media present	start position
	mode	time format
digitalvideo	audio	output
	audio alignment	palette handle
	audio bitspersample	pause mode
	audio breaks	play speed
	audio bytespersec	position
	audio input	position track *number*
	audio record	ready
	audio source	record frame rate
	audio samplespersec	reference *frame*
	audio stream	reserved size
	bass	right volume
	bitsperpel	seek exactly
	brightness	sharpness
	color	smpte
	contrast	speed
	current track	start position
	disk space *drive*	still file format
	file completion	time format
	file format	tint
	file mode	treble
	forward	unsaved
	frames skipped	video
	gamma	video key index
	input	video key color
	left volume	video record
	length	video source

	length track *number*	video source number
	media present	video stream
	mode	volume
	monitor	window handle
	monitor method	window visible
	nominal	window minimized
	nominal frame rate	window maximized
	nominal record frame rate	
	number of tracks	
overlay	media present	ready
	mode	stretch
	number of tracks	window handle
sequencer	current track	port
	division type	position
	length	position track *number*
	length track *number*	ready
	master	slave
	media present	start position
	mode	tempo
	number of tracks	time format
	offset	
vcr	assemble record	pause *timeout*
	audio monitor	play format
	audio monitor number	position
	audio record	position start
	audio record track *number*	position track *number*
	audio source	postroll *duration*
	audio source number	power on
	channel	preroll *duration*
	channel tuner *number*	ready
	clock	record format
	clock id	speed
	counter	time format
	counter format	time mode
	counter resolution	time type
	current track	timecode present
	frame rate	timecode record
	index	timecode type
	index on	tuner number
	length	video monitor
	length track *number*	video monitor number
	media present	video record
	media type	video record track *number*
	mode	video source
	number of audio tracks	video source number
	number of tracks	write protected
	number of video tracks	
videodisc	current track	number of tracks
	disc size	position
	forward	position track *number*
	length	ready
	length track *number*	side
	media present	speed
	media type	start position

	mode	time format
waveaudio	alignment	media present
	bitspersample	mode
	bytespersec	number of tracks
	channels	output
	current track	position
	format tag	position track *number*
	input	ready
	length	samplespersec
	length track *number*	start position
	level	time format

The following table lists the flags that can be specified in the *lpszRequest* parameter and their meanings:

alignment	Returns the block alignment of data, in bytes.
assemble record	Returns TRUE if the device is set to assemble mode recording.
audio	Returns "on" or "off" depending on the most recent **setaudio** "on" or "off" command. It returns "on" if either or both speakers are enabled, and "off" otherwise.
audio alignment	Returns the alignment of data blocks relative to the start of input waveform-audio data.
audio bitspersample	Returns the number of bits per sample the device uses for recording. This flag applies only to devices supporting the "pcm" algorithm.
audio breaks	Returns the number of times the audio portion of the last AVI sequence broke up. The system counts an audio break whenever it attempts to write audio data to the device driver and discovers that the driver has already played all of the available data. This flag is recognized only by the MCIAVI digital-video driver. It is meant for performance evaluation only; the return value is difficult to interpret.
audio bytespersec	Returns the average number of bytes per second used for recording.
audio input	Returns the approximate instantaneous audio level of the analog input audio signal. A value greater than 1000 implies clipping distortion. Some devices can return this value only while recording audio. The value has no associated **set** or **setaudio** command.
audio monitor	Returns "output", or one of the valid source-input types. For more information,

	see the **setaudio** "monitor" command.
audio monitor number	Returns the monitored-video number of the type specified by **status** "audio monitor". For more information, see the **setaudio** command.
audio record	Returns "on" or "off", reflecting the state set by **setaudio** "record".
audio record track *number*	Returns TRUE if the VCR is set to record audio. If no track number is given, the default value of 1 is assumed.
audio samplespersec	Returns the number of samples per second recorded.
audio source	Returns the current audio digitizer source: "left", "right", "average", or "stereo".
audio source number	Returns the audio-source number of the type returned by **status** "audio source". For more information, see the **setaudio** command.
audio stream	Returns the current audio-stream number.
bass	Returns the current audio-bass level.
bitsperpel	Returns the number of bits per pixel for saving captured or recorded data.
bitspersample	Returns the bits per sample.
brightness	Returns the current video-brightness level.
bytespersec	Returns the average number of bytes per second played or recorded.
cdaudio type track *number*	Returns the type of the specified track number. This can be "audio" or "other."
channel	Returns the integer value of the channel set on the tuner.
channel tuner *number*	If "tuner" *number* is given, then the currently selected channel on the logical tuner *number* will be returned. Note that there can be several logical tuners.
channels	Returns the number of channels set (1 for mono, 2 for stereo).
clock	Returns the external time. The time must be an unsigned long integer expressing total increments. For more information, see the **capability** "clock increment rate" command.
clock id	Returns a unique integer identifying the clock.
color	Returns the current color level.
contrast	Returns the current contrast level.
counter	Returns the counter position, in the current counter format.

counter format	Returns the current counter format. For more information, see the **set** "counter format" command.
counter resolution	Returns "frames" or "seconds", indicating the counter's resolution. This is not the same as accuracy.
current track	Returns the current track. The MCISEQ sequencer returns 1.
disc size	Returns either 8 or 12, indicating the size of the loaded disc in inches.
disk space *drive*	Returns the approximate disk space, in the current time format, that can be obtained by a **reserve** command for the specified disk *drive.* The *drive* is usually specified as a single letter or a single letter followed by a colon (:). Some devices, however, might use a path.
division type	Returns one of the following file division types: PPQN SMPTE 24 frame SMPTE 25 frame SMPTE 30 drop frame SMPTE 30 frame Use this information to determine the format of the MIDI file and the meaning of tempo and position information.
file completion	Returns the estimated percentage a **load**, **save**, **capture**, **cut**, **copy**, **delete**, **paste**, or **undo** operation has progressed. (Applications can use this to provide a visual indicator of progress.)
file format	Returns the current file format for **record** or **save** commands.
file mode	Returns "loading", "saving", "editing", or "idle". During a **load** operation, it returns "loading". During **save** and **capture** operations, it returns "saving". During **cut**, **copy**, **delete**, **paste**, or **undo** operations, it returns "editing".
format tag	Returns the format tag.
forward	Returns TRUE if the play direction is forward or if the device is not playing.
frame rate	Returns the number of frames per second that the device will use by default. NTSC devices return 30, PAL 25, and so on.
frames skipped	Returns the number of frames that were not drawn when the last AVI sequence was played. This flag is recognized only by the MCIAVI digital-video driver. It is

	meant for performance evaluation only; the return value is difficult to interpret.
gamma	Returns the value set with **setvideo** "gamma to" *value*.
index	Returns the current index display. For more information, see the **set** "index" command.
index on	Returns TRUE if the index is on.
input	Returns the input set. If one is not set, the error returned indicates that any device can be used.
	For digital-video devices, modifies the "bass", "treble", "volume", "brightness", "color", "contrast", "gamma", "sharpness", or "tint" flag so that it applies only to the input. This is the default when monitoring the input.
left volume	Returns the volume set for the left audio channel.
length	Returns the total length of the media, in the current time format.
	For PPQN files, the length is returned in song pointer units. For SMPTE files, it is returned as *hh*:*mm*:*ss*:*ff*, where *hh* is hours, *mm* is minutes, *ss* is seconds, and *ff* is frames. For VCR devices, the length is 2 hours (unless the length has been explicitly changed using the **set** command).
length track *number*	Returns the length of the track, in time or frames, specified by *number*.
	For PPQN files, the length is returned in song pointer units. For SMPTE files, it is returned as *hh*:*mm*:*ss*:*ff*, where *hh* is hours, *mm* is minutes, *ss* is seconds, and *ff* is frames.
level	Returns the current PCM audio sample value.
master	Returns "midi", "none", or "smpte" depending on the type of synchronization set.
media present	Returns TRUE if the media is inserted in the device or FALSE otherwise. Sequencer, video-overlay, digital-video, and waveform-audio devices return TRUE.
media type	Returns the type of the media. For VCRS, this is "8mm", "vhs", "svhs", "beta", "Hi8", "edbeta", or "other". For videodiscs, this is "CAV", "CLV", or "other", depending on the type of

	videodisc.
mode	Returns the current mode of the device. All devices can return the "not ready", "paused", "playing", and "stopped" values. Some devices can return the additional "open", "parked", "recording", and "seeking" values.
monitor	Returns "file" or "input". The returned value indicates the current presentation source.
monitor method	Returns "pre", "post", or "direct". The returned value indicates the method used for input monitoring.
nominal	The item modifies the "bass", "brightness", "color", "contrast", "gamma", "sharpness", "tint", "treble," and "volume" flags to return the nominal value instead of the current setting.
nominal frame rate	Returns the nominal frame rate associated with the file. The units are in frames per second multiplied by 1000.
nominal record frame rate	Returns the nominal frame rate associated with the input video signal. The units are in frames per second multiplied by 1000.
number of audio tracks	Returns the number of audio tracks on the media.
number of tracks	Returns the number of tracks on the media. The MCISEQ and MCIWAVE devices return 1, as do most VCR devices. The MCIPIONR device does not support this flag.
number of video tracks	Returns the number of video tracks on the media.
offset	Returns the offset of a SMPTE-based file. The offset is the start time of a SMPTE-based sequence. The time is returned as *hh*:*mm*:*ss*:*ff*, where *hh* is hours, *mm* is minutes, *ss* is seconds, and *ff* is frames.
output	Returns the currently set output. If no output is set, the error returned indicates that any device can be used. For digital-video devices, modifies the "bass", "treble", "volume", "brightness", "color", "contrast", "gamma", "sharpness", or "tint" flag so that it applies only to the output. This is the default when monitoring a file.
palette handle	Returns the handle of the palette used for the animation in the low-order word of

	the return value.
pause mode	Returns "recording" if the device is paused while recording. It returns "playing" if the device is paused while playing. It returns the error "Action not applicable in current mode" if the device is not paused.
pause timeout	Returns the maximum duration, in milliseconds, of a **pause** command.
play format	Returns a code indicating the format that the videotape will be played back in, if detectable: "lp", "ep", "sp", or "other". For more information, see the "record format" flag.
play speed	Returns a value representing how closely the actual playing time of the last AVI sequence matched the target playing time. The value 1000 indicates that the target time and the actual time were the same. A value of 2000, for example, would indicate that the AVI sequence took twice as long to play as it should have. This flag is recognized only by the MCIAVI digital-video driver. It is meant for performance evaluation only; the return value is difficult to interpret.
port	Returns the MIDI port number assigned to the sequence.
position	Returns the current position. For PPQN files, the position is returned in song pointer units. For SMPTE files, it is returned as *hh*:*mm*:*ss*:*ff*, where *hh* is hours, *mm* is minutes, *ss* is seconds, and *ff* is frames.
position start	Returns the position of the start of the usable media.
position track *number*	Returns the position of the start of the track specified by *number*. For PPQN files, the time format is returned in song pointer units. For SMPTE files, it is returned as *hh*:*mm*:*ss*:*ff*, where *hh* is hours, *mm* is minutes, *ss* is seconds, and *ff* is frames. The MCISEQ sequencer returns zero. The MCIPIONR device does not support this flag. The MCIWAVE device returns zero.
postroll duration	Returns the length of videotape, in the current time format, needed to brake the VCR transport when a **stop** or **pause** command is issued.
power on	Returns TRUE if the VCR's power is on.

preroll duration	Returns the length of videotape, in the current time format, needed to stabilize the VCR output.
ready	Returns TRUE if the device is ready to accept another command.
record format	Returns a code indicating the format that the videotape will be recorded in. The current return types are "lp", "ep", "sp", or "other". These formats are not VHS specific and can be applied to any VCR that has multiple selectable recording formats. The "sp" type is the fastest, highest quality recording format and is used as default on single format VCRs.
record frame rate	Returns the frame rate, in frames per second multiplied by 1000, used for compression.
reference *frame*	Returns the frame number for the nearest key frame image that precedes the specified *frame*.
reserved size	Returns the size, in the current time format, of the reserved workspace. The size corresponds to the approximate time it would take to play the compressed data from a full workspace. It returns zero if there is no reserved disk space. This flag returns the approximate size because the precise disk space for compressed data cannot, in general, be predicted until after the data has been compressed.
right volume	Returns the volume set for the right audio channel.
samplespersec	Returns the number of samples per second played or recorded.
seek exactly	Returns "on" or "off", indicating whether or not the "seek exactly" flag is set.
sharpness	Returns the current sharpness level of the device.
side	Returns 1 or 2 to indicate which side of the videodisc is loaded.
slave	Returns "file", "midi", "none", or "smpte" depending on the type of synchronization set.
smpte	Returns the SMPTE timecode associated with the current position in the workspace. This is a string with the form *dd*:*dd*:*dd*:*dd*, where each *d* denotes a digit from 0 to 9. If the workspace data does not include timecode data, then this flag returns 00:00:00:00.
speed	Returns the current speed of the device

	in frames per second (or in the same format used by the **set** "speed" command). The MCIPIONR videodisc player does not support this flag.
start position	Returns the starting position of the media.
still file format	Returns the current file format for the **capture** command.
stretch	Returns TRUE if stretching is enabled.
tempo	Returns the current tempo of a MIDI sequence in the current time format. For files with PPQN format, the tempo is in beats per minute. For files with SMPTE format, the tempo is in frames per second.
time format	Returns the current time format. For more information, see the time formats in the **set** command.
time mode	Returns the current position time mode. It can be "detect", "timecode", or "counter".
time type	Returns the current position time in use: "timecode" or "counter".
timecode present	Returns TRUE if timecode has been recorded at the current position on the tape. The timecode must advance from the current position. A VCR might need to be played to check this condition.
timecode record	Returns TRUE if the VCR is set to record timecode.
timecode type	Returns "smpte", "smpte drop", "other", or "none". Note the frames per second can be obtained from the **status** "frame rate" command, and the accuracy of the device can be returned by the **capability** "seek accuracy" command.
tint	Returns the current video-tint level.
treble	Returns the current audio-treble level.
tuner number	Returns the current logical-tuner number.
unsaved	Returns TRUE if there is recorded data in the workspace that might be lost as a result of a **close**, **load**, **record**, **reserve**, **cut**, **delete**, or **paste** command. Returns FALSE otherwise.
video	Returns "on" or "off", reflecting the state set by the **setvideo** command.
video key color	Returns the value for the key color.
video key index	Returns the value for the key index.
video monitor	Returns "output" or one of the valid source-input types. For more information,

	see the **setvideo** "monitor" command.
video monitor number	Returns the monitored-video number of the type returned by **status** "video monitor". For more information, see the **setvideo** command.
video record	Returns "on" or "off", reflecting the current state set by **setvideo** "record".
video record track *number*	Return TRUE if the VCR is set to record video. If no track number is given, the default value of 1 is assumed.
video source	Returns the video-source type. For more information, see the **setvideo** command.
video source number	Returns a number corresponding to the video source of the type in use. For example, it returns 2 if the second NTSC video source input is being used.
video stream	Returns the current video-stream number.
volume	Returns the average volume to the left and right speaker. This returns an error if the device has not been played or volume has not been set.
window handle	Returns ASCII decimal value for the window handle in the low-order word of the return value.
window maximized	Returns TRUE if the window is maximized.
window minimized	Returns TRUE if the window is minimized.
window visible	Returns TRUE if the window is not hidden.
write protected	Returns TRUE if the device detects that it cannot record (that is, if the write protect is on). If it can record, or if it is unable to determine whether or not it can record (without actually writing), the driver returns FALSE.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

Before issuing any commands that use position values, you should set the desired time format by using the **set** command.

The following command returns the current mode of the "mysound" device:

```
status mysound mode
```

step

```
wsprintf(lpstrCommand, "step %s %s %s", lpszDeviceID, lpszStepFlags,
    lpszFlags);
```

Steps the play one or more frames forward or reverse. The default action is to step forward one frame. Animation, digital-video, VCR, and CAV-format videodisc devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszStepFlags
One or both of the following flags:

by *frames*	Indicates the number of frames to step. If you specify a negative *frames* value, you cannot specify the "reverse" flag.
reverse	Step the frames in reverse.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

The following command plays five frames of the animation file associated with the "movie" device, starting at the current frame:

```
step movie by 5
```

stop

```
wsprintf(lpstrCommand, "stop %s %s %s", lpszDeviceID, lpszStopFlags,
    lpszFlags);
```

Stops playback or recording. Animation, CD audio, digital-video, MIDI sequencer, videodisc, VCR, and waveform-audio devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszStopFlags
For digital-video devices, it can be the following flag:

hold	Prevents the release of resources required to redraw a still image on the screen. The frame buffer remains available for use in updating the display when, for example, the window is moved.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

For CD audio devices, the **stop** command stops playback and resets the current track position to zero.

The following command stops playback or recording on the "mysound" device:

```
stop mysound
```

sysinfo

```
wsprintf(lpstrCommand, "sysinfo %s %s %s", lpszDeviceID, lpszRequest,
    lpszFlags);
```

Retrieves MCI system information. The **sysinfo** command is an MCI system command; it is interpreted directly by MCI.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device or device type. If a device type is specified, it must be a standard MCI device-type name, as listed in the reference material for the **capability** command. You can specify "all" when the flag specified in *lpszRequest* allows that possibility.

lpszRequest
One of the following flags:

installname	Returns the name listed in the registry or the SYSTEM.INI file used to install the open device with the specified device identifier.
quantity	Returns the number of MCI devices listed in the registry or the SYSTEM.INI file of the type specified in the *lpszDeviceID* parameter. This device identifier must be a standard MCI device-type name. Any digits after the device type are ignored. Specifying "all" for *lpszDeviceID* returns the total number of MCI devices in the system.
quantity open	Returns the number of open MCI devices of the type specified in *lpszDeviceID*. This device identifier must be a standard MCI device-type name. Specifying "all" for *lpszDeviceID* returns the total number of open MCI devices in the system.
name *index*	Returns the name of an MCI device. The device identifier must be a standard MCI device-type name. The *index* ranges from 1 to the number of devices of that type. If "all" is specified for *lpszDeviceID*, *index* ranges from 1 to the total number of devices in the system.
name *index* open	Returns the name of an open MCI device. The device identifier must be a standard MCI device-type name. The *index* ranges from 1 to the number of open devices of that device type. If "all" is specified for *lpszDeviceID*, *index* ranges from 1 to the total number of open devices in the system.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

The following command returns the number of open waveform-audio devices:

```
sysinfo waveaudio quantity open
```

The following command returns the name (device alias) of the first open waveform-audio device:

```
sysinfo waveaudio name 1 open
```

undo

```
wsprintf(lpstrCommand, "undo %s %s", lpszDeviceID, lpszFlags);
```

Reverses the action taken by the most recent successful **copy**, **cut**, **delete**, **undo**, or **paste** command. Digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszFlags
Can be "wait", "notify", "test", or a combination of these. For more information about these flags, see Chapter 3, "MCI Overview."

unfreeze

```
wsprintf(lpstrCommand, "unfreeze %s %s %s", lpszDeviceID, lpszUnfreeze,
    lpszFlags);
```

Reenables video acquisition to the frame buffer after it has been disabled by the **freeze** command. Digital-video, VCR, and video-overlay devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszUnfreeze
Flag for reenabling video acquisition to the frame buffer. The following table lists device types that recognize the **unfreeze** command and the flags used by each type:

digitalvideo	at *rectangle*
overlay	at *rectangle*
vcr	input output

The following table lists the flags that can be specified in the *lpszUnfreeze* parameter and their meanings:

at *rectangle*	Specifies the region that will have video acquisition reenabled. The rectangle is relative to the video buffer origin and is specified as *X1 Y1 X2 Y2*. The coordinates *X1 Y1* specify the upper left corner of the rectangle, and the coordinates *X2 Y2* specify the width and height.
input	Unfreeze the input image.
output	Unfreeze the output image. If neither "input" nor "output" is given, "output" is assumed.

lpszFlags
Can be "wait", "notify", or both. For digital-video and VCR devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

The following command unfreezes a region of the video buffer:

```
unfreeze vboard at 10 20 90 165
```

update

```
wsprintf(lpstrCommand, "update %s %s %s", lpszDeviceID, lpszHDC,
    lpszFlags);
```

Repaints the current frame into the specified device context (DC). Animation and digital-video devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszHDC
Handle of a DC. The following table lists device types that recognize the **update** command and the flags used by each type:

animation	hdc *hdc*	hdc *hdc* at *rect*
digitalvideo	hdc *hdc* hdc *hdc* at *rect*	paint hdc *hdc*

The following table lists the flags that can be specified in the *lpszHDC* parameter and their meanings:

hdc *hdc*	Specifies the handle of the DC to paint.
hdc *hdc* at *rect*	Specifies the clipping rectangle relative to the client rectangle.
paint hdc *hdc*	Paints the DC when the application receives a WM_PAINT message intended for a DC.

To specify the handle of the DC, use the string "hdc" followed by an ASCII representation of the handle. The rectangle is specified as *X1 Y1 X2 Y2*. The coordinates *X1 Y1* specify the upper left corner of the rectangle, and the coordinates *X2 Y2* specify the width and height.

lpszFlags
Can be "wait", "notify", or both. For digital-video devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

The following command updates the entire display window used by the "movie" device. The number 203 is a handle to a DC obtained from the **BeginPaint** function:

```
update movie hdc 203
```

where

```
wsprintf(lpstrCommand, "where %s %s %s", lpszDeviceID, lpszRequestRect,
    lpszFlags);
```

Retrieves the rectangle specifying the source or destination area. This rectangle was specified using the **put** command. Animation, digital-video, and video-overlay devices recognize this command.

- Returns a rectangle in the *lpstrReturnString* parameter of the **mciSendString** function. The rectangle describes the area specified in the *lpszRequestRect* parameter of this command. The rectangle is specified as *X1 Y1 X2 Y2*. The coordinates *X1 Y1* specify the upper left corner of the rectangle, and the coordinates *X2 Y2* specify the width and height.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszRequestRect
Flag that identifies the rectangle whose dimensions are retrieved. The following table lists device types that recognize the **where** command and the flags used by each type:

animation	destination	source
digitalvideo	destination	source max
	destination max	video
	frame	video max
	frame max	window
	source	window max
overlay	destination	source
	frame	video

The following table lists the flags that can be specified in the *lpszRequestRect* parameter and their meanings:

destination	Retrieves the destination offset and extent. For video-overlay devices, the destination rectangle defines the area of the display window client area that displays the image data from the frame buffer.
destination max	Retrieves the current size of the client rectangle.
frame	Retrieves the offset and extent of the frame buffer rectangle. The frame buffer rectangle defines the area of the frame buffer that receives incoming video data. Images from the "video" rectangle are scaled into this region.
frame max	Returns the maximum size of the frame buffer.
source	Retrieves the source offset and extent. For video-overlay devices, the source rectangle defines the region of the frame buffer that is displayed in the destination window. The device uses this rectangle to crop the image before it is stretched to fit the destination rectangle on the display.
source max	Retrieves the maximum size of the frame buffer.
video	Retrieves the offset and extent of the video rectangle. The video rectangle defines the region of the incoming video data that is transferred to the frame buffer.
video max	Returns the maximum size of the input.
window	Retrieves the current size and position of the display-

	window frame.
window max	Retrieves the size of the entire display.

lpszFlags
Can be "wait", "notify", or both. For digital-video devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

The following command returns the display rectangle of the "movie" device:

```
where movie destination
```

window

```
wsprintf(lpstrCommand, "window %s %s %s", lpszDeviceID, lpszWindowFlags,
    lpszFlags);
```

Controls the display window. You can use this command to change the display characteristics of the window or provide a destination window for the driver to use in place of the default display window. Animation, digital-video, and video-overlay devices recognize this command.

- Returns zero if successful or an error otherwise.

lpszDeviceID
Identifier of an MCI device. This identifier or alias is assigned when the device is opened.

lpszWindowFlags
Flag for controlling the display window. The following table lists device types that recognize the **window** command and the flags used by each type:

animation	fixed	state restore
	handle default	state show
	handle *hwnd*	show maximized
	state hide	show minimized
	state iconic	show min noactive
	state maximized	show na
	state minimize	show noactivate
	state minimized	show normal
	state no action	stretch
	state no activate	text *caption*
	state normal	
digitalvideo	handle *hwnd*	show minimized
	state hide	show min noactive
	state minimize	show na
	state restore	show noactivate
	state show	show normal
	show maximized	text *caption*
overlay	fixed	state restore
	handle default	state show
	handle *hwnd*	show maximized
	state hide	show minimized
	state iconic	show min noactive
	state maximized	show na
	state minimize	show noactivate
	state minimized	show normal
	state no action	stretch
	state no activate	text *caption*
	state normal	

The following table lists the flags that can be specified in the *lpszWindowFlags* parameter and their meanings:

fixed	Disables stretching of the image.
handle default	Specifies that the device should set the display window back to the default window created during the **open** operation. For video-overlay devices, specifies that the device should create and manage its own destination window.
handle *hwnd*	Specifies the handle of the destination window to

	use instead of the default window. The *hwnd* parameter contains the ASCII numeric equivalent of the window handle returned by the **CreateWindow** function. Two device instances can use the same window handle provided that each instance updates the video and image pixels in the window as if the other instance did not exist. When video output is disabled with **setvideo** "off", an **update** command will make the destination rectangle a solid color.
show maximized	Maximizes the destination window.
show min noactive	Displays the destination window as an icon.
show minimized	Minimizes the destination window.
show na	Displays the destination window in its current state; the window that is currently active remains active.
show noactivate	Displays the destination window in its most recent size and position; the window that is currently active remains active.
show normal	Activates and displays the destination window in its original size and position. (This is the same as the "state restore" flag.)
state hide	Hides the destination window.
state iconic	Displays the destination window as an icon.
state maximized	Maximizes the destination window.
state minimize	Minimizes the destination window and activates the top-level window in the window-manager's list.
state minimized	Minimizes the destination window.
state no action	Displays the destination window in its current state. The window that is currently active remains active.
state no activate	Displays the destination window in its most recent size and state. The currently active window remains active.
state normal	Activates and displays the destination window in its original size and position.
state restore	Activates and displays the destination window in its original size and position.
state show	Shows the destination window.
stretch	Enables stretching of the image.
text *caption*	Specifies the caption for the destination window. If this text contains embedded blanks, the entire caption must be enclosed in quotation marks. The default caption for the default window is blank.

lpszFlags

Can be "wait", "notify", or both. For digital-video devices, "test" can also be specified. For more information about these flags, see Chapter 3, "MCI Overview."

Generally, animation devices create a window when opened but don't display the window until they receive a **play** command. Video-overlay devices, on the other hand, typically create and display a window when opened. If your application provides a window to the driver, your application is responsible for managing the messages sent to the window.

Since you can use the **status** command to retrieve the handle to the driver display window, you can also use the standard window manager functions (such as **ShowWindow**) to manipulate the window.

The following command displays and sets the caption for the "movie" playback window:

```
window movie text "Welcome to the Movies" state show
```

Video Capture

You can easily incorporate video capture capabilities into your application by using the AVICap window class. AVICap provides applications with a simple, message-based interface to access video and waveform-audio acquisition hardware and to control the process of streaming video capture to disk.

AVICap supports streaming video capture and single-frame capture in real-time. In addition, AVICap provides control of video sources that are Media Control Interface (MCI) devices so the user can control (through an application) the start and stop positions of a video source, and augment the capture operation to include step frame capture.

The windows you create by using the AVICap window class can perform the following tasks:

- Capture audio and video streams to an audo-video interleaved (AVI) file.
- Connect and disconnect video and audio input devices dynamically.
- View a live incoming video signal by using the overlay or preview methods.
- Specify a file to use when capturing and copy the contents of the capture file to another file.
- Set the capture rate.
- Display dialog boxes that control the video source and format.
- Create, save, and load palettes.
- Copy images and palettes to the clipboard.
- Capture and save a single image as a device-independent bitmap (DIB).

Video Capture: A Minimal Approach

Video capture digitizes a stream of video and audio data, and stores it on a hard disk or some other type of persistent storage device. This section describes how to add a simple form of video capture to an application using three statements of code. It also describes how to end or abort a capture session by sending messages to the capture window.

An AVICap capture window handles the details of streaming audio and video capture to AVI files. This frees your application from involvement in the AVI file format, video and audio buffer management, and the low-level access of video and audio device drivers. AVICap provides a flexible interface for applications. You can add video capture to your application with only the following lines of code:

```
hWndC = capCreateCaptureWindow ( "My Own Capture Window",
                  WS_CHILD | WS_VISIBLE , 0, 0, 160, 120, hwndParent, nID);
SendMessage (hWndC, WM_CAP_DRIVER_CONNECT, 0 /* wIndex */, 0L);
SendMessage (hWndC, WM_CAP_SEQUENCE, 0, 0L);
```

A macro interface is also available that provides an alternative to using the **SendMessage** function and improves the readability of an application. The following example uses the macro interface to add video capture to an application.

```
hWndC = capCreateCaptureWindow (   "My Own Capture Window",
                  WS_CHILD | WS_VISIBLE , 0, 0, 160, 120, hwndParent, nID);
capDriverConnect (hWndC, 0);
capCaptureSequence (hWndC);
```

Once your application creates a capture window of the AVICap window class and connects it to a video driver, the capture window is ready to capture data. At this point, your application can simply send the WM_CAP_SEQUENCE message (or the **capCaptureSequence** macro) to begin capturing.

Using default settings, WM_CAP_SEQUENCE initiates capture of video and audio to a file named CAPTURE.AVI. Capture continues until one of the following events occurs:

- The user presses the ESC key or a mouse button.
- Your application stops or aborts capture operation.
- The disk becomes full.

In an application, you can stop streaming captured data to a file by sending the WM_CAP_STOP (or the **capCaptureStop** macro) message to a capture window. You can also abort the capture operation by sending the WM_CAP_ABORT message (or the **capCaptureAbort** macro) to a capture window.

Basic Capture Options

By modifying one or more of the capture parameters defined in the **CAPTUREPARMS** structure, you can perform the following tasks:

- Change the frame capture rate.
- Specify keyboard or mouse control for ending a capture session.
- Specify a duration for a capture session.

Capture Rate

The capture rate is the number of frames that are captured each second of a capture session. You can retrieve the current capture rate by using the WM_CAP_GET_SEQUENCE_SETUP message (or the **capCaptureGetSetup** macro). The current capture rate is stored in the **dwRequestMicroSecPerFrame** member of the **CAPTUREPARMS** structure. You can set the capture rate by specifying the number of microseconds between successive frames as the value of this member, and then sending the updated **CAPTUREPARMS** structure to the capture window by using the WM_CAP_SET_SEQUENCE_SETUP message (or the **capCaptureSetSetup** macro). The default value of **dwRequestMicroSecPerFrame** is 66667, which corresponds to 15 frames per second.

Keys Ending Capture

You can allow the user to abort a capture session by pressing a key or keystroke combination from the keyboard, or by pressing the right or left mouse button. If the user aborts a real-time capture session, the contents of the capture file are discarded. If the user aborts a step-frame capture session, the contents of the capture file up to the point of aborting the capture are saved.

You can retrieve the settings for aborting a capture session by using the WM_CAP_GET_SEQUENCE_SETUP message (or the **capCaptureGetSetup** macro). The current keystroke setting is stored in the **vKeyAbort** member of the **CAPTUREPARAMS** structure; the current mouse settings are stored in the **fAbortLeftMouse** and **fAbortRightMouse** members. You can set a new key or keystroke combination by specifying the keycode or keycode combination (as in a CTRL or SHIFT key combination) as the value of **vKeyAbort**, or set the left or right mouse button as the abort key by specifying the **fAbortLeftMouse** or **fAbortRightMouse** member. After you set these members, send the updated **CAPTUREPARMS** structure to the capture window by using the WM_CAP_SET_SEQUENCE_SETUP message (or the **capCaptureSetSetup** macro). The default value of **vKeyAbort** is VK_ESCAPE. The default values of **fAbortLeftMouse** and **fAbortRightMouse** are TRUE.

Time Limit

You can limit the duration of a capture operation by using the **fLimitEnabled** and **wTimeLimit** members of the **CAPTUREPARAMS** structure. The **fLimitEnabled** member indicates whether the capture operation is to be timed, while **wTimeLimit** specifies the maximum duration of the capture operation.

You can retrieve the values for **fLimitEnabled** and **wTimeLimit** by using the WM_CAP_GET_SEQUENCE_SETUP message (or the **capCaptureGetSetup** macro). You can enable a timer for the capture operation by specifying TRUE as the value of **fLimitEnabled**, or you can set the duration of the capture operation by specifying a value, in seconds, for **wTimeLimit**. After you set these members, send the updated **CAPTUREPARMS** structure to the capture window by using the WM_CAP_SET_SEQUENCE_SETUP message (or the **capCaptureSetSetup** macro). The default value of **fLimitEnabled** is FALSE.

Capture Windows

Capture windows are conceptually similar to standard controls (such as buttons, list boxes, or scroll bars) for the Microsoft Windows operating system. Typically, capture windows use the WS_CHILD and WS_VISIBLE window styles.

Creating an AVICap Capture Window

You can create a capture window of the AVICap window class by using the **capCreateCaptureWindow** function. This function returns a window handle that identifies the capture window and is used by an application to send subsequent messages to the window.

You can create one or more capture windows in an application and connect each capture window to a different capture device.

Connecting a Capture Window to a Capture Driver

You can dynamically connect or disconnect a capture window to a capture driver. You can connect or associate a capture window with a capture driver by using the WM_CAP_DRIVER_CONNECT message (or the **capDriverConnect** macro). After a capture window and capture driver are connected, you can send device-specific messages to the capture driver associated with a capture window.

If you have more than one capture device installed on a system, you can connect a capture window to a particular video capture device driver by specifying an integer for the *wParam* parameter of the WM_CAP_DRIVER_CONNECT message. The integer is an index that identifies a video capture driver listed in the registry or in the [drivers] section of the SYSTEM.INI file. Use zero for the first index entry.

You can retrieve the name and version of an installed capture driver by using the the **capGetDriverDescription** function. Your application can use this function to enumerate the installed capture devices and drivers, so the user can select a capture device to connect to a capture window.

You can retrieve the name of the capture device driver connected to a capture window by using the WM_CAP_DRIVER_GET_NAME message (or the **capDriverGetName** macro). To retrieve the version of an installed capture driver, use the WM_CAP_DRIVER_GET_VERSION message (or the **capDriverGetVersion** macro).

You can disconnect a capture window from a capture driver by using the WM_CAP_DRIVER_DISCONNECT message (or the **capDriverDisconnect** macro).

When an capture window is destroyed, any connected video capture device drivers are automatically disconnected.

Parent-Child Window Interaction

Some system-level messages, such as WM_PALETTECHANGED and WM_QUERYNEWPALETTE, are sent only to top-level and overlapped windows. If a capture window is a child window, its parent must forward these messages.

Similarly, if the parent window changes size, it might need to send notification messages to the capture window. Conversely, if the dimensions of the captured video change, the capture window might need to send notification messages to the parent window. The simplest way to manage this is to always keep the capture window dimensions equal to the size of the captured video stream, notifying the parent whenever these dimensions change.

Capture Window Status

You can retrieve the current status of a capture window by using the WM_CAP_GET_STATUS message (or the **capGetStatus** macro). This message retrieves a copy of the **CAPSTATUS** structure with the current values of its members. The **CAPSTATUS** structure contains information regarding the dimensions of the image, scroll position, and whether overlay or preview of the image is enabled. Because the information represented in **CAPSTATUS** is dynamic, your application should refresh the contents of the structure whenever the size or format of the captured video stream might have changed (such as after displaying the video format of the capture driver).

Changing the dimensions of the capture window has no effect on the dimensions of the actual captured video stream. The format dialog box displayed by the video capture device driver controls the dimensions of the captured video stream.

Capture and Audio Drivers

A capture driver and the underlying hardware can dictate several aspects of video capture, including acceptable video sources, display options, formats, and compression options. An audio driver specifies the audio format and possibly a compression option used with captured audio data.

Capture Driver Capabilities

You can retrieve the hardware capabilities of the currently connected capture driver by using the WM_CAP_DRIVER_GET_CAPS message (or the **capDriverGetCaps** macro). This message returns the capabilities of the capture driver and underlying hardware in the **CAPDRIVERCAPS** structure.

Video Dialog Boxes

Each capture driver can provide up to four dialog boxes to control aspects of the video digitization and capture process, and to define compression attributes used in reducing the size of the video data. The contents of these dialog boxes are defined by the video capture driver.

The Video Source dialog box controls the selection of video input channels and parameters affecting the video image being digitized in the frame buffer. This dialog box enumerates the types of signals that connect the video source to the capture card (typically SVHS and composite inputs), and provides controls to change hue, contrast or saturation. If the dialog box is supported by a video capture driver, you can display and update it by using the WM_CAP_DLG_VIDEOSOURCE message (or the **CapDlgVideoSource** macro).

The Video Format dialog box controls selection of the digitized video frame dimensions and image-depth, and compression options of the captured video. If the dialog box is supported by a video capture driver, you can display and update it by using the WM_CAP_DLG_VIDEOFORMAT message (or the **capDlgVideoFormat** macro).

The Video Display dialog box controls the appearance of the video on the monitor during capture. The controls in this dialog box have no effect on the digitized video data, but they might affect the presentation of the digitized signal. For example, capture devices that support overlay might allow altering hue and saturation, key color, or alignment of the overlay. If the dialog box is supported by a video capture driver, you can display and update it by using the WM_CAP_DLG_VIDEODISPLAY message (or the **capDlgVideoDisplay** macro).

The Video Compression dialog box controls the post-capture video compression attributes. If the dialog box is supported by a video capture driver, you can display and update it by using the WM_CAP_DLG_VIDEOCOMPRESSION message (or the **capDlgVideoCompression** macro).

Preview and Overlay Modes

A capture driver can implement two methods for viewing an incoming video stream: preview mode and overlay mode. If a capture driver implements both methods, the user can choose which method to use.

Preview mode transfers digitized frames from the capture hardware to system memory and then displays the digitized frames in the capture window by using graphics device interface (GDI) functions. Applications might decrease the preview rate when the parent window loses focus, and increase the preview rate when the parent window gains focus. This action improves general system responsiveness because the preview operation is processor intensive.

Three messages control the preview operation. You can enable or disable preview mode by sending the WM_CAP_SET_PREVIEW message (or the **capPreview** macro) to a capture window, or you can set the rate at which frames are displayed in preview mode by sending the WM_CAP_SET_PREVIEWRATE message (or the **capPreviewRate** macro), or you can enable or disable scaling of the preview video by sending the WM_CAP_SET_SCALE message (or the **capPreviewScale** macro). When preview and scaling are both enabled, the captured video frame is stretched to the dimensions of the capture window. Enabling preview mode automatically disables overlay mode.

Overlay mode is a hardware function that displays the contents of the capture buffer on the monitor without using CPU resources. You can enable and disable overlay mode by sending the WM_CAP_SET_OVERLAY message (or the **capOverlay** macro) to a capture window. Enabling overlay mode automatically disables preview mode.

You can also set the scroll position of the video frame within the client area of the capture window for preview mode or overlay mode by sending the WM_CAP_SET_SCROLL message (or the **capSetScrollPos** macro) to a capture window.

Video Format

You can retrieve the structure that specifies the video format or the size of that structure by sending the WM_CAP_GET_VIDEOFORMAT message (or the **capGetVideoFormat** and **capGetVideoFormatSize** macros) to a capture window. You can set the format of captured video data by sending the WM_CAP_SET_VIDEOFORMAT message (or the **capSetVideoFormat** macro) to a capture window.

Video Capture Settings

The **CAPTUREPARMS** structure contains the control parameters for streaming video capture. This structure controls several aspects of the capture process, and allows you to perform the following tasks:

- Specify the frame rate.
- Specify the number of allocated video buffers.
- Disable and enable audio capture.
- Specify the time interval for the capture.
- Specify if an MCI device (VCR or videodisc) is used during capture.
- Specify keyboard or mouse control for ending streaming.
- Specify the type of video averaging applied during capture.

You can retrieve the current capture settings within the **CAPTUREPARMS** structure by sending the WM_CAP_GET_SEQUENCE_SETUP message (or the **capCaptureGetSetup** macro) to a capture window. You can set one or more current capture settings by updating the appropriate members of the **CAPTUREPARMS** structure and then sending the WM_CAP_SET_SEQUENCE_SETUP message (or the **capCaptureSetSetup** macro) and **CAPTUREPARMS** to a capture window.

Audio Format

You can retrieve the current capture format for audio data or the size of the audio format structure by sending the WM_CAP_GET_AUDIOFORMAT message (or the **capGetAudioFormat** and **capGetAudioFormatSize** macros) to a capture window. The default audio capture format is mono, 8-bit, 11 kHz PCM (Pulse Code Modulation). When you retrieve the format by using WM_CAP_GET_AUDIOFORMAT, always use the **WAVEFORMATEX** structure.

You can set the capture format for audio data by sending the WM_CAP_SET_AUDIOFORMAT message (or the **capSetAudioFormat** macro) to a capture window. When setting the audio format, you can pass a pointer to a **WAVEFORMAT**, **WAVEFORMATEX,** or **PCMWAVEFORMAT** structure, depending on the specified audio format.

Capture File and Buffers

This section describes tips and options for using the capture file and for specifying buffers for the capture operation.

Capture Filename

AVICap, by default, routes video and audio stream data from a capture window to a file named CAPTURE.AVI in the root directory of the current drive. You can specify an alternate filename by sending the WM_CAP_FILE_SET_CAPTURE_FILE message (or the **capFileSetCaptureFile** macro) to a capture window. This message specifies the filename; it does not create, allocate, or open the file. You can retrieve the current capture filename by sending the WM_CAP_FILE_GET_CAPTURE_FILE message (or the **capFileGetCaptureFile** macro) to a capture window.

Saving Captured Data to a New File

If the user wants to save captured data, you should save the contents of the capture file to another file by using the WM_CAP_FILE_SAVEAS message (or the **capFileSaveAs** macro). This message does not change the name or contents of the capture file. Your application must specify a name for the new file because the capture file retains its original filename.

Typically, a capture file is preallocated for the largest capture segment anticipated and only a portion of it might be used to capture data. This message copies only the portion of the capture file containing the capture data.

Disk Space Preallocation for the Capture File

Preallocating disk space for the capture file builds a file of a specified size on the disk before the start of a capture operation. Preallocating a capture file reduces the processing required while capture is in progress and results in fewer dropped frames. You can preallocate a capture file by using the WM_CAP_FILE_ALLOCATE message (or the **capFileAlloc** macro).

Typically, your application should preallocate enough disk space to contain the largest capture file anticipated. Preallocating disk space does not restrict the size of the captured file. The file size is automatically enlarged if the captured data exceeds the allocated space. Subsequent write operations to the capture file reuse the portions of disk space allocated for the file, preserving the size and state of fragmentation of the file.

You can also improve capture performance by defragmenting the capture file. To defragment the file, use a defragmentation utility such as Disk Defragmenter. If you use a defragmented capture file and later enlarge it, you should defragment the enlarged file. Enlarging a defragmented capture file can fragment the expanded portion of the file and reduce performance in the capture operation.

You might also improve performance by using an uncompressed disk for video capture. Compressing data during capture can limit capture throughput to the disk.

An application can reserve a permanent capture file to eliminate the time required to preallocate and defragment a file each time it is started. Because a capture file can require considerable disk space, and preallocating a capture file removes all data from an existing capture file, an application should let the user decide if the file is permanent or temporary.

Index Size

Each AVI file uses an index of a specified size to locate video and audio data chunks within the file. An entry in the index locates one video frame or waveform-audio buffer. Consequently, the value of the index size indirectly sets the upper limit on the number of frames that can be captured in a file.

You can retrieve the current index size by using the WM_CAP_GET_SEQUENCE_SETUP message (or the **capCaptureGetSetup** macro). The current index size is stored in the **dwIndexSize** member of the **CAPTUREPARAMS** structure. You can specify a new index size as the value of **dwIndexSize** and then send the updated **CAPTUREPARMS** structure to the capture window by using the WM_CAP_SET_SEQUENCE_SETUP message (or the **capCaptureSetSetup** macro). The default index size is 34,952 entries (allowing 32K frames and a proportional number of audio buffers).

Video and Audio Chunk Granularity

The chunk granularity is a logical block size for an AVI file that is used for writing and retrieving audio and video data chunks. When writing audio and video chunks to disk, AVICap adds filler chunks (RIFF "JUNK" chunks) as necessary to fill each logical block of data.

You can retrieve the current chunk granularity setting by using the WM_CAP_GET_SEQUENCE_SETUP message (or the **capCaptureGetSetup** macro). The current chunk granularity is stored in the **wChunkGranularity** member of the **CAPTUREPARMS** structure. You can specify a new chunk granularity as the value of **wChunkGranularity** and then send the updated **CAPTUREPARMS** structure to the capture window by using the WM_CAP_SET_SEQUENCE_SETUP message (or the **capCaptureSetSetup** macro). You can also specify zero for this member to set the chunk granularity to the sector size of the disk.

Video Buffers

The buffers used with video capture reside in the memory heap. The number of buffers used in a capture operation can vary and depend on the value of the **wNumVideoRequested** member of the **CAPTUREPARMS** structure and available system memory.

You can retrieve the current value of requested video buffers by using the WM_CAP_GET_SEQUENCE_SETUP message (or the **capCaptureGetSetup** macro). The current requested number of video buffers is stored in the **wNumVideoRequested** member of the **CAPTUREPARMS** structure. You can request the placement and number of buffers by updating this member, and then sending the updated **CAPTUREPARMS** structure to the capture window by using the WM_CAP_SET_SEQUENCE_SETUP message (or the **capCaptureSetSetup** macro).

Audio Buffers

You can control the audio portion of a capture operation in three ways:

- Include or exclude audio from the capture operation.
- Request a specific number of audio buffers.
- Request audio buffers be a specific size.

You can retrieve the settings for audio buffers by using the WM_CAP_GET_SEQUENCE_SETUP message (or the **capCaptureGetSetup** macro). The **fCaptureAudio** member of the **CAPTUREPARMS** structure specifies whether audio is included or excluded from the capture operation. The current requested number of audio buffers is stored in the **wNumAudioRequested** member, and the current audio buffer size is stored in the **dwAudioBufferSize** member. You can specify whether to include audio capture, and the size and number of audio buffers by updating these members, and then sending the updated **CAPTUREPARMS** structure to the capture window by using the WM_CAP_SET_SEQUENCE_SETUP message (or the **capCaptureSetSetup** macro).

By default, audio is included in the capture operation and four audio buffers are allocated. The default value of **fCaptureAudio** is TRUE. The default buffer size (the value of **dwAudioBufferSize**) can contain 0.5 seconds of audio data or 10K, whichever is greater.

Capture Variations

In addition to streaming capture based on a constant time interval, AVICap supports the following types of capture:

- Manual frame capture (programmable control frames that are captured)
- Still-image capture
- Capture without using disk storage
- Streaming capture from an MCI device (real-time and step-frame)

Manual Frame Capture

If you want to individually specify the frames to capture in a video stream, you can control the sequence by using the WM_CAP_SINGLE_FRAME_OPEN, WM_CAP_SINGLE_FRAME, and WM_CAP_SINGLE_FRAME_CLOSE messages (or the **capCaptureSingleFrameOpen**, **capCaptureSingleFrame**, and **capCaptureSingleFrameClose** macros). Typically, these messages are used to create animation by appending individual frames to the capture file. WM_CAP_SINGLE_FRAME_OPEN opens a file for a manually-driven capture operation. WM_CAP_SINGLE_FRAME captures an individual frame and appends it to the capture file. WM_CAP_SINGLE_FRAME_CLOSE closes the file used for manual frame capture.

Note This capture method does not support simultaneous audio capture with video capture.

Still-Image Capture

If you want to capture a single frame as a still image, you can use the WM_CAP_GRAB_FRAME_NOSTOP or WM_CAP_GRAB_FRAME message (or the **capGrabFrameNoStop** or **capGrabFrame** macro) to capture the digitized image in an internal frame buffer. You can freeze the display on the captured image by using WM_CAP_GRAB_FRAME. Otherwise, use WM_CAP_GRAB_FRAME_NOSTOP.

Once captured, you can copy the image for use by other applications. You can copy an image from the frame buffer to the clipboard by using the WM_CAP_EDIT_COPY message (or the **capEditCopy** macro). You can also copy the image from the frame buffer to a device-independent bitmap (DIB) by using the WM_CAP_FILE_SAVEDIB message (or the **capFileSaveDIB** macro).

Your application can also use the two single-frame capture messages to edit a sequence frame by frame, or to create a time-lapse photography sequence.

Capture Without Using Disk Storage

You can use capture services without writing the data to a disk file by using the WM_CAP_SEQUENCE_NOFILE message (or the **capCaptureSequenceNoFile** macro). This message is useful only in conjunction with callback functions that allow your application to use the video and audio data directly. For example, videoconferencing applications might use this message to obtain streaming video frames. The callback functions would transfer the captured images to the remote computer.

Streaming Capture from an MCI Device

MCI devices augment the capture operation in real-time capture and step-frame capture. You can specify the MCI device, such as a videodisc or video-cassette recorder (VCR), acting as the video source for your capture operation by using the WM_CAP_SET_MCI_DEVICE message (or the **capSetMCIDeviceName** macro) and specifying the name of the device. You can also retrieve the device name currently set by using the WM_CAP_GET_MCI_DEVICE message (or the **capGetMCIDeviceName** macro).

In real-time capture, the capture window synchronizes the capture operation and compensates for delays associated with positioning the MCI video source and initializing the resources (such as capture buffers) required for capturing data. The capture window expects a valid MCI video device to be installed in the system for capturing data this way.

Specifications for controlling an MCI device are stored in the members of the **CAPTUREPARMS** structure. MCI-compatible video sources include VCRs and laserdiscs. If the **fMCIControl** member of this structure is set to TRUE, the capture window coordinates MCI operation. The capture window uses the parameters specified in the **dwMCIStartTime** and **dwMCIStopTime** members to obtain the starting and stopping positions, in milliseconds, of the sequence. If the value of **fMCIControl** is FALSE, the video source is not treated as an MCI device and the contents of **dwMCIStartTime** and **dwMCIStopTime** are ignored.

You can use Media Player to quickly verify that an MCI video device is properly connected to the system. Playing a device with Media Player verifies that the MCI configuration for the device is correct. If an image appears on the video display, the video source is connected properly to the capture hardware.

In step-frame capture, the capture window synchronizes the capture operation and compensates for the delays associated with positioning the MCI video source and initializing the resources required for capturing data. In addition, the capture window ensures that no frames are dropped; it steps through the video frames individually, ensuring that the frame is captured and stored before capturing the next frame in the video stream.

Specifications for controlling step-frame capture are stored in the members of the **CAPTUREPARMS** structure. Step-frame capture uses the following members in addition to the members used for real-time capture: **fStepMCIDevice**, **fStepCaptureAt2x**, and **wStepCaptureAverageFrames**. If the **fStepMCIDevice** member is set to TRUE, the capture window coordinates step-frame capture. The capture window uses the parameters specified in the **dwMCIStartTime** and **dwMCIStopTime** members for the starting and stopping positions, in milliseconds, of the sequence. The capture window uses **fStepCaptureAt2x** to determine if the capture hardware should capture video frames at twice the normal resolution and uses **wStepCaptureAverageFrames** to specify the number of times each frame in the capture operation is sampled.

If **fStepMCIDevice** is FALSE, real-time capture is used instead of step-frame capture and the contents of **fStepCaptureAt2x**, and **wStepCaptureAverageFrames** are ignored.

If a step-frame capture is specified and **fStepCaptureAt2x** is set to TRUE, the capture hardware captures at twice the specified resolution. (The resolutions of both the height and width are doubled.) The software interpolates the pixels in the higher resolution image to produce the image at the specified resolution. This form of averaging can improve the edge definition of images in a frame. You can enable this option if the hardware does not support hardware-based decimation and you are capturing in the RGB format.

Note If your hardware supports hardware-based decimation, it can capture samples at a higher rate than specified and use these additional samples to obtain color definitions that are more consistent with the original image. The additional samples are discarded after they are used, and the hardware passes samples to the capture driver at the specified rate.

If a step-frame capture is specified, the **wStepCaptureAverageFrames** member specifies the number of times a frame is sampled when creating a frame based on the average sample. This averaging technique reduces the random digitization noise appearing in a frame. A typical value for the number of averages is 5.

For more information about MCI, see Chapter 3, "MCI Overview."

Advanced Capture Options

This section describes other options you can include in a capture operation.

Measuring Video Quality

One way to measure video quality is to limit the number of captured frames that are dropped during the capture operation. When streaming capture has finished, the quality value is compared with the ratio of dropped frames to total frames. If the percentage of dropped frames is greater than the value of the **wPercentDropForError** member of the **CAPTUREPARMS** structure, AVICap sends an error message to the error callback function if it is installed.

You can retrieve the current limit of dropped frames (expressed as a percentage) by using the WM_CAP_GET_SEQUENCE_SETUP message (or the **capCaptureGetSetup** macro). You can set a new limit by specifying a percentage as the value of the **wPercentDropForError** member of the **CAPTUREPARAMS** structure, and then sending the updated structure to the capture window by using the WM_CAP_SET_SEQUENCE_SETUP message (or the **capCaptureSetSetup** macro). The default value of **wPercentDropForError** is 10 percent.

User-Initiated Capture

You can retrieve the current value of the user-initiated capture flag by using the WM_CAP_GET_SEQUENCE_SETUP message (or the **capCaptureGetSetup** macro). The value of the flag is stored in the **fMakeUserHitOKToCapture** member of the **CAPTUREPARAMS** structure. You can provide the user with precise control over when to start a capture session by setting this member to TRUE. AVICap displays a dialog box after allocating all video and audio buffers for a capture session. This lets the user eliminate capture delays because of software initialization. If your applications uses a small number of video buffers, this dialog box is probably unnecessary. The default value is FALSE.

Working with Palettes

Initially, if the video capture format requires a palette, the capture window uses the palette supplied by the capture driver. This palette might consist of gray-scale values for black-and-white reproduction, or a broad selection of color values. You can retrieve an existing palette to replace the default palette by using the WM_CAP_PAL_PASTE or WM_CAP_PAL_OPEN message (or the **capPalettePaste** or **capPaletteOpen** macro). Alternatively, you can create a custom palette to replace the default palette by using the WM_CAP_PAL_AUTOCREATE or WM_CAP_PAL_MANUALCREATE message (or the **capPaletteAuto** or **capPaletteManual** macro). After you replace the default palette, the capture window and driver use the replacement palette until you create or open another palette.

The WM_CAP_PAL_AUTOCREATE or WM_CAP_PAL_MANUALCREATE message creates an optimized palette based on the current video input. This custom palette gives a video sequence the best color fidelity because it is based on colors that exist in the sequence. The capture window creates a three-dimensional histogram of the colors it samples. It reduces the number of colors by examining the absolute error between adjacent colors and consolidating those with the smallest error value.

When sending WM_CAP_PAL_AUTOCREATE, you must specify the number of frames for AVICap to sample and the size of the color palette. When specifying the number of frames, include enough frames to ensure that all colors in the sequence are sampled.

You can sample the current frame by using WM_CAP_PAL_MANUALCREATE. By using this message with several manually selected frames, you can create a palette that contains the colors you want to appear in the palette.

A palette can contain up to 256 colors. If you merge palettes or if the video sequence is to be displayed simultaneously with other video or images, you should use a smaller color selection so that colors from each image or video clip can coexist.

You save a new palette by using the WM_CAP_PAL_SAVE message (or the **capPaletteSave** macro) and later retrieve it by using the WM_CAP_PAL_OPEN message. You can save a palette for post-processing of the palette or for use in another application.

You can paste a palette from the clipboard into the capture window by using the WM_CAP_PAL_PASTE message. The capture window passes the palette to the capture driver. Other applications can copy palettes to the clipboard. You can also copy a palette to the clipboard by using the WM_CAP_EDIT_COPY message (or the **capEditCopy** macro). This message copies the video frame buffer, including the palette, onto the clipboard.

Embedding Information Chunks in an AVI File

You can insert information chunks, such as text or custom data, in an AVI file by using the WM_CAP_FILE_SET_INFOCHUNK message (or the **capFileSetInfoChunk** macro). You can also use this message to clear information chunks from an AVI file.

User Data Messages

You can associate data with a capture window by using the WM_CAP_GET_USER_DATA and WM_CAP_SET_USER_DATA messages (or the **capGetUserData** and **capSetUserData** macros). You can retrieve a **LONG** data value by using the WM_CAP_GET_USER_DATA message and set a **LONG** data value by using the WM_CAP_SET_USER_DATA message.

AVICap Callback Functions

Your application can register callback functions with a capture window to have it notify your application when the status changes, when errors occur, when video frame and audio buffers become available, and to yield during streaming capture. The following messages set the callback function.

Message	Description
WM_CAP_SET_CALLBACK_CAPCONTROL	Specifies the callback function in the application called to give precise control over capture start and end. You can also use the **capSetCallbackOnCapControl** macro to send this message.
WM_CAP_SET_CALLBACK_ERROR	Specifies the callback function in the application called when an error occurs. You can also use the **capSetCallbackOnError** macro to send this message.
WM_CAP_SET_CALLBACK_FRAME	Specifies the callback function in the application called when preview frames are captured. You can also use the **capSetCallbackOnFrame** macro to send this message.
WM_CAP_SET_CALLBACK_STATUS	Specifies the callback function in the application called when the status changes. You can also use the **capSetCallbackOnStatus** macro to send this message.
WM_CAP_SET_CALLBACK_VIDEOSTREAM	Specifies the callback function in the application called during streaming capture when a new video buffer becomes available. You can also use the **capSetCallbackOnVideoStream** macro to send this message.
WM_CAP_SET_CALLBACK_WAVESTREAM	Specifies the callback function in the application called during streaming capture when a new audio buffer becomes available. You can also use the **capSetCallbackOnWaveStream** macro to send this message.
WM_CAP_SET_CALLBACK_YIELD	Specifies the callback function in the application called when yielding during streaming capture. You can also use the **capSetCallbackOnYield** macro to send this message.

The following sections describe the different callback functions, the notifications sent to the functions, and their uses.

Precise Capture Control

A capture window can provide the capture-control callback function with precise control over the moments that streaming capture begin and end. The first message sent from the capture driver to the callback procedure sets the *nState* parameter to CONTROLCALLBACK_PREROLL after allocating all buffers and all other capture preparations are complete. This message gives the application the ability to preroll the video sources. (The callback function specifies *nState* as its second parameter.) The callback function then returns at the exact moment recording is to begin. A return value of TRUE from the callback function continues capture. A return value of FALSE aborts capture. Once capture begins, the callback function is called frequently with *nState* set to CONTROLCALLBACK_CAPTURING to allow the application to end capture by returning false.

Error

A capture window uses error notification messages to notify your application of AVICap errors, such as running out of disk space, attempting to write to a read-only file, failing to access hardware, or dropping too many frames. The content of an error notification includes a message identifier and a formatted text string ready for display. Your application can use the message identifier to filter the notifications and limit the messages to present to the user. A message identifier of zero indicates a new operation is starting and the callback function should clear any displayed error message.

Frame

A capture window uses frame callback notification messages to notify your application when a new video frame is available. The capture window enables these callback notifications only if the preview rate is non-zero and streaming capture is not in progress.

Status Callback Functions

A capture window can send messages to the status callback function while capturing video to disk or during other lengthy operations to notify your application of the progress of an operation. The status information includes a message identifier and a formatted text string ready for display. Your application can use the message identifier to filter the notifications and limit the messages to present to the user. During capture operations, the first message sent to the callback function is always IDS_CAP_BEGIN and the last is always IDS_CAP_END. A message identifier of zero indicates a new operation is starting and the callback function should clear the current status.

Videostream

Applications can use videostream callback functions during streaming capture to process a captured video frame. The capture window calls a videostream callback function just before writing each captured frame to the disk.

Wavestream

Applications use the wavestream callback functions during streaming capture to process audio buffers. The capture window calls a wavestream callback function just before writing each audio buffer to the disk.

Yield Callback Functions

Applications can use yield callback functions during streaming capture. (A yield callback function typically consists of a **PeekMessage**, **TranslateMessage**, **DispatchMessage** loop.) The capture window calls the yield callback function at least once for every captured video frame, but the exact rate depends on the frame rate and the overhead of the capture driver and disk.

Disabling Callback Functions

You can permanently or temporarily disable any of the callback functions by specifying NULL in place of the callback function when sending the appropriate message to set a callback function.

Using Video Capture

This section contains examples demonstrating how to perform the following tasks:

- Create a capture window.
- Connect to a capture driver.
- Enumerate installed capture drivers.
- Obtain the capabilities of a capture driver.
- Obtain the status of a capture window.
- Display dialog boxes to set video characteristics.
- Obtain and setting the video format.
- Preview video.
- Enable video overlay.
- Name the capture file.
- Preallocate disk space for the capture file.
- Format audio capture.
- Change a video capture setting.
- Capture data.
- Adding an information chunk.
- Add callback functions to an application.
- Create a status callback function.
- Create an error callback function.
- Create a frame callback function.

Creating a Capture Window

The following example creates a capture window by using the **capCreateCaptureWindow** function.

```
hWndC = capCreateCaptureWindow (
     (LPSTR) "My Capture Window", // window name if pop-up
     WS_CHILD | WS_VISIBLE,       // window style
     0, 0, 160, 120,              // window position and dimensions
     (HWND) hwndParent,
     (int) nID /* child ID */);
```

Connecting to a Capture Driver

The following example connects the capture window with the handle *hWndC* to the MSVIDEO driver and then disconnects them:

```
fOK = SendMessage (hWndC, WM_CAP_DRIVER_CONNECT, 0, 0L);
//
// Or, use the macro to connect to the MSVIDEO driver:
// fOK = capDriverConnect(hWndC, 0);
// .
// . Place code to set up and capture video here.
// .
capDriverDisconnect (hWndC);
```

Enumerating Installed Capture Drivers

The following example uses the **capGetDriverDescription** function to obtain the names and versions of the installed capture drivers.

```
char szDeviceName[80];
char szDeviceVersion[80];

for (wIndex = 0; wIndex < 10; wIndex++) {
    if (capGetDriverDescription (wIndex, szDeviceName,
        sizeof (szDeviceName), szDeviceVersion,
        sizeof (szDeviceVersion))
    {
        \\ Append name to list of installed capture drivers
        \\ and then let the user select a driver to use.
    }
}
```

Obtaining the Capabilities of a Capture Driver

The WM_CAP_DRIVER_GET_CAPS message returns the capabilities of the capture driver and underlying hardware in the **CAPDRIVERCAPS** structure. Each time an application connects a new capture driver to the capture window, it should update the **CAPDRIVERCAPS** structure. The following example obtains the capture driver capabilities.

```
CAPDRIVERCAPS CapDrvCaps;
// .
// .
// .
SendMessage (hWndC, WM_CAP_DRIVER_GET_CAPS,
     sizeof (CAPDRIVERCAPS), (LONG) (LPVOID) &CapDrvCaps);
//
// Or, use the macro to retrieve the driver capabilities.
// capDriverGetCaps(hWndC, &CapDrvCaps, sizeof (CAPDRIVERCAPS));
```

Obtaining the Status of a Capture Window

The following example sets the size of the capture window to the overall dimensions of the incoming video stream based on information returned in the **CAPSTATUS** structure.

```
CAPSTATUS CapStatus;
// .
// .
// .
capGetStatus(hWndC, &CapStatus, sizeof (CAPSTATUS));
SetWindowPos(hWndC, NULL, 0, 0, CapStatus.uiImageWidth,
    CapStatus.uiImageHeight, SWP_NOZORDER | SWP_NOMOVE);
```

Displaying Dialog Boxes to Set Video Characteristics

Each capture driver can provide up to three different dialog boxes used to control aspects of the video digitization and capture process. The following example demonstrates how to display these dialog boxes. Before displaying each dialog box, the example checks the **CAPDRIVERCAPS** structure to see if the capture driver can display it.

```
CAPDRIVERCAPS CapDrvCaps;
// .
// .
// .
capDriverGetCaps(hWndC, &CapDrvCaps, sizeof (CAPDRIVERCAPS));

// Video source dialog box.
if (CapDriverCaps.fHasDlgVideoSource)
    capDlgVideoSource(hWndC);

// Video format dialog box.
if (CapDriverCaps.fHasDlgVideoFormat) {
    capDlgVideoFormat(hWndC);

    // New image dimensions?
    capGetStatus(hWndC, &CapStatus, sizeof (CAPSTATUS));

    // If so, notify the parent of a size change.
}

// Video display dialog box.
if (CapDriverCaps.fHasDlgVideoDisplay)
    capDlgVideoDisplay(hWndC);
```

Obtaining and Setting the Video Format

The **BITMAPINFO** structure is of variable length to accommodate standard and compressed data formats. Because this structure is of variable length, applications must always query the size of the structure and allocate memory before retrieving the current video format. The following example uses the **capGetVideoFormatSize** macro to retrieve the buffer size and then calls the **capGetVideoFormat** macro to retrieve the current video format.

```
LPBITMAPINFO lpbi;
DWORD dwSize;

dwSize = capGetVideoFormatSize(hWndC);
lpbi = GlobalAllocPtr (GHND, dwSize);
capGetVideoFormat(hWndC, lpbi, dwSize);

// Access the video format and then free the allocated memory.
```

Applications can use the **capSetVideoFormat** macro (or the WM_CAP_SET_VIDEOFORMAT message) to send a **BITMAPINFO** header structure to the capture window. Because video formats are device specific, your application should check the return value to determine if the format was accepted.

Previewing Video

The following example sets the frame display rate for preview mode to 66 milliseconds per frame and then places the capture window in preview mode.

```
capPreviewRate(hWndC, 66);     // rate, in milliseconds
capPreview(hWndC, TRUE);       // starts preview
// .
// .
// .
capPreview(hWnd, FALSE);        // disables preview
```

Enabling Video Overlay

The following example determines if a capture driver has overlay capabilities; if it does, it enables the overlay.

```
CAPDRIVERCAPS CapDrvCaps;
// .
// .
// .
capDriverGetCaps(hWndC, &CapDrvCaps, sizeof (CAPDRIVERCAPS));
if (CapDrvCaps.fHasOverlay)
    capOverlay(hWndC, TRUE);
```

Naming the Capture File

The following example specifies an alternate filename (MYCAP.AVI) for the capture file and preallocates the file to 5 MB.

```
char szCaptureFile[] = "MYCAP.AVI";

capFileSetCaptureFile( hWndC , szCaptureFile);
capFileAlloc( hWndC, (1024L * 1024L * 5));
```

Formatting Audio Capture

The following example sets the audio format to 11-kHz PCM 8-bit, stereo.

```
WAVEFORMATEX wfex;
// .
// .
// .
wfex.wFormatTag = WAVE_FORMAT_PCM;
wfex.nChannels = 2;                    // Use stereo
wfex.nSamplesPerSec = 11025;
wfex.nAvgBytesPerSec = 22050;
wfex.nBlockAlign = 2;
wfex.wBitsPerSample = 8;
wfex.cbSize = 0;

capSetAudioFormat(hWndC, &wfex, sizeof(WAVEFORMATEX));
```

Changing a Video Capture Setting

The following example changes the capture rate from the default value (15 frames per second) to 10 frames per second.

```
CAPTUREPARMS CaptureParms;
float FramesPerSec = 10.0;
// .
// .
// .
capCaptureGetSetup(hWndC, &CaptureParms, sizeof(CAPTUREPARMS));

CaptureParms.dwRequestMicroSecPerFrame = (DWORD) (1.0e6 / FramesPerSec);
capCaptureSetSetup(hWndC, &CaptureParms, sizeof (CAPTUREPARMS));
```

Capturing Data

The following example starts video capture and copies the captured data from the capture file to the file NEWFILE.AVI.

```
char szNewName[] = "NEWFILE.AVI";
// .
// . Set up the capture operation.
// .
capCaptureSequence(hWndC);
// .
// .
// .
capFileSaveAs(hWndC, szNewName);
```

Adding an Information Chunk

If your application needs to include other information in addition to audio and video, you can create information chunks and insert them into a capture file. Information chunks can contain several types of information, including the details of a copyright notice, identification of the video source, or external timing information. The following example stores external timing information – a SMPTE (Society of Motion Picture and Television Engineers) timecode – in an information chunk and adds the chunk to a capture file.

```
//  This example assumes the application controls
//  the video source for preroll and postroll.
CAPINFOCHUNK cic;
// .
// .
// .
cic.fccInfoID = infotypeSMPTE_TIME;
cic.lpData = "00:20:30:12";
cic.cbData = strlen (cic.lpData) + 1;
capFileSetInfo (hwndC, &cic);
```

Adding Callback Functions to an Application

An application can register callback functions with the capture window to have it notify the application when the status changes, when errors occur, when video frame and audio buffers become available, and to yield during streaming capture.

The following example creates a capture window and registers status, error, video stream, and frame callback functions in the message processing loop of an application. It also includes a sample statement for disabling a callback function. Subsequent examples describe the status, error, and frame callback functions.

```
case WM_CREATE:
{
    char    achDeviceName[80] ;
    char    achDeviceVersion[100] ;
    char    achBuffer[100] ;
    WORD    wDriverCount = 0 ;
    WORD    wIndex ;
    WORD    wError ;
    HMENU   hMenu ;

    // Create the capture window.
    ghWndCap = capCreateCaptureWindow((LPSTR)"Capture Window",
        WS_CHILD | WS_VISIBLE, 0, 0, 160, 120, (HWND) hWnd, (int) 0);

    // Register the error callback function before connecting capture
    // driver.
    fpErrorCallback = MakeProcInstance((FARPROC)ErrorCallbackProc,
        ghInst);
    capSetCallbackOnError(ghWndCap, fpErrorCallback);

    // Register the status callback function.
    fpStatusCallback = MakeProcInstance((FARPROC)StatusCallbackProc,
        ghInst);
    capSetCallbackOnStatus(ghWndCap, fpStatusCallback);

    // Register the video-stream callback function.
    fpVideoCallback = MakeProcInstance((FARPROC)VideoCallbackProc,
        ghInst);
    capSetCallbackOnVideoStream(ghWndCap, fpVideoCallback);

    // Register the frame callback function.
    fpFrameCallback = MakeProcInstance((FARPROC)FrameCallbackProc,
        ghInst);
    capSetCallbackOnFrame(ghWndCap, fpFrameCallback);

     // .
     // . Connect to a capture driver
     // .
     break;
}
case WM_CLOSE:
{

// Use the following macro to disable the frame callback.
```

```
// Similar macros exist for disabling other callback functions.
//
// FreeProcInstance(fpFrameCallback);
// capSetCallbackOnFrame(hWndC, NULL);
break;
}
```

Creating a Status Callback Function

The following example is a simple status callback function.

```
// StatusCallbackProc: Status Callback Function
// hWnd:           capture window handle
// nID:            status code for the current status
// lpStatusText:   status text string for the current status
//
LRESULT FAR PASCAL StatusCallbackProc(HWND hWnd, int nID,
    LPSTR lpStatusText)
{
    if (!ghWndMain)
        return FALSE;

    if (nID == 0) {              // Zero means clear old status messages
        SetWindowText(ghWndMain, (LPSTR) gachAppName);
        return (LRESULT) TRUE;
    }
    // Show the status ID and status text...
    wsprintf(gachBuffer, "Status# %d: %s", nID, lpStatusText);

    SetWindowText(ghWndMain, (LPSTR)gachBuffer);
    return (LRESULT) TRUE;
}
```

Creating an Error Callback Function

The following example is a simple error callback function.

```
// ErrorCallbackProc: Error Callback Function
// hWnd:          capture window handle
// nErrID:        error code for the encountered error
// lpErrorText:   error text string for the encountered error
//
LRESULT FAR PASCAL ErrorCallbackProc(HWND hWnd, int nErrID,
    LPSTR lpErrorText)
{

    if (!ghWndMain)
        return FALSE;

    if (nErrID == 0)            // starting a new major function
        return TRUE;            // clear out old errors

    // Show the error identifier and text
    wsprintf(gachBuffer, "Error# %d", nErrID);

    MessageBox(hWnd, lpErrorText, gachBuffer,
        MB_OK | MB_ICONEXCLAMATION);

    return (LRESULT) TRUE;
}
```

Creating a Frame Callback Function

The following example is a simple frame callback function.

```
// FrameCallbackProc: Frame Callback Function
// Called whenever a new frame is captured but not streaming
// hWnd:       capture window handle
// lpVHdr:     long pointer to VideoHdr struct containing captured
//             frame information
//
LRESULT FAR PASCAL FrameCallbackProc(HWND hWnd, LPVIDEOHDR lpVHdr)
{
    if (!ghWndMain)
        return FALSE;

    wsprintf(gachBuffer, "Preview frame# %ld ", gdwFrameNum++);
    SetWindowText(ghWndMain, (LPSTR)gachBuffer);
    return (LRESULT) TRUE ;
}
```

Video Capture Reference

This section describes the functions, messages and macros, and structures associated with the AVICap window class. These elements are grouped as follows.

Basic Capture Operations

capCreateCaptureWindow
WM_CAP_ABORT
WM_CAP_DRIVER_CONNECT
WM_CAP_SEQUENCE
WM_CAP_STOP

Capture Windows

CAPSTATUS
capGetDriverDescription
WM_CAP_DRIVER_CONNECT
WM_CAP_DRIVER_DISCONNECT
WM_CAP_GET_STATUS

Capture Drivers

CAPDRIVERCAPS
WM_CAP_DRIVER_GET_CAPS
WM_CAP_DRIVER_GET_NAME
WM_CAP_DRIVER_GET_VERSION
WM_CAP_GET_AUDIOFORMAT
WM_CAP_GET_VIDEOFORMAT
WM_CAP_SET_AUDIOFORMAT
WM_CAP_SET_VIDEOFORMAT

Capture Driver Preview and Overlay Modes

WM_CAP_SET_OVERLAY
WM_CAP_SET_PREVIEW
WM_CAP_SET_PREVIEWRATE
WM_CAP_SET_SCALE
WM_CAP_SET_SCROLL

Capture Driver Video Dialog Boxes

WM_CAP_DLG_VIDEOCOMPRESSION
WM_CAP_DLG_VIDEODISPLAY
WM_CAP_DLG_VIDEOFORMAT
WM_CAP_DLG_VIDEOSOURCE

Audio Format

WM_CAP_GET_AUDIOFORMAT
WM_CAP_SET_AUDIOFORMAT

Video Capture Settings

CAPTUREPARMS
WM_CAP_GET_SEQUENCE_SETUP
WM_CAP_SET_SEQUENCE_SETUP

Capture File and Buffers

CAPTUREPARMS
WM_CAP_FILE_ALLOCATE
WM_CAP_FILE_GET_CAPTURE_FILE
WM_CAP_FILE_SAVEAS

WM_CAP_FILE_SET_CAPTURE_FILE

Directly Using Capture Data

WM_CAP_SEQUENCE_NOFILE

Capture from MCI Device

WM_CAP_SET_MCI_DEVICE

Manual Frame Capture

WM_CAP_SINGLE_FRAME
WM_CAP_SINGLE_FRAME_CLOSE
WM_CAP_SINGLE_FRAME_OPEN

Still-Image Capture

WM_CAP_EDIT_COPY
WM_CAP_FILE_SAVEDIB
WM_CAP_GRAB_FRAME
WM_CAP_GRAB_FRAME_NOSTOP

Advanced Capture Options

WM_CAP_FILE_SET_INFOCHUNK
WM_CAP_GET_USER_DATA
WM_CAP_SET_USER_DATA

Working with Palettes

WM_CAP_EDIT_COPY
WM_CAP_PAL_AUTOCREATE
WM_CAP_PAL_MANUALCREATE
WM_CAP_PAL_OPEN
WM_CAP_PAL_PASTE
WM_CAP_PAL_SAVE

Yielding to Other Applications

WM_CAP_GET_SEQUENCE_SETUP
WM_CAP_SET_CALLBACK_YIELD
WM_CAP_SET_SEQUENCE_SETUP

AVICap Callback Functions

capControlCallback
capErrorCallback
capStatusCallback
capVideoStreamCallback
capWaveStreamCallback
capYieldCallback
WM_CAP_SET_CALLBACK_CAPCONTROL
WM_CAP_SET_CALLBACK_ERROR
WM_CAP_SET_CALLBACK_FRAME
WM_CAP_SET_CALLBACK_STATUS
WM_CAP_SET_CALLBACK_VIDEOSTREAM
WM_CAP_SET_CALLBACK_WAVESTREAM
WM_CAP_SET_CALLBACK_YIELD

Video Capture Functions

An application uses the AVICap functions to create a capture window and to retrieve information about the capture driver. A capture window uses standard window styles.

capCreateCaptureWindow

```
HWND VFWAPI capCreateCaptureWindow(LPCSTR lpszWindowName,
    DWORD dwStyle, int x, int y, int nWidth, int nHeight,
    HWND hWnd, int nID);
```

Creates a capture window.

- Returns a handle of the capture window if successful or NULL otherwise.

lpszWindowName
Null-terminated string containing the name used for the capture window.

dwStyle
Window styles used for the capture window. Window styles are described with the **CreateWindowEx** function.

x and *y*
The x- and y-coordinates of the upper left corner of the capture window.

nWidth and *nHeight*
Width and height of the capture window.

hWnd
Handle of the parent window.

nID
Window identifier.

capControlCallback

```
LRESULT CALLBACK capControlCallback(HWND hWnd, int nState);
```

Callback function used for precision control to begin and end streaming capture. The **capControlCallback** callback function is a placeholder for the application-supplied function name.

- When *nState* is set to CONTROLCALLBACK_PREROLL, this callback function must return TRUE to start capture or FALSE to abort it. When *nState* is set to CONTROLCALLBACK_CAPTURING, this callback function must return TRUE to continue capture or FALSE to end it.

hWnd
Handle of the capture window associated with the callback function.

nState
Current state of the capture operation. The CONTROLCALLBACK_PREROLL value is sent initially to enable prerolling of the video sources and to return control to the capture application at the exact moment recording is to begin. The CONTROLCALLBACK_CAPTURING value is sent once per captured frame to indicate that streaming capture is in progress and to enable the application to end capture.

The first message sent to the callback procedure sets the *nState* parameter to CONTROLCALLBACK_PREROLL after allocating all buffers and all other capture preparations are complete.

capGetDriverDescription

```
BOOL VFWAPI capGetDriverDescription(WORD wDriverIndex,
    LPSTR lpszName, INT cbName, LPSTR lpszVer, INT cbVer);
```

Retrieves the version description of the capture driver.

- Returns TRUE if successful or FALSE otherwise.

wDriverIndex
Index of the capture driver. The index can range from 0 through 9.

Plug-and-Play capture drivers are enumerated first, followed by capture drivers listed in the registry, which are then followed by capture drivers listed in SYSTEM.INI.

lpszName
Address of a buffer containing a null-terminated string corresponding to the capture driver name.

cbName
Length, in bytes, of the buffer pointed to by *lpszName*.

lpszVer
Address of a buffer containing a null-terminated string corresponding to the description of the capture driver.

cbVer
Length, in bytes, of the buffer pointed to by *lpszVer*.

If the information description is longer than its buffer, the description is truncated. The returned string is always null-terminated. If a buffer size is zero, its corresponding description is not copied.

capErrorCallback

```
LRESULT CALLBACK capErrorCallback(HWND hWnd, int nID, LPCSTR lpsz);
```

Error callback function used with video capture. The **capErrorCallback** error callback function is a placeholder for the application-supplied function name.

hWnd
Handle of the capture window associated with the callback function.

nID
Error identification number.

lpsz
Address of a textual description of the returned error.

A message identifier of zero indicates a new operation is starting and the callback function should clear the current error.

capStatusCallback

```
LRESULT CALLBACK capStatusCallback(HWND hWnd, int nID, LPCSTR lpsz);
```

Status callback function used with video capture. The **capStatusCallback** status callback function is a placeholder for the application-supplied function name.

hWnd
Handle of the capture window associated with the callback function.

nID
Message identification number.

lpsz
Address of a textual description of the returned status.

During capture operations, the first message sent to the callback function is always IDS_CAP_BEGIN and the last is always IDS_CAP_END. A message identifier of zero indicates a new operation is starting and the callback function should clear the current status.

capVideoStreamCallback

```
LRESULT CALLBACK capVideoStreamCallback(HWND hWnd, LPVIDEOHDR lpVHdr);
```

Callback function used with streaming capture to optionally process a frame of captured video. The **capVideoStreamCallback** callback function is a placeholder for the application-supplied function name.

hWnd
Handle of the capture window associated with the callback function.

lpVHdr
Address of a **VIDEOHDR** structure containing information about the captured frame.

The capture window calls a videostream callback function when a video buffer is marked done by the capture driver. When capturing to disk, this will preceed the disk write operation.

capWaveStreamCallback

```
LRESULT CALLBACK capWaveStreamCallback(HWND hWnd, LPWAVEHDR lpWHdr);
```

Callback function used with streaming capture to optionally process buffers of audio data. The **capWaveStreamCallback** callback function is a placeholder for the application-supplied function name.

hWnd
Handle of the capture window associated with the callback function.

lpWHdr
Address of a **WAVEHDR** structure containing information about the captured audio data.

The capture window calls a wavestream callback function when an audio buffer is marked done by the waveform-audio driver. When capturing to disk, this will preceed the disk write operation.

capYieldCallback

```
LRESULT CALLBACK capYieldCallback(HWND hWnd);
```

Yield callback function used with video capture. The **capYieldCallback** yield callback function is a placeholder for the application-supplied function name.

hWnd
Handle of the capture window associated with the callback function.

The capture window calls the yield callback function at least once for every captured video frame, but the exact rate depends on the frame rate and the overhead of the capture driver and disk.

Video Capture Messages and Macros

Applications communicate with capture windows through messages. AVICap macros provide a shorthand method of sending these messages, and they isolate your application from the **SendMessage** function. AVICap macros are identified with the prefix **cap**. Definitions of the AVICap macros are included with the message definitions.

WM_CAP_ABORT

```
WM_CAP_ABORT
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capCaptureAbort(hwnd);
```

Stops the capture operation. In the case of step capture, the image data collected up to the point of the WM_CAP_ABORT message will be retained in the capture file, but audio will not be captured.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

The capture operation must yield to use this message.

Use the WM_CAP_STOP message to halt step capture at the current position, and then capture audio.

WM_CAP_DLG_VIDEOCOMPRESSION

```
WM_CAP_DLG_VIDEOCOMPRESSION
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capDlgVideoCompression(hwnd);
```

Displays a dialog box in which the user can select a compressor to use during the capture process. The list of available compressors can vary with the video format selected in the capture driver's Video Format dialog box.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

Use this message with capture drivers that provide frames only in the BI_RGB format. This message is most useful in the step capture operation to combine capture and compression in a single operation. Compressing frames with a software compressor as part of a real-time capture operation is most likely too time-consuming to perform.

Compression does not affect the frames copied to the clipboard.

WM_CAP_DLG_VIDEODISPLAY

```
WM_CAP_DLG_VIDEODISPLAY
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capDlgVideoDisplay(hwnd);
```

Displays a dialog box in which the user can set or adjust the video output. This dialog box might contain controls that affect the hue, contrast, and brightness of the displayed image, as well as key color alignment.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

The controls in this dialog box do not affect digitized video data; they affect only the output or redisplay of the video signal.

The Video Display dialog box is unique for each capture driver. Some capture drivers might not support a Video Display dialog box. Applications can determine if the capture driver supports this message by checking the **fHasDlgVideoDisplay** member of **CAPDRIVERCAPS**.

WM_CAP_DLG_VIDEOFORMAT

```
WM_CAP_DLG_VIDEOFORMAT
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capDlgVideoFormat(hwnd)
```

Displays a dialog box in which the user can select the video format. The Video Format dialog box might be used to select image dimensions, bit depth, and hardware compression options.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

After this message returns, applications might need to update the **CAPSTATUS** structure because the user might have changed the image dimensions.

The Video Format dialog box is unique for each capture driver. Some capture drivers might not support a Video Format dialog box. Applications can determine if the capture driver supports this message by checking the **fHasDlgVideoFormat** member of **CAPDRIVERCAPS**.

WM_CAP_DLG_VIDEOSOURCE

```
WM_CAP_DLG_VIDEOSOURCE
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capDlgVideoSource(hwnd);
```

Displays a dialog box in which the user can control the video source. The Video Source dialog box might contain controls that select input sources; alter the hue, contrast, brightness of the image; and modify the video quality before digitizing the images into the frame buffer.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

The Video Source dialog box is unique for each capture driver. Some capture drivers might not support a Video Source dialog box. Applications can determine if the capture driver supports this message by checking the **fHasDlgVideoSource** member of the **CAPDRIVERCAPS** structure.

WM_CAP_DRIVER_CONNECT

```
WM_CAP_DRIVER_CONNECT
wParam = (WPARAM) (iIndex);
lParam = 0L;

// Corresponding macro
capDriverConnect(hwnd, iIndex);
```

Connects a capture window to a capture driver.

- Returns TRUE if successful or FALSE if the specified capture driver cannot be connected to the capture window.

iIndex
Index of the capture driver. The index can range from 0 through 9.

hwnd
Handle of a capture window.

Connecting a capture driver to a capture window automatically disconnects any previously connected capture driver.

WM_CAP_DRIVER_DISCONNECT

```
WM_CAP_DRIVER_DISCONNECT
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capDriverDisconnect(hwnd);
```

Disconnects a capture driver from a capture window.

- Returns TRUE if successful or FALSE if the capture window is not connected to a capture driver.

WM_CAP_DRIVER_GET_CAPS

```
WM_CAP_DRIVER_GET_CAPS
wParam = (WPARAM) (wSize);
lParam = (LPARAM) (LPVOID) (LPCAPDRIVERCAPS) (psCaps);

// Corresponding macro
capDriverGetCaps(hwnd, psCaps, wSize);
```

Returns the hardware capabilities of the capture driver currently connected to a capture window.

- Returns TRUE if successful or FALSE if the capture window is not connected to a capture driver.

hwnd
Handle of a capture window.

wSize
Size, in bytes, of the structure referenced by *s*.

psCaps
Address of the **CAPDRIVERCAPS** structure to contain the hardware capabilities.

The capabilities returned in **CAPDRIVERCAPS** are constant for a given capture driver. Applications need to retrieve this information once when the capture driver is first connected to a capture window.

WM_CAP_DRIVER_GET_NAME

```
WM_CAP_DRIVER_GET_NAME
wParam = (WPARAM) (wSize);
lParam = (LPARAM) (LPVOID) (LPSTR) (szName);

// Corresponding macro
capDriverGetName(hwnd, szName, wSize);
```

Returns the name of the capture driver connected to the capture window.

- Returns TRUE if successful or FALSE if the capture window is not connected to a capture driver.

hwnd
Handle of a capture window.

wSize
Size, in bytes, of the buffer referenced by *szName*.

szName
Address of an application-defined buffer used to return the device name as a null-terminated string.

The name is a text string retrieved from the driver's resource area. Applications should allocate approximately 80 bytes for this string. If the driver does not contain a name resource, the full path name of the driver listed in the registry or in the SYSTEM.INI file is returned.

WM_CAP_DRIVER_GET_VERSION

```
WM_CAP_DRIVER_GET_VERSION
wParam = (WPARAM) (wSize);
lParam = (LPARAM) (LPVOID) (LPSTR) (szVer);

// Corresponding macro
capDriverGetVersion(hwnd, szVer, wSize);
```

Returns the version information of the capture driver connected to a capture window.

- Returns TRUE if successful or FALSE if the capture window is not connected to a capture driver.

wSize
Size, in bytes, of the application-defined buffer referenced by *szVer*.

szVer
Address of an application-defined buffer used to return the version information as a null-terminated string.

The version information is a text string retrieved from the driver's resource area. Applications should allocate approximately 40 bytes for this string. If version information is not available, a NULL string is returned.

WM_CAP_EDIT_COPY

```
WM_CAP_EDIT_COPY
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capEditCopy(hwnd);
```

Copies the contents of the video frame buffer and associated palette to the clipboard.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

WM_CAP_FILE_ALLOCATE

```
WM_CAP_FILE_ALLOCATE
wParam = (WPARAM) 0;
lParam = (LPARAM) (DWORD) (dwSize);

// Corresponding macro
capFileAlloc(hwnd, dwSize);
```

Creates (preallocates) a capture file of a specified size.

- Returns TRUE if successful or FALSE otherwise. Also, if an error occurs and an error callback function is set using the WM_CAP_SET_CALLBACK_ERROR message, the error callback function is called.

hwnd
Handle of a capture window.

dwSize
Size, in bytes, to create the capture file.

You can improve streaming capture performance significantly by preallocating a capture file large enough to store an entire video clip and by defragmenting the capture file before capturing the clip.

WM_CAP_FILE_GET_CAPTURE_FILE

```
WM_CAP_FILE_GET_CAPTURE_FILE
wParam = (WPARAM) (wSize);
lParam = (LPARAM) (LPVOID) (LPSTR) (szName);

// Corresponding macro
capFileGetCaptureFile(hwnd, szName, wSize);
```

Returns the name of the current capture file.

- Returns TRUE if successful or FALSE otherwise.

wSize
Size, in bytes, of the application-defined buffer referenced by *szName*.

szName
Address of an application-defined buffer used to return the name of the capture file as a null-terminated string.

The default capture filename is C:\CAPTURE.AVI.

WM_CAP_FILE_SAVEAS

```
WM_CAP_FILE_SAVEAS
wParam = (WPARAM) 0;
lParam = (LPARAM) (LPVOID) (LPSTR) (szName);

// Corresponding macro
capFileSaveAs(hwnd, szName);
```

Copies the contents of the capture file to another file.

- Returns TRUE if successful or FALSE otherwise. Also, if an error occurs and an error callback function is set using the WM_CAP_SET_CALLBACK_ERROR message, the error callback function is called.

hwnd
Handle of a capture window.

szName
Address of the null-terminated string that contains the name of the destination file used to copy the file.

This message does not change the name or contents of the current capture file.

If the copy operation is unsuccessful due to a disk full error, the destination file is automatically deleted.

Typically, a capture file is preallocated for the largest capture segment anticipated and only a portion of it might be used to capture data. This message copies only the portion of the file containing the capture data.

WM_CAP_FILE_SAVEDIB

```
WM_CAP_FILE_SAVEDIB
wParam = (WPARAM) 0;
lParam = (LPARAM) (LPVOID) (LPSTR) (szName);

// Corresponding macro
capFileSaveDIB(hwnd, szName);
```

Copies the current frame to a DIB file.

- Returns TRUE if successful or FALSE otherwise. Also, if an error occurs and an error callback function is set using the WM_CAP_SET_CALLBACK_ERROR message, the error callback function is called.

hwnd
Handle of a capture window.

szName
Address of the null-terminated string that contains the name of the destination DIB file.

If the capture driver supplies frames in a compressed format, this call attempts to decompress the frame before writing the file.

WM_CAP_FILE_SET_CAPTURE_FILE

```
WM_CAP_FILE_SET_CAPTURE_FILE
wParam = (WPARAM) 0;
lParam = (LPARAM) (LPVOID) (LPSTR) (szName);

// Corresponding macro
capFileSetCaptureFile(hwnd, szName);
```

Names the file used for video capture.

- Returns TRUE if successful or FALSE if the filename is invalid, or if streaming or single-frame capture is in progress.

hwnd
Handle of a capture window.

szName
Address of the null-terminated string that contains the name of the capture file to use.

This message stores the filename in an internal structure. It does not create, allocate, or open the specified file. The default capture filename is C:\CAPTURE.AVI.

WM_CAP_FILE_SET_INFOCHUNK

```
WM_CAP_FILE_SET_INFOCHUNK
wParam = (WPARAM)0;
lParam = (LPARAM) (LPCAPINFOCHUNK) (lpInfoChunk);

// Corresponding macro
capFileSetInfoChunk(hwnd, lpInfoChunk);
```

Sets and clears information chunks. Information chunks can be inserted in an AVI file during capture to embed text strings or custom data.

- Returns TRUE if successful or FALSE otherwise. Also, if an error occurs and an error callback function is set using the WM_CAP_SET_CALLBACK_ERROR message, the error callback function is called.

hwnd
Handle of a capture window.

lpInfoChunk
Address of a **CAPINFOCHUNK** structure defining the information chunk to be created or deleted.

Multiple registered information chunks can be added to an AVI file. After an information chunk is set, it continues to be added to subsequent capture files until either the entry is cleared or all information chunk entries are cleared. To clear a single entry, specify the information chunk in the **fccInfoID** member and NULL in the **lpData** member of the **CAPINFOCHUNK** structure. To clear all entries, specify NULL in **fccInfoID**.

WM_CAP_GET_AUDIOFORMAT

```
WM_CAP_GET_AUDIOFORMAT
wParam = (WPARAM) (wSize);
lParam = (LPARAM) (LPVOID) (LPWAVEFORMATEX) (psAudioFormat);

// Corresponding macros
capGetAudioFormat(hwnd, psAudioFormat, wSize);
capGetAudioFormatSize(hwnd);
```

Obtains the audio format or the size of the audio format.

- Returns the size, in bytes, of the audio format.

hwnd
Handle of a capture window.

wSize
Size, in bytes, of the structure referenced by *s*.

psAudioFormat
Address of a **WAVEFORMATEX** structure, or NULL. If the value is NULL, the size, in bytes, required to hold the **WAVEFORMATEX** structure is returned.

Because compressed audio formats vary in size requirements applications must first retrieve the size, then allocate memory, and finally request the audio format data.

WM_CAP_GET_MCI_DEVICE

```
WM_CAP_GET_MCI_DEVICE
wParam = (WPARAM) (wSize);
lParam = (LPARAM) (LPVOID) (LPSTR) (szName);

// Corresponding macro
capGetMCIDeviceName(hwnd, szName, wSize);
```

Retrieves the name of an MCI device previously set with the WM_CAP_SET_MCI_DEVICE message.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

wSize
Length, in bytes, of the buffer referenced by *szName* .

szName
Address of a null-terminated string that contains the MCI device name.

WM_CAP_GET_SEQUENCE_SETUP

```
WM_CAP_GET_SEQUENCE_SETUP
wParam = (WPARAM) (wSize);
lParam = (LPARAM) (LPVOID) (LPCAPTUREPARMS) (s);

// Corresponding macro
capCaptureGetSetup(hwnd, s, wSize);
```

Retrieves the current settings of the streaming capture parameters.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

wSize
Size, in bytes, of the structure referenced by *s*.

s
Address of a **CAPTUREPARMS** structure.

For information about the parameters used to control streaming capture, see the **CAPTUREPARMS** structure.

WM_CAP_GET_STATUS

```
WM_CAP_GET_STATUS
wParam = (WPARAM) (wSize);
lParam = (LPARAM) (LPVOID) (LPCAPSTATUS) (s);

// Corresponding macro
capGetStatus(hwnd, s, wSize);
```

Retrieves the status of the capture window.

- Returns TRUE if successful or FALSE if the capture window is not connected to a capture driver.

hwnd
Handle of a capture window.

wSize
Size, in bytes, of the structure referenced by *s*.

s
Address of a **CAPSTATUS** structure.

The **CAPSTATUS** structure contains the current state of the capture window. Since this state is dynamic and changes in response to various messages, the application should initialize this structure after sending the WM_CAP_DLG_VIDEOFORMAT message (or using the **capDlgVideoFormat** macro) and whenever it needs to enable menu items or determine the actual state of the window.

WM_CAP_GET_USER_DATA

```
WM_CAP_GET_USER_DATA
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capGetUserData(hwnd);
```

Retrieves a LONG data value associated with a capture window.

- Returns a value previously saved by using the WM_CAP_SET_USER_DATA message.

hwnd
Handle of a capture window.

WM_CAP_GET_VIDEOFORMAT

```
WM_CAP_GET_VIDEOFORMAT
wParam = (WPARAM) (wSize);
lParam = (LPARAM) (LPVOID) (psVideoFormat);

// Corresponding macros
capGetVideoFormat(hwnd, psVideoFormat, wSize);
capGetVideoFormatSize(hwnd);
```

Retrieves a copy of the video format in use or the size required for the video format.

- Returns the size, in bytes, of the video format or zero if the capture window is not connected to a capture driver. For video formats that require a palette, the current palette is also returned.

hwnd
Handle of a capture window.

wSize
Size, in bytes, of the structure referenced by *s*.

psVideoFormat
Address of a **BITMAPINFO** structure. You can also specify NULL to retrieve the number of bytes needed by **BITMAPINFO**.

Because compressed video formats vary in size requirements applications must first retrieve the size, then allocate memory, and finally request the video format data.

WM_CAP_GRAB_FRAME

```
WM_CAP_GRAB_FRAME
wParam = (WPARAM)0;
lParam = (LPARAM)0L;

// Corresponding macro
capGrabFrame(hwnd);
```

Retrieves and displays a single frame from the capture driver. After capture, overlay and preview are disabled.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

For information about installing callback functions, see the WM_CAP_SET_CALLBACK_ERROR and WM_CAP_SET_CALLBACK_FRAME messages.

WM_CAP_GRAB_FRAME_NOSTOP

```
WM_CAP_GRAB_FRAME_NOSTOP
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capGrabFrameNoStop(hwnd);
```

Fills the frame buffer with a single uncompressed frame from the capture device and displays it. Unlike with the WM_CAP_GRAB_FRAME message, the state of overlay or preview is not altered by this message.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

For information about installing callback functions, see the WM_CAP_SET_CALLBACK_ERROR and WM_CAP_SET_CALLBACK_FRAME messages.

WM_CAP_PAL_AUTOCREATE

```
WM_CAP_PAL_AUTOCREATE
wParam = (WPARAM) (iFrames);
lParam = (LPARAM) (DWORD) (iColors);

// Corresponding macro
capPaletteAuto(hwnd, iFrames, iColors);
```

Requests that the capture driver sample video frames and automatically create a new palette.

- Returns TRUE if successful or FALSE otherwise. Also, if an error occurs and an error callback function is set using the WM_CAP_SET_CALLBACK_ERROR message, the error callback function is called.

hwnd
Handle of a capture window.

iFrames
Number of frames to sample.

iColors
Number of colors in the palette. The maximum value for this parameter is 256.

The sampled video sequence should include all the colors you want in the palette. To obtain the best palette, you might have to sample the whole sequence rather than a portion of it.

WM_CAP_PAL_MANUALCREATE

```
WM_CAP_PAL_MANUALCREATE
wParam = (WPARAM) (fGrab);
lParam = (LPARAM) (DWORD) (iColors);

// Corresponding macro
capPaletteManual(hwnd, fGrab, iColors);
```

Requests that the capture driver manually sample video frames and create a new palette.

- Returns TRUE if successful or FALSE otherwise. Also, if an error occurs and an error callback function is set using the WM_CAP_SET_CALLBACK_ERROR message, the error callback function is called.

hwnd
Handle of a capture window.

fGrab
Palette histogram flag. Set this parameter to TRUE for each frame included in creating the optimal palette. After the last frame has been collected, set this parameter to FALSE to calculate the optimal palette and send it to the capture driver.

iColors
Number of colors in the palette. The maximum value for this parameter is 256. This value is used only during collection of the first frame in a sequence.

WM_CAP_PAL_OPEN

```
WM_CAP_PAL_OPEN
wParam = (WPARAM) 0;
lParam = (LPARAM) (LPVOID) (LPSTR) (szName);

// Corresponding macro
capPaletteOpen(hwnd, szName);
```

Loads a new palette from a palette file and passes it to a capture driver. Palette files typically use the filename extension .PAL. A capture driver uses a palette when required by the specified digitized image format.

- Returns TRUE if successful or FALSE otherwise. Also, if an error occurs and an error callback function is set using the WM_CAP_SET_CALLBACK_ERROR message, the error callback function is called.

hwnd
Handle of a capture window.

szName
Address of a null-terminated string containing the palette filename.

WM_CAP_PAL_PASTE

```
WM_CAP_PAL_PASTE
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capPalettePaste(hwnd);
```

Copies the palette from the clipboard and passes it to a capture driver.

- Returns TRUE if successful or FALSE otherwise. Also, if an error occurs and an error callback function is set using the WM_CAP_SET_CALLBACK_ERROR message, the error callback function is called.

hwnd
Handle of a capture window.

A capture driver uses a palette when required by the specified digitized video format.

WM_CAP_PAL_SAVE

```
WM_CAP_PAL_SAVE
wParam = (WPARAM) 0;
lParam = (LPARAM) (LPVOID) (LPSTR) (szName);

// Corresponding macro
capPaletteSave(hwnd, szName);
```

Saves the current palette to a palette file. Palette files typically use the filename extension .PAL.

- Returns TRUE if successful or FALSE otherwise. Also, if an error occurs and an error callback function is set using the WM_CAP_SET_CALLBACK_ERROR message, the error callback function is called.

hwnd
Handle of a capture window.

szName
Address of a null-terminated string containing the palette filename.

WM_CAP_SEQUENCE

```
WM_CAP_SEQUENCE
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capCaptureSequence(hwnd);
```

Initiates streaming video and audio capture to a file.

- Returns TRUE if successful or FALSE otherwise. Also, if an error occurs and an error callback function is set using the WM_CAP_SET_CALLBACK_ERROR message, the error callback function is called.

hwnd
Handle of a capture window.

If you want to alter the parameters controlling streaming capture, use the WM_CAP_SET_SEQUENCE_SETUP message prior to starting the capture.

By default, the capture window does not allow other applications to continue running during capture. To override this, either set the **fYield** member of the **CAPTUREPARMS** structure to TRUE, or install a yield callback function.

During streaming capture, the capture window can optionally issue notifications to your application of specific types of conditions. To install callback procedures for these notifications, use the following messages:

WM_CAP_SET_CALLBACK_ERROR
WM_CAP_SET_CALLBACK_STATUS
WM_CAP_SET_CALLBACK_YIELD
WM_CAP_SET_CALLBACK_VIDEOSTREAM
WM_CAP_SET_CALLBACK_WAVESTREAM

WM_CAP_SEQUENCE_NOFILE

```
WM_CAP_SEQUENCE_NOFILE
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capCaptureSequenceNoFile(hwnd);
```

Initiates streaming video capture without writing data to a file.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

This message is useful in conjunction with video stream or waveform-audio stream callback functions that let your application use the video and audio data directly.

If you want to alter the parameters controlling streaming capture, use the WM_CAP_SET_SEQUENCE_SETUP message prior to starting the capture.

By default, the capture window does not allow other applications to continue running during capture. To override this, either set the **fYield** member of the **CAPTUREPARMS** structure to TRUE, or install a yield callback function.

During streaming capture, the capture window can optionally issue notifications to your application of specific types of conditions. To install callback procedures for these notifications, use the following messages:

WM_CAP_SET_CALLBACK_ERROR
WM_CAP_SET_CALLBACK_STATUS
WM_CAP_SET_CALLBACK_YIELD
WM_CAP_SET_CALLBACK_VIDEOSTREAM
WM_CAP_SET_CALLBACK_WAVESTREAM

WM_CAP_SET_AUDIOFORMAT

```
WM_CAP_SET_AUDIOFORMAT
wParam = (WPARAM) (wSize);
lParam = (LPARAM) (LPVOID) (LPWAVEFORMATEX) (psAudioFormat);

// Corresponding macro
capSetAudioFormat(hwnd, psAudioFormat, wSize);
```

Sets the audio format to use when performing streaming or step capture.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

wSize
Size, in bytes, of the structure referenced by *s*.

psAudioFormat
Address of a **WAVEFORMATEX** or **PCMWAVEFORMAT** structure that defines the audio format.

WM_CAP_SET_CALLBACK_CAPCONTROL

```
WM_CAP_SET_CALLBACK_CAPCONTROL
wParam = (WPARAM) 0;
lParam = (LPARAM) (LPVOID) (fpProc);

// Corresponding macro
capSetCallbackOnCapControl(hwnd, fpProc);
```

Sets a callback function in the application giving it precise recording control.

- Returns TRUE if successful or FALSE if a streaming capture or a single-frame capture session is in progress.

hwnd
Handle of a capture window.

fpProc
Address of the callback function. Specify NULL for this parameter to disable a previously installed callback function.

A single callback function is used to give the application precise control over the moments that streaming capture begins and completes. The capture window first calls the procedure with *nState* set to CONTROLCALLBACK_PREROLL after all buffers have been allocated and all other capture preparations have finished. This gives the application the ability to preroll video sources, returning from the callback function at the exact moment recording is to begin. A return value of TRUE from the callback function continues capture, and a return value of FALSE aborts capture. After capture begins, this callback function will be called frequently with *nState* set to CONTROLCALLBACK_CAPTURING to allow the application to end capture by returning FALSE.

WM_CAP_SET_CALLBACK_ERROR

```
WM_CAP_SET_CALLBACK_ERROR
wParam = (WPARAM) 0;
lParam = (LPARAM) (LPVOID) (fpProc);

// Corresponding macro
capSetCallbackOnError(hwnd, fpProc);
```

Sets an error callback function in the client application. AVICap calls this procedure when errors occur.

- Returns TRUE if successful or FALSE if streaming capture or a single-frame capture session is in progress.

hwnd
Handle of a capture window.

fpProc
Address of the error callback function. Specify NULL for this parameter to disable a previously installed error callback function.

Applications can optionally set an error callback function. If set, AVICap calls the error procedure in the following situations:

- The disk is full.
- A capture window cannot be connected with a capture driver.
- A waveform-audio device cannot be opened.
- The number of frames dropped during capture exceeds the specified percentage.
- The frames cannot be captured due to vertical synchronization interrupt problems.

WM_CAP_SET_CALLBACK_FRAME

```
WM_CAP_SET_CALLBACK_FRAME
wParam = (WPARAM) 0;
lParam = (LPARAM) (LPVOID) (fpProc);

// Corresponding macro
capSetCallbackOnFrame(hwnd, fpProc);
```

Sets a preview callback function in the application. AVICap calls this procedure when the capture window captures preview frames.

- Returns TRUE if successful or FALSE if streaming capture or a single-frame capture session is in progress.

hwnd
Handle of a capture window.

fpProc
Address of the preview callback function. Specify NULL for this parameter to disable a previously installed callback function.

The capture window calls the callback function before displaying preview frames. This allows an application to modify the frame if desired. This callback function is not used during streaming video capture.

WM_CAP_SET_CALLBACK_STATUS

```
WM_CAP_SET_CALLBACK_STATUS
wParam = (WPARAM) 0;
lParam = (LPARAM) (LPVOID) (fpProc);

// Corresponding macro
capSetCallbackOnStatus(hwnd, fpProc);
```

Sets a status callback function in the application. AVICap calls this procedure whenever the capture window status changes.

- Returns TRUE if successful or FALSE if streaming capture or a single-frame capture session is in progress.

hwnd
Handle of a capture window.

fpProc
Address of the status callback function. Specify NULL for this parameter to disable a previously installed status callback function.

Applications can optionally set a status callback function. If set, AVICap calls this procedure in the following situations:

- A capture session is completed.
- A capture driver successfully connected to a capture window.
- An optimal palette is created.
- The number of captured frames is reported.

WM_CAP_SET_CALLBACK_VIDEOSTREAM

```
WM_CAP_SET_CALLBACK_VIDEOSTREAM
wParam = (WPARAM) 0;
lParam = (LPARAM) (LPVOID) (fpProc);

// Corresponding macro
capSetCallbackOnVideoStream(hwnd, fpProc);
```

Sets a callback function in the application. AVICap calls this procedure during streaming capture when a video buffer is filled.

- Returns TRUE if successful or FALSE if streaming capture or a single-frame capture session is in progress.

hwnd
Handle of a capture window.

fpProc
Address of the video-stream callback function. Specify NULL for this parameter to disable a previously installed video-stream callback function.

The capture window calls the callback function before writing the captured frame to disk. This allows applications to modify the frame if desired.

If a video stream callback function is used for streaming capture, the procedure must be installed before starting the capture session and it must remain enabled for the duration of the session. It can be disabled after streaming capture ends.

WM_CAP_SET_CALLBACK_WAVESTREAM

```
WM_CAP_SET_CALLBACK_WAVESTREAM
wParam = (WPARAM) 0;
lParam = (LPARAM) (LPVOID) (fpProc);

// Corresponding macro
capSetCallbackOnWaveStream(hwnd, fpProc);
```

Sets a callback function in the application. AVICap calls this procedure during streaming capture when a new audio buffer becomes available.

- Returns TRUE if successful or FALSE if streaming capture or a single-frame capture session is in progress.

hwnd
Handle of a capture window.

fpProc
Address of the wave stream callback function. Specify NULL for this parameter to disable a previously installed wave stream callback function.

The capture window calls the procedure before writing the audio buffer to disk. This allows applications to modify the audio buffer if desired.

If a wave stream callback function is used, it must be installed before starting the capture session and it must remain enabled for the duration of the session. It can be disabled after streaming capture ends.

WM_CAP_SET_CALLBACK_YIELD

```
WM_CAP_SET_CALLBACK_YIELD
wParam = (WPARAM) 0;
lParam = (LPARAM) (LPVOID) (fpProc);

// Corresponding macro
capSetCallbackOnYield(hwnd, fpProc);
```

Sets a callback function in the application. AVICap calls this procedure when the capture window yields during streaming capture.

- Returns TRUE if successful or FALSE if streaming capture or a single-frame capture session is in progress.

hwnd
Handle of a capture window.

fpProc
Address of the yield callback function. Specify NULL for this parameter to disable a previously installed yield callback function.

Applications can optionally set a yield callback function. The yield callback function is called at least once for each video frame captured during streaming capture. If a yield callback function is installed, it will be called regardless of the state of the **fYield** member of the **CAPTUREPARMS** structure.

If the yield callback function is used, it must be installed before starting the capture session and it must remain enabled for the duration of the session. It can be disabled after streaming capture ends.

Applications typically perform some type of message processing in the callback function consisting of a **PeekMessage**, **TranslateMessage**, **DispatchMessage** loop, as in the message loop of a WinMain function. The yield callback function must also filter and remove messages that can cause reentrancy problems.

An application typically returns TRUE in the yield procedure to continue streaming capture. If a yield callback function returns FALSE, the capture window stops the capture process.

WM_CAP_SET_MCI_DEVICE

```
WM_CAP_SET_MCI_DEVICE
wParam = (WPARAM) 0;
lParam = (LPARAM) (LPVOID) (LPSTR) (szName);

// Corresponding macro
capSetMCIDeviceName(hwnd, szName);
```

Specifies the name of the MCI video device to be used to capture data.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

szName
Address of a null-terminated string containing the name of the device.

This message stores the MCI device name in an internal structure. It does not open or access the device. The default device name is NULL.

WM_CAP_SET_OVERLAY

```
WM_CAP_SET_OVERLAY
wParam = (WPARAM) (BOOL) (f);
lParam = 0L;

// Corresponding macro
capOverlay(hwnd, f);
```

Enables or disables overlay mode. In overlay mode, video is displayed using hardware overlay.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

f
Overlay flag. Specify TRUE for this parameter to enable overlay mode or FALSE to disable it.

Using an overlay does not require CPU resources.

The **fHasOverlay** member of the **CAPDRIVERCAPS** structure indicates whether the device is capable of overlay. The **fOverlayWindow** member of the **CAPSTATUS** structure indicates whether overlay mode is currently enabled.

Enabling overlay mode automatically disables preview mode.

WM_CAP_SET_PREVIEW

```
WM_CAP_SET_PREVIEW
wParam = (WPARAM) (BOOL) (f);
lParam = 0L;

// Corresponding macro
capPreview(hwnd, f);
```

Enables or disables preview mode. In preview mode, frames are transferred from the capture hardware to system memory and then displayed in the capture window using GDI functions.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

f
Preview flag. Specify TRUE for this parameter to enable preview mode or FALSE to disable it.

The preview mode uses substantial CPU resources. Applications can disable preview or lower the preview rate when another application has the focus. The **fLiveWindow** member of the **CAPSTATUS** structure indicates if preview mode is currently enabled.

Enabling preview mode automatically disables overlay mode.

WM_CAP_SET_PREVIEWRATE

```
WM_CAP_SET_PREVIEWRATE
wParam = (WPARAM) (wMS);
lParam = 0L;

// Corresponding macro
capPreviewRate(hwnd, wMS);
```

Sets the frame display rate in preview mode.

- Returns TRUE if successful or FALSE if the capture window is not connected to a capture driver.

hwnd
Handle of a capture window.

wMS
Rate, in milliseconds, at which new frames are captured and displayed.

The preview mode uses substantial CPU resources. Applications can disable preview or lower the preview rate when another application has the focus. During streaming video capture, the previewing task is lower priority than writing frames to disk, and preview frames are displayed only if no other buffers are available for writing.

WM_CAP_SET_SCALE

```
WM_CAP_SET_SCALE
wParam = (WPARAM) (BOOL)f;
lParam = 0L;

// Corresponding macro
capPreviewScale(hwnd, f);
```

Enables or disables scaling of the preview video images. If scaling is enabled, the captured video frame is stretched to the dimensions of the capture window.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

f
Preview scaling flag. Specify TRUE for this parameter to stretch preview frames to the size of the capture window or FALSE to display them at their natural size.

Scaling preview images controls the immediate presentation of captured frames within the capture window. It has no effect on the size of the frames saved to file.

Scaling has no effect when using overlay to display video in the frame buffer.

WM_CAP_SET_SCROLL

```
WM_CAP_SET_SCROLL
wParam = (WPARAM) 0;
lParam = (LPARAM) (LPPOINT) (lpP);

// Corresponding macro
capSetScrollPos(hwnd, lpP);
```

Defines the portion of the video frame to display in the capture window. This message sets the upper left corner of the client area of the capture window to the coordinates of a specified pixel within the video frame.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

lpP
Address to contain the desired scroll position.

The scroll position affects the image in both preview and overlay modes.

WM_CAP_SET_SEQUENCE_SETUP

```
WM_CAP_SET_SEQUENCE_SETUP
wParam = (WPARAM) (wSize);
lParam = (LPARAM) (LPVOID) (LPCAPTUREPARMS) (psCapParms);

// Corresponding macro
capCaptureSetSetup(hwnd, psCapParms, wSize);
```

Sets the configuration parameters used with streaming capture.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

psCapParms
Address of a **CAPTUREPARMS** structure.

wSize
Size, in bytes, of the structure referenced by *s*.

For information about the parameters used to control streaming capture, see the **CAPTUREPARMS** structure.

WM_CAP_SET_USER_DATA

```
WM_CAP_SET_USER_DATA
wParam = (WPARAM) 0;
lParam = (LPARAM)lUser;

// Corresponding macro
capSetUserData(hwnd, lUser);
```

Associates a **LONG** data value with a capture window.

- Returns TRUE if successful or FALSE if streaming capture is in progress.

hwnd
Handle of a capture window.

lUser
Data value to associate with a capture window.

Typically this message is used to point to a block of data associated with a capture window.

WM_CAP_SET_VIDEOFORMAT

```
WM_CAP_SET_VIDEOFORMAT
wParam = (WPARAM) (wSize);
lParam = (LPARAM) (LPVOID) (psVideoFormat);

// Corresponding macro
capSetVideoFormat(hwnd, psVideoFormat, wSize);
```

Sets the format of captured video data.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

psVideoFormat
Address of a **BITMAPINFO** structure.

wSize
Size, in bytes, of the structure referenced by *s*.

Because video formats are device-specific, applications should check the return value from this function to determine if the format is accepted by the driver.

WM_CAP_SINGLE_FRAME

```
WM_CAP_SINGLE_FRAME
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capCaptureSingleFrame(hwnd);
```

Appends a single frame to a capture file that was opened using the WM_CAP_SINGLE_FRAME_OPEN message.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

WM_CAP_SINGLE_FRAME_CLOSE

```
WM_CAP_SINGLE_FRAME_CLOSE
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capCaptureSingleFrameClose(hwnd);
```

Closes the capture file opened by the WM_CAP_SINGLE_FRAME_OPEN message.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

For information about installing callback functions, see the WM_CAP_SET_CALLBACK_ERROR and WM_CAP_SET_CALLBACK_FRAME messages.

WM_CAP_SINGLE_FRAME_OPEN

```
WM_CAP_SINGLE_FRAME_OPEN
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capCaptureSingleFrameOpen(hwnd);
```

Opens the capture file for single-frame capturing. Any previous information in the capture file is overwritten.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

For information about installing callback functions, see the WM_CAP_SET_CALLBACK_ERROR and WM_CAP_SET_CALLBACK_FRAME messages.

WM_CAP_STOP

```
WM_CAP_STOP
wParam = (WPARAM) 0;
lParam = 0L;

// Corresponding macro
capCaptureStop(hwnd);
```

Stops the capture operation.

In step frame capture, the image data that was collected before this message was sent is retained in the capture file. An equivalent duration of audio data is also retained in the capture file if audio capture was enabled.

- Returns TRUE if successful or FALSE otherwise.

hwnd
Handle of a capture window.

The capture operation must yield to use this message. Use the WM_CAP_ABORT message to abandon the current capture operation.

Video Capture Structures

The AVICap functions, messages, and macros use structures to describe, configure, and control the capture operation. Some of the structures described in this section, such as **BITMAPINFOHEADER**, are borrowed from other Win32 subsystems; others are specific to AVICap.

CAPDRIVERCAPS

```
typedef struct {
     UINT   wDeviceIndex;
     BOOL   fHasOverlay;
     BOOL   fHasDlgVideoSource;
     BOOL   fHasDlgVideoFormat;
     BOOL   fHasDlgVideoDisplay;
     BOOL   fCaptureInitialized;
     BOOL   fDriverSuppliesPalettes;
     HANDLE hVideoIn;
     HANDLE hVideoOut;
     HANDLE hVideoExtIn;
     HANDLE hVideoExtOut;
} CAPDRIVERCAPS;
```

Defines the capabilities of the capture driver.

An application should use the WM_CAP_DRIVER_GET_CAPS message or **capDriverGetCaps** macro to place a copy of the driver capabilities in a **CAPDRIVERCAPS** structure whenever the application connects a capture window to a capture driver.

wDeviceIndex
Index of the capture driver. An index value can range from 0 to 9.

fHasOverlay
Video-overlay flag. The value of this member is TRUE if the device supports video overlay.

fHasDlgVideoSource
Video source dialog flag. The value of this member is TRUE if the device supports a dialog box for selecting and controlling the video source.

fHasDlgVideoFormat
Video format dialog flag. The value of this member is TRUE if the device supports a dialog box for selecting the video format.

fHasDlgVideoDisplay
Video display dialog flag. The value of this member is TRUE if the device supports a dialog box for controlling the redisplay of video from the capture frame buffer.

fCaptureInitialized
Capture initialization flag. The value of this member is TRUE if a capture device has been successfully connected.

fDriverSuppliesPalettes
Driver palette flag. The value of this member is TRUE if the driver can create palettes.

hVideoIn
Not used in Win32 applications.

hVideoOut
Not used in Win32 applications.

hVideoExtIn
Not used in Win32 applications.

hVideoExtOut
Not used in Win32 applications.

CAPINFOCHUNK

```
typedef struct {
     FOURCC fccInfoID;
     LPVOID lpData;
     LONG   cbData;
} CAPINFOCHUNK;
```

Contains parameters that can be used to define an information chunk within an AVI capture file. The WM_CAP_FILE_SET_INFOCHUNK message or **capSetInfoChunk** macro is used to send a **CAPINFOCHUNK** structure to a capture window.

fccInfoID
Four-character code that identifies the representation of the chunk data. If this value is NULL and **lpData** is NULL, all accumulated information chunks are deleted.

lpData
Address of the data. If this value is NULL, all **fccInfoID** information chunks are deleted.

cbData
Size, in bytes, of the data pointed to by **lpData**. If **lpData** specifies a null-terminated string, use the string length incremented by one to save the NULL with the string.

CAPSTATUS

```
typedef struct {
     UINT     uiImageWidth;                 // image width, in pixels
     UINT     uiImageHeight;                // image height, in pixels
     BOOL     fLiveWindow;                  // see below
     BOOL     fOverlayWindow;               // see below
     BOOL     fScale;                       // see below
     POINT    ptScroll;                     // see below
     BOOL     fUsingDefaultPalette;         // see below
     BOOL     fAudioHardware;               // see below
     BOOL     fCapFileExists;               // see below
     DWORD    dwCurrentVideoFrame;          // see below
     DWORD    dwCurrentVideoFramesDropped;  // see below
     DWORD    dwCurrentWaveSamples;         // see below
     DWORD    dwCurrentTimeElapsedMS;       // see below
     HPALETTE hPalCurrent;                  // handle of current pallette
     BOOL     fCapturingNow;                // see below
     DWORD    dwReturn;                     // see below
     UINT     wNumVideoAllocated;           // see below
     UINT     wNumAudioAllocated;           // see below
} CAPSTATUS;
```

Defines the current state of the capture window.

fLiveWindow
Live window flag. The value of this member is TRUE if the window is displaying video using the preview method.

fOverlayWindow
Overlay window flag. The value of this member is TRUE if the window is displaying video using hardware overlay.

fScale
Input scaling flag. The value of this member is TRUE if the window is scaling the input video to the client area when displaying video using preview. This parameter has no effect when displaying video using overlay.

ptScroll
The x- and y-offset of the pixel displayed in the upper left corner of the client area of the window.

fUsingDefaultPalette
Default palette flag. The value of this member is TRUE if the capture driver is using its default palette.

fAudioHardware
Audio hardware flag. The value of this member is TRUE if the system has waveform-audio hardware installed.

fCapFileExists
Capture file flag. The value of this member is TRUE if a valid capture file has been generated.

dwCurrentVideoFrame
Number of frames processed during the current (or most recent) streaming capture. This count includes dropped frames.

dwCurrentVideoFramesDropped
Number of frames dropped during the current (or most recent) streaming capture. Dropped frames occur when the capture rate exceeds the rate at which frames can be saved to file. In this case, the capture driver has no buffers available for storing data. Dropping frames does not affect synchronization because the previous frame is displayed in place of the dropped frame.

dwCurrentWaveSamples
Number of waveform-audio samples processed during the current (or most recent) streaming capture.

dwCurrentTimeElapsedMS
Time, in milliseconds, since the start of the current (or most recent) streaming capture.

fCapturingNow
Capturing flag. The value of this member is TRUE when capturing is in progress.

dwReturn
Error return values. Use this member if your application does not support an error callback function.

wNumVideoAllocated
Number of video buffers allocated. This value might be less than the number specified in the **wNumVideoRequested** member of the **CAPTUREPARMS** structure.

wNumAudioAllocated
Number of audio buffers allocated. This value might be less than the number specified in the **wNumAudioRequested** member of the **CAPTUREPARMS** structure.

Because the state of a capture window changes in response to various messages, an application should update the information in this structure whenever it needs to enable menu items, determine the actual state of the capture window, or call the video format dialog box. If the application yields during streaming capture, this structure returns the progress of the capture in the **dwCurrentVideoFrame**, **dwCurrentVideoFramesDropped**, **dwCurrentWaveSamples**, and **dwCurrentTimeElapsedMS** members. Use the WM_CAP_GET_STATUS message or **capGetStatus** macro to update the contents of this structure.

CAPTUREPARMS

```
typedef struct {
     DWORD dwRequestMicroSecPerFrame;
     BOOL  fMakeUserHitOKToCapture;
     UINT  wPercentDropForError;
     BOOL  fYield;
     DWORD dwIndexSize;
     UINT  wChunkGranularity;
     BOOL  fUsingDOSMemory;
     UINT  wNumVideoRequested;
     BOOL  fCaptureAudio;
     UINT  wNumAudioRequested;
     UINT  vKeyAbort;
     BOOL  fAbortLeftMouse;
     BOOL  fAbortRightMouse;
     BOOL  fLimitEnabled;
     UINT  wTimeLimit;
     BOOL  fMCIControl;
     BOOL  fStepMCIDevice;
     DWORD dwMCIStartTime;
     DWORD dwMCIStopTime;
     BOOL  fStepCaptureAt2x;
     UINT  wStepCaptureAverageFrames;
     DWORD dwAudioBufferSize;
     BOOL  fDisableWriteCache;
     UINT  AVStreamMaster;
} CAPTUREPARMS;
```

Contains parameters that control the streaming video capture process. This structure is used to get and set parameters that affect the capture rate, the number of buffers to use while capturing, and how capture is terminated.

dwRequestMicroSecPerFrame
Requested frame rate, in microseconds. The default value is 66667, which corresponds to 15 frames per second.

fMakeUserHitOKToCapture
User-initiated capture flag. If this member is TRUE, AVICap displays a dialog box prompting the user to initiate capture. The default value is FALSE.

wPercentDropForError
Maximum allowable percentage of dropped frames during capture. Values range from 0 to 100. The default value is 10.

fYield
Yield flag. If this member is TRUE, the capture window spawns a separate background thread to perform step and streaming capture. The default value is FALSE.

Applications that set this flag must handle potential reentry issues because the controls in the application are not disabled while capture is in progress.

dwIndexSize
Maximum number of index entries in an AVI file. Values range from 1800 to 324,000. If set to 0, a default value of 34,952 (32K frames plus a proportional number of audio buffers) is used.

Each video frame or buffer of waveform-audio data uses one index entry. The value of this entry establishes a limit for the number of frames or audio buffers that can be captured.

wChunkGranularity

Logical block size, in bytes, of an AVI file. The value 0 indicates the current sector size is used as the granularity.

fUsingDOSMemory
Not used in Win32 applications.

wNumVideoRequested
Maximum number of video buffers to allocate. The memory area to place the buffers is specified with **fUsingDOSMemory**. The actual number of buffers allocated might be lower if memory is unavailable.

fCaptureAudio
Capture audio flag. If this member is TRUE, audio is captured during streaming capture. This is the default value if audio hardware is installed.

wNumAudioRequested
Maximum number of audio buffers to allocate. The maximum number of buffers is 10.

vKeyAbort
Virtual keycode used to terminate streaming capture. The default value is VK_ESCAPE.

You can combine keycodes that include CTRL and SHIFT keystrokes by using the logical OR operator with the keycodes for CTRL (0x8000) and SHIFT (0x4000).

fAbortLeftMouse
Abort flag for left mouse button. If this member is TRUE, streaming capture stops if the left mouse button is pressed. The default value is TRUE.

fAbortRightMouse
Abort flag for right mouse button. If this member is TRUE, streaming capture stops if the right mouse button is pressed. The default value is TRUE.

fLimitEnabled
Time limit enabled flag. If this member is TRUE, streaming capture stops after the number of seconds in **wTimeLimit** has elapsed. The default value is FALSE.

wTimeLimit
Time limit for capture, in seconds. This parameter is used only if **fLimitEnabled** is TRUE.

fMCIControl
MCI device capture flag. If this member is TRUE, AVICap controls an MCI-compatible video source during streaming capture. MCI-compatible video sources include VCRs and laserdiscs.

fStepMCIDevice
MCI device step capture flag. If this member is TRUE, step capture using an MCI device as a video source is enabled. If it is FALSE, real-time capture using an MCI device is enabled. (If **fMCIControl** is FALSE, this member is ignored.)

dwMCIStartTime
Starting position, in milliseconds, of the MCI device for the capture sequence. (If **fMCIControl** is FALSE, this member is ignored.)

dwMCIStopTime
Stopping position, in milliseconds, of the MCI device for the capture sequence. When this position in the content is reached, capture ends and the MCI device stops. (If **fMCIControl** is FALSE, this member is ignored.)

fStepCaptureAt2x
Double-resolution step capture flag. If this member is TRUE, the capture hardware captures at twice the specified resolution. (The resolution for the height and width is doubled.)

Enable this option if the hardware does not support hardware-based decimation and you are capturing in the RGB format.

wStepCaptureAverageFrames
Number of times a frame is sampled when creating a frame based on the average sample. A typical value for the number of averages is 5.

dwAudioBufferSize

Audio buffer size. If the default value of zero is used, the size of each buffer will be the maximum of 0.5 seconds of audio or 10K bytes.

fDisableWriteCache

Not used in Win32 applications.

AVStreamMaster

Indicates whether the audio stream controls the clock when writing an AVI file. If this member is set to AVSTREAMMASTER_AUDIO, the audio stream is considered the master stream and the video stream duration is forced to match the audio duration. If this member is set to AVSTREAMMASTER_NONE, the durations of audio and video streams can differ.

The WM_CAP_GET_SEQUENCE_SETUP message or **capCaptureGetSetup** macro is used to retrieve the current capture parameters. The WM_CAP_SET_SEQUENCE_SETUP message or **capCaptureSetSetup** macro is used to set the capture parameters.

The WM_CAP_GET_SEQUENCE_SETUP message or **capCaptureGetSetup** macro is used to retrieve the current capture parameters. The WM_CAP_SET_SEQUENCE_SETUP message or **capCaptureSetSetup** macro is used to set the capture parameters.

Video Compression Manager

The video compression manager (VCM) provides access to the interface used by installable compressors to handle real-time data. Applications can use installable compressors to perform the following tasks:

- Compress and decompress video data.
- Send a renderer compressed video data and have it draw it to the display.
- Compress, decompress, or draw data with application-defined renderers.
- Use renderers to handle text and custom data.

Typically, installable compressors operate with video-image data stored in audio-video interleaved (AVI) files. This chapter explains the programming techniques used to access installable compressors through VCM and covers the following topics:

- VCM and the Video for Windows architecture
- Compressing and decompressing image data from your application
- Using VCM renderers to draw data from your application
- VCM functions and structures

Before you read this chapter, you should be familiar with the Microsoft Win32 graphic services. In particular, bitmaps and bitmap-related structures, such as **BITMAPINFO** and **BITMAPINFOHEADER**, are used extensively by VCM. For additional information about the **BITMAPINFO** and **BITMAPINFOHEADER** structures, see Chapter 29, "Bitmaps."

Note The audio compression manager (ACM) provides system-level support for audio compression and decompression drivers. For a description of the audio compression services, see Chapter 12, "Audio Compression Manager."

VCM Architecture

VCM is an intermediary between an application and compression and decompression drivers. The compression and decompression drivers compress and decompress individual frames of data.

When an application makes a call to VCM, VCM translates the call into a message. The message is sent by using the **ICSendMessage** function to the appropriate compressor or decompressor, which compresses or decompresses the data. VCM receives the return value from the compression or decompression driver and then returns control to the application.

If a macro is defined for a message, the macro expands to an **ICSendMessage** function call supplying appropriate parameters for that message. If a macro is defined for a message, your application should use it rather than the message. In this chapter, these macros follow messages in parentheses.

System Entries for Compressors, Decompressors, and Renderers

The system uses entries in the registry to find VCM drivers. These entries are in the form of 2 four-character codes separated by a period. The first four-character code is defined by the system and can be one of the following:

Four-character code	Description
"VIDC"	Identifies compressors and decompressors.
"VIDS"	Identifies video-stream renderers.
"TXTS"	Identifies text-stream renderers.
"AUDS"	Identifies audio-stream handlers.

Custom renderers can define their own four-character codes.

The second four-character code is defined by the driver. Typically, the second four-character code corresponds to the type of data the driver can handle.

When opening a VCM driver, an application specifies the type of driver and the type of data handler it needs. Typically, this information comes from the stream header. The system tries to open the specified data handler, but if it fails, the system searches the registry for a driver that has the required handler.

When searching for the driver, the system tries to match the four-character codes specified for the driver type and data handler with those specified in the driver entry. For example, if an application specifies the compressor MSSQ, the system searches the registry for the driver entry VIDC.MSSQ. If it cannot find a match, it opens each driver corresponding to the driver type and locates one that can handle the type of data your application specifies. In the previous example, if the system could not find VIDC.MSSQ, it would open all drivers with the "VIDC" four-character code and locate one that can handle the data.

VCM Services

In general, an application uses VCM to perform the following tasks:

- Locate, open, or install a compressor or decompressor.
- Configure or obtain configuration information about the compressor or decompressor.
- Use a series of functions to compress, decompress, or draw the data.

The functions and macros of the DrawDib library perform these tasks implicitly and might provide the most convenient way to use VCM. For more information about the DrawDib library, see Chapter 10, "DrawDib Functions ."

The following sections describe tasks you can perform by using VCM.

Compressor and Decompressor Basics

To open and locate a compressor, you can use the **ICLocate** and **ICOpen** functions. You can use **ICLocate** to find a compressor of a specific type and to obtain a handle of it for use in other VCM functions. To open a compressor, you can use **ICOpen**. Your application uses the handle returned by this function to identify the opened compressor when it uses other VCM functions.

To open and locate a decompressor, applications can use the **ICDecompressOpen** and **ICDrawOpen** macros. These macros use **ICLocate** for operation.

When your application is finished using a compressor or decompressor, it must close it to free any resources used for compression or decompression. Your application can use the **ICClose** function to close the compressor or decompressor.

Also, your application can enumerate the compressors or decompressors on a system by using the **ICInfo** function.

Note The stream header in an AVI file contains information about the stream type and the specific handler for that stream. Within the stream header, the **fccType** and **fccHandler** members identify the stream type and the stream handler specified for the stream.

User-Selected Compressors

When compressing data, your application can use the **ICCompressorChoose** function to create a dialog box in which the user can select the compressor. You can specify flags for this function to allow the user to specify the key-frame frequency and the movie-data rate, or to display a preview window.

The compressor selected by the user is automatically opened and its handle is returned in the **hic** member of the **COMPVARS** structure specified in **ICCompressorChoose**.

If you use **ICCompressorChoose**, use the **ICCompressorFree** function to close the compressor and free any resources associated with the **COMPVARS** structure.

Compressor and Decompressor Installation and Removal

An application can use compressors and decompressors that are already installed on a system running the Microsoft Windows operating system. An application can also install compressors and decompressors for general or special uses. Most applications will not need to install or remove compressors or decompressors because they are usually installed by a setup program. An application might, however, install a compressor directly or install a function as a compressor.

An application can install a compressor or decompressor (or a function used as a compressor or decompressor) by using the **ICInstall** function. This function creates an entry in the registry identifying the compressor or decompressor. Your application or another application can search the registry to determine if the system contains a compressor or decompressor suitable for its data. Use **ICInstall** to install all compression and decompression drivers.

An application can locate and open an installed compressor or decompressor by using the **ICLocate** and **ICOpen** functions. When an application finishes using a compressor or decompressor, it closes it by using the **ICClose** function.

An application can remove the registry entry for an installed compressor or decompressor by using the **ICRemove** function. This function removes the registry entry of a compressor or decompressor that is not currently loaded in memory.

An application can restrict the use of a compressor or decompressor by installing, opening, closing, and removing the compressor.

Alternatively, an application can use a function internally as a compressor or decompressor without installing the function in the registry by using the **ICOpenFunction** function. This function requires the calling application to have the address of the function. When the application finishes using the function, it must close it by using **ICClose**. Because the function was not installed, the application does not need to remove the function from the registry.

The internal structure of a function used as a compressor or decompressor should be the same as the **DriverProc** entry-point function used by installable drivers. For more information about the **DriverProc** entry-point function, see Chapter 0, "Installable Drivers."

Note An application installing a function as a compressor or decompressor must remove the function before the application is closed so other applications do not try to use the function. When removing a function, the application must identify it with the four-character code used to install it.

Compressor and Decompressor Configuration

Your application can configure the compressor or decompressor automatically, or it can allow the user to configure them. If it is practical, you should allow the user to configure the compressor or decompressor because it frees you from considering all the compressor's or decompressor's options.

The user can configure the compressor or decompressor by using a configuration dialog box. You can send the ICM_CONFIGURE message to VCM or use the **ICQueryConfigure** macro to determine if a compressor or decompressor can display a configuration dialog box. If so, you can send the ICM_CONFIGURE message (or the **ICConfigure** macro) to display it.

Your application can send the ICM_GETSTATE and ICM_SETSTATE messages (or the **ICGetStateSize**, **ICGetState**, and **ICSetState** macros) to get and set the status for a compressor or decompressor. If your application creates or modifies the status, it must obtain the layout of the compressor or decompressor data before restoring its status. Alternatively, if your application obtains the status from a compressor or decompressor and uses it to restore the status in a subsequent session, it must ensure that it restores only status information obtained from the compressor or decompressor it is currently using.

Getting Information About Compressors and Decompressors

To get general information about a compressor or decompressor, your application can use the **ICGetInfo** function. This function fills an **ICINFO** structure with information about the compressor or decompressor. Your application must allocate the memory for the **ICINFO** structure and pass a pointer to it in **ICGetInfo**. Unless your application searches for a particular compressor or decompressor, the flags in the **ICINFO** structure provide the most useful information about the capabilities of a compressor or decompressor.

To get the default key-frame rate and default quality value of a compressor or decompressor, your application can send the ICM_GETDEFAULTKEYFRAMERATE and ICM_GETDEFAULTQUALITY messages (or the **ICGetDefaultKeyFrameRate** and **ICGetDefaultQuality** macros).

To determine the best display format of a compressor or decompressor, your application can use the **ICGetDisplayFormat** function.

You can determine if a compressor or decompressor can display an About dialog box by sending the ICM_ABOUT message (or the **ICQueryAbout** macro). You can also display the About dialog box of a compressor or decompressor by sending the ICM_ABOUT message and changing the value of the *wParam* parameter (or by using the **ICAbout** macro).

Single-Image Compression

You can use the **ICImageCompress** function to compress a single image. This function returns a handle of the compressed device-independent bitmap (DIB). The compressed DIB is packed using the CF_DIB format.

Sequence Compression

Your application can use the **ICSeqCompressFrame**, **ICSeqCompressFrameStart**, and **ICSeqCompressFrameEnd** functions to compress a sequence of frames. These functions use the data stored in the **COMPVARS** structure. Applications use the **ICCompressorChoose** function to allow the user to select a compressor, open it, and set the compression parameters in the **COMPVARS** structure; however, applications can set the parameters in the structure manually.

Before an application can begin compressing a sequence of frames, it must use **ICSeqCompressFrameStart** to allocate the necessary resources. After the resources are allocated, the application can use **ICSeqCompressFrame** to compress each frame individually. The frame rate and key-frame frequency used in compressing the sequence are specified in members of the **COMPVARS** structure. The return value for **ICSeqCompressFrame** points to the compressed data.

When an application finishes compressing a sequence, it can use **ICSeqCompressFrameEnd** to free system resources allocated for **ICSeqCompressFrameStart**, and **ICCompressorFree** to free the resources allocated for the **COMPVARS** structure.

Image-Data Compression

Your application can use a series of **ICCompress** functions and macros to compress data. The functions and macros can help you perform the following tasks:

- Determine the compression format to use for a specified input format.
- Prepare the compressor.
- Compress the data.
- End compression.

Your application can increase control over the compression process by using the **ICCompress** functions and macros. This group of functions and macros deals with individual frames, rather than the sequence as a whole. For example, your application can identify the frames to compress as key frames by using the **ICCompress** function.

A compressor receives data in one format, compresses the data, and returns a compressed version of the data using a specified format. The typical input format specifies DIBs using the **BITMAPINFO** structure. The typical output format specifies compressed DIBs, also using the **BITMAPINFO** structure.

Note To minimize image and audio degradation during playback, avoid compressing an AVI file more than once. Combine uncompressed pieces of video in your editing system and then compress the final product.

Compressor and Compression Format Selection

If you want to compress data and your application requires a specific output format, you can send the ICM_COMPRESS_QUERY message (or the **ICCompressQuery** macro) to query the compressor to determine if it supports the input and output formats.

If the output format is not important to your application, you need only find a compressor that can handle the input format. To determine if a compressor can handle the input format, you can send ICM_COMPRESS_QUERY, specifying NULL for the *lParam* parameter. This message does not return the output format to your application. Your application can determine the buffer size needed for the data specifying the compression format by sending the ICM_COMPRESS_GET_FORMAT message (or the **ICCompressGetFormatSize** macro). You can also retrieve the format data by sending ICM_COMPRESS_GET_FORMAT (or the **ICCompressGetFormat** macro).

If you want to determine the largest buffer that the compressor could require for compression, send the ICM_COMPRESS_GET_SIZE message (or the **ICCompressGetSize** macro). You can use the number of bytes returned by the **ICSendMessage** function to allocate a buffer for subsequent image compressions.

Compressor Initialization

After your application selects a compressor that can handle the input and output formats it needs, you can initialize the compressor by using the ICM_COMPRESS_BEGIN message (or the **ICCompressBegin** macro). This message requires the compressor handle and the input and output formats.

Data Compression

You can use the **ICCompress** function to compress a frame. Your application must use this function repeatedly until all the frames in a sequence are compressed. Your application must also track and increment the number of each frame compressed with **ICCompress**. The compressor uses this value to check if frames are sent out of order during fast temporal compression (storing differences between successive frames). If your application recompresses a frame, it should use the same frame number as when the frame was first compressed. If your application compresses a still-frame image, it can specify

a frame number of zero.

Your application can use the ICCOMPRESS_KEYFRAME flag to make the frame compressed by **ICCompress** a key frame.

When VCM returns control to your application after compressing a frame, VCM stores the compressed data in the structures referenced by the *lpbiOutput* and *lpData* parameters. If your application needs to move the compressed data, it can find its size in the **biSizeImage** member of the **BITMAPINFO** structure specified in *lpbiOutput*.

Note Your application must allocate the structures and buffers that store the uncompressed and compressed data. If the compressor supports temporal compression, your application must also allocate a structure and buffer to hold the format and data for the previous frame of information.

Ending Compression

After your application has compressed the data, it can use the **ICCompressEnd** macro to notify the compressor that it has finished. If you want to restart compression after using this function, your application must reinitialize the compressor by sending the ICM_COMPRESS_BEGIN message (or the **ICCompressBegin** macro).

Single-Image Decompression

You can use the **ICImageDecompress** function to decompress a single image. This function returns a handle of the decompressed DIB. The decompressed DIB is stored in the CF_DIB format.

Image-Data Decompression

Your application uses a series of **ICDecompressEx** functions to control the decompressor. The functions can help you perform the following tasks:

- Select a decompressor.
- Prepare the decompressor.
- Decompress the data.
- End decompression.

Your application handles decompression similarly to compression except that the input format is a compressed format and the output format is a displayable format. The input format for decompression is usually obtained from the stream header. After determining the input format, your application can use the **ICLocate** or **ICOpen** functions to find a decompressor that can handle it.

The **ICDecompressEx** functions and macros are a superset of the **ICDecompress** function group and provide more capabilities. The functionality of **ICDecompressEx**, **ICDecompressExBegin**, **ICDecompressExEnd**, and **ICDecompressExQuery** replaces that of the **ICDecompress**, **ICDecompressBegin**, **ICDecompressEnd**, and **ICDecompressQuery**. Use the **ICDecompressEx** functions and macros in place of the **ICDecompress** equivalents.

Decompressor and Decompression Format Selection

If you want to decompress data and your application requires a specific output format, you can use the **ICDecompressExQuery** function to query the decompressor to determine if it supports the input and output formats.

If the output format is not important in your application, you need only find a decompressor that can handle the input format. To determine if a decompressor can handle the input format, use **ICDecompressExQuery** and specify NULL for the *lpbiDst* parameter. Your application can determine the buffer size needed for the data specifying the decompression format by sending the ICM_DECOMPRESS_GET_FORMAT message (or the **ICDecompressGetFormatSize** macro). You can also send ICM_DECOMPRESS_GET_FORMAT (or the **ICDecompressGetFormat** macro) to retrieve the format data. The decompressor returns its suggested format in a **BITMAPINFO** structure. This format typically preserves the most information during decompression. Your application should ensure that the decompressor returns successfully before it decompresses the information.

Because your application allocates the memory required for decompression, it needs to determine the maximum memory the decompressor can require for the output format. The ICM_DECOMPRESS_GET_FORMAT message obtains the number of bytes the decompressor uses for the default format.

If your application defines its own format by using **ICDecompressExQuery**, it must also obtain a palette for the bitmap; **ICDecompressExQuery** does not provide palette definitions. (Most applications use standard formats and do not need to obtain a palette.) Your application can obtain the palette by sending the ICM_DECOMPRESS_GET_PALETTE message (or the **ICDecompressGetPalette** macro).

Decompressor Initialization

After your application selects a decompressor that can handle the input and output formats it needs, you can initialize the decompressor by using the **ICDecompressExBegin** function. This function requires the decompressor handle and the input and output formats.

Data Decompression

You can use the **ICDecompressEx** function to decompress a frame. Your application must use this function repeatedly until all the frames in a sequence are decompressed.

If your video stream lags behind other components (such as audio) during playback, your application can specify the ICDECOMPRESS_HURRYUP flag to speed decompression. To do this, a decompressor might extract only the information it needs to decompress the next frame and not fully decompress the current frame. Therefore, your application should not try to draw the decompressed data when it uses this flag.

After your application has decompressed the data, it can send the ICM_DECOMPRESSEX_END message (or the **ICDecompressExEnd** macro) to notify the decompressor that it has finished. If you want to restart decompression after using this function, your application must reinitialize the decompressor by using **ICDecompressExBegin**.

Monitoring the Progress of Compressors and Decompressors

Your application can monitor the progress of a lengthy operation performed by a compressor or decompressor by sending it the address of a callback function. You can use the **ICSetStatusProc** function to send the address to the compressor or decompressor. When the compressor or decompressor receives this address, it sends status messages to the function. These messages indicate whether the operation is starting, stopping, yielding, or proceeding.

Hardware Drawing Capabilities

Some renderers can draw directly to video hardware as they decompress video frames. These renderers return the VIDCF_DRAW flag in response to the **ICGetInfo** function. When using this type of renderer, your application does not have to handle the decompressed data. It lets the renderer retain the decompressed data for drawing.

If your application uses a renderer with drawing capabilities, it must handle the following tasks:

- Select a renderer.
- Specify image formats.
- Initialize the renderer.
- Draw the data.
- Control drawing parameters.

Renderer Selection

The **ICDrawOpen** macro opens a renderer that can draw images with the specified format. It returns a handle of a renderer if it is successful or zero otherwise. This macro uses the **ICLocate** function to open the renderer.

Specifying Image Formats

Because your application does not need to draw the decompressed data, it does not require a specific output format. It must, however, ensure that the renderer can draw using the input format by using the ICM_DRAW_QUERY message (or the **ICDrawQuery** macro). This message cannot determine if a renderer can draw a bitmap. If your application must determine if the renderer can draw the bitmap, use this message with the **ICDrawBegin** function.

Your application can have a renderer suggest an input format by using the **ICDrawSuggestFormat** function. This function is used when a renderer separates the drawing capabilities from the decompressing capabilities. Most applications using the drawing functions will not need to determine the output format.

Renderer Initialization

The **ICDrawBegin** function initializes a renderer and tells it the drawing destination. This function can also perform the following tasks:

- Determine whether the renderer supports a specific input format.
- Specify whether the drawing operation occupies a window or the entire screen.
- Specify the part of the image to display using the source rectangle.
- Define the playback rate of the image sequence.

Some renderers buffer the compressed data to operate more efficiently. Your application can send the ICM_GETBUFFERSWANTED message (or the **ICGetBuffersWanted** macro) to determine the number of buffers the renderer requests. Your application should preload these buffers and send them to the renderer before drawing.

Drawing the Data

You can use the **ICDraw** function to decompress the data for drawing. The renderer, however, does not start drawing data until your application sends the ICM_DRAW_START message (or the **ICDrawStart** macro). When your application calls this function, the renderer begins to draw the frames at the rate specified by the *dwRate* parameter divided by the *dwScale* parameter; these parameters were supplied when the application initialized the renderer by using the **ICDrawBegin** function. Drawing continues until your application stops the renderer drawing clock by sending the ICM_DRAW_STOP message (or

the **ICDrawStop** macro).

Note If a renderer buffers the data before drawing, your application should not use **ICDrawStart** until it has sent the number of frames the renderer returned for the **ICGetBuffersWanted** macro.

The *lTime* parameter of **ICDraw** specifies the time to draw a frame. The renderer divides this integer by the time scale specified with **ICDrawBegin** to obtain the actual time. Times for **ICDraw** functions are relative to **ICDrawStart**. **ICDrawStart** sets the clock to zero. For example, if your application specifies 1000 for the time scale and 75 for *lTime*, the renderer draws the frame 75 milliseconds into the sequence.

Controlling Drawing Parameters

You can monitor the clock of a renderer by sending the ICM_DRAW_GETTIME message (or the **ICDrawGetTime** macro), and you can set the clock of a renderer that can draw data by sending the ICM_DRAW_SETTIME message (or the **ICDrawSetTime** macro).

To change the current position while a renderer is drawing, your application can send the ICM_DRAW_WINDOW message (or the **ICDrawWindow** macro) for repositioning the window. Applications typically use this message whenever the window changes.

If the playback window gets a palette-realize message, your application must send the ICM_DRAW_REALIZE message (or the **ICDrawRealize** macro) to have the renderer realize the palette again for playback. Applications can change the palette by sending the ICM_DRAW_CHANGEPALETTE message (or the **ICDrawChangePalette** macro) and obtain the current palette by sending the ICM_DRAW_GET_PALETTE message.

Some renderers must be specifically instructed to display frames passed to them. Sending the ICM_DRAW_RENDERBUFFER message (or the **ICDrawRenderBuffer** macro) causes these renderers to draw the frame.

Using the Video Compression Manager

This section contains examples demonstrating how to perform the following tasks:

- Locate and open compressors and decompressors.
- Install compressors and decompressors.
- Configure compressors and decompressors.
- Get information about compressors and decompressors.
- Determine a compressor's output format.
- Compress data.
- Determine a decompressor's output format.
- Decompress data.
- Determine if a driver can handle the input format.
- Prepare to draw data.
- Draw data.
- Monitor compressor and decompressor progress.

Locating and Opening Compressors and Decompressors

The following example attempts to find a compressor that can compress an 8-bits-per-pixel bitmap:

```
BITMAPINFOHEADER bih;
HIC              hIC

// Initialize the bitmap structure.
bih.biSize = sizeof(BITMAPINFOHEADER);
bih.biWidth = bih.biHeight = 0;
bih.biPlanes = 1;
bih.biCompression = BI_RGB;      // standard RGB bitmap
bih.biBitcount = 8;              // 8 bits-per-pixel format
bih.biSizeImage = 0;
bih.biXPelsPerMeter = bih.biYPelsPerMeter = 0;
bih.biClrUsed = bih.biClrImportant = 256;

hIC = ICLocate (ICTYPE_VIDEO, 0L, (LPBITMAPINFOHEADER) &bih,
    NULL, ICMODE_COMPRESS);
```

The following example enumerates the decompressors in the system to find one that can handle the format of its images. This example uses ICTYPE_VIDEO (which is equivalent to the "VIDC" four-character code) and the **ICDecompressQuery** macro to determine if a compressor or decompressor supports the image format.

```
for (i=0; ICInfo(fccType, i, &icinfo); i++)
{
    hic = ICOpen(icinfo.fccType, icinfo.fccHandler, ICMODE_QUERY);
    if (hic)
    {
        // Skip this compressor if it can't handle the specified format.
        if (fccType == ICTYPE_VIDEO && pvIn != NULL &&
            ICDecompressQuery(hic, pvIn, NULL) != ICERR_OK)
        {
            ICClose(hic);
            continue;
        }

        // Find out the compressor name.
        ICGetInfo(hic, &icinfo, sizeof(icinfo));

        // Add it to the combo box.
        n = ComboBox_AddString(hwndC,icinfo.szDescription);
        ComboBox_SetItemData(hwndC, n, hic);
    }
}
```

The following example attempts to locate a specific compressor to compress the 8-bit RGB format to an 8-bit RLE format.

```
BITMAPINFOHEADER    bihIn, bihOut;
HIC                 hIC

// Initialize the bitmap structure.
biSize = bihOut.biSize = sizeof(BITMAPINFOHEADER);
bihIn.biWidth = bihIn.biHeight = bihOut.biWidth = bihOut.biHeight = 0;
```

```
bihIn.biPlanes = bihOut.biPlanes= 1;
bihIn.biCompression = BI_RGB;        // standard RGB bitmap for input
bihOut.biCompression = BI_RLE8;      // 8-bit RLE for output format
bihIn.biBitcount = bihOut.biBitCount = 8;  // 8 bits-per-pixel format
bihIn.biSizeImage = bihOut.biSizeImage = 0;
bihIn.biXPelsPerMeter = bih.biYPelsPerMeter =
    bihOut.biXPelsPerMeter = bihOut.biYPelsPerMeter = 0;
bihIn.biClrUsed = bih.biClrImportant =
    bihOut.biClrUsed = bihOut.biClrImportant = 256;
hIC = ICLocate (ICTYPE_VIDEO, 0L,
    (LPBITMAPINFOHEADER)&bihIn,
    (LPBITMAPINFOHEADER)&bihOut, ICMODE_COMPRESS);
```

Installing Compressors and Decompressors

The following example shows how an application can install a function as a compressor or decompressor.

```
// This function looks like a DriverProc entry point.
LRESULT MyCodecFunction(DWORD dwID, HDRVR hDriver,
    UINT uiMessage, LPARAM lParam1, LPARAM lParam2);

// This function installs the MyCodecFunction as a compressor.
result = ICInstall ( ICTYPE_VIDEO, mmioFOURCC('s','a','m','p'),
     (LPARAM)(FARPROC)&MyCodecFunction, NULL, ICINSTALL_FUNCTION);
```

Configuring Compressors and Decompressors

The following example demonstrates how to test if a compressor supports the configuration dialog box and to display it if it does.

```
// If the compressor handles a configuration dialog box, display it
// using our application window as the parent window.
if (ICQueryConfigure(hIC)) ICConfigure(hIC, hwndApp);
```

The following example shows how to obtain the state data.

```
dwStateSize = ICGetStateSize(hIC);    // gets size of buffer required
h = GlobalAlloc(GHND, dwStateSize);   // allocates buffer
ICGetState(hIC, (LPVOID)lpData, dwStateSize);  // gets the state data

// Store the state data as required.
```

The following example shows how to restore state data. State data restored by applications should not contain any changes to the state data obtained from a driver.

```
ICSetState(hIC, (LPVOID)lpData, dwStateSize); // sets the new state data
```

Obtaining Information About Compressors and Decompressors

The following example shows how to obtain information about a compressor or decompressor.

```
ICINFO ICInfo;
ICGetInfo(hIC, &ICInfo, sizeof(ICInfo));
```

The following example uses the **ICGetDefaultKeyFrameRate** and **ICGetDefaultQuality** macros to obtain the default values:

```
DWORD dwKeyFrameRate, dwQuality;
dwKeyFrameRate = ICGetDefaultKeyFrameRate(hIC);
dwQuality = ICGetDefaultQuality(hIC);
```

The following example uses the **ICQueryAbout** and **ICAbout** macros to display an About dialog box for the compressor or decompressor, if the dialog box exists.

```
// If the compressor has an About dialog box, display it.
if ( ICQueryAbout(hIC)) ICAbout(hIC, hwndApp);
```

Determining a Compressor's Output Format

The following example determines the buffer size needed for the data specifying the compression format, allocates a buffer of the appropriate size, and retrieves the compression format information.

```
LPBITMAPINFOHEADER     lpbiIn, lpbiOut;
.
. // *lpbiIn must be initialized to the input format.
.
dwFormatSize = ICCompressGetFormatSize(hIC, lpbiIn);
h = GlobalAlloc(GHND, dwFormatSize);
lpbiOut = (LPBITMAPINFOHEADER)GlobalLock(h);
ICCompressGetFormat(hIC, lpbiIn, lpbiOut);
```

The following example uses the **ICCompressQuery** macro to determine if a compressor can handle the input and output formats.

```
LPBITMAPINFOHEADER     lpbiIn, lpbiOut;

// Both *lpbiIn and *lpbiOut must be initialized to the respective
// formats.
if (ICCompressQuery(hIC, lpbiIn, lpbiOut) == ICERR_OK){
.
. // Format is supported; use the compressor.
.
}
```

The following example determines the buffer size and allocates a buffer of that size.

```
// Find the worst-case buffer size.
dwCompressBufferSize = ICCompressGetSize(hIC, lpbiIn, lpbiOut);

// Allocate a buffer and get lpOutput to point to it.
h = GlobalAlloc(GHND, dwCompressBufferSize);
lpOutput = (LPVOID)GlobalLock(h);
```

Compressing Data

The following example compresses image data for use in an AVI file. It assumes the compressor does not support the VIDCF_CRUNCH or VIDCF_TEMPORAL flags, but it does support VIDCF_QUALITY.

```
DWORD dwCkID;
DWORD dwCompFlags;
DWORD dwQuality;
LONG  lNumFrames, lFrameNum;
// Assume dwNumFrames is initialized to the total number of frames.
// Assume dwQuality holds the proper quality value (0-10000).
// Assume lpbiOut, lpOut, lpbiIn, and lpIn are initialized properly.

// If OK to start, compress each frame.
if (ICCompressBegin(hIC, lpbiIn, lpbiOut) == ICERR_OK){
    for ( lFrameNum = 0; lFrameNum < lNumFrames; lFrameNum++){
        if (ICCompress(hIC, 0, lpbiOut, lpOut, lpbiIn, lpIn,
            &dwCkID, &dwCompFlags, lFrameNum,
            0, dwQuality, NULL, NULL) == ICERR_OK){
            // Write compressed data to the AVI file.
            .
            .  // Set lpIn to the next frame in the sequence.
            .
        }
        else {
            // Handle compressor error.
        }
    }
    ICCompressEnd(hIC);    // terminate compression
}
else {
    // Handle the error identifying the unsupported format.
}
```

Determining a Decompressor's Output Format

The following example determines the buffer size needed for the data specifying the decompression format, allocates a buffer of the appropriate size, and retrieves the decompression format information.

```
LPBITMAPINFOHEADER lpbiIn, lpbiOut;

// Assume *lpbiIn points to the input (compressed) format.
dwFormatSize = ICDecompressGetFormatSize(hIC, lpbiIn);
h = GlobalAlloc(GHND, dwFormatSize);
lpbiOut = (LPBITMAPINFOHEADER)GlobalLock(h);
ICDecompressGetFormat(hIC, lpbiIn, lpbiOut);
```

The following example shows how an application can use the **ICDecompressQuery** macro to determine if a decompressor can handle the input and output formats.

```
LPBITMAPINFOHEADER lpbiIn, lpbiOut;
// Assume *lpbiIn & *lpbiOut are initialized to the respective
// formats.
if (ICDecompressQuery(hIC, lpbiIn, lpbiOut) == ICERR_OK){
    .
    . // Format is supported - use the decompressor.
    .
}
```

The following fragment shows how to get the palette information:

```
ICDecompressGetPalette(hIC, lpbiIn, lpbiOut);

// Move up to the palette.
lpPalette = (LPBYTE)lpbiOut + lpbi->biSize;
```

Decompressing Data

The following example shows how an application can initialize a decompressor, decompress a frame sequence, and terminate decompression.

```
LPBITMAPINFOHEADER lbpiIn, lpbiOut;
LPVOID             lpIn, lpOut;
LONG               lNumFrames, lFrameNum;

// Assume lpbiIn and lpbiOut are initialized to the input and output
// format and lpIn and lpOut are pointing to the buffers.
if (ICDecompressBegin(hIC, lpbiIn, lpbiOut) == ICERR_OK){
    for (lFrameNum = 0; lFrameNum < lNumFrames, lFrameNum++){
        if (ICDecompress(hIC, 0, lpbiIn, lpIn, lpbiOut,
            lpOut) == ICERR_OK)
        {
            // Frame decompressed OK so we can process it as required.
        } else {
            // Handle the decompression error that occurred.
        }
    }
    ICDecompressEnd(hIC);
} else {
    // Handle the error identifying an unsupported format.
}
```

Determining If a Driver Can Handle the Input Format

The following example shows how to check the input format with the **ICDrawQuery** macro.

```
// lpbiIn points to BITMAPINFOHEADER structure indicating the input
// format.
if (ICDrawQuery(hIC, lpbiIn) == ICERR_OK)
{
    // Driver recognizes the input format.
} else {
    // Need a different decompressor.
}
```

Preparing to Draw Data

The following example shows the initialization sequence that instructs the decompressor to draw full-screen.

```
// Assume lpbiIn has the input format, dwRate has the data rate.
if (ICDrawBegin(hIC, ICDRAW_QUERY | ICDRAW_FULLSCREEN, NULL, NULL,
    NULL, 0, 0, 0, 0, lpbiIn, 0, 0, 0, 0, dwRate,
    dwScale) == ICERR_OK)
{
    // Decompressor supports this drawing so set up to draw.
    ICDrawBegin(hIC, ICDRAW_FULLSCREEN, hPal, NULL, NULL, 0, 0, 0,
        0, lpbiIn, 0, 0, lbpi->biWidth, lpbi->biHeight, dwRate,
        dwScale);
    .
    . // Start decompressing and drawing frames.
    .

    // Drawing done. Terminate procedure.
    ICDrawEnd(hIC);
} else {
    .
    . // Use another renderer to draw data on the screen;
    . // ICDraw does not support the format.
}
```

Drawing Data

The following example uses the **ICDraw** functions to draw data on the screen.

```
DWORD    dwNumBuffers;

// Find out how many buffers need filling before drawing starts.
ICGetBuffersWanted(hIC, &dwNumBuffers);
for (dw = 0; dw < dwNumBuffers; dw++){
    ICDraw(hIC, 0, lpFormat, lpData, cbData, dw); // fill the pipeline
    .
    . // Point lpFormat and lpData to next format and buffer.
    .
}
ICDrawStart(hIC);  // starts the clock
dw = 0;
while (fPlaying) {
    ICDraw(hIC, 0, lpFormat, lpData, chData, dw); // fill more buffers
    .
    . // Point lpFormat and lpData to next format and buffer,
    . // update dw.
}

ICDrawStop(hIC);   // stops drawing and decompressing when done
ICDrawFlush(hIC);  // flushes any existing buffers
ICDrawEnd(hIC);    // ends decompression
```

Monitoring Compressor and Decompressor Progress

The following example shows how the **ICSetStatusProc** function is used to inform the compressor or decompressor of the callback function address:

```
ICSetStatusProc(compvars.hic, 0, (LPARAM) (UINT) hwndApp,
    &PreviewStatusProc);
```

The following example shows the callback function installed by the previous fragment:

```
LONG CALLBACK export PreviewStatusProc(LPARAM lParam,
    UINT message, LONG l)
{
    switch (message) {
    MSG msg;
    case ICSTATUS_START:
         .
         . // Create and display status dialog box.
         .
         break;
    case ICSTATUS_STATUS:
        ProSetBarPos((int) l); // sets status bar positions

        // Watch for abort message
        while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
            if (msg.message == WM_KEYDOWN && msg.wParam == VK_ESCAPE)
                return 1;
            if (msg.message == WM_SYSCOMMAND &&
                (msg.wParam & 0xFFF0) == SC_CLOSE)
                   return 1;

            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        break;
    case ICSTATUS_END:
         .
         . // Close and destroy status dialog box.
         .
        break;
    case ICSTATUS_YIELD:
         .
         .
         .
        break;
    }
    return 0;
}
```

Video Compression Manager Reference

This section describes the functions, messages and macros, notifications, and structures associated with VCM. These elements are grouped as follows.

Compressor Installation and Removal

ICInstall
ICLocate
ICOPEN
ICClose
ICRemove
ICOpenFunction

Locating and Opening a Compressor

ICLocate
ICOPEN
ICDecompressOpen
ICDrawOpen
ICINFO
ICClose

Selecting Compressors

ICCompressorChoose
ICCompressorFree
COMPVARS

Configuring Compressors

ICM_CONFIGURE
ICM_CONFIGURE
ICM_GETSTATE
ICM_GETSTATE
ICM_SETSTATE
ICSendMessage

Compressor Information

ICGetInfo
ICINFO
ICM_GETDEFAULTKEYFRAMERATE
ICGetDisplayFormat
ICM_GETDEFAULTQUALITY
ICM_ABOUT
ICM_ABOUT

Single Image Compression

ICImageCompress

Sequence Compression

ICSeqCompressFrame
ICSeqCompressFrameStart
ICSeqCompressFrameEnd

COMPVARS

ICCompressorChoose

Image Data Compression

ICM_COMPRESS_GET_FORMAT

ICM_COMPRESS_GET_FORMAT
ICM_COMPRESS_QUERY
ICM_COMPRESS_GET_SIZE
ICM_COMPRESS_BEGIN
ICCOMPRESS
ICM_COMPRESS_END
ICM_COMPRESS_BEGIN

Compressor Monitoring

ICSETSTATUSPROC

Decompressing Single Images

ICImageDecompress

Decompressing Image Data

ICDECOMPRESSEX
ICDecompressExBegin
ICM_DECOMPRESSEX_END
ICM_DECOMPRESS_GET_FORMAT
ICM_DECOMPRESS_GET_FORMAT
ICM_DECOMPRESS_GET_PALETTE
ICDecompressExQuery
ICDECOMPRESS
ICM_DECOMPRESS_BEGIN
ICM_DECOMPRESS_END
ICM_DECOMPRESS_QUERY

Using Hardware-Drawing Capabilities

ICGetInfo
ICDRAWBEGIN
ICM_DRAW_END
ICM_DRAW_FLUSH
ICM_DRAW_QUERY
ICDrawSuggestFormat
ICM_DRAW_START
ICM_DRAW_STOP
ICM_GETBUFFERSWANTED
ICM_DRAW_REALIZE
ICDrawOpen
ICDRAW
ICM_DRAW_GETTIME
ICM_DRAW_SETTIME
ICM_DRAW_WINDOW
ICM_DRAW_REALIZE
ICM_DRAW_CHANGEPALETTE
ICM_DRAW_RENDERBUFFER

Video Compression Manager Functions and Macros

An application uses VCM functions to initialize and control compression and decompression operations. These macros extend the feature set.

ICClose

```
LRESULT ICClose(HIC hic);
```

Closes a compressor or decompressor.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of a compressor or decompressor.

ICCompress

```
DWORD ICCompress(HIC hic, DWORD dwFlags, LPBITMAPINFOHEADER lpbiOutput,
    LPVOID lpData, LPBITMAPINFOHEADER lpbiInput, LPVOID lpBits,
    LPDWORD lpckid, LPDWORD lpdwFlags, LONG lFrameNum,
    DWORD dwFrameSize, DWORD dwQuality, LPBITMAPINFOHEADER lpbiPrev,
    LPVOID lpPrev);
```

Compresses a single video image.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of the compressor to use.

dwFlags
Compression flag. The following value is defined:

ICCOMPRESS_KEYFRAME
Compressor should make this frame a key frame.

lpbiOutput
Address of a **BITMAPINFO** structure containing the output format.

lpData
Address of an output buffer large enough to contain a compressed frame.

lpbiInput
Address of a **BITMAPINFO** structure containing the input format.

lpBits
Address of the input buffer.

lpckid
Reserved; do not use.

lpdwFlags
Address of the return flags used in the AVI index. The following value is defined:

AVIIF_KEYFRAME
Current frame is a key frame.

lFrameNum
Frame number.

dwFrameSize
Requested frame size, in bytes. Specify a nonzero value if the compressor supports a suggested frame size, as indicated by the presence of the VIDCF_CRUNCH flag returned by the **ICGetInfo** function. If this flag is not set or a data rate for the frame is not specified, specify zero for this parameter.

A compressor might have to sacrifice image quality or make some other trade-off to obtain the size goal specified in this parameter.

dwQuality
Requested quality value for the frame. Specify a nonzero value if the compressor supports a suggested quality value, as indicated by the presence of the VIDCF_QUALITY flag returned by **ICGetInfo**. Otherwise, specify zero for this parameter.

lpbiPrev
Address of a **BITMAPINFO** structure containing the format of the previous frame.

lpPrev
Address of the uncompressed image of the previous frame. This parameter is not used for fast temporal compression. Specify NULL for this parameter when compressing a key frame, if the compressor does not support temporal compression, or if the compressor does not require an external buffer to store the format and data of the previous image.

You can obtain the required by size of the output buffer by sending the ICM_COMPRESS_GET_SIZE message (or by using the **ICCompressGetSize** macro).

The compressor sets the contents of *lpdwFlags* to AVIIF_KEYFRAME when it creates a key frame. If your application creates AVI files, it should save the information returned for *lpckid* and *lpdwFlags* in the file.

Compressors use *lpbiPrev* and *lpPrev* to perform temporal compression and require an external buffer to store the format and data of the previous frame. Specify NULL for *lpbiPrev* and *lpPrev* when compressing a key frame, when performing fast compression, or if the compressor has its own buffer to store the format and data of the previous image. Specify non-NULL values for these parameters if **ICGetInfo** returns the VIDCF_TEMPORAL flag, the compressor is performing normal compression, and the frame to compress is not a key frame.

ICCompressorChoose

```
BOOL ICCompressorChoose(HWND hwnd, UINT uiFlags, LPVOID pvIn,
    LPVOID lpData, PCOMPVARS pc, LPSTR lpszTitle);
```

Displays a dialog box in which a user can select a compressor. This function can display all registered compressors or list only the compressors that support a specific format.

- Returns TRUE if the user chooses a compressor and presses OK. Returns FALSE on error or if the user presses CANCEL.

hwnd
Handle of a parent window for the dialog box.

uiFlags
Applicable flags. The following values are defined:

ICMF_CHOOSE_ALLCOMPRESSORS
All compressors should appear in the selection list. If this flag is not specified, only the compressors that can handle the input format appear in the selection list.

ICMF_CHOOSE_DATARATE
Displays a check box and edit box to enter the data rate for the movie.

ICMF_CHOOSE_KEYFRAME
Displays a check box and edit box to enter the frequency of key frames.

ICMF_CHOOSE_PREVIEW
Displays a button to expand the dialog box to include a preview window. The preview window shows how frames of your movie will appear when compressed with the current settings.

pvIn
Uncompressed data input format. Only compressors that support the specified data input format are included in the compressor list. This parameter is optional.

lpData
Address of an AVI stream interface to use in the preview window. You must specify a video stream. This parameter is optional.

pc
Address of a **COMPVARS** structure. The information returned initializes the structure for use with other functions.

lpszTitle
Address of a null-terminated string containing a title for the dialog box. This parameter is optional.

Before using this function, set the **cbSize** member of the **COMPVARS** structure to the size of the structure. Initialize the rest of the structure to zeros unless you want to specify some valid defaults for the dialog box. If specifying defaults, set the **dwFlags** member to ICMF_COMPVARS_VALID and initialize the other members of the structure. For more information about initializing the structure, see the **ICSeqCompressFrameStart** function and **COMPVARS**.

ICCompressorFree

```
void ICCompressorFree(PCOMPVARS pc);
```

Frees the resources in the **COMPVARS** structure used by other VCM functions.

pc
Address of the **COMPVARS** structure containing the resources to be freed.

Use this function to release the resources in the **COMPVARS** structure after using the **ICCompressorChoose**, **ICSeqCompressFrameStart**, **ICSeqCompressFrame**, and **ICSeqCompressFrameEnd** functions.

ICDecompress

```
DWORD ICDecompress(HIC hic, DWORD dwFlags,
    LPBITMAPINFOHEADER lpbiFormat, LPVOID lpData,
    LPBITMAPINFOHEADER lpbi, LPVOID lpBits);
```

Decompresses a single video frame.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of the decompressor to use.

dwFlags
Applicable decompression flags. The following values are defined:

ICDECOMPRESS_HURRYUP
Tries to decompress at a faster rate. When an application uses this flag, the driver should buffer the decompressed data but not draw the image.

ICDECOMPRESS_NOTKEYFRAME
Current frame is not a key frame.

ICDECOMPRESS_NULLFRAME
Current frame does not contain data and the decompressed image should be left the same.

ICDECOMPRESS_PREROLL
Current frame precedes the point in the movie where playback starts and, therefore, will not be drawn.

ICDECOMPRESS_UPDATE
Screen is being updated or refreshed.

lpbiFormat
Address of a **BITMAPINFO** structure containing the format of the compressed data.

lpData
Address of the input data.

lpbi
Address of a **BITMAPINFO** structure containing the output format.

lpBits
Address of a buffer that is large enough to contain the decompressed data.

ICDecompressEx

```
DWORD ICDecompressEx(HIC hic, DWORD dwFlags, LPBITMAPINFOHEADER lpbiSrc,
    LPVOID lpSrc, int xSrc, int ySrc, int dxSrc, int dySrc,
    LPBITMAPINFOHEADER lpbiDst, LPVOID lpDst, int xDst, int yDst,
    int dxDst, int dyDst);
```

Decompresses a single video frame.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of the decompressor.

dwFlags
Decompression flags. The following values are defined:

ICDECOMPRESS_HURRYUP
Tries to decompress at a faster rate. When an application uses this flag, the driver should buffer the decompressed data but not draw the image.

ICDECOMPRESS_NOTKEYFRAME
Current frame is not a key frame.

ICDECOMPRESS_NULLFRAME
Current frame does not contain data and the decompressed image should be left the same.

ICDECOMPRESS_PREROLL
Current frame precedes the point in the movie where playback starts and, therefore, will not be drawn.

ICDECOMPRESS_UPDATE
Screen is being updated or refreshed.

lpbiSrc
Address of a **BITMAPINFOHEADER** structure containing the format of the compressed data.

lpSrc
Address of the input data.

xSrc, ySrc
The x- and y- coordinates of the source rectangle for the DIB specified by *lpbiSrc*.

dxSrc, dySrc
Width and height of the source rectangle.

lpbiDst
Address of a **BITMAPINFOHEADER** structure containing the output format.

lpDst
Address of a buffer that is large enough to contain the decompressed data.

xDst, yDst
The x- and y-coordinates of the destination rectangle for the DIB specified by *lpbiDst*.

dxDst, dyDst
Width and height of the destination rectangle.

Typically, applications use the ICDECOMPRESS_PREROLL flag to seek to a key frame in a compressed stream. The flag is sent with the key frame and with subsequent frames required to decompress the desired frame.

ICDecompressExBegin

```
DWORD ICDecompressExBegin(HIC hic, DWORD dwFlags,
    LPBITMAPINFOHEADER lpbiSrc, LPVOID lpSrc, int xSrc, int ySrc,
    int dxSrc, int dySrc, LPBITMAPINFOHEADER lpbiDst, LPVOID lpDst,
    int xDst, int yDst, int dxDst, int dyDst);
```

Prepares a decompressor for decompressing data.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of the decompressor to use.

dwFlags
Decompression flags. The following values are defined:

ICDECOMPRESS_HURRYUP
Tries to decompress at a faster rate. When an application uses this flag, the driver should buffer the decompressed data but not draw the image.

ICDECOMPRESS_NOTKEYFRAME
Current frame is not a key frame.

ICDECOMPRESS_NULLFRAME
Current frame does not contain data and the decompressed image should be left the same.

ICDECOMPRESS_PREROLL
Current frame precedes the point in the movie where playback starts and, therefore, will not be drawn.

ICDECOMPRESS_UPDATE
Screen is being updated or refreshed.

lpbiSrc
Address of a **BITMAPINFOHEADER** structure containing the format of the compressed data.

lpSrc
Address of the input data.

xSrc, ySrc
The x- and y-coordinates of the source rectangle for the DIB specified by *lpbiSrc*.

dxSrc, dySrc
Width and height of the source rectangle.

lpbiDst
Address of a **BITMAPINFOHEADER** structure containing the output format.

lpDst
Address of a buffer that is large enough to contain the decompressed data.

xDst, yDst
The x- and y-coordinates of the destination rectangle for the DIB specified by *lpbiDst*.

dxDst, dyDst
Width and height of the destination rectangle.

ICDecompressExQuery

```
DWORD ICDecompressExQuery(HIC hic, DWORD dwFlags,
    LPBITMAPINFOHEADER lpbiSrc, LPVOID lpSrc, int xSrc, int ySrc,
    int dxSrc, int dySrc, LPBITMAPINFOHEADER lpbiDst, LPVOID lpDst,
    int xDst, int yDst, int dxDst, int dyDst);
```

Determines if a decompressor can decompress data with a specific format.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of the decompressor to use.

dwFlags
Reserved; do not use.

lpbiSrc
Address of a **BITMAPINFOHEADER** structure containing the format of the compressed data to decompress.

lpSrc
Reserved; must be NULL.

xSrc, ySrc
The x- and y-coordinates of the source rectangle for the DIB specified by *lpbiSrc*.

dxSrc, dySrc
Width and height of the source rectangle.

lpbiDst
Address of a **BITMAPINFOHEADER** structure containing the output format. If the value of this parameter is NULL, the function determines whether the input format is supported and this parameter is ignored.

lpDst
Address of a buffer that is large enough to contain the decompressed data.

xDst, yDst
The x- and y-coordinates of the destination rectangle for the DIB specified by *lpbiDst*.

dxDst, dyDst
Width and height of the destination rectangle.

ICDecompressOpen

```
HIC ICDecompressOpen(DWORD fccType, DWORD fccHandler,
    LPBITMAPINFOHEADER lpbiIn, LPBITMAPINFOHEADER lpbiOut)
```

Opens a decompressor that is compatible with the specified formats.

- Returns a handle of a decompressor if successful or zero otherwise.

fccType
Four-character code indicating the type of compressor to open. For video streams, the value of this parameter is "VIDC" or ICTYPE_VIDEO.

fccHandler
Four-character code indicating the preferred stream handler to use. Typically, this information is stored in the stream header in an AVI file.

lpbiIn
Address of a structure defining the input format. A decompressor handle is not returned unless it can decompress this format. For bitmaps, this parameter refers to a **BITMAPINFOHEADER** structure.

lpbiOut
Address of a structure defining an optional decompression format. You can also specify zero to use the default output format associated with the input format.

If this parameter is nonzero, a compressor handle is not returned unless it can create this output format. For bitmaps, this parameter refers to a **BITMAPINFOHEADER** structure.

The **ICDecompressOpen** macro is defined as follows:

```
#define ICDecompressOpen(fccType, fccHandler, lpbiIn, lpbiOut) \
    ICLocate(fccType, fccHandler, lpbiIn, lpbiOut, ICMODE_DECOMPRESS);
```

ICDraw

```
DWORD ICDraw(HIC hic, DWORD dwFlags, LPVOID lpFormat,
    LPVOID lpData, DWORD cbData, LONG lTime);
```

Decompresses an image for drawing.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of an decompressor.

dwFlags
Decompression flags. The following values are defined:

ICDRAW_HURRYUP
Data is buffered and not drawn to the screen. Use this flag for fastest decompression.

ICDRAW_NOTKEYFRAME
Current frame is not a key frame.

ICDRAW_NULLFRAME
Current frame does not contain any data and the previous frame should be redrawn.

ICDRAW_PREROLL
Current frame of video occurs before playback should start. For example, if playback will begin on frame 10, and frame 0 is the nearest previous key frame, frames 0 through 9 are sent to the driver with the ICDRAW_PREROLL flag set. The driver needs this data to display frame 10 properly.

ICDRAW_UPDATE
Updates the screen based on previously received data. Set *lpData* to NULL when this flag is used.

lpFormat
Address of a **BITMAPINFOHEADER** structure containing the input format of the data.

lpData
Address of the input data.

cbData
Size of the input data, in bytes.

lTime
Time, in samples, to draw this frame. The units for video data are frames. For a definition of the playback rate, see the **dwRate** and **dwScale** members of the **ICDRAWBEGIN** structure.

You can initiate drawing the frames by sending the ICM_DRAW_START message (or by using the **ICDrawStart** macro). The application should be sure to buffer the required number of frames before drawing is started. Send the KM_GETBUFFERSWANTED message (or use the **ICGetBuffersWanted** macro) to obtain this value.

ICDrawBegin

```
DWORD ICDrawBegin(HIC hic, DWORD dwFlags, HPALETTE hpal, HWND hwnd,
    HDC hdc, int xDst, int yDst, int dxDst, int dyDst,
    LPBITMAPINFOHEADER lpbi, int xSrc, int ySrc, int dxSrc,
    int dySrc, DWORD dwRate, DWORD dwScale);
```

Initializes the renderer and prepares the drawing destination for drawing.

- Returns ICERR_OK if the renderer can decompress the data or ICERR_UNSUPPORTED otherwise.

hic
Handle of the decompressor to use.

dwFlags
Decompression flags. The following values are defined:

ICDRAW_ANIMATE
Application can animate the palette.

ICDRAW_CONTINUE
Drawing is a continuation of the previous frame.

ICDRAW_FULLSCREEN
Draws the decompressed data on the full screen.

ICDRAW_HDC
Draws the decompressed data to a window or a DC.

ICDRAW_MEMORYDC
DC is off-screen.

ICDRAW_QUERY
Determines if the decompressor can decompress the data. The driver does not decompress the data.

ICDRAW_UPDATING
Current frame is being updated rather than played.

hpal
Handle of the palette used for drawing.

hwnd
Handle of the window used for drawing.

hdc
DC used for drawing.

xDst, yDst
The x- and y-coordinates of the upper right corner of the destination rectangle.

dxDst, dyDst
Width and height of the destination rectangle.

lpbi
Address of a **BITMAPINFO** structure containing the format of the input data to be decompressed.

xSrc, ySrc
The x- and y-coordinates of the upper right corner of the source rectangle.

dxSrc, dySrc
Width and height of the source rectangle.

dwRate
Frame rate numerator. The frame rate, in frames per second, is obtained by dividing *dwRate* by *dwScale*.

dwScale

Frame rate denominator. The frame rate, in frames per second, is obtained by dividing *dwRate* by *dwScale*.

The ICDRAW_HDC and ICDRAW_FULLSCREEN flags are mutually exclusive. If an application sets the ICDRAW_HDC flag in *dwFlags*, the decompressor uses *hwnd*, *hdc*, and the parameters defining the destination rectangle (*xDst*, *yDst*, *dxDst*, and *dyDst)*. Your application should set these parameters to the size of the destination rectangle. Specify destination rectangle values relative to the current window or DC.

If an application sets the ICDRAW_FULLSCREEN flag in *dwFlags*, the *hwnd* and *hdc* parameters are not used and should be set to NULL. Also, the destination rectangle is not used and its parameters can be set to zero.

The source rectangle is relative to the full video frame. The portion of the video frame specified by the source rectangle is stretched or shrunk to fit the destination rectangle.

The *dwRate* and *dwScale* parameters specify the decompression rate. The integer value specified for *dwRate* divided by the integer value specified for *dwScale* defines the frame rate in frames per second. This value is used by the renderer when it is responsible for timing frames during playback.

ICDrawOpen

```
HIC ICDrawOpen(DWORD fccType, DWORD fccHandler,
    LPBITMAPINFOHEADER lpbiIn)
```

Opens a driver that can draw images with the specified format.

- Returns a handle of a driver if successful or zero otherwise.

fccType
Four-character code indicating the type of driver to open. For video streams, the value of this parameter is "VIDC" or ICTYPE_VIDEO.

fccHandler
Four-character code indicating the preferred stream handler to use. Typically, this information is stored in the stream header in an AVI file.

lpbiIn
Address of the structure defining the input format. A driver handle will not be returned unless it can decompress this format. For images, this parameter refers to a **BITMAPINFOHEADER** structure.

The **ICDrawOpen** macro is defined as follows:

```
#define ICDrawOpen(fccType, fccHandler, lpbiIn) \
    ICLocate(fccType, fccHandler, lpbiIn, NULL, ICMODE_DRAW);
```

ICDrawSuggestFormat

```
DWORD ICDrawSuggestFormat(HIC hic, LPBITMAPINFOHEADER lpbiIn,
    LPBITMAPINFOHEADER lpbiOut, int dxSrc, int dySrc, int dxDst,
    int dyDst, HIC hicDecompressor);
```

Notifies the drawing handler to suggest the input data format.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of the driver to use.

lpbiIn
Address of a structure containing the format of the compressed data. For bitmaps, this is a **BITMAPINFOHEADER** structure.

lpbiOut
Address of a structure to return the suggested format. The drawing handler can receive and draw data from this format. For bitmaps, this is a **BITMAPINFOHEADER** structure.

dxSrc, dySrc
Width and height of the source rectangle.

dxDst, dyDst
Width and height of the destination rectangle.

hicDecompressor
Decompressor that can use the format of data in *lpbiIn*.

Applications can use this function to determine alternative input formats that a drawing handler can decompress and if the drawing handler can stretch data. If the drawing handler cannot stretch data as requested, the application might have to stretch the data.

If the drawing handler cannot decompress a format provided by an application, use the **ICDecompress**, **ICDecompressEx**, **ICDecompressExBegin**, **ICDecompressExQuery**, and **ICDecompressOpen** functions to obtain alternate formats.

ICGetDisplayFormat

```
HIC ICGetDisplayFormat(HIC hic, LPBITMAPINFOHEADER lpbiIn,
    LPBITMAPINFOHEADER lpbiOut, int BitDepth, int dx, int dy);
```

Determines the best format available for displaying a compressed image. The function also opens a compressor if a handle of an open compressor is not specified.

- Returns a handle of a decompressor if successful or zero otherwise.

hic
Handle of the compressor to use. Specify NULL to have VCM select and open an appropriate compressor.

lpbiIn
Address of **BITMAPINFOHEADER** structure containing the compressed format.

lpbiOut
Address of a buffer to return the decompressed format. The buffer should be large enough for a **BITMAPINFOHEADER** structure and 256 color entries.

BitDepth
Preferred bit depth, if nonzero.

dx, dy
Width and height multipliers to stretch the image. If this parameter is zero, that dimension is not stretched.

ICGetInfo

```
LRESULT ICGetInfo(HIC hic, ICINFO FAR * lpicinfo, DWORD cb);
```

Obtains information about a compressor.

- Returns the number of bytes copied into the structure or zero if an error occurs.

hic
Handle of a compressor.

lpicinfo
Address of the **ICINFO** structure to return information about the compressor.

cb
Size, in bytes, of the structure pointed to by *lpicinfo*.

ICImageCompress

```
HANDLE ICImageCompress(HIC hic, UINT uiFlags, LPBITMAPINFO lpbiIn,
    LPVOID lpBits, LPBITMAPINFO lpbiOut, LONG lQuality,
    LONG FAR * plSize);
```

Compresses an image to a given size. This function does not require initialization functions.

- Returns a handle of a compressed DIB. The image data follows the format header.

hic
Handle of a compressor opened with the **ICOpen** function. Specify NULL to have VCM select an appropriate compressor for the compression format. An application can have the user select the compressor by using the **ICCompressorChoose** function, which opens the selected compressor and returns a handle of the compressor in this parameter.

uiFlags
Reserved; must be zero.

lpbiIn
Address of the **BITMAPINFO** structure containing the input data format.

lpBits
Address of input data bits to compress. The data bits exclude header and format information.

lpbiOut
Address of the **BITMAPINFO** structure containing the compressed output format. Specify NULL to have the compressor use an appropriate format.

lQuality
Quality value used by the compressor. Values range from 0 to 10,000.

plSize
Maximum size desired for the compressed image. The compressor might not be able to compress the data to fit within this size. When the function returns, this parameter points to the size of the compressed image. Image sizes are specified in bytes.

To obtain the format information from the **LPBITMAPINFOHEADER** structure, use the **GlobalLock** function to lock the data. Use the **GlobalFree** function to free the DIB when you are finished.

ICImageDecompress

```
HANDLE ICImageDecompress(HIC hic, UINT uiFlags, LPBITMAPINFO lpbiIn,
    LPVOID lpBits, LPBITMAPINFO lpbiOut);
```

Decompresses an image without using initialization functions.

- Returns a handle of an uncompressed DIB in the CF_DIB format if successful or NULL otherwise. Image data follows the format header.

hic
Handle of a decompressor opened with the **ICOpen** function. Specify NULL to have VCM select an appropriate decompressor for the compressed image.

uiFlags
Reserved; must be zero.

lpbiIn
Compressed input data format.

lpBits
Address of input data bits to compress. The data bits exclude header and format information.

lpbiOut
Decompressed output format. Specify NULL to let decompressor use an appropriate format.

To obtain the format information from the **LPBITMAPINFOHEADER** structure, use the **GlobalLock** function to lock the data. Use the **GlobalFree** function to free the DIB when you are finished.

ICInfo

```
BOOL ICInfo(DWORD fccType, DWORD fccHandler, ICINFO FAR * lpicinfo);
```

Retrieves information about specific installed compressors or enumerates the installed compressors.

- Returns TRUE if successful or FALSE otherwise.

fccType
Four-character code indicating the type of compressor. Specify zero to match all compressor types.

fccHandler
Four-character code identifying a specific compressor or a number between zero and the number of installed compressors of the type specified by *fccType*.

lpicinfo
Address of a **ICINFO** structure to return information about the compressor.

To enumerate the compressors or decompressors, specify an integer for *fccHandler*. This function returns information for integers between zero and the number of installed compressors or decompressors of the type specified for *fccType*.

ICInstall

```
BOOL ICInstall(DWORD fccType, DWORD fccHandler, LPARAM lParam,
    LPSTR szDesc, UINT wFlags);
```

Installs a new compressor or decompressor.

- Returns TRUE if successful or FALSE otherwise.

fccType
Four-character code indicating the type of data used by the compressor or decompressor. Specify "VIDC" for a video compressor or decompressor.

fccHandler
Four-character code identifying a specific compressor or decompressor.

lParam
Address of a null-terminated string containing the name of the compressor or decompressor, or the address of a function used for compression or decompression. The contents of this parameter are defined by the flags set for *wFlags*.

szDesc
Reserved; do not use.

wFlags
Flags defining the contents of *lParam*. The following values are defined:

ICINSTALL_DRIVER
The *lParam* parameter contains the address of a null-terminated string that names the compressor to install.

ICINSTALL_FUNCTION
The *lParam* parameter contains the address of a compressor function. This function should be structured like the **DriverProc** entry point function used by compressors.

Applications must open an installed compressor or decompressor before using it.

If your application installs a function as a compressor or decompressor, it should remove the function with the **ICRemove** function before it terminates. This prevents other applications from trying to access the function when it is not available.

ICLocate

```
HIC ICLocate(DWORD fccType, DWORD fccHandler, LPBITMAPINFOHEADER lpbiIn,
    LPBITMAPINFOHEADER lpbiOut, WORD wFlags);
```

Finds a compressor or decompressor that can handle images with the specified formats, or finds a driver that can decompress an image with a specified format directly to hardware.

- Returns a handle of a compressor or decompressor if successful or zero otherwise.

fccType
Four-character code indicating the type of compressor or decompressor to open. For video streams, the value of this parameter is "VIDC".

fccHandler
Preferred handler of the specified type. Typically, the handler type is stored in the stream header in an AVI file. Specify NULL if your application can use any handler type or it does not know the handler type to use.

lpbiIn
Address of a **BITMAPINFOHEADER** structure defining the input format. A compressor handle is not returned unless it supports this format.

lpbiOut
Address of a **BITMAPINFOHEADER** structure defining an optional decompressed format. You can also specify zero to use the default output format associated with the input format.

If this parameter is nonzero, a compressor handle is not returned unless it can create this output format.

wFlags
Flags that describe the search criteria for a compressor or decompressor. The following values are defined:

ICMODE_COMPRESS
Finds a compressor that can compress an image with a format defined by *lpbiIn* to the format defined by *lpbiOut*.

ICMODE_DECOMPRESS
Finds a decompressor that can decompress an image with a format defined by *lpbiIn* to the format defined by *lpbiOut*.

ICMODE_DRAW
Finds a decompressor that can decompress an image with a format defined by *lpbiIn* and draw it directly to hardware.

ICMODE_FASTCOMPRESS
Has the same meaning as ICMODE_COMPRESS except the compressor is used for a real-time operation and emphasizes speed over quality.

ICMODE_FASTDECOMPRESS
Has the same meaning as ICMODE_DECOMPRESS except the decompressor is used for a real-time operation and emphasizes speed over quality.

ICOpen

```
HIC ICOpen(DWORD fccType, DWORD fccHandler, UINT wMode);
```

Opens a compressor or decompressor.

- Returns a handle of a compressor or decompressor if successful or zero otherwise.

fccType
: Four-character code indicating the type of compressor or decompressor to open. For video streams, the value of this parameter is "VIDC".

fccHandler
: Preferred handler of the specified type. Typically, the handler type is stored in the stream header in an AVI file.

wMode
: Flag defining the use of the compressor or decompressor. The following values are defined:

  ICMODE_COMPRESS
  : Compressor will perform normal compression.

  ICMODE_DECOMPRESS
  : Decompressor will perform normal decompression.

  ICMODE_DRAW
  : Decompressor will decompress and draw the data directly to hardware.

  ICMODE_FASTCOMPRESS
  : Compressor will perform fast (real-time) compression.

  ICMODE_FASTDECOMPRESS
  : Decompressor will perform fast (real-time) decompression.

  ICMODE_QUERY
  : Queries the compressor or decompressor for information.

ICOpenFunction

```
HIC ICOpenFunction(DWORD fccType, DWORD fccHandler, UINT wMode,
    FARPROC lpfnHandler);
```

Opens a compressor or decompressor defined as a function.

- Returns a handle of a compressor or decompressor if successful or zero otherwise.

fccType
Type of compressor to open. For video, the value of this parameter is ICTYPE_VIDEO.

fccHandler
Preferred handler of the specified type. Typically, this comes from the stream header in an AVI file.

wMode
Flag to define the use of the compressor or decompressor. The following values are defined:

ICMODE_COMPRESS
Compressor will perform normal compression.

ICMODE_DECOMPRESS
Decompressor will perform normal decompression.

ICMODE_DRAW
Decompressor will decompress and draw the data directly to hardware.

ICMODE_FASTCOMPRESS
Compressor will perform fast (real-time) compression.

ICMODE_FASTDECOMPRESS
Decompressor will perform fast (real-time) decompression.

ICMODE_QUERY
Queries the compressor or decompressor for information.

lpfnHandler
Address of the function used as the compressor or decompressor.

ICRemove

```
BOOL ICRemove(DWORD fccType, DWORD fccHandler, UINT wFlags);
```

Removes an installed compressor.

- Returns TRUE if successful or FALSE otherwise.

fccType
: Four-character code indicating the type of data used by the compressor or decompressor. Specify "VIDC" for a video compressor or decompressor.

fccHandler
: Four-character code identifying a specific compressor or a number between zero and the number of installed compressors of the type specified by *fccType*.

wFlags
: Reserved; do not use.

ICSendMessage

```
LRESULT ICSendMessage(HIC hic, UINT wMsg, DWORD dw1, DWORD dw2);
```

Sends a message to a compressor.

- Returns a message-specific result.

hic
Handle of the compressor to receive the message.

wMsg
Message to send.

dw1
Additional message-specific information.

dw2
Additional message-specific information.

ICSeqCompressFrame

```
LPVOID ICSeqCompressFrame(PCOMPVARS pc, UINT uiFlags,
    LPVOID lpBits, BOOL FAR * pfKey, LONG FAR * plSize);
```

Compresses one frame in a sequence of frames.

- Returns the address of the compressed bits or NULL if an error occurs.

pc
Address of a **COMPVARS** structure initialized with information about the compression.

uiFlags
Reserved; must be zero.

lpBits
Address of the data bits to compress. (The data bits exclude header or format information.)

pfKey
Returns whether or not the frame was compressed into a key frame.

plSize
Maximum size desired for the compressed image. The compressor might not be able to compress the data to fit within this size. When the function returns, the parameter points to the size of the compressed image. Images sizes are specified in bytes.

This function uses a **COMPVARS** structure to provide settings for the specified compressor and intersperses key frames at the rate specified by the **ICSeqCompressorFrameStart** function. You can specify values for the data rate for the sequence and the key-frame frequency by using the appropriate members of **COMPVARS**.

Use this function instead of the **ICCompress** function to compress a video sequence.

You can allow the user to specify a compressor and initialize a **COMPVARS** structure by using the **ICCompressorChoose** function. Or, you can initialize a **COMPVARS** structure manually.

Use the **ICSeqCompressFrameStart**, **ICSeqCompressFrame**, and **ICSeqCompressFrameEnd** functions to compress a sequence of frames to a specified data rate and number of key frames. Use **ICSeqCompressFrame** once for each frame to be compressed.

When finished with compression, use the **ICCompressorFree** function to release the resources specified by **COMPVARS**.

ICSeqCompressFrameEnd

```
void ICSeqCompressFrameEnd(PCOMPVARS pc);
```

Ends sequence compression that was initiated by using the **ICSeqCompressFrameStart** and **ICSeqCompressFrame** functions.

pc
Address of a **COMPVARS** structure used during sequence compression.

When finished with compression, use the **ICCompressorFree** function to release the resources specified by **COMPVARS**.

ICSeqCompressFrameStart

```
BOOL ICSeqCompressFrameStart(PCOMPVARS pc, LPBITMAPINFO lpbiIn);
```

Initializes resources for compressing a sequence of frames using the **ICSeqCompressFrame** function.

- Returns TRUE if successful or FALSE otherwise.

pc
Address of a **COMPVARS** structure initialized with information for compression.

lpbiIn
Format of the data to be compressed.

This function uses a **COMPVARS** structure to provide settings for the specified compressor and intersperses key frames at the rate specified by the **lKey** member of **COMPVARS**. You can specify values for the data rate for the sequence and the key-frame frequency by using the appropriate members of **COMPVARS**.

Use the **ICSeqCompressFrameStart**, **ICSeqCompressFrame**, and **ICSeqCompressFrameEnd** functions to compress a sequence of frames to a specified data rate and number of key frames.

When finished with compression, use the **ICCompressorFree** function to release the resources specified in **COMPVARS**.

COMPVARS needs to be initialized before you use this function. You can initialize the structure manually or you can allow the user to specify a compressor and initialize a **COMPVARS** structure by using the **ICCompressorChoose** function.

ICSetStatusProc

```
DWORD ICSetStatusProc(HIC hic, DWORD dwFlags, LONG lParam,
    LONG (CALLBACK * ()) fpfnStatus);
```

Sends the address of a status callback function to a compressor. The compressor calls this function during lengthy operations.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of the compressor.

dwFlags
Applicable flags. Set to zero.

lParam
Constant specified with the status callback address.

fpfnStatus
Address of the status callback function. Specify NULL to indicate no status callbacks should be sent.

MyStatusProc

```
LONG MyStatusProc(LPARAM lParam, UINT Message);
```

Describes an application-defined status callback function used by the ICM_SET_STATUS_PROC message and the **ICSetStatusProc** function.

- Returns zero if processing should continue or a nonzero value if it should end.

lParam
Constant specified with the status callback address.

Message
Status flag. It can be one of the following values:

ICSTATUS_END
A lengthy operation is finishing.

ICSTATUS_START
A lengthy operation is starting.

ICSTATUS_STATUS
Operation is proceeding, and is *lParam* percent done.

ICSTATUS_YIELD
A lengthy operation is proceeding. This value has the same meaning as ICSTATUS_STATUS but does not indicate a value for percentage done.

Video Compression Manager Messages

Applications use messages to communicate with VCM, compression drivers, decompression drivers, and rendering drivers. VCM macros provide a shorthand method of sending these messages. VCM messages are based on the **ICSendMessage** function. Definitions of VCM macros are included with the associated message definitions.

ICM_ABOUT

```
ICM_ABOUT
wParam = (DWORD) (UINT) hwnd;
lParam = 0;

// Corresponding macros
DWORD ICAbout(hic, hwnd);
DWORD ICQueryAbout(hic);
```

Notifies a video compression driver to display its About dialog box or queries a video compression driver to determine if it has an About dialog box.

- Returns ICERR_OK if the driver supports this message or ICERR_UNSUPPORTED otherwise.

hic
Handle of the compressor.

hwnd
Handle of the parent window of the displayed dialog box.

You can also determine if a driver has an About dialog box by specifying -1 in this parameter, as in the **ICQueryAbout** macro. The driver returns ICERR_OK if it has an About dialog box or ICERR_UNSUPPORTED otherwise.

ICM_COMPRESS

```
ICM_COMPRESS
wParam = (DWORD) (LPVOID) &icc;
lParam = sizeof(ICCOMPRESS);
```

Notifies a video compression driver to compress a frame of data into an application-defined buffer.

- Returns ICERR_OK if successful or an error otherwise.

icc

Address of an **ICCOMPRESS** structure. The following members of this structure specify the compression parameters: **lpbiInput**, **lpInput**, **lpbiOutput**, **lpOutput**, **lpbiPrev**, **lpPrev**, **lpckid**, **lpdwFlags**, **dwFrameSize**, and **dwQuality**.

The driver should also use the **biSizeImage** member of the **BITMAPINFOHEADER** structure associated with **lpbiOutput** of **ICCOMPRESS** to return the size of the compressed frame.

lParam

Size, in bytes, of **ICCOMPRESS**.

ICM_COMPRESS_BEGIN

```
ICM_COMPRESS_BEGIN
wParam = (DWORD) (LPVOID) lpbiInput;
lParam = (DWORD) (LPVOID) lpbiOutput;

// Corresponding macro
DWORD ICCompressBegin(hic, lpbiInput, lpbiOutput);
```

Notifies a video compression driver to prepare to compress data.

- Returns ICERR_OK if the specified compression is supported or ICERR_BADFORMAT if the input or output format is not supported.

hic
Handle of a compressor.

lpbiInput
Address of a **BITMAPINFO** structure containing the input format.

lpbiOutput
Address of a **BITMAPINFO** structure containing the output format.

The driver should allocate and initialize any tables or memory that it needs for compressing the data formats when it receives the ICM_COMPRESS message.

VCM saves the settings of the most recent ICM_COMPRESS_BEGIN message. The ICM_COMPRESS_BEGIN and ICM_COMPRESS_END messages do not nest. If your driver receives ICM_COMPRESS_BEGIN before compression is stopped with ICM_COMPRESS_END, it should restart compression with new parameters.

ICM_COMPRESS_END

```
ICM_COMPRESS_END
wParam = 0;
lParam = 0;

// Corresponding macro
DWORD ICCompressEnd(hic);
```

Notifies a video compression driver to end compression and free resources allocated for compression.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of the compressor.

VCM saves the settings of the most recent ICM_COMPRESS_BEGIN message. ICM_COMPRESS_BEGIN and ICM_COMPRESS_END do not nest. If your driver receives ICM_COMPRESS_BEGIN before compression is stopped with ICM_COMPRESS_END, it should restart compression with new parameters.

ICM_COMPRESS_FRAMES_INFO

```
ICM_COMPRESS_FRAMES_INFO
wParam = (DWORD) (LPVOID) &icf;
lParam = sizeof(ICCOMPRESSFRAMES);
```

Notifies a compression driver to set the parameters for the pending compression.

- Returns ICERR_OK if successful or an error otherwise.

wParam
Address of an **ICCOMPRESSFRAMES** structure. The **GetData** and **PutData** members of this structure are not used with this message.

lParam
Size, in bytes, of **ICCOMPRESSFRAMES**.

A compressor can use this message to determine how much space to allocate for each frame while compressing.

ICM_COMPRESS_GET_FORMAT

```
ICM_COMPRESS_GET_FORMAT
wParam = (DWORD) (LPVOID) lpbiInput;
lParam = (DWORD) (LPVOID) lpbiOutput;

// Corresponding macros
DWORD ICCompressGetFormat(hic, lpbiInput, lpbiOutput);
DWORD ICCompressGetFormatSize(hic, lpbi);
```

Requests the output format of the compressed data from a video compression driver.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of the compressor.

lpbiInput
Address of a **BITMAPINFO** structure containing the input format.

lpbiOutput
Address of a **BITMAPINFO** structure to contain the output format. You can specify zero for this parameter to request only the size of the output format, as in the **ICCompressGetFormatSize** macro.

If *lpbiOutput* is nonzero, the driver should fill the **BITMAPINFO** structure with the default output format corresponding to the input format specified for *lpbiInput*. If the compressor can produce several formats, the default format should be the one that preserves the greatest amount of information.

ICM_COMPRESS_GET_SIZE

```
ICM_COMPRESS_GET_SIZE
wParam = (DWORD) (LPVOID) lpbiInput;
lParam = (DWORD) (LPVOID) lpbiOutput;

// Corresponding macro
DWORD ICCompressGetSize(hic, lpbiInput, lpbiOutput);
```

Requests that the video compression driver supply the maximum size of one frame of data when compressed into the specified output format.

- Returns the maximum number of bytes a single compressed frame can occupy.

hic
Handle of a compressor.

lpbiInput
Address of a **BITMAPINFO** structure containing the input format.

lpbiOutput
Address of a **BITMAPINFO** structure containing the output format.

Typically, applications send this message to determine how large a buffer to allocate for the compressed frame.

The driver should calculate the size of the largest possible frame based on the input and output formats.

ICM_COMPRESS_QUERY

```
ICM_COMPRESS_QUERY
wParam = (DWORD) (LPVOID) lpbiInput;
lParam = (DWORD) (LPVOID) lpbiOutput;

// Corresponding macro
DWORD ICCompressQuery(hic, lpbiInput, lpbiOutput);
```

Queries a video compression driver to determine if it supports a specific input format or if it can compress a specific input format to a specific output format.

- Returns ICERR_OK if the specified compression is supported or ICERR_BADFORMAT otherwise.

hic
Handle of a compressor.

lpbiInput
Address of a **BITMAPINFO** structure containing the input format.

lpbiOutput
Address of a **BITMAPINFO** structure containing the output format. You can specify zero for this parameter to indicate any output format is acceptable.

When a driver receives this message, it should examine the **BITMAPINFO** structure associated with *lpbiInput* to determine if it can compress the input format.

ICM_CONFIGURE

```
ICM_CONFIGURE
wParam = (DWORD) (UINT) hwnd;
lParam = 0;

// Corresponding macros
DWORD ICQueryConfigure(hic);
DWORD ICConfigure(hic, hwnd);
```

Notifies a video compression driver to display its configuration dialog box or queries a video compression driver to determine if it has a configuration dialog box.

- Returns ICERR_OK if the driver supports this message or ICERR_UNSUPPORTED otherwise.

hic
Handle of the compressor.

hwnd
Handle of the parent window of the displayed dialog box.

You can determine if a driver has a configuration dialog box by specifying - 1 in this parameter, as in the **ICQueryConfigure** macro.

This message is different from the DRV_CONFIGURE message used for hardware configuration. The dialog box for this message should let the user set and edit the internal state referenced by the ICM_GETSTATE and ICM_SETSTATE messages. For example, this dialog box can let the user change parameters affecting the quality level and other similar compression options.

ICM_DECOMPRESS

```
ICM_DECOMPRESS
wParam = (DWORD) (LPVOID) &icd;
lParam = sizeof(ICDECOMPRESS);
```

Notifies a video decompression driver to decompress a frame of data into an application-defined buffer.

- Returns ICERR_OK if successful or an error otherwise.

wParam
Address of an **ICDECOMPRESS** structure.

lParam
Size, in bytes, of **ICDECOMPRESS**.

If you want the driver to decompress data directly to the screen, send the ICM_DRAW message.

The driver returns an error if this message is received before the ICM_DECOMPRESS_BEGIN message.

ICM_DECOMPRESS_BEGIN

```
ICM_DECOMPRESS_BEGIN
wParam = (DWORD) (LPVOID) lpbiInput;
lParam = (DWORD) (LPVOID) lpbiOutput;

// Corresponding macro
DWORD ICDecompressBegin(hic, lpbiInput, lpbiOutput);
```

Notifies a video decompression driver to prepare to decompress data.

- Returns ICERR_OK if the specified decompression is supported or ICERR_BADFORMAT otherwise.

hic
Handle of a decompressor.

lpbiInput
Address of a **BITMAPINFO** structure containing the input format.

lpbiOutput
Address of a **BITMAPINFO** structure containing the output format.

When the driver receives this message, it should allocate buffers and do any time-consuming operations so that it can process ICM_DECOMPRESS messages efficiently.

If you want the driver to decompress data directly to the screen, send the ICM_DRAW message.

The ICM_DECOMPRESS_BEGIN and ICM_DECOMPRESS_END messages do not nest. If your driver receives ICM_DECOMPRESS_BEGIN before decompression is stopped with ICM_DECOMPRESS_END, it should restart decompression with new parameters.

ICM_DECOMPRESS_END

```
ICM_DECOMPRESS_END
wParam = 0;
lParam = 0;

// Corresponding macro
DWORD ICDecompressEnd(hic);
```

Notifies a video decompression driver to end decompression and free resources allocated for decompression.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of a decompressor.

The driver should free any resources allocated for the ICM_DECOMPRESS_BEGIN message.

ICM_DECOMPRESS_BEGIN and ICM_DECOMPRESS_END do not nest. If your driver receives ICM_DECOMPRESS_BEGIN before decompression is stopped with ICM_DECOMPRESS_END, it should restart decompression with new parameters.

ICM_DECOMPRESS_GET_FORMAT

```
ICM_DECOMPRESS_GET_FORMAT
wParam = (DWORD) (LPVOID) lpbiInput;
lParam = (DWORD) (LPVOID) lpbiOutput;

// Corresponding macros
DWORD ICDecompressGetFormat(hic, lpbiInput, lpbiOutput);
DWORD ICDecompressGetFormatSize(hic, lpbi);
```

Requests the output format of the decompressed data from a video decompression driver.

- Return ICERR_OK if successful or an error otherwise.

hic
Handle of a decompressor.

lpbiInput
Address of a **BITMAPINFO** structure containing the input format.

lpbiOutput
Address of a **BITMAPINFO** structure to contain the output format. You can specify zero to request only the size of the output format, as in the **ICDecompressGetFormatSize** macro.

If *lpbiOutput* is nonzero, the driver should fill the **BITMAPINFO** structure with the default output format corresponding to the input format specified for *lpbiInput*. If the compressor can produce several formats, the default format should be the one that preserves the greatest amount of information.

ICM_DECOMPRESS_GET_PALETTE

```
ICM_DECOMPRESS_GET_PALETTE
wParam = (DWORD) (LPVOID) lpbiInput;
lParam = (DWORD) (LPVOID) lpbiOutput;

// Corresponding macro
DWORD ICDecompressGetPalette(hic, lpbiInput, lpbiOutput);
```

Requests that the video decompression driver supply the color table of the output **BITMAPINFOHEADER** structure.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of a decompressor.

lpbiInput
Address of a **BITMAPINFOHEADER** structure containing the input format.

lpbiOutput
Address of a **BITMAPINFOHEADER** structure to contain the color table. The space reserved for the color table is always at least 256 colors. You can specify zero for this parameter to return only the size of the color table.

If *lpbiOutput* is nonzero, the driver sets the **biClrUsed** member of **BITMAPINFOHEADER** to the number of colors in the color table. The driver fills the **bmiColors** members of **BITMAPINFO** with the actual colors.

The driver should support this message only if it uses a palette other than the one specified in the input format.

ICM_DECOMPRESS_QUERY

```
ICM_DECOMPRESS_QUERY
wParam = (DWORD) (LPVOID) lpbiInput;
lParam = (DWORD) (LPVOID) lpbiOutput;

// Corresponding macro
DWORD ICDecompressQuery(hic, lpbiInput, lpbiOutput);
```

Queries a video decompression driver to determine if it supports a specific input format or if it can decompress a specific input format to a specific output format.

- Returns ICERR_OK if the specified decompression is supported or ICERR_BADFORMAT otherwise.

hic
Handle of a decompressor.

lpbiInput
Address of a **BITMAPINFO** structure containing the input format.

lpbiOutput
Address of a **BITMAPINFO** structure containing the output format. You can specify zero for this parameter to indicate any output format is acceptable.

ICM_DECOMPRESS_SET_PALETTE

```
ICM_DECOMPRESS_SET_PALETTE
wParam = (DWORD) (LPVOID) lpbiPalette;
lParam = 0;

// Corresponding macro
ICDecompressSetPalette(hic, lpbiPalette);
```

Specifies a palette for a video decompression driver to use if it is decompressing to a format that uses a palette.

- Returns ICERR_OK if the decompression driver can precisely decompress images to the suggested palette using the set of colors as they are arranged in the palette. Returns ICERR_UNSUPPORTED otherwise.

lpbiPalette
Address of a **BITMAPINFOHEADER** structure whose color table contains the colors that should be used if possible. You can specify zero to use the default set of output colors.

This message should not affect decompression already in progress; rather, colors passed using this message should be returned in response to future ICM_DECOMPRESS_GET_FORMAT and ICM_DECOMPRESS_GET_PALETTE messages. Colors are sent back to the decompression driver in a future ICM_DECOMPRESS_BEGIN message.

This message is used primarily when a driver decompresses images to the screen and another application that uses a palette is in the foreground, forcing the decompression driver to adapt to a foreign set of colors.

ICM_DECOMPRESSEX

```
ICM_DECOMPRESSEX
wParam = (DWORD) (LPVOID) &icdex;
lParam = sizeof(ICDECOMPRESSEX);
```

Notifies a video compression driver to decompress a frame of data directly to the screen, decompress to an upside-down DIB, or decompress images described with source and destination rectangles.

- Returns ICERR_OK if successful or an error otherwise.

wParam
Address of an **ICDECOMPRESSEX** structure.

lParam
Size, in bytes, of **ICDECOMPRESSEX**.

This message is similar to ICM_DECOMPRESS except that it uses the **ICDECOMPRESSEX** structure to define its decompression information.

If you want the driver to decompress data directly to the screen, send the ICM_DRAW message.

The driver returns an error if this message is received before the ICM_DECOMPRESSEX_BEGIN message.

ICM_DECOMPRESSEX_BEGIN

```
ICM_DECOMPRESSEX_BEGIN
wParam = (DWORD) (LPVOID) &icdex;
lParam = sizeof(ICDECOMPRESSEX);
```

Notifies a video compression driver to prepare to decompress data.

- Returns ICERR_OK if the specified decompression is supported or ICERR_BADFORMAT otherwise.

wParam
Address of a **ICDECOMPRESSEX** structure containing the input and output formats.

lParam
Size, in bytes, of **ICDECOMPRESSEX**.

When the driver receives this message, it should allocate buffers and do any time-consuming operations so that it can process ICM_DECOMPRESSEX messages efficiently.

If you want the driver to decompress data directly to the screen, send the ICM_DRAW_BEGIN message.

The ICM_DECOMPRESSEX_BEGIN and ICM_DECOMPRESSEX_END messages do not nest. If your driver receives ICM_DECOMPRESSEX_BEGIN before decompression is stopped with ICM_DECOMPRESSEX_END, it should restart decompression with new parameters.

ICM_DECOMPRESSEX_END

```
ICM_DECOMPRESSEX_END
wParam = 0;
lParam = 0;

// Corresponding macro
DWORD ICDecompressExEnd(hic);
```

Notifies a video decompression driver to end decompression and free resources allocated for decompression.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of a decompressor.

The driver frees any resources allocated for the ICM_DECOMPRESSEX_BEGIN message.

ICM_DECOMPRESSEX_BEGIN and ICM_DECOMPRESSEX_END do not nest. If your driver receives ICM_DECOMPRESSEX_BEGIN before decompression is stopped with ICM_DECOMPRESSEX_END, it should restart decompression with new parameters.

ICM_DECOMPRESSEX_QUERY

```
ICM_DECOMPRESSEX_QUERY
wParam = (DWORD) (LPVOID) &icdex;
lParam = sizeof(ICDECOMPRESSEX);
```

Queries a video compression driver to determine if it supports a specific input format or if it can decompress a specific input format to a specific output format.

- Returns ICERR_OK if the specified decompression is supported or ICERR_BADFORMAT otherwise.

wParam
Address of a **ICDECOMPRESSEX** structure containing the input format.

lParam
Size, in bytes, of **ICDECOMPRESSEX**.

ICM_DRAW

```
ICM_DRAW
wParam = (DWORD) (LPVOID) &icdraw;
lParam = sizeof(ICDRAW);
```

Notifies a rendering driver to decompress a frame of data and draw it to the screen.

- Returns ICERR_OK if successful or an error otherwise.

wParam
Address of an **ICDRAW** structure.

lParam
Size, in bytes, of **ICDRAW**.

If the ICDRAW_UPDATE flag is set in the **dwFlags** member of **ICDRAW**, the area of the screen used for drawing is invalid and needs to be updated. The extent of updating depends on the contents of the **lpData** member.

If **lpData** is NULL, the driver should update the entire destination rectangle with the current image. If the driver maintains a copy of the image in an off-screen buffer, it can fail this message. If **lpData** is not NULL, the driver should draw the data and make sure the entire destination is updated.

If the ICDRAW_HURRYUP flag is set in **dwFlags**, the calling application wants the driver to proceed as quickly as possible, possibly not even updating the screen.

If the ICDRAW_PREROLL flag is set in **dwFlags**, this video frame is preliminary information and should not be displayed if possible. For example, if play is to start from frame 10, and frame 0 is the nearest previous key frame, frames 0 through 9 will have ICDRAW_PREROLL set.

If you want the driver to decompress data into a buffer, send the ICM_DECOMPRESS message.

ICM_DRAW_BEGIN

```
ICM_DRAW_BEGIN
wParam = (DWORD) (LPVOID) &icdrwBgn;
lParam = sizeof(ICDRAW);
```

Notifies a rendering driver to prepare to draw data.

- Returns ICERR_OK if the driver supports drawing the data to the screen in the specified manner and format, or an error code otherwise. Possible error values include the following:

ICERR_BADFORMAT	Input or output format is not supported.
ICERR_NOTSUPPORTED	Driver does not draw directly to the screen or does not support this message.

wParam
Address of an **ICDRAWBEGIN** structure containing the input format.

lParam
Size, in bytes, of **ICDRAWBEGIN**.

If you want the driver to decompress data into a buffer, send the ICM_DECOMPRESS_BEGIN message.

The ICM_DRAW_BEGIN and ICM_DRAW_END messages do not nest. If your driver receives ICM_DRAW_BEGIN before decompression is stopped with ICM_DRAW_END, it should restart decompression with new parameters.

ICM_DRAW_CHANGEPALETTE

```
ICM_DRAW_CHANGEPALETTE
wParam = (DWORD) (LPVOID) lpbiInput;
lParam = 0;

// Corresponding macro
DWORD ICDrawChangePalette(hic, lpbiInput);
```

Notifies a rendering driver that the movie palette is changing.

- Returns ICERR_OK if successful or FALSE otherwise.

hic
Handle of a rendering driver.

lpbiInput
Address of a **BITMAPINFO** structure containing the new format and optional color table.

This message should be supported by installable rendering handlers that draw DIBs with an internal structure that includes a palette.

ICM_DRAW_END

```
ICM_DRAW_END
wParam = 0;
lParam = 0;

// Corresponding macro
DWORD ICDrawEnd(hic);
```

Notifies a rendering driver to decompress the current image to the screen and to release resources allocated for decompression and drawing.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of a driver.

The ICM_DRAW_BEGIN and ICM_DRAW_END messages do not nest. If your driver receives ICM_DRAW_BEGIN before decompression is stopped with ICM_DRAW_END, it should restart decompression with new parameters.

ICM_DRAW_FLUSH

```
ICM_DRAW_FLUSH
wParam = 0;
lParam = 0;

// Corresponding macro
DWORD ICDrawFlush(hic);
```

Notifies a rendering driver to render the contents of any image buffers that are waiting to be drawn.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of a driver.

This message is used only by hardware that performs its own asynchronous decompression, timing, and drawing.

ICM_DRAW_GET_PALETTE

```
ICM_DRAW_GET_PALETTE
wParam = 0;
lParam = 0;
```

Requests a rendering driver to return a palette.

- The driver should return one of the following: a handle of the palette being used, NULL if it doesn't have a handle to return, or ICERR_UNSUPPORTED if it doesn't support palettes.

ICM_DRAW_GETTIME

```
ICM_DRAW_GETTIME
wParam = (DWORD) (LPVOID) lplTime;
lParam = 0;

// Corresponding macro
DWORD ICDrawGetTime(hic, lplTime);
```

Requests a rendering driver that controls the timing of drawing frames to return the current value of its internal clock.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of a driver.

lplTime
Address to contain the current time. The return value should be specified in samples.

This message is generally supported by hardware that performs its own asynchronous decompression, timing, and drawing. The message can also be sent if the hardware is being used as the synchronization master.

ICM_DRAW_QUERY

```
ICM_DRAW_QUERY
wParam = (DWORD) (LPVOID) lpbiInput;
lParam = 0;

// Corresponding macro
DWORD ICDrawQuery(hic, lpbiInput);
```

Queries a rendering driver to determine if it can render data in a specific format.

- Returns ICERR_OK if the driver can render data in the specified format or ICERR_BADFORMAT otherwise.

hic
Handle of a driver.

lpbiInput
Address of a **BITMAPINFO** structure containing the input format.

This message differs from the ICM_DRAW_BEGIN message in that it queries the driver in a general sense. ICM_DRAW_BEGIN determines if the driver can draw the data using the specified format under specific conditions, such as stretching the image.

ICM_DRAW_REALIZE

```
ICM_DRAW_REALIZE
wParam = (DWORD) (UINT) (HDC) hdc;
lParam = (DWORD) (BOOL) fBackground;

// Corresponding macro
DWORD ICDrawRealize(hic, hdc, fBackground);
```

Notifies a rendering driver to realize its drawing palette while drawing.

- Returns ICERR_OK if the drawing palette is realized or ICERR_UNSUPPORTED if the palette associated with the decompressed data is realized.

hic
Handle of a driver.

hdc
Handle of the DC used to realize the palette.

fBackground
Background flag. Specify TRUE to realize the palette as a background task or FALSE to realize the palette in the foreground.

Drivers need to respond to this message only if the drawing palette is different from the decompressed palette.

ICM_DRAW_RENDERBUFFER

```
ICM_DRAW_RENDERBUFFER
wParam = 0;
lParam = 0;

// Corresponding macro
DWORD ICDrawRenderBuffer(hic);
```

Notifies a rendering driver to draw the frames that have been passed to it.

- No return value.

hic
Handle of a driver.

Use this message with hardware that performs its own asynchronous decompression, timing, and drawing.

This message is typically used to perform a seek operation when the driver must be specifically instructed to display each video frame passed to it rather than playing a sequence of video frames.

ICM_DRAW_SETTIME

```
ICM_DRAW_SETTIME
wParam = (DWORD) lpTime;
lParam = 0;

// Corresponding macro
DWORD ICDrawSetTime(hic, lpTime);
```

Provides synchronization information to a rendering driver that handles the timing of drawing frames. The synchronization information is the sample number of the frame to draw.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of a driver.

lpTime
Sample number of the frame to render.

Typically, the driver compares the specified value with the frame number associated with the time of its internal clock and attempts to synchronize the two if the difference is significant.

Use this message when the hardware performs its own asynchronous decompression, timing, and drawing, and the hardware relies on an external synchronization signal (the hardware is not being used as the synchronization master).

ICM_DRAW_START

```
ICM_DRAW_START
wParam = 0;
lParam = 0;

// Corresponding macro
DWORD ICDrawStart(hic);
```

Notifies a rendering driver to start its internal clock for the timing of drawing frames.

- No return value.

hic
Handle of a driver.

This message is used by hardware that performs its own asynchronous decompression, timing, and drawing.

When the driver receives this message, it should start rendering data at the rate specified with the ICM_DRAW_BEGIN message.

The ICM_DRAW_START and ICM_DRAW_STOP messages do not nest. If your driver receives ICM_DRAW_START before rendering is stopped with ICM_DRAW_STOP, it should restart rendering with new parameters.

ICM_DRAW_START_PLAY

```
ICM_DRAW_START_PLAY
wParam = (DWORD) lFrom;
lParam = (DWORD) lTo;

// Corresponding macro
ICDrawStartPlay(hic, lFrom, lTo);
```

Provides the start and end times of a play operation to a rendering driver.

- No return value.

lFrom
Start time.

lTo
End time.

This message precedes any frame data sent to the rendering driver.

Units for *lFrom* and *lTo* are specified with the ICM_DRAW_BEGIN message. For video data this is normally a frame number. For more information about the playback rate, see the **dwRate** and **dwScale** members of the **ICDRAWBEGIN** structure.

If the end time is less than the start time, the playback direction is reversed.

ICM_DRAW_STOP

```
ICM_DRAW_STOP
wParam = 0;
lParam = 0;

// Corresponding macro
DWORD ICDrawStop(hic);
```

Notifies a rendering driver to stop its internal clock for the timing of drawing frames.

- No return value.

hic
Handle of a driver.

This message is used by hardware that performs its own asynchronous decompression, timing, and drawing.

ICM_DRAW_STOP_PLAY

```
ICM_DRAW_STOP_PLAY
wParam = 0;
lParam = 0;

// Corresponding macro
ICDrawStopPlay(hic);
```

Notifies a rendering driver when a play operation is complete.

- No return value.

Use this message when the play operation is complete. Use the ICM_DRAW_STOP message to end timing.

ICM_DRAW_SUGGESTFORMAT

```
ICM_DRAW_SUGGESTFORMAT
wParam = (DWORD) (LPVOID) &icdrwSuggest;
lParam = sizeof(ICDRAWSUGGEST);
```

Queries a rendering driver to suggest a decompressed format that it can draw.

- Returns ICERR_OK if successful. If the **lpbiSuggest** member of the **ICDRAWSUGGEST** structure is NULL, this message returns the amount of memory required to contain the suggested format.

wParam
Address of an **ICDRAWSUGGEST** structure.

lParam
Size, in bytes, of **ICDRAWSUGGEST**.

The driver should examine the format specified in the **lpbiIn** member of the **ICDRAWSUGGEST** structure and use the **lpbiSuggest** member to return a format it can draw. The output format should preserve as much data as possible from the input format.

Optionally, the driver can use the installable compressor handle passed in the **hicDecompressor** member of **ICDRAWSUGGEST** to make more complex selections. For example, if the input format is 24-bit JPEG data, a renderer could query the decompressor to find out if it can decompress to a YUV format (which might be drawn more efficiently) before selecting the format to suggest.

ICM_DRAW_WINDOW

```
ICM_DRAW_WINDOW
wParam = (DWORD) (LPVOID) prc;
lParam = 0;

// Corresponding macro
DWORD ICDrawWindow(hic, prc);
```

Notifies a rendering driver that the window specified for the ICM_DRAW_BEGIN message needs to be redrawn. The window has moved or become temporarily obscured.

- Returns ICERR_OK if successful or an error otherwise.

hic
Handle of a driver.

prc
Address of the destination rectangle in screen coordinates. If this parameter points to an empty rectangle, drawing should be turned off.

This message is supported by hardware that performs its own asynchronous decompression, timing, and drawing.

Video-overlay drivers use this message to draw when the window is obscured or moved. When a window specified for ICM_DRAW_BEGIN is completely hidden by other windows, the destination rectangle is empty. Drivers should turn off video-overlay hardware when this condition occurs.

ICM_GET

```
ICM_GET
wParam = (DWORD) (LPVOID) pv;
lParam = (DWORD) cb;
```

Retrieves an application-defined doubleword from a video compression driver.

- Returns the amount of memory, in bytes, required to store the status information.

wParam
Address of a block of memory to be filled with the current state. You can also specify NULL to determine the amount of memory required by the state information.

lParam
Size, in bytes, of the block of memory.

The structure used to represent state information is driver specific and is defined by the driver.

ICM_GETBUFFERSWANTED

```
ICM_GETBUFFERSWANTED
wParam = (DWORD) (LPVOID) lpdwBuffers;
lParam = 0;

// Corresponding macro
DWORD ICGetBuffersWanted(hic, lpdwBuffers);
```

Queries a driver for the number of buffers to allocate.

- Returns ICERR_OK if successful or ICERR_UNSUPPORTED otherwise.

hic
Handle of a driver.

lpdwBuffers
Address to contain the number of samples the driver needs to efficiently render the data.

This message is used by drivers that use hardware to render data and want to ensure a minimal time lag caused by waiting for buffers to arrive. For example, if a driver controls a video decompression board that can hold 10 frames of video, it could return 10 for this message. This instructs applications to try to stay 10 frames ahead of the frame it currently needs.

ICM_GETDEFAULTKEYFRAMERATE

```
ICM_GETDEFAULTKEYFRAMERATE
wParam = (DWORD) (LPVOID) &dwICValue;
lParam = 0;

// Corresponding macro
DWORD ICGetDefaultKeyFrameRate(hic);
```

Queries a video compression driver for its default (or preferred) key-frame spacing.

- Returns ICERR_OK if the driver supports this message or ICERR_UNSUPPORTED otherwise. The **ICGetDefaultKeyFrameRate** macro returns the default key-frame rate.

hic
Handle of a compressor.

wParam
Address to contain the preferred key-frame spacing.

ICM_GETDEFAULTQUALITY

```
ICM_GETDEFAULTQUALITY
wParam = (DWORD) (LPVOID) &dwICValue;
lParam = 0;

// Corresponding macro
DWORD ICGetDefaultQuality(hic);
```

Queries a video compression driver to provide its default quality setting.

- Returns ICERR_OK if the driver supports this message or ICERR_UNSUPPORTED otherwise. The **ICGetDefaultQuality** macro returns the default quality value.

hic
Handle of a compressor.

wParam
Address to contain the default quality value. Quality values range from 0 to 10,000.

ICM_GETINFO

```
ICM_GETINFO
wParam = (DWORD) (LPVOID) lpicinfo;
lParam = sizeof(icinfo);
```

Queries a video compression driver to return a description of itself in an **ICINFO** structure.

- Returns the size, in bytes, of **ICINFO** or zero if an error occurs.

wParam
Address of an **ICINFO** structure to contain information.

lParam
Size, in bytes, of **ICINFO**.

Typically, applications send this message to display a list of the installed compressors.

The driver should fill all members of the **ICINFO** structure except **szDriver**.

ICM_GETQUALITY

```
ICM_GETQUALITY
wParam = (DWORD) (LPVOID) &dwICValue;
lParam = 0;
```

Queries a video compression driver to return its current quality setting.

- Returns ICERR_OK if the driver supports this message or ICERR_UNSUPPORTED otherwise.

wParam

Address to contain the current quality value. Quality values range from 0 to 10,000.

ICM_GETSTATE

```
ICM_GETSTATE
wParam = (DWORD) (LPVOID) pv;
lParam = (DWORD) cb;

// Corresponding macros
DWORD ICGetState(hic, pv, cb);
DWORD ICGetStateSize(hic);
```

Queries a video compression driver to return its current configuration in a block of memory or to determine the amount of memory required to retrieve the configuration information.

- Returns the amount of memory, in bytes, required by the state information. The **ICGetStateSize** macro returns the number of bytes used by the state data.

hic
Handle of the compressor.

pv
Address of a block of memory to contain the current configuration information. You can specify NULL for this parameter to determine the amount of memory required for the configuration information, as in **ICGetStateSize**.

cb
Size, in bytes, of the block of memory.

The structure used to represent configuration information is driver specific and is defined by the driver.

Use **ICGetStateSize** before calling the **ICGetState** macro to determine the size of buffer to allocate for the call.

ICM_SET_STATUS_PROC

```
ICM_SET_STATUS_PROC
wParam = (DWORD) (LPVOID) icsetstatusProc;
lParam = sizeof(icsetstatusProc);
```

Provides a status callback function with the status of a lengthy operation.

- Return ICERR_OK if successful or an error otherwise.

wParam
Address of an **ICSETSTATUSPROC** structure.

lParam
Size, in bytes, of **ICSETSTATUSPROC**.

Support of this message is optional but strongly recommended if compression or decompression takes longer than approximately one-tenth of a second.

An application can send this message periodically to a status callback function during lengthy operations.

ICM_SETQUALITY

```
ICM_SETQUALITY
wParam = (DWORD) (LPVOID) &dwICValue;
lParam = 0;
```

Provides a video compression driver with a quality level to use during compression.

- Returns ICERR_OK if the driver supports this message or ICERR_UNSUPPORTED otherwise.

wParam

New quality value. Quality values range from 0 to 10,000.

ICM_SETSTATE

```
ICM_SETSTATE
wParam = (DWORD) (LPVOID) pv;
lParam = (DWORD) cb;

// Corresponding macro
DWORD ICSetState(hic, pv, cb);
```

Notifies a video compression driver to set the state of the compressor.

- Returns the number of bytes used by the compressor if successful or zero otherwise.

hic
Handle of the compressor.

pv
Address of a block of memory containing configuration data. You can specify NULL for this parameter to reset the compressor to its default state.

cb
Size, in bytes, of the block of memory.

The information used by this message is private and specific to a given compressor. Client applications should use this message only to restore information previously obtained with the ICM_GETSTATE message and should use the ICM_CONFIGURE message to adjust the configuration of a video compression driver.

Video Compression Manager Structures

Applications use structures to transfer data in VCM messages and functions. The following structures are specific to VCM and provide support for VCM functions and messages.

COMPVARS

```
typedef struct {
    LONG          cbSize;      // see below
    DWORD         dwFlags;     // see below
    HIC           hic;         // see below
    DWORD         fccType;     // see below
    DWORD         fccHandler;  // see below
    LPBITMAPINFO lpbiIn;       // reserved; do not use
    LPBITMAPINFO lpbiOut;      // see below
    LPVOID        lpBitsOut;   // reserved; do not use
    LPVOID        lpBitsPrev;  // reserved; do not use
    LONG          lFrame;      // reserved; do not use
    LONG          lKey;        // see below
    LONG          lDataRate;   // see below
    LONG          lQ;          // see below
    LONG          lKeyCount;   // reserved; do not use
    LPVOID        lpState;     // reserved; do not use
    LONG          cbState;     // reserved; do not use
} COMPVARS;
```

Describes compressor settings for functions such as **ICCompressorChoose**, **ICSeqCompressFrame**, and **ICCompressorFree**.

cbSize
Size, in bytes, of this structure. This member must be set to validate the structure before calling any function using this structure.

dwFlags
Applicable flags. The following value is defined:

ICMF_COMPVARS_VALID
Data in this structure is valid and has been manually entered. Set this flag before you call any function if you fill this structure manually. Do not set this flag if you let **ICCompressorChoose** initialize this structure.

hic
Handle of the compressor to use. You can open a compressor and obtain a handle of it by using the **ICOpen** function. You can also choose a compressor by using **ICCompressorChoose**. **ICCompressorChoose** opens the chosen compressor and returns the handle of the compressor in this member. You can close the compressor by using **ICCompressorFree**.

fccType
Type of compressor used. Currently only ICTYPE_VIDEO (VIDC) is supported. This member can be set to zero.

fccHandler
Four-character code of the compressor. Specify NULL to indicate the data is not to be recompressed. Specify "DIB" to indicate the data is an uncompressed, full frame. You can use this member to specify which compressor is selected by default when the dialog box is displayed.

lpbiOut
Address of a **BITMAPINFO** structure containing the image output format. You can specify a specific format to use or you can specify NULL to use the default compressor associated with the input format. You can also set the image output format by using **ICCompressorChoose**.

lKey
Key-frame rate. Specify an integer to indicate the frequency that key frames are to occur in the compressed sequence or zero to not use key frames. You can also let **ICCompressorChoose** set the key-frame rate selected in the dialog box. The **ICSeqCompressFrameStart** function uses the

value of this member for making key frames.

lDataRate

Data rate, in kilobytes per second. **ICCompressorChoose** copies the selected data rate from the dialog box to this member.

lQ

Quality setting. Specify a quality setting of 1 to 10,000 or specify ICQUALITY_DEFAULT to use the default quality setting. You can also let **ICCompressorChoose** set the quality value selected in the dialog box. **ICSeqCompressFrameStart** uses the value of this member as its quality setting.

You can let **ICCompressorChoose** fill the contents of this structure or you can do it manually. If you manually fill the structure, you must provide information for the following members: **cbSize**, **hic**, **lpbiOut**, **lKey**, and **lQ**. Also, you must set the ICMF_COMPVARS_VALID flag in the **dwFlags** member.

ICCOMPRESS

```
typedef struct {
    DWORD              dwFlags;      // see below
    LPBITMAPINFOHEADER lpbiOutput;   // see below
    LPVOID             lpOutput;     // see below
    LPBITMAPINFOHEADER lpbiInput;    // see below
    LPVOID             lpInput;      // see below
    LPDWORD            lpckid;       // see below
    LPDWORD            lpdwFlags;    // see below
    LONG               lFrameNum;    // see below
    DWORD              dwFrameSize;  // see below
    DWORD              dwQuality;    // quality setting
    LPBITMAPINFOHEADER lpbiPrev;     // see below
    LPVOID             lpPrev;       // see below
} ICCOMPRESS;
```

Contains compression parameters used with the ICM_COMPRESS message.

dwFlags
Flags used for compression. The following value is defined:

ICCOMPRESS_KEYFRAME
Input data should be treated as a key frame.

lpbiOutput
Address of a **BITMAPINFOHEADER** structure containing the output (compressed) format. The **biSizeImage** member must contain the size of the compressed data.

lpOutput
Address of the buffer where the driver should write the compressed data.

lpbiInput
Address of a **BITMAPINFOHEADER** structure containing the input (uncompressed) format.

lpInput
Address of the buffer containing input data.

lpckid
Address to contain the chunk identifier for data in the AVI file. If the value of this member is not NULL, the driver should specify a two-character code for the chunk identifier corresponding to the chunk identifier used in the AVI file.

lpdwFlags
Address to contain flags for the AVI index. If the returned frame is a key frame, the driver should set the AVIIF_KEYFRAME flag.

lFrameNum
Number of the frame to compress.

dwFrameSize
Desired maximum size, in bytes, for compressing this frame. The size value is used for compression methods that can make tradeoffs between compressed image size and image quality. Specify zero for this member to use the default setting.

lpbiPrev
Address of a **BITMAPINFOHEADER** structure containing the format of the previous frame, which is typically the same as the input format.

lpPrev
Address of the buffer containing input data of the previous frame.

Drivers that perform temporal compression use data from the previous frame (found in the **lpbiPrev**

and **lpPrev** members) to prune redundant data from the current frame.

ICCOMPRESSFRAMES

```
typedef struct {
    DWORD               dwFlags;      // see below
    LPBITMAPINFOHEADER lpbiOutput;    // see below
    LPARAM              lOutput;      // reserved; do not use
    LPBITMAPINFOHEADER lpbiInput;     // see below
    LPARAM              lInput;       // reserved; do not use
    LONG                lStartFrame;  // see below
    LONG                lFrameCount;  // see below
    LONG                lQuality;     // quality setting
    LONG                lDataRate;    // see below
    LONG                lKeyRate;     // see below
    DWORD               dwRate;       // see below
    DWORD               dwScale;      // see below
    DWORD               dwOverheadPerFrame;   // reserved; do not use
    DWORD               dwReserved2;  // reserved; do not use
    LONG (CALLBACK* GetData) (LPARAM lInput, LONG lFrame,
        LPVOID lpBits, LONG len);
    LONG (CALLBACK* PutData) (LPARAM lInput, LONG lFrame,
        LPVOID lpBits, LONG len);
} ICCOMPRESSFRAMES;
```

Contains compression parameters used with the ICM_COMPRESS_FRAMES_INFO message.

dwFlags
Applicable flags. The following value is defined:

ICDECOMPRESSFRAMES_PADDING
Padding is used with the frame.

lpbiOutput
Address of a **BITMAPINFOHEADER** structure containing the output format.

lpbiInput
Address of a **BITMAPINFOHEADER** structure containing the input format.

lStartFrame
Number of the first frame to compress.

lFrameCount
Number of frames to compress.

lDataRate
Maximum data rate, in bytes per second.

lKeyRate
Maximum number of frames between consecutive key frames.

dwRate
Compression rate in an integer format. To obtain the rate in frames per second, divide this value by the value in **dwScale**.

dwScale
Value used to scale **dwRate** to frames per second.

GetData
Reserved; do not use.

PutData
Reserved; do not use.

ICDECOMPRESS

```
typedef struct {
    DWORD              dwFlags;
    LPBITMAPINFOHEADER lpbiInput;
    LPVOID             lpInput;
    LPBITMAPINFOHEADER lpbiOutput;
    LPVOID             lpOutput;
    DWORD              ckid;
} ICDECOMPRESS;
```

Contains decompression parameters used with the ICM_DECOMPRESS message.

dwFlags
Applicable flags. The following values are defined:

ICDECOMPRESS_HURRYUP
Tries to decompress at a faster rate. When an application uses this flag, the driver should buffer the decompressed data but not draw the image.

ICDECOMPRESS_NOTKEYFRAME
Current frame is not a key frame.

ICDECOMPRESS_NULLFRAME
Current frame does not contain data and the decompressed image should be left the same.

ICDECOMPRESS_PREROLL
Current frame precedes the point in the movie where playback starts and, therefore, will not be drawn.

ICDECOMPRESS_UPDATE
Screen is being updated or refreshed.

lpbiInput
Address of a **BITMAPINFOHEADER** structure containing the input format.

lpInput
Address of a buffer containing the input data.

lpbiOutput
Address of a **BITMAPINFOHEADER** structure containing the output format.

lpOutput
Address of a buffer where the driver should write the decompressed image.

ckid
Chunk identifier from the AVI file.

ICDECOMPRESSEX

```
typedef struct {
    DWORD              dwFlags;
    LPBITMAPINFOHEADER lpbiSrc;
    LPVOID             lpSrc;
    LPBITMAPINFOHEADER lpbiDst;
    LPVOID             lpDst;
    int                xDst;
    int                yDst;
    int                dxDst;
    int                dyDst;
    int                xSrc;
    int                ySrc;
    int                dxSrc;
    int                dySrc;
} ICDECOMPRESSEX;
```

Contains decompression parameters used with the ICM_DECOMPRESSEX message

dwFlags
Applicable flags. The following values are defined:

ICDECOMPRESS_HURRYUP
Tries to decompress at a faster rate. When an application uses this flag, the driver should buffer the decompressed data but not draw the image.

ICDECOMPRESS_NOTKEYFRAME
Current frame is not a key frame.

ICDECOMPRESS_NULLFRAME
Current frame does not contain data and the decompressed image should be left the same.

ICDECOMPRESS_PREROLL
Current frame precedes the point in the movie where playback starts and, therefore, will not be drawn.

ICDECOMPRESS_UPDATE
Screen is being updated or refreshed.

lpbiSrc
Address of a **BITMAPINFOHEADER** structure containing the input format.

lpSrc
Address of a buffer containing the input data.

lpbiDst
Address of a **BITMAPINFOHEADER** structure containing the output format.

lpDst
Address of a buffer where the driver should write the decompressed image.

xDst, **yDst**
The x- and y-coordinates of the destination rectangle within the DIB specified by **lpbiDst**.

dxDst, **dyDst**
Width and height of the destination rectangle.

xSrc, **ySrc**
The x- and y- coordinates of the source rectangle within the DIB specified by **lpbiSrc**.

dxSrc, **dySrc**
Width and height of the source rectangle.

ICDRAW

```
typedef struct {
    DWORD  dwFlags;   // see below
    LPVOID lpFormat;  // see below
    LPVOID lpData;    // address of the data to render
    DWORD  cbData;    // number of bytes of data to render
    LONG   lTime;     // see below
} ICDRAW;
```

Contains parameters for drawing video data to the screen. This structure is used with the ICM_DRAW message.

dwFlags

Flags from the AVI file index. The following values are defined:

ICDRAW_HURRYUP
Data is buffered and not drawn to the screen. Use this flag for fastest decompression.

ICDRAW_NOTKEYFRAME
Current frame is not a key frame.

ICDRAW_NULLFRAME
Current frame does not contain any data, and the previous frame should be redrawn.

ICDRAW_PREROLL
Current frame of video occurs before playback should start. For example, if playback will begin on frame 10, and frame 0 is the nearest previous key frame, frames 0 through 9 are sent to the driver with this flag set. The driver needs this data to display frame 10 properly.

ICDRAW_UPDATE
Updates the screen based on data previously received. In this case, **lpData** should be ignored.

lpFormat

Address of a structure containing the data format. For video streams, this is a **BITMAPINFOHEADER** structure.

lTime

Time, in samples, when this data should be drawn. For video data this is normally a frame number.

ICDRAWBEGIN

```
typedef struct {
    DWORD              dwFlags;
    HPALETTE           hpal;
    HWND               hwnd;
    HDC                hdc;
    int                xDst;
    int                yDst;
    int                dxDst;
    int                dyDst;
    LPBITMAPINFOHEADER lpbi;
    int                xSrc;
    int                ySrc;
    int                dxSrc;
    int                dySrc;
    DWORD              dwRate;
    DWORD              dwScale;
} ICDRAWBEGIN;
```

Contains decompression parameters used with the ICM_DRAW_BEGIN message.

dwFlags
Applicable flags. The following values are defined:

ICDRAW_ANIMATE
Application can animate the palette.

ICDRAW_BUFFER
Buffers this data off-screen; it will need to be updated.

ICDRAW_CONTINUE
Drawing is a continuation of the previous frame.

ICDRAW_FULLSCREEN
Draws the decompressed data on the full screen.

ICDRAW_HDC
Draws the decompressed data to a window or a DC.

ICDRAW_MEMORYDC
DC is off-screen.

ICDRAW_QUERY
Determines if the decompressor can handle the decompression. The driver does not actually decompress the data.

ICDRAW_RENDER
Renders but does not draw the data.

ICDRAW_UPDATING
Current frame is being updated rather than played.

hpal
Handle of the palette used for drawing.

hwnd
Handle of the window used for drawing.

hdc
Handle of the DC used for drawing. Specify NULL to use a DC associated with the specified window.

xDst, **yDst**
The x- and y-coordinates of the destination rectangle.

dxDst, **dyDst**
Width and height of the destination rectangle.

lpbi
Address of a **BITMAPINFOHEADER** structure containing the input format.

xSrc, **ySrc**
The x- and y-coordinates of the source rectangle.

dxSrc, **dySrc**
Width and height of the source rectangle.

dwRate
Decompression rate in an integer format. To obtain the rate in frames per second, divide this value by the value in **dwScale**.

dwScale
Value used to scale **dwRate** to frames per second.

ICDRAWSUGGEST

```
typedef struct {
    LPBITMAPINFOHEADER lpbiIn;           // see below
    LPBITMAPINFOHEADER lpbiSuggest;      // see below
    int                dxSrc;            // see below
    int                dySrc;            // see below
    int                dxDst;            // see below
    int                dyDst;            // see below
    HIC                hicDecompressor;  // see below
} ICDRAWSUGGEST;
```

Contains compression parameters used with the ICM_DRAW_SUGGESTFORMAT message to suggest an appropriate input format.

lpbiIn
Address of the structure containing the compressed input format.

lpbiSuggest
Address of a buffer to return a compatible input format for the renderer.

dxSrc, **dySrc**
Width and height of the source rectangle.

dxDst, **dyDst**
Width and height of the destination rectangle.

hicDecompressor
Handle of a decompressor that supports the format of data described in **lpbiIn**.

ICINFO

```
typedef struct {
    DWORD dwSize;              // see below
    DWORD fccType;             // see below
    DWORD fccHandler;          // see below
    DWORD dwFlags;             // see below
    DWORD dwVersion;           // version member of driver
    DWORD dwVersionICM;        // see below
    WCHAR szName[16];          // see below
    WCHAR szDescription[128];  // see below
    WCHAR szDriver[128];       // see below
} ICINFO;
```

Contains compression parameters supplied by a video compression driver. The driver fills or updates the structure when it receives the ICM_GETINFO message.

dwSize
Size, in bytes, of the **ICINFO** structure.

fccType
Four-character code indicating the type of stream being compressed or decompressed. Specify "VIDC" for video streams.

fccHandler
A four-character code identifying a specific compressor.

dwFlags
Applicable flags. Zero or more of the following flags can be set:

VIDCF_COMPRESSFRAMES
Driver is requesting to compress all frames. For information about compressing all frames, see the ICM_COMPRESS_FRAMES_INFO message.

VIDCF_CRUNCH
Driver supports compressing to a frame size.

VIDCF_DRAW
Driver supports drawing.

VIDCF_FASTTEMPORALC
Driver can perform temporal compression and maintains its own copy of the current frame. When compressing a stream of frame data, the driver doesn't need image data from the previous frame.

VIDCF_FASTTEMPORALD
Driver can perform temporal decompression and maintains its own copy of the current frame. When decompressing a stream of frame data, the driver doesn't need image data from the previous frame.

VIDCF_QUALITY
Driver supports quality values.

VIDCF_TEMPORAL
Driver supports inter-frame compression.

dwVersionICM
Version of VCM supported by the driver. This member should be set to ICVERSION.

szName
Short version of the compressor name. The name in the null-terminated string should be suitable for use in list boxes.

szDescription
Long version of the compressor name.

szDriver

Name of the module containing VCM compression driver. Normally, a driver does not need to fill this out.

ICOPEN

```
typedef struct {
    DWORD    dwSize;       // size, in bytes, of the structure
    DWORD    fccType;      // see below
    DWORD    fccHandler;   // see below
    DWORD    dwVersion;    // see below
    DWORD    dwFlags;      // see below
    LPRESULT dwError;      // error return values
    LPVOID   pV1Reserved;  // reserved; do not use
    LPVOID   pV2Reserved;  // reserved; do not use
    DWORD    dnDevNode;    // device node for Plug and Play devices
} ICOPEN;
```

Contains information about the data stream being compressed or decompressed, the version number of the driver, and how the driver is used.

fccType
Four-character code indicating the type of stream being compressed or decompressed. Specify "VIDC" for video streams.

fccHandler
Four-character code identifying a specific compressor.

dwVersion
Version of the installable driver interface used to open the driver.

dwFlags
Applicable flags indicating why the driver is opened. The following values are defined:

ICMODE_COMPRESS
Driver is opened to compress data.

ICMODE_DECOMPRESS
Driver is opened to decompress data.

ICMODE_DRAW
Device driver is opened to decompress data directly to hardware.

ICMODE_QUERY
Driver is opened for informational purposes, rather than for compression.

This structure is passed to video capture drivers when they are opened. This allows a single installable driver to function as either an installable compressor or a video capture device. By examining the **fccType** member of the **ICOPEN** structure, the driver can determine its function. For example, a **fccType** value of "VIDC" indicates that it is opened as an installable video compressor.

ICSETSTATUSPROC

```
typedef struct {
    DWORD  dwFlags;                     // reserved; set to zero
    LPARAM lParam;                      // see below
    LONG   (CALLBACK * ()) fpfnStatus;  // see below
} ICSETSTATUSPROC;
```

Contains status information used with the ICM_SET_STATUS_PROC message.

lParam
Parameter that contains a constant to pass to the status procedure.

fpfnStatus
Address of the status function. Specify NULL if status messages should not be sent. For more information about the callback function, see the **MyStatusProc** function.

Custom File and Stream Handlers

File and stream handlers are drivers that provide consistent interfaces to an application controlling multimedia data. The file and stream handlers included in the Microsoft Windows operating system access video and waveform-audio data stored in audio-video interleaved (AVI) and waveform-audio files.

You can write handlers to allow your application to write or access multimedia data from another source, such as a file using a proprietary format, an AVI file that has been expanded to contain additional data streams, or a handler that generates its own multimedia data. If you have a custom file format for AVI data that you would like to use with the AVIFile functions provided with the Microsoft Win32 application programming interface (API), you need to write a custom handler.

Your application can use a custom file handler to read from a file or write to a file that is in a nonstandard format. To do this, your application simply uses the name of your file handler when opening the file or allocating the file interface. The AVIFile library then uses the functions from your file handler instead of those from another file handler. The nonstandard format appears as standard AVI data to your application or to any other application using your custom file handler.

Similarly, your application can use a custom stream handler to read a stream that is in a nonstandard format. A stream − whether it constitutes audio, video, MIDI (Musical Instrument Digital Interface), text, or custom data − is a component of an AVI file. For example, an AVI file that contains a video sequence, an English soundtrack, and a French soundtrack consists of three streams. Your application can specify the streams in an AVI file to process and direct each of those streams to a handler that can optimally process the appropriate type of multimedia data.

Note You must place custom stream and file handlers in one or more DLLs, separated from main application files.

Handler Architecture

The internal function of a file or stream handler is defined by the handler itself. To an application, a file handler typically appears as a module to read and write AVI files. Similarly, a stream handler appears as a module to read and write a specific type of data stream. The consistent stream interface makes the source and destination of the stream unimportant to the application that uses the handler.

A file handler provides access to a data source consisting of one or more data streams. File handlers typically provide access to disk files containing one or more data streams and the internal functions of the file handler read and write the multimedia data; however, file handlers can work with any data source, such as a digital transmission channel containing several intermingled data streams.

In contrast, a stream handler processes one type of data and appears as a data stream to an application. A stream handler can provide data that it manufactures or it can retrieve data from a file or an external source. It supplies its data in a format that your application can use.

C++ and OLE Programming Concepts

The file and stream handlers included with Windows use an object-oriented design to promote a standard interface and to share functionality. These handlers are written in C++ and use the OLE Component Object Model.

You can develop custom handlers using the C or C++ development systems; however, C++ is recommended primarily because it provides an easier and more straightforward approach to implement a handler. Using C++, you can explicitly define data as objects and you can associate the functions that manipulate the data as member functions of an object.

This section identifies and briefly summarizes the important concepts of C++ and the OLE Component Object Model that apply to designing and implementing file and stream handlers.

Classes, Objects, Methods, and Interfaces

For file and stream handlers, an *object* can be considered as simply a pointer to a structure. For example, a pointer to the **RECT** data type can be considered an object. A *method* is an object with a function associated with it. A *class* is a composite group of member objects, functions to manipulate these members, and (optionally) access-control specifications to member objects and functions. An *interface* is a set of related methods.

The Scope Resolution Operator in C++

Two colons (::) are used in C+ + as a *scope resolution* operator. This operator gives you more freedom in naming your variables by letting you distinguish between variables with the same name. For example, **MyFile::Read** refers to the **Read** method of the **MyFile** class of objects, as opposed to **YourFile::Read**, which refers to a **Read** method in the **YourFile** class.

Virtual Function Tables

A virtual function table (Vtbl) is an array of pointers to the methods an object supports. If you're using C, an object appears as a structure whose first member is a long pointer to the Vtbl (*lpVtbl*); that is, the first member points to a structure containing the function pointers. Thus, the following example calls the **Read** method of a *pStream* object:

```
pStream->lpVtbl->Read(pStream, parameters)
```

In C+ +, the pointer to Vtbl is implicit, and C+ + automatically passes the object itself as a first parameter. (Objects can obtain this information from the *this* parameter that is implicitly defined.) The following is equivalent to the previous example when using C+ +:

```
pStream->Read(parameters)
```

The OLE Component Object Model

The objects used by the AVIFile library are all part of the OLE Component Object Model. Primarily, this means that they share certain methods that make them easier to work with, and the values they return are common to most OLE interface methods.

The OLE Component Object Model of the file and stream handlers uses the OLE **IClassFactory** interface to create instances of their object class. As component objects, they implement the **QueryInterface**, **Release**, and **AddRef** methods defined by the **IUnknown** interface. The **IUnknown** interface lets an application obtain pointers to other interfaces supported by the same object.

You can determine if an object supports a specific interface by using the **QueryInterface** method. If an object supports a specified interface, **QueryInterface** returns a pointer to that interface.

You can increment and decrement the reference count associated with an object by using the **AddRef** and **Release** methods. The reference count lets multiple clients access an object. When an object is used by the first application, its reference count is set to 1. Applications subsequently use the **AddRef** method to increment the count to let the object keep track of how many times it is being accessed.

When an application is done using an object, it calls the **Release** method to decrement the reference count. When the reference count is zero, the object is no longer needed and **Release** frees any resources it uses and destroys the object. Because an object uses internal resources transparent to the application, the object is responsible for freeing them. For example, a file handler might need to close open disk files and free buffer memory when released.

Most OLE interface methods and API functions return result handles that are defined by using the **HRESULT** data type. This data type is made of a severity code, contextual information, a facility code, and a status code. A return result handle that indicates success has the value zero. A nonzero value indicates failure and the status code member of the return result handle provides a basis for additional interpretation. For additional information about OLE return result handles, see the *Microsoft OLE 2 Programmer's Reference*.

IAVIStream and IAVIFile Interfaces

The **IAVIStream** and **IAVIFile** interfaces contain the methods used by file and stream handlers. These interfaces use the **PAVISTREAM** and **PAVIFILE** data types as object pointers. From a C point of view, an object pointer like **PAVISTREAM** or **PAVIFILE** is a pointer to a structure whose first member is a pointer to a Vtbl table of functions. Other members of the structure retain data used by the interfaces.

For example, to create an object pointer, first allocate space for a structure large enough to contain the pointer to the Vtbl and any other members you want in the structure. Then, create a function table with the read, write, and other functions to operate on your type of stream, and then set the first member of the object pointer to point at the table.

File and Stream Handler Installation

The AVIFile library uses installed stream and file handlers for reading and writing AVI files and streams. A handler is installed when it resides in the Windows SYSTEM directory and the registry contains the following information needed to describe and identify a handler:

- The 16-byte class identifier for the handler
- A brief description of the handler
- The name of the file containing the handler
- The file extension that a file handler can process
- File-access and non-RGB properties associated with a file handler
- Four-character codes identifying stream types that a stream handler can process

The AVIFile library queries the registry for handlers that are external to an application when opening files and accessing streams. The result of a successful query returns the filename of a handler that can process the file or stream type specified in the query. The registry lists each handler by creating three entries that have the following form:

```
[HKEY_CLASSES_ROOT\Clsid\{00010023-0000-0000-C000-000000000046}]
@="Wave File reader/writer"
[HKEY_CLASSES_ROOT\Clsid\{00010023-0000-0000-C000-
000000000046}\InprocServer32]
@="wavefile.dll"
"ThreadingModel"="Apartment"
[HKEY_CLASSES_ROOT\Clsid\{00010023-0000-0000-C000-
000000000046}\AVIFile]
@="3"
```

These entries consist of the following parts:

Part	Description
HKEY_CLASSES_ROOT	Identifies the root entry of the registry.
Clsid	Identifies this entry as a class identifier.
{00010023-0000-0000-C000-000000000046}	Specifies an interface identifier (IID) or class identifier. This value is a unique 16-byte identifier. (The identifier might also be referred to as a GUID or a UUID in other manuals.)
Wave File reader/writer	Specifies a string to describe the handler. This string can be displayed in dialog boxes for selecting stream and file handlers.
InProcServer32	Specifies the file (in this example, WAVEFILE.DLL) that can be loaded to handle

	this class.
AVIFile	Specifies the properties of a file handler. In this example, the handler can read and write to an AVI file.

A file handler can have one or more of its properties stored in the registry. The following constants identify the properties currently associated with a file:

Constant	Description
AVIFILEHANDLER_CANACCEPTNONRGB	Indicates that a file handler can process non-RGB image data.
AVIFILEHANDLER_CANREAD	Indicates that a file handler can open a file with read access.
AVIFILEHANDLER_CANWRITE	Indicates that a file handler can open a file with write access.

When creating a file or stream handler, you can obtain a new identifier by running UUIDGEN.EXE. Always use UUIDGEN.EXE to create a new identifier. The 16-byte hexadecimal number created by this executable will uniquely identify your handler.

The AVIFile library uses additional entries in the registry to identify a class identifier based on the file extension that a file handler can process or a four-character code that a file or stream handler can process. For example, the following entries associate a class identifier with the file extension .WAV and the four-character code "WAVE":

```
[HKEY_CLASSES_ROOT\AVIFile\Extensions\WAV]
@="{00010023-0000-0000-C000-000000000046}"
[HKEY_CLASSES_ROOT\AVIFile\RIFFHandlers\WAVE]
@="{00010023-0000-0000-C000-000000000046}"
```

These entries consist of the following parts:

Part	Description
HKEY_CLASSES_ROOT	Identifies the root entry of the registry.
AVIFile	Identifies this entry as an entry used by AVIFile.
Extensions	Specifies the file extension (in this example, .WAV) that a file handler can process.
RIFFHandlers	Specifies the four-character code (in this example, "WAVE") a file or stream handler can process.
{00010023-0000-0000-C000-000000000046}	Specifies an interface identifier (IID) or class identifier.

If you distribute your stream or file handler without a setup application to install it in the user's system, you must include a .REG file so the user can install the handler. The user will use the registry editor to create registry entries for your stream or file handler.

The following example shows the contents of a typical .REG file. The first entry in the following example holds the descriptive string for the handler. The second entry identifies the file containing the handler. The third entry identifies the properties of the file handler (in this case, read-only access to files). The fourth entry associates the type of file the handler processes (in this case, files with a .JPG filename extension) with the class identifier.

```
[HKEY_CLASSES_ROOT\Clsid\{5C2B8200-E2C8-1068-B1CA-6066188C6002}]
@="JFIF (JPEG) Files"
[HKEY_CLASSES_ROOT\Clsid\{5C2B8200-E2C8-1068-B1CA-6066188C6002}]
\InprocServer32]
@="jfiffile.dll"
[HKEY_CLASSES_ROOT\AVIFile\Extensions\JPG]
@="{5C2B8200-E2C8-1068-B1CA-6066188C6002}"
```

When creating this file, save it with a .REG extension to identify it as an update file for the registry. Also, substitute a unique IID for the 16-byte code used in the example.

Users can update the registry on their system by using the following procedure:

1. From the Program Manager File menu, choose the Run command.
2. In the Run dialog box, type the following command and press ENTER:
   regedit -s *filename*.reg

Using Custom File and Stream Handlers

This section contains examples demonstrating how to perform the following tasks:

- Create a file or stream handler.
- Create a virtual function table for a stream handler.
- Create an object pointer.
- Obtain the address of a virtual function table.
- Create a file-handler instance in a dynamic-link library (DLL).
- Determine which interface an object supports.
- Increment the handler reference counter.
- Delete an object.

Creating a File or Stream Handler

In C, a file or stream handler usually creates a function for each method. Your application accesses these functions through an array of function pointers the stream handler creates. An **IAVIStreamVtbl** structure contains the array of function pointers. A stream handler can assign any name it wants to functions it creates for the methods. The position of the function pointer in the structure implies the correspondence of the function to the method. For example, the first function pointer in the structure corresponds to the **QueryInterface** method. Your stream handler must provide all the functions of an interface.

Creating a Virtual Function Table for a Stream Handler

The following example (written in C) shows how an application (AVIBall) creates the Vtbl used to reference its services.

```
HRESULT STDMETHODCALLTYPE AVIBallQueryInterface (PAVISTREAM ps,
     REFIID riid, LPVOID FAR* ppvObj);
HRESULT STDMETHODCALLTYPE AVIBallCreate (PAVISTREAM ps,
     LONG lParam1, LONG lParam2);
ULONG STDMETHODCALLTYPE AVIBallAddRef (PAVISTREAM ps);
ULONG STDMETHODCALLTYPE AVIBallRelease (PAVISTREAM ps);
HRESULT STDMETHODCALLTYPE AVIBallInfo (PAVISTREAM ps,
     AVIStreamHeader FAR * psi, LONG lSize);
LONG STDMETHODCALLTYPE AVIBallFindSample (PAVISTREAM ps,
     LONG lPos, LONG lFlags);
HRESULT STDMETHODCALLTYPE AVIBallReadFormat (PAVISTREAM ps,
     LONG lPos, LPVOID lpFormat, LONG FAR *lpcbFormat);
HRESULT STDMETHODCALLTYPE AVIBallSetFormat (PAVISTREAM ps,
     LONG lPos, LPVOID lpFormat, LONG cbFormat);
HRESULT STDMETHODCALLTYPE AVIBallRead (PAVISTREAM ps,
     LONG lStart, LONG lSamples, LPVOID lpBuffer, LONG cbBuffer,
     LONG FAR * plBytes,LONG FAR * plSamples);
HRESULT STDMETHODCALLTYPE AVIBallWrite (PAVISTREAM ps, LONG lStart,
     LONG lSamples, LPVOID lpBuffer, LONG cbBuffer, DWORD dwFlags);
HRESULT STDMETHODCALLTYPE AVIBallDelete (PAVISTREAM ps,
     LONG lStart, LONG lSamples);
HRESULT STDMETHODCALLTYPE AVIBallReadData (PAVISTREAM ps,
     DWORD fcc, LPVOID lp,LONG FAR *lpcb);
HRESULT STDMETHODCALLTYPE AVIBallWriteData (PAVISTREAM ps,
     DWORD fcc, LPVOID lp,LONG cb);

IAVIStreamVtbl AVIBallHandler = {
    AVIBallQueryInterface,  // Function pointer for ::QueryInterface
    AVIBallAddRef,          // Function pointer for ::AddRef
    AVIBallRelease,         // Function pointer for ::Release
    AVIBallCreate,          // Function pointer for ::Create
    AVIBallInfo,            // Function pointer for ::Info
    AVIBallFindSample,      // Function pointer for ::FindSample
    AVIBallReadFormat,      // Function pointer for ::ReadFormat
    AVIBallSetFormat,       // Function pointer for ::SetFormat
    AVIBallRead,            // Function pointer for ::Read
    AVIBallWrite,           // Function pointer for ::Write
    AVIBallDelete,          // Function pointer for ::Delete
    AVIBallReadData,        // Function pointer for ::ReadData
    AVIBallWriteData        // Function pointer for ::WriteData
};
```

File handlers use a similar procedure, except they use a different definition for the Vtbl.

Creating an Object Pointer

AVIBall uses the following structure as its object pointer. The first member of this structure points to the Vtbl AVIBall uses to access its functions. Applications can cast this structure to the PAVISTREAM data type. Methods that use the PAVISTREAM data type use only the pointer to the Vtbl. The members following the pointer to the Vtbl are used internally by AVIBall.

```
typedef struct {
    IAVIStreamVtbl FAR * lpvtbl;

    // Ball instance data.
    ULONG    ulRefCount;
    DWORD    fccType;  // is this audio/video?
    int             width;    // size, in pixels, of each frame
    int             height;
    int             length;   // length, in frames
    int             size;
    COLORREF      color;    // ball color
} AVIBALL, FAR * PAVIBALL;
```

Obtaining the Address of a Virtual Function Table

Applications written in C can retrieve the address of the **IAVIStreamVtbl** structure by using the NewBall function. This function returns the address of a structure containing a pointer to **IAVIStreamVtbl**. Information following the **IAVIStreamVtbl** pointer specifies data used internally by AVIBall. Your stream handler can append its own information after the **IAVIStreamVtbl** pointer. This information is returned in subsequent calls to your stream handler.

```
PAVISTREAM FAR PASCAL NewBall(VOID)
{
    PAVIBALL pball;
    pball = (PAVIBALL) GlobalAllocPtr(GHND, sizeof(AVIBALL));
    if (!pball)
           return 0;
    pball->lpvtbl = &AVIBallHandler;
    pball->lpvtbl->Create((PAVISTREAM) pball, 0, 0);
    return (PAVISTREAM) pball;
}
```

Creating a File-Handler Instance in a DLL

When an application specifies your file-handler DLL or stream handler, it is looked up in the registry by its class identifier and loaded. The system then calls the **DllGetClassObject** function of the DLL to create an instance of the file or stream handler. The following example (written in C++) shows how a file handler creates an instance:

```
// Main DLL entry point.
STDAPI DllGetClassObject(const CLSID FAR& rclsid,
    const IID FAR& riid, void FAR* FAR* ppv)
{
    HRESULT hresult;
     hresult = CAVIFileCF::Create(rclsid, riid, ppv);
     return hresult;
}
HRESULT CAVIFileCF::Create(const CLSID FAR&    rclsid,
     const IID FAR& riid, void FAR* FAR* ppv)
{
// The following is the class factory creation and not an
// actual PAVIFile.
     CAVIFileCF FAR*   pAVIFileCF;
     IUnknown FAR*     pUnknown;
     HRESULT hresult;

// Create the instance.
     pAVIFileCF = new FAR CAVIFileCF(rclsid, &pUnknown);
     if (pAVIFileCF == NULL)
           return ResultFromScode(E_OUTOFMEMORY);

// Set the interface pointer.
     hresult = pUnknown->QueryInterface(riid, ppv);
     if (FAILED(GetScode(hresult)))
           delete pAVIFileCF;
     return hresult;
}
```

Determining Which Interface an Object Supports

The **QueryInterface** method lets an application query an object to determine which interfaces it supports. The sample application sets the *ppv* pointer to the current interface.

```
STDMETHODIMP CAVIFileCF::QueryInterface(
     const IID FAR&    iid,
     void FAR* FAR*    ppv)
{
     if (iid == IID_IUnknown)
           *ppv = this;                    // set the interface pointer
                                          // to this instance
     else if (iid == IID_IClassFactory)
           *ppv = this;                    // second chance to set the
                                          // interface pointer to this
                                          // instance

     else
           return ResultFromScode(E_NOINTERFACE);
     AddRef();  //Increment the reference count
     return NULL;
}
```

Incrementing the Handler Reference Count

The **AddRef** method increments the stream- or file-handler reference count.

```
STDMETHODIMP_(ULONG) CAVIFileCF::AddRef()
{
     return ++m_refs;
}
```

Deleting an Object

The **Release** method deletes the object when its reference count is zero.

```
STDMETHODIMP_(ULONG) CAVIFileCF::Release()
{
     if (!--m_refs) {
           delete this;   // if 0, delete this instance
           return 0;
     }
     return m_refs;
}
```

Custom File and Stream Handler Reference

This section describes the functions, interfaces, and member functions that make up the standard AVI file and stream handlers. These elements are grouped as follows:

Handler Entry Point

DllGetClassObject

AVI File Interface

IAVIFile
IAVIFile::CreateStream
IAVIFile::EndRecord
IAVIFile::GetStream
IAVIFile::Info
IAVIFile::Open
IAVIFile::WriteData

Stream Interface

IAVIStream
IAVIStream::Create
IAVIStream::Delete
IAVIStream::Info
IAVIStream::FindSample
IAVIStream::Read
IAVIStream::ReadData
IAVIStream::ReadFormat
IAVIStream::SetFormat
IAVIStream::Write
IAVIStream::WriteData

Data Streaming Interface

IAVIStreaming
IGetFrame::Begin
IGetFrame::End

Editable Stream Interface

IAVIEditStream
IAVIEditStream::Clone
IAVIEditStream::Copy
IAVIEditStream::Cut
IAVIEditStream::Paste
IAVIEditStream::SetInfo

Frame Extraction Interface

IGetFrame
IGetFrame::SetFormat
IGetFrame::End
IGetFrame::GetFrame
IGetFrame::SetFormat

IUnknown Interface

IUnknown
IUnknown::QueryInterface
IUnknown::AddRef
IUnknown::Release

Custom File and Stream Handler Function

A file or stream handler needs to export the **DllGetClassObject** function entry point so an application can access the handler.

DllGetClassObject

```
STDAPI DllGetClassObject(const CLSID FAR& rclsid, const IID FAR& riid,
    void FAR* FAR* ppv)
```

Entry point used by C+ + file and stream handlers to create an instance of the handler.

- Returns a handle of an instance of the file or stream handler.

rclsid
Class identifier of the file or stream handler.

riid
Interface identifier of the file or stream handler.

ppv
Address returned for the object of the interface query. If the interface specified in *riid* is not supported by the object, S_FALSE is returned and the *ppvObj* parameter used in the **IUnknown** interface must be set to NULL.

DllGetClassObject is the only export function your DLL needs. The OLE component object DLL calls this function to obtain an instance of the stream- or file-handler interface.

Your file or stream handler should ensure that the system requests the correct class identifier before creating an instance of it.

Custom File and Stream Handler Interfaces and Members

A file or stream handler uses one or more interfaces to define the member functions for manipulating a file or data stream. Each of the interfaces described in this section is based on the OLE Component Object Model. The prefixes of member function names correspond to the interface names they are associated with.

IAVIEditStream

Interface for manipulating and modifying editable streams. Uses **IUnknown::QueryInterface**, **IUnknown::AddRef**, **IUnknown::Release** in addition to the following custom methods:

Clone	Duplicates a stream.
Copy	Copies a stream or a portion of it to a temporary stream.
Cut	Removes a portion of a stream and places it in a temporary stream.
Paste	Copies a stream or a portion of it and places it in another stream.
SetInfo	Changes the characteristics of a stream.

IAVIEditStream::Clone

```
HRESULT STDMETHODCALLTYPE Clone(PAVISTREAM pavi,
    PAVISTREAM FAR *ppResult);
```

Duplicates a stream. Called when an application uses the **EditStreamClone** function.

- Returns the HRESULT defined by OLE.

pavi
Address of the interface to the stream being cloned.

ppResult
Address to contain a pointer to the interface to the new stream.

For handlers written in C++, **Clone** has the following syntax:

```
STDMETHODIMP Clone(PAVISTREAM FAR *ppResult);
```

IAVIEditStream::Copy

```
HRESULT STDMETHODCALLTYPE Copy(PAVISTREAM pavi, LONG FAR *plStart,
    LONG FAR *plLength, PAVISTREAM FAR * ppResult);
```

Copies a stream or a portion of it to a temporary stream. Called when an application uses the **EditStreamCopy** function.

- Returns the HRESULT defined by OLE.

pavi
Address of the interface to the stream to copy.

plStart
Address that contains the starting position of the operation.

plLength
Address that contains the length, in frames, of the operation.

ppResult
Address to contain a pointer to the interface to the new stream.

For handlers written in C++, **Copy** has the following syntax:

```
STDMETHODIMP Copy(LONG FAR *plStart, LONG FAR *plLength,
    PAVISTREAM FAR * ppResult);
```

IAVIEditStream::Cut

```
HRESULT STDMETHODCALLTYPE Cut(PAVISTREAM pavi, LONG FAR *plStart,
    LONG FAR *plLength, PAVISTREAM FAR * ppResult);
```

Removes a portion of a stream and places it in a temporary stream. Called when an application uses the **EditStreamCut** function.

- Returns the HRESULT defined by OLE.

pavi
Address of the interface to the stream to cut.

plStart
Address that contains the starting position of the operation.

plLength
Address that contains the length, in frames, of the operation.

ppResult
Address to contain a pointer to the interface to the new stream.

For handlers written in C++, **Cut** has the following syntax:

```
STDMETHODIMP Cut(LONG FAR *plStart, LONG FAR *plLength,
    PAVISTREAM FAR * ppResult);
```

IAVIEditStream::Paste

```
HRESULT STDMETHODCALLTYPE Paste(PAVISTREAM pavi, LONG FAR *plPos,
    LONG FAR *plLength, PAVISTREAM pstream, LONG lStart,
    LONG lLength);
```

Copies a stream or a portion of it in another stream. Called when an application uses the **EditStreamPaste** function.

- Returns the HRESULT defined by OLE.

pavi
Address of the interface to the stream to receive the pasted data.

plPos
Address that contains the starting position of the operation.

plLength
Address that contains the length, in bytes, of the data to paste from the source stream.

pstream
Address of the interface to the source stream.

lStart
Starting position of the copy operation within the source stream.

lLength
Length, in frames, of the copy operation within the source stream.

For handlers written in C++, **Paste** has the following syntax:

```
STDMETHODIMP Paste(LONG FAR *plPos, LONG FAR *plLength,
    PAVISTREAM pstream, LONG lStart, LONG lLength);
```

IAVIEditStream::SetInfo

```
HRESULT STDMETHODCALLTYPE SetInfo(PAVISTREAM pavi,
    AVISTREAMINFO FAR *lpInfo, LONG cbInfo);
```

Changes the characteristics of a stream. Called when an application uses the **EditStreamSetInfo** function.

- Returns the HRESULT defined by OLE.

pavi
Address of the interface to a stream.

lpInfo
Address of an **AVISTREAMINFO** structure containing the new stream characteristics.

cbInfo
Size, in bytes, of the buffer.

For handlers written in C++, **SetInfo** has the following syntax:

```
STDMETHODIMP SetInfo(AVISTREAMINFO FAR *lpInfo, LONG cbInfo);
```

IAVIFile

Interface for opening and manipulating files and file headers, and creating and obtaining stream interfaces. Uses **IUnknown::QueryInterface**, **IUnknown::AddRef**, and **IUnknown::Release** in addition to the following custom methods:

CreateStream	Creates a stream for writing.
EndRecord	Writes the "REC" chunk in a tightly interleaved AVI file.
GetStream	Opens a stream by accessing it in a file.
Info	Fills and returns an **AVIFILEINFO** structure with information about a file.
Open	Initializes a file handler.
ReadData	Reads file headers data, format data, or nonaudio and nonvideo data.
WriteData	Writes file headers data, format data, or nonaudio and nonvideo data.

IAVIFile::CreateStream

```
HRESULT STDMETHODCALLTYPE CreateStream(PAVIFILE pf,
    PAVISTREAM FAR * ppstream, AVISTREAMINFO FAR * psi);
```

Creates a stream for writing. Called when an application uses the **AVIFileCreateStream** function.

- Returns HRESULT defined by OLE.

pf
Address of the interface to a file.

ppStream
Address to contain a pointer to the interface to the new stream.

psi
Address of an **AVISTREAMINFO** structure defining the stream to create.

For handlers written in C++, **CreateStream** has the following syntax:

```
STDMETHODIMP CreateStream(PAVISTREAM FAR * ppStream,
    AVISTREAMINFO FAR * psi);
```

IAVIFile::EndRecord

```
HRESULT STDMETHODCALLTYPE EndRecord(PAVISTREAM pf);
```

Writes the "REC" chunk in a tightly interleaved AVI file (having a one-to-one interleave factor of audio to video). Called when an application uses the **AVIFileEndRecord** function.

- Returns the HRESULT defined by OLE.

pf
Address of the interface to a file.

This file handler method is typically not used.

For handlers written in C++, **EndRecord** has the following syntax:

```
STDMETHODIMP EndRecord(VOID);
```

IAVIFile::GetStream

```
STDMETHODIMP GetStream(PAVIFILE pf, PAVISTREAM FAR * ppStream,
    DWORD fccType, LONG lParam);
```

Opens a stream by accessing it in a file. Called when an application uses the **AVIFileGetStream** function.

- Returns the HRESULT defined by OLE.

pf
Address of the interface to a file.

ppStream
Address to contain a pointer to the interface to a stream.

fccType
Four-character code indicating the type of stream to locate.

lParam
Stream number.

It is typically easier to implement this method by creating all of the stream objects in advance by using the **IAVIFile::Open** method. Then, this method accesses the interface to the specified stream.

Remember to increment the reference count maintained by the **AddRef** method for the stream when this method is used.

For handlers written in C++, **GetStream** has the following syntax:

```
STDMETHODIMP GetStream(PAVISTREAM FAR * ppStream,
    DWORD fccType, LONG lParam);
```

IAVIFile::Info

```
HRESULT STDMETHODCALLTYPE Info (PAVISTREAM pf,
    AVISTREAMINFO FAR * pfi, LONG lSize);
```

Fills and returns an **AVIFILEINFO** structure with information about a file. Called when an application uses the **AVIFileInfo** function.

- Returns the HRESULT defined by OLE.

pf
Address of the interface to a file.

pfi
Address of an application-defined buffer to contain file information.

lSize
Size, in bytes, of the buffer specified by *pfi*.

If the buffer allocated is too small for the structure, this method should fail the call by returning AVIERR_BUFFERTOOSMALL. Otherwise, it should fill the structure and return its size.

For handlers written in C++, **Info** has the following syntax:

```
STDMETHODIMP Info(AVIFILEINFO FAR * psi, LONG lSize)
```

IAVIFile::Open

```
HRESULT STDMETHODIMP Open(PAVISTREAM pf, LPCSTR szFile, UINT mode);
```

Initializes a file handler. Called when an application uses the **AVIFileOpen** function.

- Returns the HRESULT defined by OLE.

pf
Address to contain a pointer to the interface to a file.

szFile
Address of a null-terminated string that contains the filename.

mode
Flags for the open operation.

This method is always the first method called, regardless of whether your application is reading or writing a file.

For handlers written in C++, **Open** has the following syntax:

```
STDMETHODIMP Open(LPCSTR szFile, UINT mode);
```

IAVIFile::ReadData

```
HRESULT STDMETHODCALLTYPE ReadData (PAVISTREAM ps, DWORD fcc,
    LPVOID lpBuffer, LONG FAR * lpcbBuffer);
```

Reads file headers. Called when an application uses the **AVIFileReadData** function.

- Returns the HRESULT defined by OLE.

ps
Address of the interface to a file.

fcc
Four-character code of the header to read.

lpBuffer
Address of the buffer for the data.

lpcbBuffer
Size, in bytes, of the buffer specified by *lpBuffer*. When this method returns control to the application, the contents of this parameter specifies the amount of data read.

For handlers written in C++, **ReadData** has the following syntax:

```
STDMETHODIMP ReadData(DWORD fcc, LPVOID lp, LONG FAR *lpcb);
```

IAVIFile::WriteData

```
HRESULT STDMETHODCALLTYPE AVIBallWriteData(PAVISTREAM ps, DWORD fcc,
    LPVOID lpBuffer, LONG cbBuffer);
```

Writes file headers. Called when an application uses the **AVIFileWriteData** function.

- Returns the HRESULT defined by OLE.

ps
Address of the interface to a file.

fcc
Four-character code of the header to write.

lpBuffer
Address of the buffer for the data.

cbBuffer
Size, in bytes, of the buffer specified by *lpBuffer*.

For handlers written in C++, **WriteData** has the following syntax:

```
STDMETHODIMP WriteData(DWORD fcc, LPVOID lpBuffer, LONG cbBuffer);
```

IAVIStream

Interface for creating and manipulating data streams within a file. Uses **IUnknown::QueryInterface**, **IUnknown::AddRef**, **IUnknown::Release** in addition to the following custom methods:

Create	Initializes a stream handler that is not associated with any file.
Delete	Deletes data from a stream.
Info	Fills and returns an **AVISTREAMINFO** structure with information about a stream.
FindSample	Obtains the position in a stream of a key frame or a nonempty frame.
Read	Reads data from a stream and copies it to an application-defined buffer.
ReadData	Reads data headers, format data, or nonaudio and nonvideo data. (Use the **Read** method to read audio and video data.)
ReadFormat	Obtains format information from a stream.
SetFormat	Sets format information in a stream.
Write	Writes data to a stream.
WriteData	Writes data headers, format data, or nonaudio and nonvideo data. (Use the **Write** method to write audio and video data.)

IAVIStream::Create

```
HRESULT STDMETHODCALLTYPE Create(PAVISTREAM ps, LONG lParam1,
    LONG lParam2);
```

Initializes a stream handler that is not associated with any file. Called when an application uses the **AVIStreamCreate** function.

- Returns the HRESULT defined by OLE.

ps
Address of the interface to a stream.

lParam1 and *lParam2*
Stream handler-specific data.

For handlers written in C++, **Create** has the following syntax:

```
STDMETHODIMP Create(LONG lParam1, LONG lParam2)
```

IAVIStream::Delete

```
HRESULT STDMETHODCALLTYPE Delete(PAVISTREAM ps, LONG lStart,
    LONG lSamples);
```

Deletes data from a stream.

- Returns the HRESULT defined by OLE.

ps
Address of the interface to a stream.

lStart
Starting sample or frame number to delete.

lSamples
Number of samples to delete.

For handlers written in C++, **Delete** has the following syntax:

```
STDMETHODIMP Delete(LONG lStart, LONG lSamples);
```

IAVIStream::Info

```
HRESULT STDMETHODCALLTYPE Info(PAVISTREAM ps, AVISTREAMINFO FAR * psi,
    LONG lSize);
```

Fills and returns an **AVISTREAMINFO** structure with information about a stream. Called when an application uses the **AVIStreamInfo** function.

- Returns the HRESULT defined by OLE.

ps
Address of the interface to a stream.

psi
Address of an **AVISTREAMINFO** structure to contain stream information.

lSize
Size, in bytes, of the structure specified by *psi*.

If the buffer allocated is too small for the structure, the **Info** method should fail the call by returning AVIERR_BUFFERTOOSMALL. Otherwise, it should fill the structure and return its size.

For handlers written in C++, **Info** has the following syntax:

```
STDMETHODIMP Info(AVIFILEINFO FAR * psi, LONG lSize)
```

IAVIStream::FindSample

```
LONG STDMETHODCALLTYPE FindSample(PAVISTREAM ps, LONG lPos,
    LONG lFlags);
```

Obtains the position in a stream of a key frame or a nonempty frame. Called when an application uses the **AVIStreamFindSample** function.

- Returns the location of the key frame corresponding to the frame specified by the application.

ps
Address of the interface to a stream.

lPos
Position of the sample or frame.

lFlags
Applicable flags. The following values are defined:

FIND_ANY
Searches for a nonempty frame.

FIND_FORMAT
Searches for a format change.

FIND_KEY
Searches for a key frame.

FIND_NEXT
Searches forward through a stream, beginning with the current frame.

FIND_PREV
Searches backward through a stream, beginning with the current frame.

The FIND_ANY, FIND_KEY, and FIND_FORMAT flags are mutually exclusive, as are the FIND_NEXT and FIND_PREV flags. You must specify one value from each group.

If key frames are not significant in your custom format, return the position specified for *lPos*.

For handlers written in C++, **FindSample** has the following syntax:

```
STDMETHODIMP_(LONG) FindSample(LONG lPos, LONG lFlags)
```

IAVIStream::Read

```
HRESULT STDMETHODCALLTYPE Read(PAVISTREAM ps, LONG lStart,
    LONG lSamples, LPVOID lpBuffer, LONG cbBuffer,
    LONG FAR * plBytes, LONG FAR * plSamples);
```

Reads data from a stream and copies it to an application-defined buffer. If no buffer is supplied, it determines the buffer size needed to retrieve the next buffer of data. Called when an application uses the **AVIStreamRead** function.

- Returns AVIERR_OK if successful or AVIERR_BUFFERTOOSMALL if the buffer is not large enough to hold the data. If successful, **Read** also returns either a buffer of data with the number of frames (samples) included in the buffer or the required buffer size, in bytes.

ps
Address of the interface to a stream.

lStart
Starting sample or frame number to read.

lSamples
Number of samples to read.

lpBuffer
Address of the application-defined buffer to contain the stream data. You can also specify NULL to request the required size of the buffer. Many applications precede each read operation with a query for the buffer size to see how large a buffer is needed.

cbBuffer
Size, in bytes, of the buffer specified by *lpBuffer*.

plBytes
Address to contain the number of bytes read.

plSamples
Address to contain the number of samples read.

For handlers written in C++, **Read** has the following syntax:

```
STDMETHODIMP Read(LONG lStart, LONG lSamples,
    LPVOID lpBuffer, LONG cbBuffer,
    LONG FAR * plBytes, LONG FAR * plSamples);
```

IAVIStream::ReadData

```
HRESULT STDMETHODCALLTYPE ReadData(PAVISTREAM ps, DWORD fcc,
    LPVOID lpBuffer, LONG FAR * lpcbBuffer);
```

Reads data headers of a stream. Called when an application uses the **AVIStreamReadData** function.

- Returns the HRESULT defined by OLE.

ps
Address of the interface to a stream.

fcc
Four-character code of the stream header to read.

lpBuffer
Address of the buffer to contain the header data.

lpcbBuffer
Size, in bytes, of the buffer specified by *lpBuffer*. When this method returns control to the application, the contents of this parameter specifies the amount of data read.

For handlers written in C++, **ReadData** has the following syntax:

```
STDMETHODIMP ReadData(DWORD fcc, LPVOID lp, LONG FAR *lpcb);
```

IAVIStream::ReadFormat

```
HRESULT STDMETHODCALLTYPE ReadFormat(PAVISTREAM ps, LONG lPos,
    LPVOID lpFormat, LONG FAR * lpcbFormat);
```

Obtains format information from a stream. Fills and returns a structure with the data in an application-defined buffer. If no buffer is supplied, determines the buffer size needed to retrieve the buffer of format data. Called when an application uses the **AVIStreamReadFormat** function.

- Returns the HRESULT defined by OLE.

ps
Address of the interface to a stream.

lPos
Position of the sample or frame.

lpFormat
Address of the buffer for the format data. Specify NULL to request the required size of the buffer.

lpcbFormat
Address that contains the size, in bytes, of the buffer specified by *lpFormat*. When this method is called, the contents of this parameter indicates the size of the buffer specified by *lpFormat*. When this method returns control to the application, the contents of this parameter specifies the amount of data read or the required size of the buffer.

The type of data stored in a stream dictates the format information and the structure that contains the format information. A stream handler should return all applicable format information in this structure, including palette information when the format uses a palette. A stream handler should not return stream data with the structure.

Standard video stream handlers provide format information in a **BITMAPINFOHEADER** structure. Standard audio stream handlers provide format information in a **PCMWAVEFORMAT** structure. Other data streams can use other structures that describe the stream data.

For handlers written in C++, **ReadFormat** has the following syntax:

```
STDMETHODIMP ReadFormat(LONG lPos, LPVOID lpFormat,
    LONG FAR *lpcbFormat)
```

IAVIStream::SetFormat

```
HRESULT STDMETHODCALLTYPE SetFormat (PAVISTREAM ps, LPVOID lpFormat,
    LONG cbFormat);
```

Sets format information in a stream. Called when an application uses the **AVIStreamSetFormat** function.

- Returns the HRESULT defined by OLE.

ps
Address of the interface to a stream.

lpFormat
Address of the buffer for the format data.

cbFormat
Address containing the size, in bytes, of the buffer specified by *lpFormat*.

Standard video stream handlers provide format information in a **BITMAPINFOHEADER** structure. Standard audio stream handlers provide format information in a **PCMWAVEFORMAT** structure. Other data streams can use other structures that describe the stream data.

For handlers written in C++, **SetFormat** has the following syntax:

```
STDMETHODIMP SetFormat(LONG lPos, LPVOID lpFormat, LONG cbFormat)
```

IAVIStream::Write

```
HRESULT STDMETHODCALLTYPE Write (PAVISTREAM ps, LONG lStart,
    LONG lSamples, LPVOID lpBuffer, LONG cbBuffer, DWORD dwFlags,
    LONG FAR *plSampWritten, LONG FAR *plBytesWritten);
```

Writes data to a stream. Called when an application uses the **AVIStreamWrite** function.

- Returns the HRESULT defined by OLE.

ps
Address of the interface to a stream.

lStart
Starting sample or frame number to write.

lSamples
Number of samples to write.

lpBuffer
Address of the buffer for the data.

cbBuffer
Size, in bytes, of the buffer specified by *lpBuffer*.

dwFlags
Applicable flags. The AVIF_KEYFRAME flag is defined and indicates that this frame contains all the information needed for a complete image.

plSampWritten
Address of a buffer used to contain the number of samples written.

plBytesWritten
Address to contain the number of bytes written.

For handlers written in C++, **Write** has the following syntax:

```
STDMETHODIMP Write(LONG lStart, LONG lSamples, LPVOID lpBuffer,
    LONG cbBuffer, DWORD dwFlags, LONG FAR *plSampWritten,
    LONG FAR *plBytesWritten);
```

IAVIStream::WriteData

```
HRESULT STDMETHODCALLTYPE WriteData (PAVISTREAM ps, DWORD fcc,
    LPVOID lpBuffer, LONG cbBuffer);
```

Writes headers for a stream. Called when an application uses the **AVIStreamWriteData** function.

- Returns the HRESULT defined by OLE.

ps
Address of the interface to a stream.

fcc
Four-character code of the stream header to write.

lpBuffer
Address of the buffer that contains the header data to write.

cbBuffer
Size, in bytes, of the buffer specified by *lpBuffer*.

For handlers written in C++, **WriteData** has the following syntax:

```
STDMETHODIMP WriteData(DWORD fcc, LPVOID lpBuffer, LONG cbBuffer);
```

IAVIStreaming

Interface for preparing open data streams for playback in streaming operations. Uses **IUnknown::QueryInterface**, **IUnknown::AddRef**, **IUnknown::Release** in addition to the **Begin** and **End** custom methods.

IAVIStreaming::Begin

```
STDMETHODCALLTYPE Begin(PAVISTREAM ps, LONG lStart, LONG lEnd,
    LONG lRate);
```

Prepares for the streaming operation. Called when an application uses the **AVIStreamBeginStreaming** function.

- Returns the HRESULT defined by OLE.

ps
Address of the interface to a stream.

lStart
Starting frame for streaming.

lEnd
Ending frame for streaming.

lRate
Speed at which the file is read relative to its normal playback rate. Normal speed is 1000. Larger values indicate faster speeds; smaller values indicate slower speeds.

For handlers written in C++, **Begin** has the following syntax:

```
STDMETHODIMP Begin(LONG lStart, LONG lEnd, LONG lRate);
```

IAVIStreaming::End

```
STDMETHODCALLTYPE End(PAVISTREAM ps);
```

Ends the streaming operation. Called when an application uses the **AVIStreamEndStreaming** function.

- Returns the HRESULT defined by OLE.

ps
Address of the interface to a stream.

For handlers written in C++, **End** has the following syntax:

```
STDMETHODIMP End(VOID);
```

IGetFrame

Interface for extracting, decompressing, and displaying individual frames from an open stream. Uses **IUnknown::QueryInterface**, **IUnknown::AddRef**, **IUnknown::Release** in addition to the following custom methods:

Begin	Prepares to extract and decompress copies of video frames from a stream.
End	Ends frame extraction and decompression.
GetFrame	Retrieves a decompressed copy of a frame from a stream.
SetFormat	Sets the image format of the frames being extracted.

IGetFrame::Begin

```
STDMETHODCALLTYPE Begin(PAVISTREAM ps, LONG lStart, LONG lEnd,
    LONG lRate);
```

Prepares to extract and decompress copies of frames from a stream. Called when an application uses the **AVIStreamGetFrameOpen** function.

- Returns the HRESULT defined by OLE.

ps
Address of the interface to a stream.

lStart
Starting frame for extracting and decompressing.

lEnd
Ending frame for extracting and decompressing.

lRate
Speed at which the file is read relative to its normal playback rate. Normal speed is 1000. Larger values indicate faster speeds; smaller values indicate slower speeds.

For handlers written in C++, **Begin** has the following syntax:

```
STDMETHODIMP Begin(LONG lStart, LONG lEnd, LONG lRate);
```

IGetFrame::End

```
STDMETHODCALLTYPE End(PAVISTREAM ps);
```

Ends frame extraction and decompression. Called when an application uses the **AVIStreamGetFrameClose** function.

- Returns the HRESULT defined by OLE.

ps
Address of the interface to a stream.

For handlers written in C++, **Begin** has the following syntax:

```
STDMETHODIMP End(VOID);
```

IGetFrame::GetFrame

```
LPVOID STDMETHODCALLTYPE GetFrame(PAVISTREAM ps, LONG lPos);
```

Retrieves a decompressed copy of a frame from a stream. Called when an application uses the **AVIStreamGetFrame** function.

- Returns the address of the decompressed frame data.

ps
Address of the interface to a stream.

lPos
Frame to copy and decompress.

For handlers written in C++, **GetFrame** has the following syntax:

```
STDMETHODIMP_(LPVOID) GetFrame(LONG lPos);
```

IGetFrame::SetFormat

```
STDMETHODCALLTYPE SetFormat(PAVISTREAM ps, LPBITMAPINFOHEADER lpbi,
    LPVOID lpBits, int x, int y, int dx, int dy);
```

Sets the decompressed image format of the frames being extracted and optionally provides a buffer for the decompression operation.

- Returns NOERROR if successful, E_OUTOFMEMORY if the decompressed image is larger than the buffer size, or E_FAIL otherwise.

ps
Address of the interface to a stream.

lpbi
Address of a **BITMAPINFOHEADER** structure defining the decompressed image format. You can also specify NULL or the value ((LPBITMAPINFOHEADER) 1) for this parameter. NULL causes the decompressor to choose a format that is appropriate for editing (normally a 24-bit image depth format). The value ((LPBITMAPINFOHEADER) 1) causes the decompressor to choose a format appropriate for the current display mode.

lpBits
Address of a buffer to contain the decompressed image data. Specify NULL to have this method allocate a buffer.

x and *y*
The x- and y-coordinates of the destination rectangle within the DIB specified by *lpbi*. This parameter is used when *lpBits* is not NULL.

dx and *dy*
Width and height of the destination rectangle. These parameters are used when *lpBits* is not NULL.

The *x*, *y*, *dx*, and *dy* parameters identify the portion of the bitmap specified by *lpbi* and *lpBits* that receives the decompressed image.

For handlers written in C++, **SetFormat** has the following syntax:

```
STDMETHODIMP SetFormat(LPBITMAPINFOHEADER lpbi, LPVOID lpBits, int x,
    int y, int dx, int dy);
```

IUnknown

OLE interface from which AVIFile and AVIStream interfaces are derived. Interfaces used with AVI files rely on definitions of the **QueryInterface**, **AddRef**, and **Release** methods from this factory.

IUnknown::QueryInterface

```
HRESULT STDMETHODCALLTYPE QueryInterface(LPUNKNOWN ps,
    const IID FAR& riid, void FAR* ppvObj);
```

Determines if an interface can be used with an object. Used by the following interfaces: **IAVIEditStream**, **IAVIFile**, **IAVIStream**, **IAVIStreaming**, and **IGetFrame**.

- Returns a pointer to the current interface if successful or E_NOINTERFACE otherwise.

ps
Address of an **IAVIEditStream**, **IAVIFile**, **IAVIStream**, **IAVIStreaming**, or **IGetFrame** interface.

riid
Identifier of the interface being queried.

ppvObj
Address to contain a pointer to the object whose interface is queried or NULL when an interface is not supported.

For handlers written in C++, **QueryInterface** has the following syntax:

```
STDMETHODIMP QueryInterface(const IID FAR& riid, void FAR* ppvObj);
```

IUnknown::AddRef

```
ULONG STDMETHODCALLTYPE AddRef(LPUNKNWON ps);
```

Increments the reference count of the appropriate handler: **IAVIEditStream**, **IAVIFile**, **IAVIStream**, **IAVIStreaming**, or **IGetFrame**. When the reference count is nonzero, the handler must retain resources for the file or stream.

- Returns the resulting reference count.

ps
Address of an **IAVIEditStream**, **IAVIFile**, **IAVIStream**, **IAVIStreaming**, or **IGetFrame** interface.

For handlers written in C++, **AddRef** has the following syntax:

```
STDMETHODIMP AddRef(VOID);
```

IUnknown::Release

```
ULONG STDMETHODCALLTYPE Release(LPUNKNOWN ps);
```

Decrements the reference count of the appropriate handler: **IAVIEditStream**, **IAVIFile**, **IAVIStream**, **IAVIStreaming**, or **IGetFrame**. When the reference count reaches zero, the handler must free resources established for the file or stream.

- Returns the resulting reference count.

ps
Address of an **IAVIEditStream**, **IAVIFile**, **IAVIStream**, **IAVIStreaming**, or **IGetFrame** interface.

For handlers written in C++, **Release** has the following syntax:

```
STDMETHODIMP Release(VOID);
```

AVIFile Functions and Macros

AVIFile functions and macros provide access to time-based files that use the Resource Information File Format (RIFF), such as waveform-audio and audio-video interleaved (AVI) files. These functions and macros manage the internals of RIFF files, making it unnecessary for you to manage and navigate through the RIFF architecture.

The AVIFile functions and macros handle the information in time-based files as one or more *data streams* instead of tagged blocks of data called chunks. Data streams refer to the components of a time-based file. An AVI file can contain several different types of data, such as a video sequence, an English soundtrack, and a French soundtrack. Using AVIFile, an application can access each of these components separately.

Note Although the AVIFile functions and macros work with any RIFF file, this chapter demonstrates their use with AVI files only. AVI files are typically the time-based files used with the AVIFile macros and functions.

AVIFile functions and macros are contained in a dynamic-link library. You can initialize the library by using the **AVIFileInit** function. After you initialize the library, you can use any of the AVIFile functions or macros. You can release the library by using the **AVIFileExit** function. AVIFile maintains a reference count of the applications that are using the library, but not those that have released it. Your applications should balance each use of **AVIFileInit** with a call to **AVIFileExit** to completely release the library after each application finishes using it.

Function Data Types and Return Values

The AVIFile functions and macros use file and stream handlers implemented with OLE technology. The standard data type of an OLE function is **STDAPI**, and the function returns an **HRESULT** value (zero for success or an error otherwise). If a function returns a value other than an **HRESULT**, the type of the function's prototype has a slightly different syntax that embeds the return value type in parentheses following **STDAPI_**. For example, a function that returns a **LONG** data value uses **STDAPI_(LONG)** in the prototype statement.

AVIFile Operations

This section describes the AVIFile file input and output (I/O) operations.

Opening and Closing Files

An application must open an AVI file before reading or writing. You can open an AVI file by using the **AVIFileOpen** function. **AVIFileOpen** returns the address of an AVI file interface that contains the handle of the open file and increments the reference count of the file.

AVIFileOpen supports the OF flags used with the **OpenFile** function. If an application writes to an existing file, it must include the OF_WRITE flag in **AVIFileOpen**. Similarly, if your application creates and writes to a new file, you must include the OF_CREATE and OF_WRITE flags in **AVIFileOpen**.

When you open a file using **AVIFileOpen**, you can use a default file handler or you can specify a custom file handler to read and write to the file and its data streams. In either case, AVIFile searches the registry for the correct file handler to use. You must ensure custom file handlers are in the registry before an application can access them.

You can increment the reference count of a file by using the **AVIFileAddRef** function. For example, you might want to do this when passing a handle of the file interface to another application or when you want to keep a file open when using a function that would normally close the file.

You can close a file by using the **AVIFileRelease** function. **AVIFileRelease** decrements the reference count of an AVI file, saves changes made to the file, and when the reference count reaches zero, closes the file. Your applications should balance the reference count by including a call to **AVIFileRelease** for each use of **AVIFileOpen** and **AVIFileAddRef**.

Note An application can open a file with one or more program threads; however, only one thread should access the file at a time for the best possible performance.

Reading from a File

You can retrieve information about an open file by using the **AVIFileInfo** function. This function fills the **AVIFILEINFO** structure with information such as the maximum data rate, the number of streams in the file, whether the file uses an index, and whether the file is copyrighted.

You can retrieve supplementary information in an AVI file by using the **AVIFileReadData** function. Supplementary information is applicable to the entire file and is not included in the normal file headers. For example, the name of the company or person who holds the copyrights of a file could be supplementary information. Supplementary information does not adhere to a specific format; it can be file specific. **AVIFileReadData** returns the supplementary information in an application-supplied buffer.

Writing to a File

You can write supplementary information to a file that has been opened with write privileges by using the **AVIFileWriteData** function. This function copies the information from an application-supplied buffer and places it in one or more chunks in the file. The "INFO" chunk is a common RIFF chunk type in which the function stores supplementary information. For a description of RIFF files and their data chunks, see Chapter 15, "File Input and Output."

Using the Clipboard with AVI Files

The clipboard provides convenient, temporary storage for an application to copy or transfer AVI files. AVIFile includes clipboard functions that you can use with disk or memory files.

You can copy a file to the clipboard by using the **AVIPutFileOnClipboard** function. You can write a file that is on the clipboard to memory or to disk by using the **AVIGetFromClipboard** function.

You can clear a file from the clipboard by using the **AVIClearClipboard** function. This function does not clear other types of information, such as text, from the clipboard. If you use clipboard functions in your application, clear the clipboard with **AVIClearClipboard** before closing the application or closing the file on the clipboard. You can call **AVIClearClipboard** in your application whether or not **AVIPutFileOnClipboard** has been used.

Note If your application copies a file to the clipboard and the file contains stream data that might change, you can create a memory file of cloned streams by using the **AVIMakeFileFromStreams** function. You can then place the file on the clipboard and then release the original file without invalidating it.

Stream Operations

Most of the features of AVIFile focus on data streams. This section describes the functions and macros that deal with streams and stream data.

Opening and Closing Streams

Opening data streams is similar to opening files. You can open a stream by using the **AVIFileGetStream** function. This function creates a stream interface, places a handle of the stream in the interface, and returns a pointer to the interface. **AVIFileGetStream** also maintains a reference count of the applications that have opened a stream, but not of those that have closed it.

If you want to access a single stream in a file, you can open the file and the stream by using the **AVIStreamOpenFromFile** function. This function combines the operations and function arguments of the **AVIFileOpen** and **AVIFileGetStream** functions.

If you want to access more than one stream in a file, you can use **AVIFileOpen** once followed by multiple calls to **AVIFileGetStream**.

You can increment the reference count of a stream by using the **AVIStreamAddRef** function to keep a stream open when using a function that would normally close the stream.

You can close a stream by using the **AVIStreamRelease** function. This function decrements the reference count of the stream and closes it when the reference count reaches zero. Your applications should balance the reference count by including a call to **AVIStreamRelease** for each use of the **AVIFileGetStream**, **AVIFileCreateStream**, **AVIStreamAddRef**, or **AVIStreamOpenFromFile** function. When you release a stream that has been opened by using **AVIStreamOpenFromFile**, AVIFile closes the file containing the stream. If your application releases a file that has open data streams, AVIFile will not close the streams until all of the streams are released.

Reading from a Stream

You can retrieve information about an open stream by using the **AVIStreamInfo** function. This function fills the **AVISTREAMINFO** structure with information such as the type of data in the stream, the compression method used when writing stream data, the suggested buffer size, the playback rate, and a text description of the stream.

Some members of the **AVISTREAMINFO** structure are also present in the **AVIFILEINFO** structure. The information in the **AVIFILEINFO** structure applies to the entire file. The information in the **AVISTREAMINFO** structure is specific to the accessed stream and has precedence over the information in the **AVIFILEINFO** structure.

If a stream has supplementary information associated with it, you can retrieve this information by using the **AVIStreamReadData** function. **AVIStreamReadData** returns the information in an application-supplied buffer. Supplementary stream information might include configuration settings for the compression and decompression methods used on a stream. You can obtain the required buffer size by using the **AVIStreamDataSize** macro.

You can retrieve formatting information about a stream by using the **AVIStreamReadFormat** function. This function returns a stream-specific structure in an application-supplied buffer. For a video stream, the buffer contains formatting information in a **BITMAPINFO** structure. For an audio stream, the buffer contains formatting information in a **WAVEFORMATEX** or **PCMWAVEFORMAT** structure. For other stream types, the stream handler returns information specific to the stream. You can determine the required buffer size by using **AVIStreamReadFormat** and specifying a NULL buffer address or by using the **AVIStreamFormatSize** macro.

You can retrieve the multimedia content in a stream by using the **AVIStreamRead** function. This function copies raw data from the stream into an application-supplied buffer. For video streams, this function retrieves the bitmapped images that make up the frame content. For audio streams, this function retrieves waveform-audio samples that make up the sound content. You can determine the required buffer size by using **AVIStreamRead** and specifying a NULL buffer address or by using the **AVIStreamSampleSize** macro.

Some AVI stream handlers introduce delays associated with software and hardware initialization or coordination. You can inform these handlers to prepare for data streaming by using the **AVIStreamBeginStreaming** function. This function lets the stream handler allocate the resources it needs and initialize them. You can also inform these handlers when streaming has ended by using the **AVIStreamEndStreaming** function. This function lets the stream handler deallocate the resources it allocated for **AVIStreamBeginStreaming**.

AVIStreamRead does not provide decompression services. For information about compressing and decompressing audio streams, see Chapter 12, "Audio Compression Manager." For information about compressing and decompressing video streams, see Chapter 7, "Video Compression Manager." For information about compressing and decompressing individual frames in a video stream, see "Working with Compressed Video Data in a Stream" later in this chapter.

Working with Compressed Video Data in a Stream

AVIFile provides several ways for you to access compressed video streams.

If you want to display one or more frames of a compressed video stream, you can retrieve the frames by using the **AVIStreamRead** function and forwarding the compressed frame data to DrawDib functions to display them. DrawDib functions can display images of several image depths and automatically dither images for displays that cannot handle certain types of device-independent bitmaps (DIBs). These functions work with uncompressed and compressed images. For more information about DrawDib functions, see Chapter 10, "DrawDib Functions ."

AVIFile provides functions for decompressing a single video frame. You can allocate resources by using the **AVIStreamGetFrameOpen** function. This function creates a buffer for the decompressed data.

You can decompress a single video frame by using the **AVIStreamGetFrame** function. This function decompresses the frame and retrieves a complete frame of image data and returns the address of the DIB in the **BITMAPINFOHEADER** structure. Your application can display the DIB by using standard Microsoft Win32 drawing functions or by using the DrawDib functions.

If your application makes successive calls to **AVIStreamGetFrame**, the function overwrites its buffer with each retrieved frame.

When you finish using **AVIStreamGetFrame** and the decompressed DIB is in its buffer, you can release the allocated resources by using the **AVIStreamGetFrameClose** function.

For more information about decompressing images, see Chapter 7, "Video Compression Manager."

Creating a File from Existing Streams

One way to create a file that contains data streams is to combine existing streams into a new file. The streams that provide data for the new file can reside in memory or in one or more files.

You can build a file from several streams by using the **AVISave** function. This function creates a file and writes the data streams specified in its calling sequence to the file. The calling sequence for **AVISave** uses a variable number of arguments that include interfaces for the streams combined in the new file.

You can also combine data streams in a new file by using the **AVISaveV** function. This function provides the same functionality as **AVISave**, but when you use **AVISaveV**, your application specifies the data streams as an array, not as a variable number of arguments.

You can create a dialog box in which the user can select compression settings for the new file by using the **AVISaveOptions** function. The dialog box displays the current compression settings and lets the user edit them. Compression setting changes are stored in an **AVICOMPRESSOPTIONS** structure.

You can also include a callback function with **AVISave** and **AVISaveV** that your application can use to display the progress of writing the file and, if needed, let the user cancel the save operation. You can include the address of the callback function in the calling sequence of **AVISave** or **AVISaveV**.

You can let the user select a filename for the new file by using the **GetSaveFileNamePreview** function. This function displays the Save As dialog box in which the user can preview the first stream (normally the video stream) of an AVI file.

You can create a file interface pointer (and a virtual file) for a group of streams by using the **AVIMakeFileFromStreams** function. Other AVIFile functions can use the file interface pointer returned by this function to access the streams in the virtual file. When you finish using the virtual file, delete the file interface pointer by using the **AVIFileRelease** function.

Note To minimize image and audio degradation, avoid compressing an AVI file more than once. Combine uncompressed pieces of video in your editing system and then compress the final product. For information about compression options, see Chapter 7, "Video Compression Manager."

Writing Streams to a File

You can also create a file containing data streams by writing a new data stream to a file.

You can create a new stream in a new or existing file by using the **AVIFileCreateStream** function. This function defines a new stream according to the characteristics described in an **AVISTREAMINFO** structure, creates a stream interface for the new stream, increments the reference count of the stream, and returns the address of the stream-interface pointer.

Before you write the content of the stream, you must specify the stream format. You can set the stream format by using the **AVIStreamSetFormat** function. When setting the format of a video stream, you must supply this function with a **BITMAPINFO** structure containing the appropriate information. When setting the format of an audio stream, you must supply a **WAVEFORMAT** or **WAVEFORMATEX** structure containing the appropriate information. The information you need to supply to the function for other stream types depends on the stream type and the stream handler.

You can write the multimedia content in a stream by using the **AVIStreamWrite** function. This function copies raw data from an application-supplied buffer into the specified stream. The default AVI file handler appends information to the end of a stream. The default WAVE handler can write waveform-audio data within a stream as well as at the end of a stream.

You can write supplementary information about the file or stream that is not included in the **AVIFileCreateStream** or **AVIStreamSetFormat** function by using the **AVIFileWriteData** and **AVIStreamWriteData** functions. You can record data that is applicable to the entire file, such as copyright information and modification history, by using **AVIFileWriteData**. You can record stream-specific information, such as compression and decompression settings, by using **AVIStreamWriteData**. The supplementary information is stored in separate chunks within the file.

You can close the stream after you finish writing to the new stream by using the **AVIStreamRelease** function. This function clears buffers used in recording the stream data, and it completes and closes any incomplete data chunks in the file.

Positioning in Streams

AVIFile provides several ways to locate and move to a position in a data stream. The functions and macros in this section let your application find the starting position, length, and key frames (containing a complete image in the sample) within a stream. The functions and macros also associate time with positions in a stream by calculating the elapsed time needed to play a stream from its beginning to any point in a stream.

Finding the Starting Position

You can retrieve the sample number of the first frame in a video stream by using the **AVIStreamStart** function. (The frames of a movie might start at sample 0 or 1 depending on the preference of the author.) You can also obtain this information by using the **AVIStreamInfo** function. This function stores the sample number in the **dwStart** member of the **AVISTREAMINFO** structure. You can retrieve the starting time of a stream's first sample by using the **AVIStreamStartTime** macro.

You can retrieve the stream length by using the **AVIStreamLength** function. This function returns the number of samples in the stream. You can also obtain this information by using the **AVIStreamInfo** function. This function stores the stream length in the **dwLength** member of the **AVISTREAMINFO** structure. You can retrieve the length of a stream in milliseconds by using the **AVIStreamLengthTime** macro.

In a video stream, each sample generally corresponds to a frame of video. There might, however, be samples for which no video data is present. If you call the **AVIStreamRead** function specifying one of those positions, it returns a data length of 0 bytes. You can find samples that contain data by using the **AVIStreamFindSample** function and specifying the FIND_ANY flag.

In an audio stream, each sample corresponds to one data block of audio data. For example, if the audio data has a 22 kHz ADPCM (Adaptive Differential Pulse Code Modulation) format, each sample for **AVIStreamLength** corresponds to a block of 256 bytes of compressed audio data. This block of audio data contains approximately 500 audio samples when uncompressed. The functions and macros of AVIFile, however, treat each 256-byte block as a single sample.

Note Valid positions within a stream range from the beginning to the end of the stream, which is the sum of the stream starting point and its length. The position represented by the sum of the starting position and the length corresponds to a time after the last data has been rendered; it does not contain any data. You can retrieve the sample number that represents the end of the stream by using the **AVIStreamEnd** macro. You can retrieve the time value in milliseconds that represents the end of the stream by using the **AVIStreamEndTime** macro.

Finding Sample and Key Frames

You can search for different types of samples in a stream by using the **AVIStreamFindSample** function. This function searches backward or forward through a stream for a sample of the appropriate type, beginning with the sample number you specify. You can search for different types of samples in a stream by specifying a flag in the **AVIStreamFindSample** calling sequence. Specify the FIND_ANY flag to locate nonempty samples or to skip samples that lack data. Specify the FIND_KEY flag to search for key frames that contain the data to render a complete image without needing to reference previous frames. Specify the FIND_FORMAT flag to search for changes to the format. **AVIStreamFindSample** is used mainly with video streams.

Several macros that use AVIFile functions supplement the stream search features. The following list provides a brief description of each macro. The macros that search for a specific position or type of data require a starting location to be specified in the stream.

Macro	Description
AVIStreamIsKeyFrame	Indicates whether a sample in a

	specified stream is a key frame.
AVIStreamNearestKeyFrame	Locates the key frame at or preceding a specified position in a stream.
AVIStreamNearestKeyFrameTime	Determines the time corresponding to the beginning of the key frame nearest (at or preceding) a specified time in a stream.
AVIStreamNearestSample	Locates the nearest nonempty sample at or preceding a specified position in a stream.
AVIStreamNearestSampleTime	Determines the time corresponding to the beginning of a sample that is nearest to a specified time in a stream.
AVIStreamNextKeyFrame	Locates the next key frame following a specified position in a stream.
AVIStreamNextKeyFrameTime	Returns the time of the next key frame in a stream, starting at a given time.
AVIStreamNextSample	Locates the next nonempty sample from a specified position in a stream.
AVIStreamNextSampleTime	Returns the time that a sample changes to the next sample in the stream. This macro finds the next interesting time in a stream.
AVIStreamPrevKeyFrame	Locates the key frame that precedes a specified position in a stream.
AVIStreamPrevKeyFrameTime	Returns the time of the previous key frame in the stream, starting at a given time.
AVIStreamPrevSample	Locates the nonempty sample that precedes a specified position in a stream.
AVIStreamPrevSampleTime	Determines the time that the previous sample replaces its predecessor in the stream.
AVIStreamSampleToSample	Returns the sample in a stream that occurs at the same time as a sample that occurs in a second stream.

Switching Between Samples and Time

You can determine the elapsed time from the beginning of a stream to a sample using the **AVIStreamSampleToTime** function. This function converts the sample number to a time value expressed in milliseconds. For a video frame (which spans several milliseconds), this value represents the time the sample begins to play since playback began and assumes the video clip plays at normal speed. For an audio sample (which has several samples in a millisecond), the time value corresponds to the time the sample begins to play and assumes the audio stream plays at normal speed.

Conversely, you can find the sample number associated with a time value by using the **AVIStreamTimeToSample** function. This function converts the millisecond value to a sample number and assumes the video clip plays at normal speed.

Because **AVIStreamSampleToTime** returns the time a frame begins to play, **AVIStreamSampleToTime** and **AVIStreamTimeToSample** are not true inverses. They more accurately determine the position in a file than time. For example, two consecutive audio samples might both play in the same millisecond. Using **AVIStreamSampleToTime** to convert the sample numbers would result in identical time values. If you convert the time value back to a sample number by using **AVIStreamTimeToSample**, a single sample would be referenced.

Creating Temporary Streams

Temporary streams can be beneficial in several ways. You can use a temporary stream as a work stream, such as when changing the stream format. Or you can create a temporary stream to hold portions of other streams that have been copied.

You can create a stream in memory that is not associated with any file by using the **AVIStreamCreate** function. This function returns the address of the interface to the new stream in a specified location and is used internally by other functions that create temporary streams.

You can create a compressed stream from an uncompressed stream by using the **AVIMakeCompressedStream** function. You identify the stream to compress, the compression method and compression options, and the handler for the new stream.

When you finish using a stream created with **AVIStreamCreate** or **AVIMakeCompressedStream**, close the stream by using the **AVIStreamRelease** function. **AVIStreamRelease** frees the resources the temporary stream used.

Editing Streams

You can create a stream that you can edit by using the **CreateEditableStream** function. This function initializes the environment for editing a stream. This includes creating an interface to the new stream and internal edit tables that track changes made to the stream. **CreateEditableStream** returns a stream pointer to an editable stream that is required by other stream editing functions. The editable stream pointer can also be used by other AVIFile functions that accept stream pointers.

You can cut one or more samples from an editable stream by using the **EditStreamCut** function. This function adds an entry to the edit table to remove the samples from the editable stream and then places a copy of the cut samples in a new temporary stream whose interface pointer is returned in a variable.

You can copy one or more samples from an editable stream into a temporary stream by using the **EditStreamCopy** function. **EditStreamCopy** places copies of the samples in a new temporary stream whose interface pointer is returned in a variable.

You can copy one or more samples from a stream and paste them into an editable stream by using the **EditStreamPaste** function. This function adds an entry in the edit table of the target editable stream to insert the samples at the specified position.

You can duplicate an editable stream by using the **EditStreamClone** function. This function returns a pointer to the stream interface of the new stream. You can copy these streams to the clipboard or use them to maintain a trail of edits made to a stream.

You can change several of the characteristics of an editable stream by using the **EditStreamSetInfo** function. This function updates the priority setting, language, scale and rate, starting time, quality setting, destination rectangle dimensions and coordinates, and the textual description of the stream. These items are stored in the **AVISTREAMINFO** structure associated with the editable stream.

You can also change the textual description of an editable stream by using the **EditStreamSetName** function. This function updates the **szName** member of the **AVISTREAMINFO** structure associated with the editable stream.

The editing functions work on streams. You need to cut and paste each stream individually, and then use the **AVIMakeFileFromStreams** function to create a new file pointer.

Note The edit tables in an editable stream maintain all the changes for a stream. The source stream is never changed.

Using AVIFile Functions and Macros

This section contains examples demonstrating how to perform the following tasks:

- Open an AVI file.
- Open streams in an AVI file and close the file.
- Read streams from an AVI file.
- Read from one stream and write to another.
- Use the editing functions and put a file on the clipboard.

Opening an AVI File

The following example initializes the AVIFile library and opens an AVI file. The function uses a default file handler:

```
// LoadAVIFile - loads AVIFile and opens an AVI file.
//
// szfile - filename
// hwnd - window handle
//
VOID LoadAVIFile(LPCSTR szFile, HWND hwnd)
{
    LONG hr;
    PAVIFILE *pfile;

    AVIFileInit();          // opens AVIFile library

    hr = AVIFileOpen(&pfile, szFile, OF_SHARE_DENY_WRITE, 0L);
    if (hr != 0){
        ErrMsg("Unable to open %s", szFile);
        return;
    }

//  .
//  . Place functions here that interact with the open file.
//  .

    AVIFileRelease(pfile);  // closes the file
    AVIFileExit();          // releases AVIFile library
}
```

Opening Streams in an AVI File and Closing the File

The following example opens all the streams in an AVI file. If an error is encountered, the file is closed.

```
// InsertAVIFile - opens the streams in an AVI file.
//
// pfile - file-interface pointer from AVIFileOpen
//
// Global variables
// gcpavi - count of the number of streams in an AVI file
// gapavi[] = array of stream-interface pointers

void InsertAVIFile(PAVIFILE pfile, HWND hwnd, LPSTR lpszFile)
{
    int    i;
    gcpavi = 0;

    // Open the streams until a stream is not available.
    for (i = gcpavi; i < MAXNUMSTREAMS; i++) {
        gapavi[i] = NULL;
        if (AVIFileGetStream(pfile, &gapavi[i], 0L, i - gcpavi)
            != AVIERR_OK)
        break;

    if (gapavi[i] == NULL)
        break;
    }
    // Display error message-stream not found.
    if (gcpavi == i)
    {
        ErrMsg("Unable to open %s", lpszFile);
        if (pfile) // If file is open, close it
        AVIFileRelease(pfile);
        return;
    }
    else {
        gcpavi = i - 1;
    }

//  .
//  . Place functions to process data here.
//  .
}
```

Reading Streams from an AVI File

The following subroutine obtains stream information from an AVI file and determines the stream type from the **AVISTREAMINFO** structure.

```
// StreamTypes - opens the streams in an AVI file and determines
// stream types.
//
// Global variables
// gcpavi - count of streams in an AVI file
// gapavi[] = array of stream-interface pointers

void StreamTypes(HWND hwnd)
{
    AVISTREAMINFO     avis;
    LONG    r, lHeight = 0;
    WORD    w;
    int     i;
    RECT    rc;

// Walk through all streams.
    for (i = 0; i < gcpavi; i++) {
        AVIStreamInfo(gapavi[i], &avis, sizeof(avis));

        if (avis.fccType == streamtypeVIDEO) {

        // Place video-processing functions here.

        }
        else if (avis.fccType == streamtypeAUDIO) {

        // Place audio-processing functions here.

        }
        else if (avis.fccType == streamtypeTEXT) {

        // Place text-processing functions here.

        }
    .
    .
    .
    }
}
```

Reading from One Stream and Writing to Another

The following example reads data from a stream, compresses it into a new stream, and writes the compressed data into a stream of a new file.

```
// SaveSmall - copies a stream of data from one file, compresses
// the stream, and writes the compressed stream to a new file.
//
// ps stream interface pointer
// lpFilename - new AVI file to build
//

void SaveSmall(PAVISTREAM ps, LPSTR lpFilename)
{
    PAVIFILE         pf;
    PAVISTREAM       psSmall;
    HRESULT          hr;
    AVISTREAMINFO    strhdr;
    BITMAPINFOHEADER bi;
    BITMAPINFOHEADER biNew;
    LONG             lStreamSize;
    LPVOID           lpOld;
    LPVOID           lpNew;

    // Determine the size of the format data.
    AVIStreamFormatSize(ps, 0, &lStreamSize);
    if (lStreamSize > sizeof(bi)) // Format larger than space allocated?
        return;

    lStreamSize = sizeof(bi);
    hr = AVIStreamReadFormat(ps, 0, &bi, &lStreamSize); // Read format
    if (bi.biCompression != BI_RGB) // Wrong compression format?
        return;

    hr = AVIStreamInfo(ps, &strhdr, sizeof(strhdr));

    // Create new AVI file.
    hr = AVIFileOpen(&pf, lpFilename, OF_WRITE | OF_CREATE, NULL);
    if (hr != 0)
        return;

    // Set parameters for the new stream.
    biNew = bi;
    biNew.biWidth /= 2;
    biNew.biHeight /= 2;
    biNew.biSizeImage = ((((UINT)biNew.biBitCount * biNew.biWidth
                        + 31)&~31) / 8) * biNew.biHeight;
    SetRect(&strhdr.rcFrame, 0, 0, (int) biNew.biWidth,
            (int) biNew.biHeight);

    // Create a stream.
    hr = AVIFileCreateStream(pf, &psSmall, &strhdr);
    if (hr != 0) {            //Stream created OK? If not close file.
        AVIFileRelease(pf);
        return;
```

```
    }

    // Set format of new stream.
    hr = AVIStreamSetFormat(psSmall, 0, &biNew, sizeof(biNew));
    if (hr != 0) {
        AVIStreamRelease(psSmall);
        AVIFileRelease(pf);
        return;
    }

    // Allocate memory for the bitmaps.
    lpOld = GlobalAllocPtr(GMEM_MOVEABLE, bi.biSizeImage);
    lpNew = GlobalAllocPtr(GMEM_MOVEABLE, biNew.biSizeImage);

    // Read the stream data.
    for (lStreamSize = AVIStreamStart(ps); lStreamSize <
        AVIStreamEnd(ps); lStreamSize++) {
        hr = AVIStreamRead(ps, lStreamSize, 1, lpOld, bi.biSizeImage,
            NULL, NULL);
        //  .
        //  . Place error check here.
        //  .

        // Compress the data.
        CompressDIB(&bi, lpOld, &biNew, lpNew);

        // Save the compressed data.
        hr = AVIStreamWrite(psSmall, lStreamSize, 1, lpNew,
            biNew.biSizeImage, AVIIF_KEYFRAME, NULL, NULL);
    }

     // Close the stream and file.
    AVIStreamRelease(psSmall);
    AVIFileRelease(pf);
}
```

Using the Editing Functions and Putting a File on the Clipboard

The following example cuts, copies, or deletes segments from an array of streams. The cut and copied streams are merged into a new file and placed on the clipboard.

```
// Global variables
// gcpavi - count of streams in an AVI file
// gapavi[] - array of stream-interface pointers, used as data source
// gapaviSel[] - stream-interface pointers of edited streams
// galSelStart[] - edit starting point in each stream
// galSelLen[] - length of edit to make in each stream
// gapaviTemp[] - array of stream-interface pointers put on clipboard
//
// Comment:
//     The editable streams in gapaviSel have been
//     initialized with CreateEditableStream.
//

case MENU_CUT:
case MENU_COPY:
case MENU_DELETE:
{
    PAVIFILE pf;
    int      i;

    // Walk list of selections and make streams out of each section.
    gcpaviSel = 0;  // index counter for destination streams
    for (i = 0; i < gcpavi; i++) {
        if (galSelStart[i] != -1) {
            if (wParam == MENU_COPY)
                EditStreamCopy(gapavi[i], &galSelStart[i],
                &galSelLen[i], &gapaviSel[gcpaviSel++]);
            else {
                EditStreamCut(gapavi[i], &galSelStart[i],
                &galSelLen[i], &gapaviSel[gcpaviSel++]);
            }
        }
    }

.
.
.
    // Put on the clipboard if segment is not deleted.
    if (gcpaviSel && wParam != MENU_DELETE) {
        PAVISTREAM   gapaviTemp[MAXNUMSTREAMS];
        int i;

        // Clone the edited streams, so that if the user does
        // more editing, the clipboard won't change.
        for (i = 0; i < gcpaviSel; i++) {
            gapaviTemp[i] = NULL;
            EditStreamClone(gapaviSel[i], &gapaviTemp[i]);
            //  .
            //  . Place error check here.
            //  .
```

```
        }

        // Create a file from the streams and put on clipboard.
        AVIMakeFileFromStreams(&pf, gcpaviSel, gapaviTemp);
        AVIPutFileOnClipboard(pf);

        // Release clone streams.
        for (i = 0; i < gcpaviSel; i++) {
            AVIStreamRelease(gapaviTemp[i]);
        }

        // Release file put on clipboard.
        AVIFileRelease(pf);
    }

    // Release streams created.
    for (i = 0; i < gcpaviSel; i++)
        AVIStreamRelease(gapaviSel[i]);
.
.
.
}
```

AVIFile Reference

This section describes the functions, macros, and structures for applications using the AVIFile services. These elements are grouped as follows:

AVIFile Library Operations

AVIFileInit
AVIFileExit

Opening and Closing AVI Files

AVIFileOpen
AVIFileAddRef
AVIFileRelease
GetOpenFileNamePreview

Reading from a File

AVIFileInfo
AVIFILEINFO
AVIFileReadData

Writing to a File

AVIFileWriteData

Using the Clipboard

AVIPutFileOnClipboard
AVIGetFromClipboard
AVIClearClipboard

Opening and Closing Streams

AVIFileGetStream
AVIStreamOpenFromFile
AVIStreamAddRef
AVIStreamRelease

Reading Stream Information

AVISTREAMINFO
AVIStreamReadData
AVIStreamDataSize
AVIStreamReadFormat
AVIStreamFormatSize
AVIStreamRead
AVIStreamSampleSize
AVIStreamBeginStreaming
AVIStreamEndStreaming

Decompressing Video Data in a Stream

AVIStreamGetFrameOpen
AVIStreamGetFrame
AVIStreamGetFrameClose

Creating a File from Existing Streams

AVISave
AVISaveV
AVISaveOptions
GetSaveFileNamePreview
AVIMakeFileFromStreams

Writing Individual Streams

AVIFileCreateStream
AVIStreamSetFormat
AVIStreamWrite
AVIFileWriteData
AVIStreamWriteData
AVIStreamRelease

Finding the Starting Position in a Stream

AVIStreamStart
AVIStreamStartTime
AVIStreamLength
AVIStreamLengthTime
AVIStreamFindSample
AVIStreamEnd
AVIStreamEndTime

Finding Sample and Key Frames

AVIStreamFindSample
AVIStreamIsKeyFrame
AVIStreamNearestKeyFrame
AVIStreamNearestKeyFrameTime
AVIStreamNearestSample
AVIStreamNearestSampleTime
AVIStreamNextKeyFrame
AVIStreamNextKeyFrameTime
AVIStreamNextSample
AVIStreamNextSampleTime
AVIStreamPrevKeyFrame
AVIStreamPrevKeyFrameTime
AVIStreamPrevSample
AVIStreamPrevSampleTime
AVIStreamSampleToSample

Switching Between Samples and Time

AVIStreamSampleToTime
AVIStreamTimeToSample

Creating Temporary Streams

AVIStreamCreate
AVIMakeCompressedStream
AVIStreamRelease

Editing AVI Streams

CreateEditableStream
EditStreamCut
EditStreamCopy
EditStreamPaste
EditStreamClone
EditStreamSetInfo
EditStreamSetName

AVIBuildFilter

```
STDAPI AVIBuildFilter(LPTSTR lpszFilter, LONG cbFilter, BOOL fSaving);
```

Builds a filter specification that is subsequently used by the **GetOpenFileName** or **GetSaveFileName** function.

- Returns AVIERR_OK if successful or an error otherwise. The following error values are defined:

AVIERR_BUFFERTOOSMALL	The buffer size *cbFilter* was smaller than the generated filter specification.
AVIERR_MEMORY	There was not enough memory to complete the read operation.

lpszFilter
Address of the buffer containing the filter string.

cbFilter
Size, in bytes, of buffer pointed to by *lpszFilter*.

fSaving
Flag that indicates whether the filter should include read or write formats. Specify TRUE to include write formats or FALSE to include read formats.

This function accesses the registry for all filter types that the AVIFile library can use to open, read, or write multimedia files. It does not search the hard disk for filter DLLs and formats.

AVIClearClipboard

```
STDAPI AVIClearClipboard(VOID);
```

Removes an AVI file from the clipboard.

- Returns zero if successful or an error otherwise.

AVIFileAddRef

```
STDAPI_(ULONG) AVIFileAddRef(PAVIFILE pfile);
```

Increments the reference count of an AVI file.

- Returns the updated reference count for the file interface.

pfile
Handle of an open AVI file.

AVIFileCreateStream

```
STDAPI AVIFileCreateStream(PAVIFILE pfile, PAVISTREAM FAR * ppavi,
    AVISTREAMINFO FAR * psi);
```

Creates a new stream in an existing file and creates an interface to the new stream.

- Returns zero if successful or an error otherwise. Unless the file has been opened with write permission, this function returns AVIERR_READONLY.

pfile
Handle of an open AVI file.

ppavi
Address of the new stream interface.

psi
Address of a structure containing information about the new stream, including the stream type and its sample rate.

This function starts a reference count for the new stream.

AVIFileEndRecord

```
STDAPI AVIFileEndRecord(PAVIFILE pfile);
```

Marks the end of a record when writing an interleaved file that uses a 1:1 interleave factor of video to audio data. (Each frame of video is interspersed with an equivalent amount of audio data.)

- Returns zero if successful or an error otherwise.

pfile
Handle of an open AVI file.

The **AVISave** function uses this function internally. In general, applications should not need to use this function.

AVIFileExit

```
STDAPI_(VOID) AVIFileExit(VOID);
```

Exits the AVIFile library and decrements the reference count for the library.

This function supercedes the obsolete **AVIStreamExit** function.

AVIFileGetStream

```
STDAPI AVIFileGetStream(PAVIFILE pfile, PAVISTREAM FAR * ppavi,
    DWORD fccType, LONG lParam);
```

Returns the address of a stream interface that is associated with a specified AVI file.

- Returns zero if successful or an error otherwise. Possible error values include the following:

AVIERR_NODATA	The file does not contain a stream corresponding to the values of *fccType* and *lParam*.
AVIERR_MEMORY	Not enough memory.

pfile
Handle of an open AVI file.

ppavi
Address of the new stream interface.

fccType
Four-character code indicating the type of stream to open. Zero indicates any stream can be opened. The following definitions apply to the data commonly found in AVI streams:

streamtypeAUDIO	Indicates an audio stream.
streamtypeMIDI	Indicates a MIDI stream.
streamtypeTEXT	Indicates a text stream.
streamtypeVIDEO	Indicates a video stream.

lParam
Count of the stream type. Identifies which occurrence of the specified stream type to access.

AVIFileInfo

```
STDAPI AVIFileInfo(PAVIFILE pfile, AVIFILEINFO FAR * pfi, LONG lSize);
```

Obtains information about an AVI file.

- Returns zero if successful or an error otherwise.

pfile
Handle of an open AVI file.

pfi
Address of the structure used to return file information. Typically, this parameter points to an **AVIFILEINFO** structure.

lSize
Size, in bytes, of the structure.

AVIFileInit

```
STDAPI_(VOID) AVIFileInit(VOID);
```

Initializes the AVIFile library.

The AVIFile library maintains a count of the number of times it is initialized, but not the number of times it was released. Use the **AVIFileExit** function to release the AVIFile library and decrement the reference count. Call **AVIFileInit** before using any other AVIFile functions.

This function supercedes the obsolete **AVIStreamInit** function.

AVIFileOpen

```
STDAPI AVIFileOpen(PAVIFILE FAR * ppfile, LPCTSTR szFile,
    UINT mode, CLSID FAR * pclsidHandler);
```

Opens an AVI file and returns the address of a file interface used to access it. The AVIFile library maintains a count of the number of times a file is opened, but not the number of times it was released. Use the **AVIFileRelease** function to release the file and decrement the count.

- Returns zero if successful or an error otherwise. Possible error values include the following:

AVIERR_BADFORMAT	The file couldn't be read, indicating a corrupt file or an unrecognized format.
AVIERR_MEMORY	The file could not be opened because of insufficient memory.
AVIERR_FILEREAD	A disk error occurred while reading the file.
AVIERR_FILEOPEN	A disk error occurred while opening the file.
REGDB_E_CLASSNOTREG	According to the registry, the type of file specified in **AVIFileOpen** does not have a handler to process it.

ppfile
Address to contain the new file interface pointer.

szFile
Null-terminated string containing the name of the file to open.

mode
Access mode to use when opening the file. The default access mode is OF_READ. The following access modes can be specified with **AVIFileOpen**:

OF_CREATE
Creates a new file. If the file already exists, it is truncated to zero length.

OF_SHARE_DENY_NONE
Opens the file nonexclusively. Other processes can open the file with read or write access. **AVIFileOpen** fails if another process has opened the file in compatibility mode.

OF_SHARE_DENY_READ
Opens the file nonexclusively. Other processes can open the file with write access. **AVIFileOpen** fails if another process has opened the file in compatibility mode or has read access to it.

OF_SHARE_DENY_WRITE
Opens the file nonexclusively. Other processes can open the file with read access. **AVIFileOpen** fails if another process has opened the file in compatibility mode or has write access to it.

OF_SHARE_EXCLUSIVE
Opens the file and denies other processes any access to it. **AVIFileOpen** fails if any other process has opened the file.

OF_READ
Opens the file for reading.

OF_READWRITE
Opens the file for reading and writing.

OF_WRITE
Opens the file for writing.

pclsidHandler
Address of a class identifier of the standard or custom handler you want to use. If the value is NULL, the system chooses a handler from the registry based on the file extension or the RIFF type specified in the file.

AVIFileReadData

```
STDAPI AVIFileReadData(PAVIFILE pfile, DWORD ckid, LPVOID lpData,
    LONG FAR * lpcbData);
```

Reads optional header data that applies to the entire file, such as author or copyright information.

- Returns zero if successful or an error otherwise. The return value AVIERR_NODATA indicates that data with the requested chunk identifier does not exist.

pfile
Handle of an open AVI file.

ckid
RIFF chunk identifier (four-character code) of the data.

lpData
Address of the buffer used to return the data read.

lpcbData
Address of a location indicating the size of the memory block referenced by *lpData*. If the data is read successfully, the value is changed to indicate the amount of data read.

The optional header information is custom and does not have a set format.

AVIFileRelease

```
STDAPI_(ULONG) AVIFileRelease(PAVIFILE pfile);
```

Decrements the reference count of an AVI file interface handle and closes the file if the count reaches zero.

- Returns the reference count of the file. This return value should be used only for debugging purposes.

pfile
Handle of an open AVI file.

This function supercedes the obsolete **AVIFileClose** function.

AVIFileWriteData

```
STDAPI AVIFileWriteData(PAVIFILE pfile, DWORD ckid, LPVOID lpData,
    LONG cbData);
```

Writes supplementary data (other than normal header, format, and stream data) to the file.

- Returns zero if successful or an error otherwise. If an application has read-only access to the file, the error code AVIERR_READONLY is returned.

pfile
Handle of an open AVI file.

ckid
RIFF chunk identifier (four-character code) of the data.

lpData
Address of the buffer used to write the data.

cbData
Size, in bytes, of the memory block referenced by *lpData*.

Use the **AVIStreamWriteData** function to write data that applies to an individual stream.

AVIGetFromClipboard

```
STDAPI AVIGetFromClipboard(PAVIFILE * lppf);
```

Copies an AVI file from the clipboard.

- Returns zero if successful or an error otherwise.

lppf
Address of the location used to return the handle created for the AVI file.

If the clipboard does not contain an AVI file, **AVIGetFromClipboard** also can copy data with the CF_DIB or CF_WAVE clipboard flags to an AVI file. In this case, the function creates an AVI file with one DIB stream and one waveform-audio stream, and fills each stream with the data from the clipboard.

AVIMakeCompressedStream

```
STDAPI AVIMakeCompressedStream(PAVISTREAM FAR * ppsCompressed,
    PAVISTREAM psSource, AVICOMPRESSOPTIONS FAR * lpOptions,
    CLSID FAR * pclsidHandler);
```

Creates a compressed stream from an uncompressed stream and a compression filter, and returns the address of a pointer to the compressed stream. This function supports audio and video compression.

- Returns AVIERR_OK if successful or an error otherwise. Possible error values include the following:

AVIERR_NOCOMPRESSOR	A suitable compressor cannot be found.
AVIERR_MEMORY	There is not enough memory to complete the operation.
AVIERR_UNSUPPORTED	Compression is not supported for this type of data. This error might be returned if you try to compress data that is not audio or video.

ppsCompressed
Address to contain the compressed stream pointer.

psSource
Address of the stream to be compressed.

lpOptions
Address of a structure that identifies the type of compression to use and the options to apply. You can specify video compression by identifying an appropriate handler in the **AVICOMPRESSOPTIONS** structure. For audio compression, specify the compressed data format.

pclsidHandler
Address of a class identifier used to create the stream.

Applications can read from or write to the compressed stream.

AVIMakeFileFromStreams

```
STDAPI AVIMakeFileFromStreams(PAVIFILE FAR * ppfile,
    int nStreams, PAVISTREAM FAR * papStreams);
```

Constructs an AVIFile interface pointer from separate streams.

- Returns zero if successful or an error otherwise.

ppfile
Address to contain the new file interface pointer.

nStreams
Count of the number of streams in the array of stream interface pointers referenced by *papStreams*.

papStreams
Address of an array of stream interface pointers.

Use the **AVIFileRelease** function to close the file.

Other functions can use the AVIFile interface created by this function to copy and edit the streams associated with the interface. For example, you can retrieve a specific stream by using **AVIFileGetStream** with the file interface pointer.

AVIMakeStreamFromClipboard

```
SDTAPI AVIMakeStreamFromClipboard(UINT cfFormat,
    HANDLE hGlobal, PAVISTREAM FAR * ppstream);
```

Creates an editable stream from stream data on the clipboard.

- Returns zero if successful or an error otherwise.

cfFormat
Clipboard flag.

hGlobal
Handle of stream data on the clipboard.

ppstream
Handle of the created stream.

When an application finishes using the editable stream, it must release the stream to free the resources associated with it.

AVIPutFileOnClipboard

```
STDAPI AVIPutFileOnClipboard(PAVIFILE pf);
```

Copies an AVI file to the clipboard.

- Returns zero if successful or an error otherwise.

pf
: Handle of an open AVI file.

This function also copies data with the CF_DIB, CF_PALETTE, and CF_WAVE clipboard flags onto the clipboard using the first frame of the first video stream of the file as a DIB and using the audio stream as CF_WAVE.

AVISave

```
HRESULT AVISave(LPCTSTR szFile, CLSID FAR * pclsidHandler,
    AVISAVECALLBACK lpfnCallback, int nStreams, PAVISTREAM pavi,
    LPAVICOMPRESSOPTIONS lpOptions, . . .);
```

Builds a file by combining data streams from other files or from memory.

- Returns AVIERR_OK if successful or an error otherwise.

szFile
Null-terminated string containing the name of the file to save.

pclsidHandler
Address of the file handler used to write the file. The file is created by calling the **AVIFileOpen** function using this handler. If a handler is not specified, a default is selected from the registry based on the file extension.

lpfnCallback
Address of a callback function for the save operation.

nStreams
Number of streams saved in the file.

pavi
Address of an AVI stream. This parameter is paired with *lpOptions*. The parameter pair can be repeated as a variable number of arguments.

lpOptions
Address of an application-defined **AVICOMPRESSOPTIONS** structure containing the compression options for the stream referenced by *pavi*. This parameter is paired with *pavi*. The parameter pair can be repeated as a variable number of arguments.

This function creates a file, copies stream data into the file, closes the file, and releases the resources used by the new file. The last two parameters of this function identify a stream to save in the file and define the compression options of that stream. When saving more than one stream in an AVI file, repeat these two stream-specific parameters for each stream in the file.

A callback function (referenced by using *lpfnCallback*) can display status information and let the user cancel the save operation. The callback function uses the following format:

```
LONG FAR PASCAL SaveCallback(int nPercent)
```

The *nPercent* parameter specifies the percentage of the file saved.

The callback function should return AVIERR_OK if the operation should continue and AVIERR_USERABORT if the user wishes to abort the save operation.

AVISaveOptions

```
BOOL AVISaveOptions(HWND hwnd, UINT uiFlags, int nStreams,
    PAVISTREAM FAR * ppavi, LPAVICOMPRESSOPTIONS FAR * plpOptions);
```

Retrieves the save options for a file and returns them in a buffer.

- Returns TRUE if the user pressed OK, FALSE for CANCEL, or an error otherwise.

hwnd
Handle of the parent window for the Compression Options dialog box.

uiFlags
Flags for displaying the Compression Options dialog box. The following flags are defined:

ICMF_CHOOSE_KEYFRAME
Displays a Key Frame Every dialog box for the video options. This is the same flag used in the **ICCompressorChoose** function.

ICMF_CHOOSE_DATARATE
Displays a Data Rate dialog box for the video options. This is the same flag used in **ICCompressorChoose**.

ICMF_CHOOSE_PREVIEW
Displays a Preview button for the video options. This button previews the compression by using a frame from the stream. This is the same flag used in **ICCompressorChoose**.

nStreams
Number of streams that have their options set by the dialog box.

ppavi
Address of an array of stream interface pointers. The *nStreams* parameter indicates the number of pointers in the array.

plpOptions
Address of an array of pointers to **AVICOMPRESSOPTIONS** structures. These structures hold the compression options set by the dialog box. The *nStreams* parameter indicates the number of pointers in the array.

This function presents a standard Compression Options dialog box using *hwnd* as the parent window handle. When the user is finished selecting the compression options for each stream, the options are returned in the **AVICOMPRESSOPTIONS** structure in the array referenced by *plpOptions*. The calling application must pass the interface pointers for the streams in the array referenced by *ppavi*.

An application must allocate memory for the **AVICOMPRESSOPTIONS** structures and the array of pointers to these structures.

AVISaveOptionsFree

```
LONG AVISaveOptionsFree(int nStreams, LPAVICOMPRESSOPTIONS
    FAR * plpOptions);
```

Frees the resources allocated by the **AVISaveOptions** function.

- Returns AVIERR_OK.

nStreams
Count of the **AVICOMPRESSOPTIONS** structures referenced in *plpOptions*.

plpOptions
Address of an array of pointers to **AVICOMPRESSOPTIONS** structures. These structures hold the compression options set by the dialog box. The resources allocated by **AVISaveOptions** for each of these structures will be freed.

AVISaveV

```
STDAPI AVISaveV(LPCTSTR szFile, CLSID FAR * pclsidHandler,
    AVISAVECALLBACK lpfnCallback, int nStreams, PAVISTREAM FAR * ppavi,
    LPAVICOMPRESSOPTIONS FAR * plpOptions);
```

Builds a file by combining data streams from other files or from memory.

- Returns AVIERR_OK if successful or an error otherwise.

szFile
Null-terminated string containing the name of the file to save.

pclsidHandler
Address of the file handler used to write the file. The file is created by calling the **AVIFileOpen** function using this handler. If a handler is not specified, a default is selected from the registry based on the file extension.

lpfnCallback
Address of a callback function used to display status information and to let the user cancel the save operation.

nStreams
Number of streams to save.

ppavi
Address of an array of pointers to the **AVISTREAM** function structures. The array uses one pointer for each stream.

plpOptions
Address of an array of pointers to **AVICOMPRESSOPTIONS** structures. The array uses one pointer for each stream.

This function is equivalent to the **AVISave** function except the streams are passed in an array instead of as a variable number of arguments.

This function creates a file, copies stream data into the file, closes the file, and releases the resources used by the new file. The last two parameters of this function are arrays that identify the streams to save in the file and define the compression options of those streams.

An application must allocate memory for the **AVICOMPRESSOPTIONS** structures and the array of pointers to these structures.

AVIStreamAddRef

```
STDAPI_(LONG) AVIStreamAddRef(PAVISTREAM pavi);
```

Increments the reference count of an AVI stream.

- Returns the current reference count of the stream. This value should be used only for debugging purposes.

pavi
Handle of an open AVI stream.

AVIStreamBeginStreaming

```
STDAPI AVIStreamBeginStreaming(PAVISTREAM pavi, LONG lStart,
    LONG lEnd, LONG lRate);
```

Specifies the parameters used in streaming and lets a stream handler prepare for streaming.

- Returns zero if successful or an error otherwise.

pavi
Address of a stream.

lStart
Starting frame for streaming.

lEnd
Ending frame for streaming.

lRate
Speed at which the file is read relative to its natural speed. Specify 1000 for the normal speed. Values less than 1000 indicate a slower-than-normal speed; values greater than 1000 indicate a faster-than-normal speed.

AVIStreamCreate

```
STDAPI AVIStreamCreate(PAVISTREAM FAR * ppavi, LONG lParam1,
    LONG lParam2, CLSID FAR * pclsidHandler);
```

Creates a stream not associated with any file.

- Returns zero if successful or an error otherwise.

ppavi
Address to contain the new stream interface.

lParam1
Stream-handler specific information.

lParam2
Stream-handler specific information.

pclsidHandler
Address of the class identifier used for the stream.

You should not need to call this function. Some functions, such as **CreateEditableStream** and **AVIMakeCompressedStream**, use it internally.

AVIStreamDataSize

```
AVIStreamDataSize(pavi, fcc, plSize)
```

Determines the buffer size, in bytes, needed to retrieve optional header data for a specified stream.

- Returns zero if successful or an error otherwise. The return value AVIERR_NODATA indicates the system could not find any data with the specified four-character code.

pavi
Handle of an open stream.

fcc
Four-character code specifying the stream type.

plSize
Address to contain the buffer size for the optional header data.

The **AVIStreamDataSize** macro is defined as follows:

```
#define AVIStreamDataSize(pavi, fcc, plSize) \
    AVIStreamReadData(pavi, fcc, NULL, plSize)
```

AVIStreamEnd

```
AVIStreamEnd(pavi)
```

Calculates the sample associated with the end of a stream.

- Returns the sample number associated with the end of a stream, or, if an error occurs, one less than the first sample or one less than the stream length.

pavi
Handle of an open stream.

The sample number returned is not a valid sample number for reading data. It represents the end of the file. (The end of the file is equal to the start of the file plus its length.)

The **AVIStreamEnd** macro is defined as follows:

```
#define AVIStreamEnd(pavi) \
    (AVIStreamStart(pavi) + AVIStreamLength(pavi))
```

AVIStreamEndStreaming

```
STDAPI AVIStreamEndStreaming(PAVISTREAM pavi);
```

Ends streaming.

- Returns zero if successful or an error otherwise.

pavi
: Address of a stream.

Many stream implementations ignore this function.

AVIStreamEndTime

```
AVIStreamEndTime(pavi)
```

Returns the time representing the end of the stream.

- Returns the time if successful or - 1 otherwise.

pavi
Handle of an open stream.

The **AVIStreamEndTime** macro is defined as follows:

```
#define AVIStreamEndTime(pavi) \
    AVIStreamSampleToTime(pavi, AVIStreamEnd(pavi))
```

AVIStreamFindSample

```
STDAPI_(LONG) AVIStreamFindSample(PAVISTREAM pavi, LONG lPos,
    LONG lFlags);
```

Returns the position of a sample (key frame, nonempty frame, or a frame containing a format change) relative to the specified position.

- Returns the position of the frame found or -1 if the search is unsuccessful.

pavi
Handle of an open stream.

lPos
Starting frame for the search.

lFlags
Flags that designate the type of frame to locate, the direction in the stream to search, and the type of return information. The following flags are defined:

FIND_ANY
Finds a nonempty frame. This flag supercedes the SEARCH_ANY flag.

FIND_KEY
Finds a key frame. This flag supercedes the SEARCH_KEY flag.

FIND_FORMAT
Finds a format change.

FIND_NEXT
Finds nearest sample, frame, or format change searching forward. The current sample is included in the search. Use this flag with the FIND_ANY, FIND_KEY, or FIND_FORMAT flag. This flag supercedes the SEARCH_FORWARD flag.

FIND_PREV
Finds nearest sample, frame, or format change searching backward. The current sample is included in the search. Use this flag with the FIND_ANY, FIND_KEY, or FIND_FORMAT flag. This flag supercedes the SEARCH_NEAREST and SEARCH_BACKWARD flags.

FIND_FROM_START
Finds first sample, frame, or format change beginning from the start of the stream. Use this flag with the FIND_ANY, FIND_KEY, or FIND_FORMAT flag.

The FIND_KEY, FIND_ANY, and FIND_FORMAT flags are mutually exclusive, as are the FIND_NEXT and FIND_PREV flags.

This function supercedes the obsolete **AVIStreamFindKeyFrame** function.

AVIStreamFormatSize

```
AVIStreamFormatSize(pavi, lPos, plSize)
```

Determines the buffer size, in bytes, needed to store format information for a sample in a stream.

- Returns zero if successful or an error otherwise.

pavi
Handle of an open stream.

lPos
Position of a sample in the stream.

plSize
Address to contain the buffer size.

The **AVIStreamFormatSize** macro is defined as follows:

```
#define AVIStreamFormatSize(pavi, lPos, plSize) \
    AVIStreamReadFormat(pavi, lPos, NULL, plSize)
```

AVIStreamGetFrame

```
STDAPI_(LPVOID) AVIStreamGetFrame(PGETFRAME pgf, LONG lPos);
```

Returns the address of a decompressed video frame.

- Returns a pointer to the frame data if successful or NULL otherwise. The frame data is returned as a packed DIB.

pgf
Address of a **GetFrame** object.

lPos
Position, in samples, within the stream of the desired frame.

The returned frame is valid only until the next call to this function or the **AVIStreamGetFrameClose** function.

AVIStreamGetFrameClose

```
STDAPI AVIStreamGetFrameClose(PGETFRAME pget);
```

Releases resources used to decompress video frames.

- Returns zero if successful or an error otherwise.

pget
Handle returned from the **AVIStreamGetFrameOpen** function. After calling this function, the handle is invalid.

AVIStreamGetFrameOpen

```
STDAPI_(PGETFRAME) AVIStreamGetFrameOpen(PAVISTREAM pavi,
    LPBITMAPINFOHEADER lpbiWanted);
```

Prepares to decompress video frames from the specified video stream.

- Returns a **GetFrame** object that can be used with the **AVIStreamGetFrame** function. If the system cannot find a decompressor that can decompress the stream to the given format, or to any RGB format, the function returns NULL.

pavi
Address of the video stream used as the video source.

lpbiWanted
Address of a structure that defines the desired video format. Specify NULL to use a default format. You can also specify AVIGETFRAMEF_BESTDISPLAYFMT to decode the frames to the best format for your display.

AVIStreamInfo

```
STDAPI AVIStreamInfo(PAVISTREAM pavi, AVISTREAMINFO FAR * psi,
    LONG lSize);
```

Obtains stream header information.

- Returns zero if successful or an error otherwise.

pavi
Handle of an open stream.

psi
Address of a structure to contain the stream information.

lSize
Size, in bytes, of the structure used for *psi*.

AVIStreamIsKeyFrame

```
AVIStreamIsKeyFrame(pavi, lPos)
```

Indicates whether a sample in a specified stream is a key frame.

- Returns TRUE if the sample is a key frame or FALSE otherwise.

pavi
Handle of an open stream.

lPos
Position to search in the stream.

The **AVIStreamIsKeyFrame** macro is defined as follows:

```
#define AVIStreamIsKeyFrame(pavi, lPos) \
    (AVIStreamNearestKeyFrame(pavi, lPos) == 1)
```

AVIStreamLength

```
STDAPI_(LONG) AVIStreamLength(PAVISTREAM pavi);
```

Returns the length of the stream.

- Returns the stream's length, in samples, if successful or - 1 otherwise.

pavi
Handle of an open stream.

AVIStreamLengthTime

```
AVIStreamLengthTime(pavi)
```

Returns the length of a stream in time.

- Returns the time if successful or - 1 otherwise.

pavi
Handle of an open stream.

The **AVIStreamLengthTime** macro is defined as follows:

```
#define AVIStreamLengthTime(pavi) \
    AVIStreamSampleToTime(pavi, AVIStreamLength(pavi))
```

AVIStreamNearestKeyFrame

```
AVIStreamNearestKeyFrame(pavi, lPos)
```

Locates the key frame at or preceding a specified position in a stream.

- Returns the position of the key frame if successful or - 1 otherwise.

pavi
Handle of an open stream.

lPos
Starting position to search in the stream.

The **AVIStreamNearestKeyFrame** macro is defined as follows:

```
#define AVIStreamNearestKeyFrame(pavi, lPos) \
    AVIStreamFindSample(pavi, lPos , FIND_PREV | FIND_KEY)
```

AVIStreamNearestKeyFrameTime

```
AVIStreamNearestKeyFrameTime(pavi, lTime)
```

Determines the time corresponding to the beginning of the key frame nearest (at or preceding) a specified time in a stream.

- Returns the time of the nearest key frame if successful or - 1 otherwise.

pavi
Handle of an open stream.

lTime
Starting time, in milliseconds, to search in the stream.

The **AVIStreamNearestKeyFrameTime** macro is defined as follows:

```
#define AVIStreamNearestKeyFrameTime(pavi, lTime) \
    AVIStreamSampleToTime(pavi, AVIStreamNearestKeyFrame(pavi,
    AVIStreamTimeToSample(pavi, lTime)))
```

AVIStreamNearestSample

```
AVIStreamNearestSample(pavi, lPos)
```

Locates the nearest nonempty sample at or preceding a specified position in a stream.

- Returns the sample position if successful or - 1 otherwise.

pavi
Handle of an open stream.

lPos
Starting position to search in the stream.

The **AVIStreamNearestSample** macro is defined as follows:

```
#define AVIStreamNearestSample(pavi, lPos) \
    AVIStreamFindSample(pavi, lPos, FIND_PREV | FIND_ANY)
```

AVIStreamNearestSampleTime

```
AVIStreamNearestSampleTime(pavi, lTime)
```

Determines the time corresponding to the beginning of a sample that is nearest to a specified time in a stream.

- Returns the time of the nearest sample if successful or - 1 otherwise.

pavi
Handle of an open stream.

lTime
Starting time, in milliseconds, to search in the stream.

The **AVIStreamNearestSampleTime** macro is defined as follows:

```
#define AVIStreamNearestSampleTime(pavi, lTime) \
    AVIStreamSampleToTime(pavi, AVIStreamNearestSample(pavi,
    AVIStreamTimeToSample(pavi, lTime)))
```

AVIStreamNextKeyFrame

```
AVIStreamNextKeyFrame(pavi, lPos)
```

Locates the next key frame following a specified position in a stream.

- Returns the position of the key frame if successful or - 1 otherwise.

pavi
Handle of an open stream.

lPos
Starting position to search in the stream.

The search performed by this macro does not include the frame at the specified position.

The **AVIStreamNextKeyFrame** macro is defined as follows:

```
#define AVIStreamNextKeyFrame(pavi, lPos) \
    AVIStreamFindSample(pavi, lPos + 1, FIND_NEXT | FIND_KEY)
```

AVIStreamNextKeyFrameTime

```
AVIStreamNextKeyFrameTime(pavi, time)
```

Returns the time of the next key frame in the stream, starting at a given time.

- Returns the time if successful or - 1 otherwise.

pavi
Handle of an open stream.

time
Position in the stream to begin searching.

The search performed by this macro includes the frame that corresponds to the specified time.

The **AVIStreamNextKeyFrameTime** macro is defined as follows:

```
#define AVIStreamNextKeyFrameTime(pavi, time) \
    AVIStreamSampleToTime(pavi, \
    AVIStreamNextKeyFrame(pavi, \
    AVIStreamTimeToSample(pavi, time)))
```

AVIStreamNextSample

```
AVIStreamNextSample(pavi, lPos)
```

Locates the next nonempty sample from a specified position in a stream.

- Returns the sample position if successful or - 1 otherwise.

pavi
Handle of an open stream.

lPos
Starting position to search in the stream.

The sample position returned does not include the sample specified by *lPos*.

The **AVIStreamNextSample** macro is defined as follows:

```
#define AVIStreamNextSample(pavi, lPos) \
    AVIStreamFindSample(pavi, lPos + 1, FIND_NEXT | FIND_ANY)
```

AVIStreamNextSampleTime

```
AVIStreamNextSampleTime(pavi, time)
```

Returns the time that a sample changes to the next sample in the stream. This macro finds the the next interesting time in a stream.

- Returns the time if successful or - 1 otherwise.

pavi
Handle of an open stream.

time
Position information of the sample in the stream.

The **AVIStreamNextSampleTime** macro is defined as follows:

```
#define AVIStreamNextSampleTime(pavi, time) \
    AVIStreamSampleToTime(pavi, \
    AVIStreamNextSample(pavi, \
    AVIStreamTimeToSample(pavi, t)))
```

AVIStreamOpenFromFile

```
STDAPI AVIStreamOpenFromFile(PAVISTREAM FAR * ppavi,
    LPCTSTR szFile, DWORD fccType, LONG lParam, UINT mode,
    CLSID FAR * pclsidHandler);
```

Opens a single stream from a file.

- Returns zero if successful or an error otherwise.

ppavi
Address to contain the new stream handle.

szFile
Null-terminated string containing the name of the file to open.

fccType
Four-character code indicating the type of stream to be opened. Zero indicates that any stream can be opened. The following definitions apply to the data commonly found in AVI streams:

streamtypeAUDIO	Indicates an audio stream.
streamtypeMIDI	Indicates a MIDI stream.
streamtypeTEXT	Indicates a text stream.
streamtypeVIDEO	Indicates a video stream.

lParam
Stream of the type specified in *fccType* to access. This parameter is zero-based; use zero to specify the first occurrence.

mode
Access mode to use when opening the file. This function can open only existing streams, so the OF_CREATE mode flag cannot be used. For more information about the available flags for the *mode* parameter, see the **OpenFile** function.

pclsidHandler
Address of a class identifier of the handler you want to use. If the value is NULL, the system chooses one from the registry based on the file extension or the file RIFF type.

This function calls the **AVIFileOpen**, **AVIFileGetStream**, and **AVIFileRelease** functions.

AVIStreamPrevKeyFrame

```
AVIStreamPrevKeyFrame(pavi, lPos)
```

Locates the key frame that precedes a specified position in a stream.

- Returns the position of the key frame if successful or - 1 otherwise.

pavi
Handle of an open stream.

lPos
Starting position to search in the stream.

The search performed by this macro does not include the frame at the specified position.

The **AVIStreamPrevKeyFrame** macro is defined as follows:

```
#define AVIStreamPrevKeyFrame(pavi, lPos) \
    AVIStreamFindSample(pavi, lPos - 1, FIND_PREV | FIND_KEY)
```

AVIStreamPrevKeyFrameTime

```
AVIStreamPrevKeyFrameTime(pavi, time)
```

Returns the time of the previous key frame in the stream, starting at a given time.

- Returns the time if successful or - 1 otherwise.

pavi
Handle of an open stream.

time
Position in the stream to begin searching.

The search performed by this macro includes the frame that corresponds to the specified time.

The **AVIStreamPrevKeyFrameTime** macro is defined as follows:

```
#define AVIStreamPrevKeyFrameTime(pavi, time) \
    AVIStreamSampleToTime(pavi, AVIStreamPrevKeyFrame(pavi,
    AVIStreamTimeToSample(pavi, time)))
```

AVIStreamPrevSample

```
AVIStreamPrevSample(pavi, lPos)
```

Locates the nearest nonempty sample that precedes a specified position in a stream.

- Returns the sample position if successful or - 1 otherwise.

pavi
Handle of an open stream.

lPos
Starting position to search in the stream.

The sample position returned does not include the sample specified by *lPos*.

The **AVIStreamPrevSample** macro is defined as follows:

```
#define AVIStreamPrevSample(pavi, lPos) \
    AVIStreamFindSample(pavi, lPos - 1, FIND_PREV | FIND_ANY)
```

AVIStreamPrevSampleTime

```
AVIStreamPrevSampleTime(pavi, time)
```

Determines the time of the nearest nonempty sample that precedes a specified time in a stream.

- Returns the time if successful or - 1 otherwise.

pavi
Handle of an open stream.

time
Position information of the sample in the stream.

The **AVIStreamPrevSampleTime** macro is defined as follows:

```
#define AVIStreamPrevSampleTime(pavi, time) \
    AVIStreamSampleToTime(pavi, \
    AVIStreamPrevSample(pavi, \
    AVIStreamTimeToSample(pavi, t)))
```

AVIStreamRead

```
STDAPI AVIStreamRead(PAVISTREAM pavi, LONG lStart, LONG lSamples,
    LPVOID lpBuffer, LONG cbBuffer, LONG FAR * plBytes,
    LONG FAR * plSamples);
```

Reads audio, video or other data from a stream according to the stream type.

- Returns zero if successful or an error otherwise. The values returned in *plBytes* and *plSamples* report the amount of data read by this function. Possible error values include the following:

AVIERR_BUFFERTOOSMALL	The buffer size *cbBuffer* was smaller than a single sample of data.
AVIERR_MEMORY	There was not enough memory to complete the read operation.
AVIERR_FILEREAD	A disk error occurred while reading the file.

pavi
Handle of an open stream.

lStart
First sample to read.

lSamples
Number of samples to read. You can also specify the value AVISTREAMREAD_CONVENIENT to let the stream handler determine the number of samples to read.

lpBuffer
Address of a buffer to contain the data.

cbBuffer
Size, in bytes, of the buffer pointed to by *lpBuffer*.

plBytes
Address to contain the number of bytes of data written in the buffer referenced by *lpBuffer*. This value can be NULL.

plSamples
Address to contain the number of samples written in the buffer referenced by *lpBuffer*. This value can be NULL.

If *lpBuffer* is NULL, this function does not read any data; it returns information about the size of data that would be read.

AVIStreamReadData

```
STDAPI AVIStreamReadData(PAVISTREAM pavi, DWORD ckid,
    LPVOID lpData, LONG FAR * lpcbData);
```

Reads optional header data from a stream.

- Returns zero if successful or an error otherwise. The return value AVIERR_NODATA indicates the system could not find any data with the specified chunk identifier.

pavi
Handle of an open stream.

ckid
Four-character code identifying the data.

lpData
Address of the buffer to contain the optional header data.

lpcbData
Address of the location that specifies the buffer size used for *lpData*. If the read is successful, AVIFile changes this value to indicate the amount of data written into the buffer for *lpData*.

This function retrieves only optional header information from the stream. This information is custom and does not have a set format.

AVIStreamReadFormat

```
STDAPI AVIStreamReadFormat(PAVISTREAM pavi, LONG lPos,
    LPVOID lpFormat, LONG FAR * lpcbFormat);
```

Reads the stream format data.

- Returns zero if successful or an error otherwise.

pavi
Handle of an open stream.

lPos
Position in the stream used to obtain the format data.

lpFormat
Address of a buffer to contain the format data.

lpcbFormat
Address of a location indicating the size of the memory block referenced by *lpFormat*. On return, the value is changed to indicate the amount of data read. If *lpFormat* is NULL, this parameter can be used to obtain the amount of memory needed to return the format.

AVIStreamRelease

```
STDAPI_(LONG) AVIStreamRelease(PAVISTREAM pavi);
```

Decrements the reference count of an AVI stream interface handle, and closes the stream if the count reaches zero.

- Returns the current reference count of the stream. This value should be used only for debugging purposes.

pavi
Handle of an open stream.

This function supercedes the obsolete **AVIStreamClose** function.

AVIStreamSampleSize

```
AVIStreamSampleSize(pavi, lPos, plSize)
```

Determines the size of the buffer needed to store one sample of information from a stream. The size corresponds to the sample at the position specified by *lPos*.

- Returns zero if successful or an error otherwise. The value returned in *plSize* reports the number of bytes in the sample read by this function. Possible error values include the following:

AVIERR_BUFFERTOOSMALL	The buffer size was smaller than a single sample of data.
AVIERR_MEMORY	There was not enough memory to complete the read operation.
AVIERR_FILEREAD	A disk error occurred while reading the file.

pavi
Handle of an open stream.

lPos
Position of a sample in the stream.

plSize
Address to contain the buffer size.

The **AVIStreamSampleSize** macro is defined as follows:

```
#define AVIStreamSampleSize(pavi, lPos, plSize) \
    AVIStreamRead(pavi, lPos, 1, NULL, 0, plSize, NULL)
```

AVIStreamSampleToSample

```
AVIStreamSampleToSample(pavi1, pavi2, lSample)
```

Returns the sample in a stream that occurs at the same time as a sample that occurs in a second stream.

- Returns the sample if successful or - 1 otherwise.

pavi1
Handle of an open stream that contains the sample that is returned.

pavi2
Handle of a second stream that contains the reference sample.

lSample
Position information of the sample in the stream referenced by *pavi2*.

The **AVIStreamSampleToSample** macro is defined as follows:

```
#define AVIStreamSampleToSample(pavi1, pavi2, lsample) \
    AVIStreamTimeToSample(pavi1, AVIStreamSampleToTime \
    (pavi2, lsample))
```

AVIStreamSampleToTime

```
STDAPI_(LONG) AVIStreamSampleToTime(PAVISTREAM pavi, LONG lSample);
```

Converts a stream position from samples to milliseconds.

- Returns the converted time if successful or - 1 otherwise.

pavi
Handle of an open stream.

lSample
Position information. A sample can correspond to blocks of audio, a video frame, or other format, depending on the stream type.

AVIStreamSetFormat

```
STDAPI AVIStreamSetFormat(PAVISTREAM pavi, LONG lPos,
    LPVOID lpFormat, LONG cbFormat);
```

Sets the format of a stream at the specified position.

- Returns zero if successful or an error otherwise.

pavi
Handle of an open stream.

lPos
Position in the stream to receive the format.

lpFormat
Address of a structure containing the new format.

cbFormat
Size, in bytes, of the block of memory referenced by *lpFormat*.

The handler for writing AVI files does not accept format changes. Besides setting the initial format for a stream, only changes in the palette of a video stream are allowed in an AVI file. The palette change must occur after any frames already written to the AVI file. Other handlers might impose different restrictions.

AVIStreamStart

```
STDAPI_(LONG) AVIStreamStart(PAVISTREAM pavi);
```

Returns the starting sample number for the stream.

- Returns the number if successful or - 1 otherwise.

pavi
Handle of an open stream.

AVIStreamStartTime

```
AVIStreamStartTime(pavi)
```

Returns the starting time of a stream's first sample.

- Returns the time if successful or - 1 otherwise.

pavi
Handle of an open stream.

The **AVIStreamStartTime** macro is defined as follows:

```
#define AVIStreamStartTime(pavi) \
    AVIStreamSampleToTime(pavi, AVIStreamStart(pavi))
```

AVIStreamTimeToSample

```
STDAPI_(LONG) AVIStreamTimeToSample(PAVISTREAM pavi, LONG lTime);
```

Converts from milliseconds to samples.

- Returns the converted time if successful or - 1 otherwise.

pavi
Handle of an open stream.

lTime
Time, expressed in milliseconds.

Samples typically correspond to audio samples or video frames. Other stream types might support different formats than these.

AVIStreamWrite

```
STDAPI AVIStreamWrite(PAVISTREAM pavi, LONG lStart, LONG lSamples,
    LPVOID lpBuffer, LONG cbBuffer, DWORD dwFlags,
    LONG FAR * plSampWritten, LONG FAR * plBytesWritten);
```

Writes data to a stream.

- Returns zero if successful or an error otherwise.

pavi
Handle of an open stream.

lStart
First sample to write.

lSamples
Number of samples to write.

lpBuffer
Address of a buffer containing the data to write.

cbBuffer
Size of the buffer referenced by *lpBuffer*.

dwFlags
Flag associated with this data. The following flag is defined:

AVIIF_KEYFRAME
Indicates this data does not rely on preceding data in the file.

plSampWritten
Address to contain the number of samples written. This can be set to NULL.

plBytesWritten
Address to contain the number of bytes written. This can be set to NULL.

The default AVI file handler supports writing only at the end of a stream. The "WAVE" file handler supports writing anywhere.

This function overwrites existing data, rather than inserting new data.

AVIStreamWriteData

```
STDAPI AVIStreamWriteData(PAVISTREAM pavi, DWORD ckid,
    LPVOID lpData, LONG cbData);
```

Writes optional header information to the stream.

- Returns zero if successful or an error otherwise. The return value AVIERR_READONLY indicates the file was opened without write access.

pavi
Handle of an open stream.

ckid
Four-character code identifying the data.

lpData
Address of a buffer containing the data to write.

cbData
Number of bytes of data to write into the stream.

Use the **AVIStreamWrite** function to write the multimedia content of the stream. Use **AVIFileWriteData** to write data that applies to an entire file.

CreateEditableStream

```
STDAPI CreateEditableStream(PAVISTREAM FAR * ppsEditable,
    PAVISTREAM psSource);
```

Creates an editable stream. Use this function before using other stream editing functions.

- Returns zero if successful or an error otherwise.

ppsEditable
Address to contain the new stream handle.

psSource
Handle of the stream supplying data for the new stream. Specify NULL to create an empty editable string that you can copy and paste data into.

The stream pointer returned in *ppsEditable* must be used as the source stream in the other stream editing functions.

Internally, this function creates tables to keep track of changes to a stream. The original stream is never changed by the stream editing functions. The stream pointer created by this function can be used in any AVIFile function that accepts stream pointers. You can use this function on the same stream multiple times. A copy of a stream is not affected by changes in another copy.

EditStreamClone

```
STDAPI EditStreamClone(PAVISTREAM pavi, PAVISTREAM FAR * ppResult);
```

Creates a duplicate editable stream.

- Returns zero if successful or an error otherwise.

pavi
Handle of an editable stream that will be copied.

ppResult
Address to contain the new stream handle.

The editable stream that is being cloned must have been created by the **CreateEditableStream** function or one of the stream editing functions.

The new stream can be treated as any other AVI stream.

EditStreamCopy

```
STDAPI EditStreamCopy(PAVISTREAM pavi, LONG FAR * plStart,
    LONG FAR * plLength, PAVISTREAM FAR * ppResult);
```

Copies an editable stream (or a portion of it) into a temporary stream.

- Returns zero if successful or an error otherwise.

pavi
Handle of the stream being copied.

plStart
Starting position within the stream being copied. The starting position is returned.

plLength
Amount of data to copy from the stream referenced by *pavi*. The length of the copied data is returned.

ppResult
Address to contain the handle created for the new stream.

The stream that is copied must be created by the **CreateEditableStream** function or one of the stream editing functions.

The temporary stream can be treated as any other AVI stream.

EditStreamCut

```
STDAPI EditStreamCut(PAVISTREAM pavi, LONG FAR * plStart,
    LONG FAR * plLength, PAVISTREAM FAR * ppResult);
```

Deletes all or part of an editable stream and creates a temporary editable stream from the deleted portion of the stream.

- Returns zero if successful or an error otherwise.

pavi
Handle of the stream being edited.

plStart
Starting position of the data to cut from the stream referenced by *pavi*.

plLength
Amount of data to cut from the stream referenced by *pavi*.

ppResult
Address of the handle created for the new stream.

The stream being edited must have been created by the **CreateEditableStream** function or one of the stream editing functions.

The temporary stream is an editable stream and can be treated as any other AVI stream. An application must release the temporary stream to free the resources associated with it.

EditStreamPaste

```
STDAPI EditStreamPaste(PAVISTREAM pavi, LONG FAR * plPos,
    LONG FAR * plLength, PAVISTREAM pstream, LONG lStart,
    LONG lLength);
```

Copies a stream (or a portion of it) from one stream and pastes it within another stream at a specified location.

- Returns zero if successful or an error otherwise.

pavi
Handle of an editable stream that will receive the copied stream data.

plPos
Starting position to paste the data within the destination stream (referenced by *pavi*).

plLength
Address to contain the amount of data pasted in the stream.

pstream
Handle of a stream supplying the data to paste. This stream does not need to be an editable stream.

lStart
Starting position of the data to copy within the source stream.

lLength
Amount of data to copy from the source stream. If *lLength* is -1, the entire stream referenced by *pstream* is pasted in the other stream.

The stream referenced by *pavi* must have been created by the **CreateEditableStream** function or one of the stream editing functions.

This function inserts data into the specified stream as a continuous block of data. It opens the specified data stream at the insertion point, pastes the specified stream segment at the insertion point, and appends the stream segment that trails the insertion point to the end of pasted segment.

EditStreamSetInfo

```
SDTAPI EditStreamSetInfo(PAVISTREAM pavi, AVISTREAMINFO FAR * lpInfo,
    LONG cbInfo);
```

Changes characteristics of an editable stream.

- Returns zero if successful or an error otherwise.

pavi
Handle of an open stream.

lpInfo
Address of an **AVISTREAMINFO** structure containing new information.

cbInfo
Size, in bytes, of the structure pointed to by *lpInfo*.

You must supply information for the entire **AVISTREAMINFO** structure, including the members you will not use. You can use the **AVIStreamInfo** function to initialize the structure and then update selected members with your data.

This function does not change the following members:

dwCaps

dwEditCount

dwFlags

dwInitialFrames

dwLength

dwSampleSize

dwSuggestedBufferSize

fccHandler

fccType

The function changes the following members:

dwRate

dwQuality

dwScale

dwStart

rcFrame

szName

wLanguage

wPriority

EditStreamSetName

```
SDTAPI EditStreamSetName(PAVISTREAM pavi, LPCSTR lpszName);
```

Assigns a descriptive string to a stream.

- Returns zero if successful or an error otherwise.

pavi
Handle of an open stream.

lpszName
Null-terminated string containing the description of the stream.

This function updates the **szName** member of the **AVISTREAMINFO** structure.

AVICOMPRESSOPTIONS

```
typedef struct {
      DWORD  fccType;
      DWORD  fccHandler;
      DWORD  dwKeyFrameEvery;
      DWORD  dwQuality;
      DWORD  dwBytesPerSecond;
      DWORD  dwFlags;
      LPVOID lpFormat;
      DWORD  cbFormat;
      LPVOID lpParms;
      DWORD  cbParms;
      DWORD  dwInterleaveEvery;
} AVICOMPRESSOPTIONS;
```

Contains information about a stream and how it is compressed and saved. This structure passes data to the **AVIMakeCompressedStream** function (or the **AVISave** function, which uses **AVIMakeCompressedStream**).

fccType

Four-character code indicating the stream type. The following constants have been defined for the data commonly found in AVI streams:

streamtypeAUDIO	Indicates an audio stream.
streamtypeMIDI	Indicates a MIDI stream.
streamtypeTEXT	Indicates a text stream.
streamtypeVIDEO	Indicates a video stream.

fccHandler

Four-character code for the compressor handler that will compress this video stream when it is saved (for example, mmioFOURCC('M','S','V','C')). This member is not used for audio streams.

dwKeyFrameEvery

Maximum period between video key frames. This member is used only if the AVICOMPRESSF_KEYFRAMES flag is set; otherwise every video frame is a key frame.

dwQuality

Quality value passed to a video compressor. This member is not used for an audio compressor.

dwBytesPerSecond

Video compressor data rate. This member is used only if the AVICOMPRESSF_DATARATE flag is set.

dwFlags

Flags used for compression. The following values are defined:

AVICOMPRESSF_DATARATE
: Compresses this video stream using the data rate specified in **dwBytesPerSecond**.

AVICOMPRESSF_INTERLEAVE
: Interleaves this stream every **dwInterleaveEvery** frames with respect to the first stream.

AVICOMPRESSF_KEYFRAMES
: Saves this video stream with key frames at least every **dwKeyFrameEvery** frames. By default, every frame will be a key frame.

AVICOMPRESSF_VALID
: Uses the data in this structure to set the default compression values for **AVISaveOptions**. If an empty structure is passed and this flag is not set, some defaults will be chosen.

lpFormat

Address of a structure defining the data format. For an audio stream, this is an **LPWAVEFORMAT** structure.

cbFormat
Size, in bytes, of the data referenced by **lpFormat**.

lpParms
Video-compressor-specific data; used internally.

cbParms
Size, in bytes, of the data referenced by **lpParms**

dwInterleaveEvery
Interleave factor for interspersing stream data with data from the first stream. Used only if the AVICOMPRESSF_INTERLEAVE flag is set.

AVIFILEINFO

```
typedef struct {
     DWORD dwMaxBytesPerSec;       \\ approximate max. data rate of file
     DWORD dwFlags;                \\ see below
     DWORD dwCaps;                 \\ see below
     DWORD dwStreams;              \\ see below
     DWORD dwSuggestedBufferSize;  \\ see below
     DWORD dwWidth;                \\ width, in pixels, of the AVI file
     DWORD dwHeight;               \\ height, in pixels, of the AVI file
     DWORD dwScale;                \\ see below
     DWORD dwRate;                 \\ see dwScale
     DWORD dwLength;               \\ see below
     DWORD dwEditCount;            \\ see below
     char  szFileType[64];         \\ see below
} AVIFILEINFO;
```

Contains global information for an entire AVI file.

dwFlags

Applicable flags. The following flags are defined:

AVIFILEINFO_HASINDEX
The AVI file has an index at the end of the file. For good performance, all AVI files should contain an index.

AVIFILEINFO_MUSTUSEINDEX
The file index contains the playback order for the chunks in the file. Use the index rather than the physical ordering of the chunks when playing back the data. This could be used for creating a list of frames for editing.

AVIFILEINFO_ISINTERLEAVED
The AVI file is interleaved.

AVIFILEINFO_WASCAPTUREFILE
The AVI file is a specially allocated file used for capturing real-time video. Applications should warn the user before writing over a file with this flag set because the user probably defragmented this file.

AVIFILEINFO_COPYRIGHTED
The AVI file contains copyrighted data and software. When this flag is used, software should not permit the data to be duplicated.

dwCaps

Capability flags. The following flags are defined:

AVIFILECAPS_CANREAD
An application can open the AVI file with with the read privilege.

AVIFILECAPS_CANWRITE
An application can open the AVI file with the write privilege.

AVIFILECAPS_ALLKEYFRAMES
Every frame in the AVI file is a key frame.

AVIFILECAPS_NOCOMPRESSION
The AVI file does not use a compression method.

dwStreams

Number of streams in the file. For example, a file with audio and video has at least two streams.

dwSuggestedBufferSize

Suggested buffer size, in bytes, for reading the file. Generally, this size should be large enough to contain the largest chunk in the file. For an interleaved file, this size should be large enough to read

an entire record, not just a chunk.

If the buffer size is too small or is set to zero, the playback software will have to reallocate memory during playback, reducing performance.

dwScale

Time scale applicable for the entire file. Dividing **dwRate** by **dwScale** gives the number of samples per second.

Any stream can define its own time scale to supersede the file time scale.

dwLength

Length of the AVI file. The units are defined by **dwRate** and **dwScale**.

dwEditCount

Number of streams that have been added to or deleted from the AVI file.

szFileType

Null-terminated string containing descriptive information for the file type.

AVISTREAMINFO

```
typedef struct {                    \\ see below
     DWORD fccType;                  \\ see below
     DWORD fccHandler;               \\ see below
     DWORD dwFlags;                  \\ see below
     DWORD dwCaps;                   \\ capability flags; currently unused
     WORD  wPriority;                \\ priority of the stream
     WORD  wLanguage;                \\ language of the stream
     DWORD dwScale;                  \\ see below
     DWORD dwRate;                   \\ see dwScale
     DWORD dwStart;                  \\ see below
     DWORD dwLength;                 \\ see below
     DWORD dwInitialFrames;          \\ see below
     DWORD dwSuggestedBufferSize;    \\ see below
     DWORD dwQuality;                \\ see below
     DWORD dwSampleSize;             \\ see below
     RECT  rcFrame;                  \\ see below
     DWORD dwEditCount;              \\ see below
     DWORD dwFormatChangeCount;      \\ see below
     char  szName[64];               \\ see below
} AVISTREAMINFO;                    \\ see below
```

Contains information for a single stream.

fccType

Four-character code indicating the stream type. The following constants have been defined for the data commonly found in AVI streams:

streamtypeAUDIO	Indicates an audio stream.
streamtypeMIDI	Indicates a MIDI stream.
streamtypeTEXT	Indicates a text stream.
streamtypeVIDEO	Indicates a video stream.

fccHandler

Four-character code of the compressor handler that will compress this video stream when it is saved (for example, **mmioFOURCC**('M','S','V','C')). This member is not used for audio streams.

dwFlags

Applicable flags for the stream. The bits in the high-order word of these flags are specific to the type of data contained in the stream. The following flags are defined:

AVISTREAMINFO_DISABLED

Indicates this stream should be rendered when explicitly enabled by the user.

AVISTREAMINFO_FORMATCHANGES

Indicates this video stream contains palette changes. This flag warns the playback software that it will need to animate the palette.

dwScale

Time scale applicable for the stream. Dividing **dwRate** by **dwScale** gives the playback rate in number of samples per second.

For video streams, this rate should be the frame rate. For audio streams, this rate should correspond to the audio block size (the **nBlockAlign** member of the **WAVEFORMAT** or **PCMWAVEFORMAT** structure), which for PCM (Pulse Code Modulation) audio reduces to the sample rate.

dwStart

Sample number of the first frame of the AVI file. The units are defined by **dwRate** and **dwScale**.

Normally, this is zero, but it can specify a delay time for a stream that does not start concurrently with the file.

The 1.0 release of the AVI tools does not support a nonzero starting time.

dwLength

Length of this stream. The units are defined by **dwRate** and **dwScale**.

dwInitialFrames

Audio skew. This member specifies how much to skew the audio data ahead of the video frames in interleaved files. Typically, this is about 0.75 seconds.

dwSuggestedBufferSize

Recommended buffer size, in bytes, for the stream. Typically, this member contains a value corresponding to the largest chunk in the stream. Using the correct buffer size makes playback more efficient. Use zero if you do not know the correct buffer size.

dwQuality

Quality indicator of the video data in the stream. Quality is represented as a number between 0 and 10,000. For compressed data, this typically represents the value of the quality parameter passed to the compression software. If set to - 1, drivers use the default quality value.

dwSampleSize

Size, in bytes, of a single data sample. If the value of this member is zero, the samples can vary in size and each data sample (such as a video frame) must be in a separate chunk. A nonzero value indicates that multiple samples of data can be grouped into a single chunk within the file.

For video streams, this number is typically zero, although it can be nonzero if all video frames are the same size. For audio streams, this number should be the same as the **nBlockAlign** member of the **WAVEFORMAT** or **WAVEFORMATEX** structure describing the audio.

rcFrame

Dimensions of the video destination rectangle. The values represent the coordinates of upper left corner, the height, and the width of the rectangle.

dwEditCount

Number of times the stream has been edited. The stream handler maintains this count.

dwFormatChangeCount

Number of times the stream format has changed. The stream handler maintains this count.

szName

Null-terminated string containing a description of the stream.

DrawDib Functions

The DrawDib functions are a group of functions that provide high performance image-drawing capabilities for device-independent bitmaps (DIBs). DrawDib functions support DIBs of 8-bit, 16-bit, 24-bit, and 32-bit image depths.

DrawDib functions write directly to video memory. They do not rely upon functions of the graphics device interface (GDI).

Do I Need DrawDib?

Collectively, the DrawDib functions are similar to the **StretchDIBits** function in that they provide image-stretching and dithering capabilities. But, the DrawDib functions support image decompression, data-streaming, and a greater number of display adapters.

You will find it beneficial to use the DrawDib functions in some circumstances. Still, **StretchDIBits** is more diverse than the DrawDib functions and should be used when the DrawDib functions cannot provide the desired functionality. The following list describes factors to consider when deciding whether to use the DrawDib functions or **StretchDIBits**.

- **Color table information format.** DrawDib functions display images that use the DIB_RGB_COLORS format for their color table. If images in your application store color table information with the DIB_PAL_COLORS or DIB_PAL_INDICES format, you must use **StretchDIBits** to display them.
- **Transfer mode.** DrawDib functions require that your application use the SRCCOPY transfer mode. If your application uses **StretchDIBits** with a transfer mode other than SRCCOPY, you should continue to use **StretchDIBits**. Similarly, if you need to use other raster operations in your application, such as an XOR, use **StretchDIBits**.
- **Quality of video and animation playback.** You can use the DrawDib functions for data-streaming applications, such as playing video clips and animated sequences. The DrawDib functions outperform **StretchDIBits** in that they provide higher-quality images and improve motion during playback.
- **Display adapters.** DrawDib functions support a greater number of display adapters than **StretchDIBits**. The DrawDib functions support VGA color adapters that provide 16-color palettes using 4-bit image depth, SVGA adapters that provide 256-color palettes using 8-bit image depth, and true-color display adapters that provide thousands of colors using 16-bit, 24-bit, and 32-bit image depths.

  The DrawDib functions also improve the speed and quality of displaying images on display adapters with more limited capabilities. For example, when using an 8-bit display adapter, the DrawDib functions efficiently dither true-color images to 256 colors. They also dither 8-bit images when using 4-bit display adapters.
- **Image-stretching.** Like **StretchDIBits**, the DrawDib functions use source and destination rectangles to control the portion of an image that is displayed. You can crop unwanted portions of an image or stretch an image by varying the position and size of the source and destination rectangles. If a display driver does not support image-stretching, the DrawDib functions provide more efficient stretching capabilities than **StretchDIBits**.
- **Compressed images.** The DrawDib functions support several compression and decompression methods, including run-length encoding, JPEG, Cinepak, 411 YUV, and Indeo™.

DrawDib Operations

You can access the entire group of DrawDib functions by using the **DrawDibOpen** function. **DrawDibOpen** loads the dynamic-link library (DLL), allocates memory resources, creates a DrawDib device context (DC), and maintains a reference count of the number of DCs that are initialized. **DrawDibOpen** also returns a handle of the new DC that you use with the other DrawDib functions.

You can release a DrawDib DC when you finish using it by using the **DrawDibClose** function. **DrawDibClose** also decrements the reference count of the applications accessing the DLL. The call to **DrawDibClose** should be the last DrawDib function in your application.

You can create as many DrawDib DCs as you want. You can use multiple DrawDib DCs to draw several bitmaps simultaneously. You can also create multiple DrawDib DCs, each with unique characteristics, so your application can choose and then use the DC with the most appropriate settings. For example, you can create two DrawDib DCs in an application, one that displays an image at its normal resolution and the other that displays an enlarged portion of the image.

To run efficiently, DrawDib functions require information about the display adapter and its driver. The display profile is obtained by running a series of tests on the display adapter the first time the DLL containing the DrawDib functions is accessed. The DrawDib functions use this information for all applications. You can repeat these tests when necessary by using the **DrawDibProfileDisplay** function.

Note Retrieving and storing the display profile is typically a one-time occurrence. If, however, the profile information deleted or another display driver is installed in the system, DrawDib reruns the tests.

Image Rendering

After you create a DrawDib DC, you can draw a DIB to the screen by using the **DrawDibDraw** function. **DrawDibDraw** dithers true-color bitmaps when displaying them with 8-bit display adapters.

DrawDibDraw also supports video compressors transparently when displaying compressed bitmaps. You can access the buffer that contains the decompressed image by using the **DrawDibGetBuffer** function. **DrawDibGetBuffer** returns NULL when drawing an uncompressed bitmap. You should prepare your application to handle compressed and uncompressed bitmaps.

You can refresh an image or a portion of an image displayed by your application by using the **DrawDibUpdate** macro.

Sequences of Images

You can display a sequence of bitmaps with the same dimensions and formats by using the **DrawDibDraw** function with the **DrawDibBegin** function. **DrawDibBegin** improves the efficiency of **DrawDibDraw** by preparing the DrawDib DC for drawing.

Note If your application does not use **DrawDibBegin**, **DrawDibDraw** implicitly executes it prior to drawing. If your application uses **DrawDibBegin** prior to **DrawDibDraw**, **DrawDibDraw** does not have to process the function and wait for it to complete.

DrawDibBegin provides **DrawDibDraw** with the DrawDib DC, the DC handle, the address of the **BITMAPINFOHEADER** structure, and the source and destination rectangle dimensions. When you display a sequence of bitmaps, **DrawDibDraw** checks the values of these items for each image in the sequence. If **DrawDibDraw** detects changes to any of these items, it implicitly calls **DrawDibBegin** again to adjust the DrawDib DC settings.

After using **DrawDibBegin**, you can draw the image sequence by using **DrawDibDraw** and specifying one or more flags as appropriate. Specify the DDF_SAME_HDC flag as long as the DC handle has not changed. Specify the DDF_SAME_DRAW flag when the following parameters for **DrawDibDraw** have not changed: the address of the **BITMAPINFOHEADER** structure and the source and destination rectangle dimensions.

You can update the flags set with **DrawDibBegin** by using the **DrawDibEnd** function followed by another call to **DrawDibBegin**. **DrawDibEnd** clears the DrawDib DC of its current flags and settings. The subsequent call to **DrawDibBegin** reinitializes the DrawDib DC with the appropriate flags and settings. Alternatively, you can update the flags for a DrawDib DC by using **DrawDibBegin** without **DrawDibEnd**, as long as you change at least one of the following settings concurrently with the flags: the address of the **BITMAPINFOHEADER** structure or the source or destination rectangle dimensions.

You can increase the efficiency of **DrawDibDraw** for data-streaming operations that use compressed images, such as playing a video clip, by using the **DrawDibStart** and **DrawDibStop** functions. **DrawDibStart** prepares the DrawDib DC to receive a stream of images by sending a message to the video compression manager (VCM). When streaming has ended, **DrawDibStop** sends a message to VCM indicating that it can release resources it allocated for the data-streaming operation. For more information about VCM, see Chapter 7, "Video Compression Manager."

Note You must specify the width and height of the source and destination rectangles in your application; however, you do not need to specify the origins of the rectangles. Your application can redefine the origins in **DrawDibDraw** to use different portions of the image or to update different portions of the display.

Palettes

The DrawDib functions require that an application respond to two palette-oriented messages: WM_QUERYNEWPALETTE and WM_PALETTECHANGED. If your application is not palette-aware, you will need to add a handler for each of these messages. For more information about processing the WM_QUERYNEWPALETTE and WM_PALETTECHANGED messages, see "Adding Palette Message Handlers" later in this chapter.

You can realize the current DrawDib palette to the DC by using the **DrawDibRealize** function. You should realize the palette in response to the WM_QUERYNEWPALETTE or WM_PALETTECHANGED message, or when you prepare to display an image sequence by using the **DrawDibDraw** function.

You can draw an image mapped to another palette by using the **DrawDibSetPalette** function. **DrawDibSetPalette** forces the DrawDib DC to use the specified palette. This can affect the image quality. For example, an application that is palette-aware might have realized a palette and needs to prevent DrawDib from realizing its own palette. The application can use **DrawDibSetPalette** to notify DrawDib of the palette to use.

You can obtain a handle of the current foreground palette by using the **DrawDibGetPalette** function. If your application uses the current foreground palette, it does not have exclusive use of the palette and another application can invalidate the palette handle. Your application should not free the palette when you finish using it. Freeing the palette could invalidate the palette handle for another application.

You can prepare DrawDib to receive new color values for its palette by using the **DrawDibChangePalette** function. In the code following **DrawDibChangePalette**, you assign new values for the palette color table. If the DDF_ANIMATE flag is not set in the DrawDib DC when you call **DrawDibChangePalette**, you can enact the palette changes by using **DrawDibRealize** to realize the palette and by using **DrawDibDraw** to redraw the image. If the DDF_ANIMATE flag is set in the DrawDib DC, you can animate the palette and the colors of the displayed bitmap by using **DrawDibDraw** or **DrawDibRealize**. You can update the DDF_ANIMATE flag by using the **DrawDibEnd** and **DrawDibBegin** functions.

Note If you free the DrawDib palette while it is selected by a DC, a GDI error can result when the DC uses the palette. Instead, your application should use **DrawDibSetPalette** to change the DrawDib DC to use the default palette or another palette.

The following functions can free the DrawDib palette and should not be used until the palette is not selected by the DC: **DrawDibEnd**, **DrawDibClose**, and **DrawDibBegin**. **DrawDibDraw** can also free the palette when it uses the same DrawDib DC but specifies different drawing parameters (*lpbi*, *dxDst*, *dyDst*, *dxSrc*, or *dySrc*) or a different format.

Timing

As part of debugging an application, you can obtain information about the amount of time it takes to complete repetitive DrawDib operations a specified number of times by using the **DrawDibTime** function. **DrawDibTime** returns timing information for the following operations:

- Drawing a bitmap
- Decompressing a bitmap
- Dithering a bitmap
- Stretching a bitmap
- Transferring a bitmap by using the **BitBlt** function
- Transferring a bitmap by using the **StretchDIBits** function

After retrieving a set of values, **DrawDibTime** resets the count and value for each operation.

DrawDibTime is available only in the debug version of the DrawDib functions.

Using DrawDib

This section contains examples demonstrating how to perform the following tasks:

- Add palette message handlers.
- Draw a DC.
- Animate a palette.

Adding Palette Message Handlers

The following example illustrates simple message handlers for the WM_PALETTECHANGED and WM_QUERYNEWPALETTE messages. The example uses the **DrawDibRealize** function to process the WM_QUERYNEWPALETTE message.

Your application should respond to the WM_QUERYNEWPALETTE message by invalidating the destination window to let the **DrawDibDraw** function redraw an image. You should respond to the WM_PALETTECHANGED message by using the **DrawDibRealize** function to realize the palette.

```
case WM_PALETTECHANGED:
     if ((HWND)wParam == hwnd)
           break;
case WM_QUERYNEWPALETTE:
     hdc = GetDC(hwnd);
     f = DrawDibRealize(hdd, hdc, FALSE) > 0;
     ReleaseDC(hwnd, hdc);
     if (f)
           InvalidateRect(hwnd, NULL, TRUE);
     break;
```

Drawing a Display Context

The following example prepares a DrawDib DC by using the **DrawDibRealize** function prior to displaying several images in a bitmap sequence.

```
hdc = GetDC(hwnd);
DrawDibBegin(hdd, hdc, dxDest, dyDest, lpbi, dxSrc, dySrc, NULL);
DrawDibRealize(hdd, hdc, fBackground);
DrawDibDraw(hdd, hdc, xDst, yDst, dxDst, dyDst, lpbi, lpBits,
    xSrc, ySrc, dxSrc, dySrc, DDF_SAME_DRAW|DDF_SAME_HDC);
DrawDibDraw(hdd, hdc, xDst, yDst, dxDst, dyDst, lpbi, lpBits,
    xSrc, ySrc, dxSrc, dySrc, DDF_SAME_DRAW|DDF_SAME_HDC);
DrawDibDraw(hdd, hdc, xDst, yDst, dxDst, dyDst, lpbi, lpBits,
    xSrc, ySrc, dxSrc, dySrc, DDF_SAME_DRAW|DDF_SAME_HDC);
ReleaseDC(hwnd, hdc);
```

Animating a Palette

The following example animates a palette by using the **DrawDibRealize**, **DrawDibChangePalette**, and **DrawDibDraw** functions.

You can change the colors of a bitmap by using the **DrawDibBegin** function in combination with **DrawDibChangePalette**. First, allow palette changes by specifying the DDF_ANIMATE flag in the call to **DrawDibBegin**. Second, set the color table values from the palette entries by using **DrawDibChangePalette**.

For example, if *lppe* is an address of the PALETTEENTRY array containing the new colors, and *lpbi* is the **LPBITMAPINFOHEADER** structure used in **DrawDibBegin** or **DrawDibDraw**, the following fragment updates the DIB color table.

```
hdc = GetDC(hwnd);
DrawDibBegin(hdd, ....., DDF_ANIMATE);
DrawDibRealize(hdd, hdc, fBackground);
DrawDibDraw(hdd, hdc, ...., DDF_SAME_DRAW|DDF_SAME_HDC);

// Call to change color.
DrawDibChangePalette(hDD, iStart, iLen, lppe);
.
.
.
ReleaseDC(hwnd, hdc);
```

DrawDib Reference

This section describes the DrawDib functions and associated structures. These elements are grouped as follows:

DrawDib Library Operations

DrawDibOpen
DrawDibClose
DrawDibProfileDisplay

Image Rendering

DrawDibDraw
DrawDibGetBuffer
DrawDibUpdate

Sequences of Images

DrawDibBegin
DrawDibEnd
DrawDibStart
DrawDibStop

Palettes

DrawDibRealize
DrawDibSetPalette
DrawDibGetPalette
DrawDibChangePalette

Timing DrawDib

DRAWDIBTIME
DRAWDIBTIME

DrawDib Function Reference

An application uses DrawDib functions to create and manage a DrawDib DC, display and update images on-screen, manipulate palettes, and to close the DrawDib DC when it's no longer needed. The DrawDib functions also include a timing function and a test function to determine display characteristics.

DrawDibBegin

```
BOOL DrawDibBegin(HDRAWDIB hdd, HDC hdc, int dxDest, int dyDest,
    LPBITMAPINFOHEADER lpbi, int dxSrc, int dySrc, UINT wFlags);
```

Changes parameters of a DrawDib DC or initializes a new DrawDib DC.

- Returns TRUE if successful or FALSE otherwise.

hdd
Handle of a DrawDib DC.

hdc
Handle of a DC for drawing. This parameter is optional.

dxDest and *dyDest*
Width and height, in MM_TEXT client units, of the destination rectangle.

lpbi
Address of a **BITMAPINFOHEADER** structure containing the image format. The color table for the DIB follows the image format and the **biHeight** member must be a positive value.

dxSrc and *dySrc*
Width and height, in pixels, of the source rectangle.

wFlags
Applicable flags for the function. The following values are defined:

DDF_ANIMATE
Allows palette animation. If this value is present, DrawDib reserves as many entries as possible by setting PC_RESERVED in the *palPalEntry* members of the **LOGPALETTE** structure, and the palette can be animated by using the **DrawDibChangePalette** function. If your application uses the **DrawDibBegin** function with the **DrawDibDraw** function, set this value with **DrawDibBegin** rather than **DrawDibDraw**.

DDF_BACKGROUNDPAL
Realizes the palette used for drawing as a background task, leaving the current palette used for the display unchanged. (This value is mutually exclusive of DDF_SAME_HDC.)

DDF_BUFFER
Causes DrawDib to try to use an off-screen buffer so DDF_UPDATE can be used. This disables decompression and drawing directly to the screen. If DrawDib is unable to create an off-screen buffer, it will decompress or draw directly to the screen. For more information, see the DDF_UPDATE and DDF_DONTDRAW values described for **DrawDibDraw**.

DDF_DONTDRAW
Current image is not drawn, but is decompressed. DDF_UPDATE can be used later to draw the image. This flag supercedes the DDF_PREROLL flag.

DDF_FULLSCREEN
Not supported.

DDF_HALFTONE
Always dithers the DIB to a standard palette regardless of the palette of the DIB. If your application uses **DrawDibBegin** with **DrawDibDraw**, set this value with **DrawDibBegin** rather than **DrawDibDraw**.

DDF_JUSTDRAWIT
Draws the image by using GDI. Prohibits DrawDib functions from decompressing, stretching, or dithering the image. This strips DrawDib of capabilities that differentiate it from the **StretchDIBits** function.

DDF_SAME_DRAW
Use the current drawing parameters for **DrawDibDraw**. Use this value only if *lpbi*, *dxDest*, *dyDest*, *dxSrc*, and *dySrc* have not changed since using **DrawDibDraw** or **DrawDibBegin**. This

flag supercedes the DDF_SAME_DIB and DDF_SAME_SIZE flags.

DDF_SAME_HDC
Use the current DC handle and the palette currently associated with the DC.

DDF_UPDATE
Last buffered bitmap needs to be redrawn. If drawing fails with this value, a buffered image is not available and a new image needs to be specified before the display can be updated.

This function prepares to draw a DIB specified by *lpbi* to the DC. The image is stretched to the size specified by *dxDest* and *dyDest*. If *dxDest* and *dyDest* are set to - 1, the DIB is drawn to a 1:1 scale without stretching.

You can update the flags of a DrawDib DC by reissuing **DrawDibBegin**, specifying the new flags, and changing at least one of the following settings: *dxDest*, *dyDest*, *lpbi*, *dxSrc*, or *dySrc*.

If the parameters of **DrawDibBegin** have not changed, subsequent calls to the function have no effect.

DrawDibChangePalette

```
BOOL DrawDibChangePalette(HDRAWDIB hdd, int iStart, int iLen,
    LPPALETTEENTRY lppe);
```

Sets the palette entries used for drawing DIBs.

- Returns TRUE if successful or FALSE otherwise.

hdd
Handle of a DrawDib DC.

iStart
Starting palette entry number.

iLen
Number of palette entries.

lppe
Address of an array of palette entries.

This function changes the physical palette only if the current DrawDib palette is realized by calling the **DrawDibRealize** function.

If the color table is not changed, the next call to the **DrawDibDraw** function that does not specify DDF_SAME_DRAW calls the **DrawDibBegin** function implicitly.

DrawDibClose

```
BOOL DrawDibClose(HDRAWDIB hdd);
```

Closes a DrawDib DC and frees the resources DrawDib allocated for it.

- Returns TRUE if successful or FALSE otherwise.

hdd
Handle of a DrawDib DC.

DrawDibDraw

```
BOOL DrawDibDraw(HDRAWDIB hdd, HDC hdc, int xDst, int yDst, int dxDst,
    int dyDst, LPBITMAPINFOHEADER lpbi, LPVOID lpBits, int xSrc,
    int ySrc, int dxSrc, int dySrc, UINT wFlags);
```

Draws a DIB to the screen.

- Returns TRUE if successful or FALSE otherwise.

hdd
Handle of a DrawDib DC.

hdc
Handle of the DC.

xDst and *yDst*
The x- and y-coordinates, in MM_TEXT client coordinates, of the upper left corner of the destination rectangle.

dxDst and *dyDst*
Width and height, in MM_TEXT client coordinates, of the destination rectangle. If *dxDst* is - 1, the width of the bitmap is used. If *dyDst* is - 1, the height of the bitmap is used.

lpbi
Address of the **BITMAPINFOHEADER** structure containing the image format. The color table for the DIB within **BITMAPINFOHEADER** follows the format and the **biHeight** member must be a positive value; **DrawDibDraw** will not draw inverted DIBs.

lpBits
Address of the buffer that contains the bitmap bits.

xSrc and *ySrc*
The x- and y-coordinates, in pixels, of the upper left corner of the source rectangle. The coordinates (0,0) represent the upper left corner of the bitmap.

dxSrc and *dySrc*
Width and height, in pixels, of the source rectangle.

wFlags
Applicable flags for drawing. The following values are defined:

DDF_BACKGROUNDPAL
Realizes the palette used for drawing in the background, leaving the actual palette used for display unchanged. This value is valid only if DDF_SAME_HDC is not set.

DDF_DONTDRAW
Current image is decompressed but not drawn. This flag supercedes the DDF_PREROLL flag.

DDF_FULLSCREEN
Not supported.

DDF_HALFTONE
Always dithers the DIB to a standard palette regardless of the palette of the DIB. If your application uses the **DrawDibBegin** function, set this value in **DrawDibBegin** rather than in **DrawDibDraw**.

DDF_HURRYUP
Data does not have to be drawn (that is, it can be dropped) and DDF_UPDATE will not be used to recall this information. DrawDib checks this value only if it is required to build the next frame; otherwise, the value is ignored.

This value is usually used to synchronize video and audio. When synchronizing data, applications should send the image with this value in case the driver needs to buffer the frame to decompress subsequent frames.

DDF_NOTKEYFRAME

DIB data is not a key frame.

DDF_SAME_HDC
: Use the current DC handle and the palette currently associated with the DC.

DDF_SAME_DRAW
: Use the current drawing parameters for **DrawDibDraw**. Use this value only if *lpbi*, *dxDst*, *dyDst*, *dxSrc*, and *dySrc* have not changed since using **DrawDibDraw** or **DrawDibBegin**. **DrawDibDraw** typically checks the parameters, and if they have changed, **DrawDibBegin** prepares the DrawDib DC for drawing. This flag supercedes the DDF_SAME_DIB and DDF_SAME_SIZE flags.

DDF_UPDATE
: Last buffered bitmap is to be redrawn. If drawing fails with this value, a buffered image is not available and a new image needs to be specified before the display can be updated.

DDF_DONTDRAW causes **DrawDibDraw** to decompress but not display an image. A subsequent call to **DrawDibDraw** specifying DDF_UPDATE displays the image.

If the DrawDib DC does not have an off-screen buffer specified, specifying DDF_DONTDRAW causes the frame to be drawn to the screen immediately. Subsequent calls to **DrawDibDraw** specifying DDF_UPDATE fail.

Although they are set at different times, DDF_UPDATE and DDF_DONTDRAW can be used together to create composite images off-screen. When the off-screen image is complete, you can display the image by calling **DrawDibDraw**.

DrawDibEnd

```
BOOL DrawDibEnd(HDRAWDIB hdd);
```

Clears the flags and other settings of a DrawDib DC that are set by the **DrawDibBegin** or **DrawDibDraw** functions.

- Returns TRUE if successful or FALSE otherwise.

hdd
Handle of the DrawDib DC to free.

DrawDibGetBuffer

```
LPVOID DrawDibGetBuffer(HDRAWDIB hdd, LPBITMAPINFOHEADER lpbi,
    DWORD dwSize, DWORD dwFlags);
```

Retrieves the location of the buffer used by DrawDib for decompression.

- Returns the address of the buffer or NULL if no buffer is used. If *lpbi* is not NULL, it is filled with a copy of the **BITMAPINFO** structure describing the buffer.

hdd
Handle of a DrawDib DC.

lpbi
Address of a **BITMAPINFO** structure. This structure is made up of a **BITMAPINFOHEADER** structure and a 256-entry table defining the colors used by the bitmap.

dwSize
Size, in bytes, of the **BITMAPINFO** structure pointed to by *lpbi*

dwFlags
Reserved; must be zero.

DrawDibGetPalette

```
HPALETTE DrawDibGetPalette(HDRAWDIB hdd);
```

Retrieves the palette used by a DrawDib DC.

- Returns a handle of the palette if successful or NULL otherwise.

hdd
Handle of a DrawDib DC.

This function assumes the DrawDib DC contains a valid palette entry, implying that a call to this function must follow calls to the **DrawDibDraw** or **DrawDibBegin** functions.

You should rarely need to call this function because you can realize the correct palette in response to a window message by using the **DrawDibRealize** function.

DrawDibOpen

```
HDRAWDIB DrawDibOpen(VOID);
```

Opens the DrawDib library for use and creates a DrawDib DC for drawing.

- Returns a handle of a DrawDib DC if successful or NULL otherwise.

When drawing multiple DIBs simultaneously, create a DrawDib DC for each of the images that will be simultaneously on-screen.

DrawDibProfileDisplay

```
BOOL DrawDibProfileDisplay(LPBITMAPINFOHEADER lpbi);
```

Determines settings for the display system when using DrawDib functions.

- Returns a value that indicates the fastest drawing and stretching capabilities of the display system. This value can be zero if the bitmap format is not supported, or one or more of the following flags:

PD_CAN_DRAW_DIB	DrawDib can draw images using this format. Stretching might or might not also be supported.
PD_CAN_STRETCHDIB	DrawDib can stretch and draw images using this format.
PD_STRETCHDIB_1_1_OK	**StretchDIBits** draws unstretched images using this format faster than an alternative method.
PD_STRETCHDIB_1_2_OK	**StretchDIBits** draws stretched images (in a 1:2 ratio) using this format faster than an alternative method.
PD_STRETCHDIB_1_N_OK	**StretchDIBits** draws stretched images (in a 1:N ratio) using this format faster than an alternative method.

lpbi

Address of a **BITMAPINFOHEADER** structure that contains bitmap information. You can also specify NULL to verify that the profile information is current. If the profile information is not current, DrawDib will rerun the profile tests to obtain a current set of information. When you call **DrawDibProfileDisplay** with this parameter set to NULL, the return value is meaningless.

DrawDibRealize

```
UINT DrawDibRealize(HDRAWDIB hdd, HDC hdc, BOOL fBackground);
```

Realizes the palette of the DrawDib DC for use with the specified DC.

- Returns the number of entries in the logical palette mapped to different values in the system palette. If an error occurs or no colors were updated, it returns zero.

hdd
Handle of a DrawDib DC.

hdc
Handle of the DC containing the palette.

fBackground
Background palette flag. If this value is nonzero, the palette is a background palette. If this value is zero and the DC is attached to a window, the logical palette becomes the foreground palette when the window has the input focus. (A DC is attached to a window when the window class style is CS_OWNDC or when the DC is obtained by using the **GetDC** function.)

To select the palette of the DrawDib DC as a background palette, use the **DrawDibDraw** function and specify the DDF_BACKGROUNDPAL flag.

DrawDibSetPalette

```
BOOL DrawDibSetPalette(HDRAWDIB hdd, HPALETTE hpal);
```

Sets the palette used for drawing DIBs.

- Returns TRUE if successful or FALSE otherwise.

hdd
Handle of a DrawDib DC.

hpal
Handle of the palette. Specify NULL to use the default palette.

DrawDibStart

```
BOOL DrawDibStart(HDRAWDIB hdd, LONG rate);
```

Prepares a DrawDib DC for streaming playback.

- Returns TRUE if successful or FALSE otherwise.

hdd
Handle of a DrawDib DC.

rate
Playback rate, in microseconds per frame.

DrawDibStop

```
BOOL DrawDibStop(HDRAWDIB hdd);
```

Frees the resources used by a DrawDib DC for streaming playback.

- Returns TRUE if successful or FALSE otherwise.

hdd
Handle of a DrawDib DC.

DrawDibTime

```
BOOL DrawDibTime(HDRAWDIB hdd, LPDRAWDIBTIME lpddtime);
```

Retrieves timing information about the drawing operation and is used during debug operations.

- Returns TRUE if successful or FALSE otherwise.

hdd
Handle of a DrawDib DC.

lpddtime
Address of a **DRAWDIBTIME** structure.

This function is present only in the debug version of the Win32 Software Development Kit libraries.

DrawDibUpdate

```
BOOL DrawDibUpdate(HDRAWDIB hdd, HDC hdc, int xDst, int yDst);
```

Draws the last frame in the DrawDib off-screen buffer.

- Returns TRUE if successful or FALSE otherwise.

hdd
Handle of a DrawDib DC.

hdc
Handle of the DC.

xDst and *yDst*
The x- and y-coordinates, in MM_TEXT client coordinates, of the upper left corner of the destination rectangle.

The **DrawDibUpdate** macro is defined as follows:

```
#define DrawDibUpdate( hdd, hdc, x, y) \
    DrawDibDraw( hdd, hdc, x, y, 0, 0, NULL, NULL, 0, 0, \
    0, 0, DDF_UPDATE)
```

This macro can be used to refresh an image or a portion of an image displayed by your application.

DrawDib Structure

The DrawDib functions use three structures: **DRAWDIBTIME**, **BITMAPINFO**, and **BITMAPINFOHEADER**. The description for **DRAWDIBTIME** follows. For full descriptions of **BITMAPINFO** and **BITMAPINFOHEADER**, see the Microsoft Win32 Programmer's Reference, Volume 5.

DRAWDIBTIME

```
typedef struct {
    LONG timeCount;        // see below
    LONG timeDraw;         // time to draw bitmaps
    LONG timeDecompress;   // time to decompress bitmaps
    LONG timeDither;       // time to dither bitmaps
    LONG timeStretch;      // time to stretch bitmaps
    LONG timeBlt;          // time to transfer bitmaps (BitBlt)
    LONG timeSetDIBits;    // time to transfer bitmaps (SetDIBits)
}   DRAWDIBTIME, *LPDRAWDIBTIME;
```

Contains elapsed timing information for performing a set of DrawDib operations. The **DrawDibTime** function resets the count and the elapsed time value for each operation each time it is called.

timeCount

Number of times the following operations have been performed since **DrawDibTime** was last called:

- Draw a bitmap on the screen.
- Decompress a bitmap.
- Dither a bitmap.
- Stretch a bitmap.
- Transfer bitmap data by using the **BitBlt** function.
- Transfer bitmap data by using the **SetDIBits** function.

Audio Compression Manager

The audio compression manager (ACM) adds system-level support for the following services:

- Transparent run-time audio compression and decompression
- Waveform-audio data format selection
- Waveform-audio data filter selection
- Waveform-audio data format conversion
- Waveform-audio data filtering

This chapter describes the services available in the ACM and explains the programming techniques used to access these services.

Mapping Waveform-Audio Devices

The Microsoft® Win32® application programming interface (API) provides a set of standard functions for audio devices. These functions issue calls to device drivers that manage hardware devices. The system uses a module called a "mapper" to manage installed devices. The mapper uses special hooks in the driver interface to intercept calls and to act as an intermediary between the system and the drivers installed in the system. The mapper is responsible for matching an application's requests for access to a device with the available devices and for finding a device that meets the current application's audio requirements. The system provides mappers for standard driver types: waveform-audio, MIDI (Musical Instrument Digital Interface), and auxiliary devices.

The ACM is an extension of the basic multimedia system and is installed as a mapper. This means the ACM uses the driver-interface mapper hooks for waveform-audio devices. Using these hooks allows the ACM to decode or encode waveform-audio data before passing it to or from a waveform-audio device driver. The difference between the ACM and the standard system mapper is that the ACM can search for a waveform-audio device that supports a given format or find a combination of a waveform-audio device and an ACM compressor or decompressor that supports a given format.

When an application requests that the system open a waveform-audio device for input or output, the request specifies the format and device. When the specified device is the mapper, the mapper must find a device that supports the given format. The mapper implemented in the ACM searches for an installed waveform-audio device that supports the given format. If the ACM cannot find such a device, it searches for a waveform-audio device and a compressor or decompressor that together support the format. Specifically, the ACM searches for a compressor or decompressor that converts the specified format into a format that is supported by an installed waveform-audio device. After the ACM finds a device that supports the converted format, it can honor requests to play or record the format originally requested, even though no installed waveform-audio device directly supports that format.

In addition to format conversion, the ACM also offers services to support compression, decompression, filtering, format selection, and filter selection. It provides a standard API that it supports by calling its own drivers.

How the Audio Compression Manager Works

The ACM uses existing driver interface hooks to override the default mapping algorithm for waveform-audio devices. This allows the ACM to intercept device-open calls. After a call has been intercepted, the ACM can perform a variety of tasks to process the audio data, such as inserting an external compressor or decompressor into the sequence.

The ACM manages the following types of drivers:

- Compressor and decompressor (codec) drivers
- Format converter drivers
- Filter drivers

Compressors and decompressors change one format type to another. For example, a compressor or decompressor can change a PCM (Pulse Code Modulation) file to an ADPCM (Adaptive Differential Pulse Code Modulation) file. Format converters change the format, but not the data type. For example, a converter can change 44-kHz, 16-bit data to 44-kHz, 8-bit data. Filters do not change the data format at all, but they change the waveform-audio data in some manner. For example, a filter could combine a data stream and an echo of itself. A single ACM driver, or a filter tag or format tag within a driver, might also support combinations of the above types.

For waveform-audio output, the ACM passes each buffer of data to the converter as it arrives. The converter decompresses the data and returns the decompressed data to the ACM in a "shadow" buffer. The ACM then passes the decompressed shadow buffer to the waveform-audio driver. The ACM allocates the shadow buffers whenever it receives a prepare message.

For waveform-audio input, the ACM passes empty shadow buffers to the driver. It uses a background task to receive a notification after the driver has filled the shadow buffer. The ACM then passes the buffers to the driver for compression. After compression is finished, the driver passes the data to the application.

Audio Compression Manager Functions and Structures

The ACM functions fall into several categories. Naming conventions for the functions make it easy to identify these categories. Function names (with two exceptions) are of the form *acmGroupFunction*, where *Group* designates the ACM category (such as "Driver," "Format," "FormatTag," "Filter," "FilterTag," or "Stream"), and *Function* describes the action performed by the function.

The functions in the filter and format groups are very similar. Almost every function that acts on filters has a parallel function that acts on formats.

In the format group, some functions use waveform-audio format tags (the **wFormatTag** member of a **WAVEFORMATEX** structure) while others require full waveform-audio formats (the full **WAVEFORMATEX** structure). (For reference information about the **WAVEFORMATEX** structure, see Error.)

In the filter group, some functions use waveform-audio filter tags (the **dwFilterTag** member of a **WAVEFILTER** structure) while others require full waveform-audio filters (the full **WAVEFILTER** structure).

The functions in the stream group represent the many steps involved in a conversion: opening a conversion instance, preparing for the conversion, performing the conversion, cleaning up after the conversion is complete, and closing the conversion instance.

Functions Called by the System

The system calls three different kinds of application-defined functions. Callback functions are functions in your application that the system calls in response to a request made by an application. Hook procedures help an application handle the customization of dialog boxes. A driver procedure is an application's implementation of its own codec, converter, or filter. In the reference section of this chapter, the prototypes of these three function types are alphabetized with the other function descriptions.

The callback functions have the following names:

- **acmDriverEnumCallback**
- **acmFilterEnumCallback**
- **acmFilterTagEnumCallback**
- **acmFormatEnumCallback**
- **acmFormatTagEnumCallback**
- **acmStreamConvertCallback**

Most of the enumeration functions in the ACM use callback functions. For example, when you call an enumeration function, the ACM enumerates the items by repeatedly calling the application through the callback function.

Some functions cannot be called from within these callback functions. Functions that cannot be called manipulate internal ACM structures that are used by the enumeration functions. The following functions should not be called from within a callback function:

- **acmDriverAdd**
- **acmDriverPriority**
- **acmDriverRemove**

The system calls the following functions to help an application handle the customization of dialog boxes:

- **acmFilterChooseHookProc**
- **acmFormatChooseHookProc**

The following function is given as a prototype that allows an application to use a customized codec, converter, or filter. A function conforming to this prototype may be passed as an argument to the **acmDriverAdd** function.

- **acmDriverProc**

Using the Audio Compression Manager

This section contains examples demonstrating how to perform the following tasks:

- Retrieve a string that describes a filter.
- Produce a dialog box for selecting a filter.
- Produce a dialog box for selecting a specific type of format.
- Produce a dialog box for selecting restricted formats.
- Produce a dialog box for selecting a format for saving.
- Produce a dialog box for selecting a format for recording.
- Convert data from one format to another.
- Multistep format conversion.
- Find a specific format.
- Find a specific driver.
- Add drivers within an application.
- Generate a nonstandard format.

Retrieving a String That Describes a Filter

An application often needs to display a string that describes the current format. This task can be accomplished easily with the **acmFilterTagDetails** and **acmFilterDetails** functions. These functions must be called with the appropriate filter or filter tag. The following example shows how to use these functions.

```
BOOL GetFilterDescription
(
    LPWAVEFILTER  pwfltr,
    LPTSTR        pszFilterTag,
    LPTSTR        pszFilter
)
{
    MMRESULT      mmr;

    // Retrieve the name for the filter tag of the specified filter.
    if (NULL != pszFilterTag) {
        ACMFILTERTAGDETAILS aftd;

        // Initialize all unused members of the ACMFILTERTAGDETAILS
        // structure to zero.
        memset(&aftd, 0, sizeof(aftd));

        // Fill in the required members of the ACMFILTERTAGDETAILS
        // structure for the ACM_FILTERTAGDETAILSF_FILTERTAG query.
        aftd.cbStruct = sizeof(aftd);
        aftd.dwFilterTag = pwfltr->dwFilterTag;

        // Ask the ACM to find the first available driver that
        // supports the specified filter tag.
        mmr = acmFilterTagDetails(NULL, &aftd,
            ACM_FILTERTAGDETAILSF_FILTERTAG);
        if (MMSYSERR_NOERROR != mmr) {
            // No ACM driver is available that supports the
            // specified filter tag.
            return (FALSE);
        }

        // Copy the filter tag name into the calling application's
        // buffer.
        lstrcpy(pszFilterTag, aftd.szFilterTag);
    }

    // Retrieve the description of the attributes for the specified
    // filter.
    if (NULL != pszFilter) {
        ACMFILTERDETAILS afd;

        // Initialize all unused members of the ACMFILTERDETAILS
        // structure to zero.
        memset(&afd, 0, sizeof(afd));

        // Fill in the required members of the ACMFILTERDETAILS
        // structure for the ACM_FILTERDETAILSF_FILTER query.
```

```
        afd.cbStruct    = sizeof(afd);
        afd.dwFilterTag = pwfltr->dwFilterTag;
        afd.pwfltr      = pwfltr;
        afd.cbwfltr     = pwfltr->cbStruct;

        // Ask the ACM to find the first available driver that
        // supports the specified filter.
        mmr = acmFilterDetails(NULL, &afd, ACM_FILTERDETAILSF_FILTER);
        if (MMSYSERR_NOERROR != mmr) {
            // No ACM driver is available that supports the
            // specified filter.
            return (FALSE);
        }

        // Copy the filter attributes description into the calling
        // application's buffer.
        lstrcpy(pszFilter, afd.szFilter);
    }

    return (TRUE);
}
```

Producing a Dialog Box for Selecting a Filter

An application can allow users to select an arbitrary filter operation and apply it to waveform-audio data. In the following example, the application allocates a buffer to hold the filter and then uses the **acmFilterChoose** function to select the filter. The functions in this example must be called with the appropriate filter or filter tag.

```
MMRESULT        mmr;
ACMFILTERCHOOSE afc;
PWAVEFILTER     pwfltr;
DWORD           cbwfltr;

// Determine the maximum size required for any valid filter
// for which the ACM has a driver installed and enabled.
mmr = acmMetrics(NULL, ACM_METRIC_MAX_SIZE_FILTER, &cbwfltr);
if (MMSYSERR_NOERROR != mmr) {

    // The ACM probably has no drivers installed and
    // enabled for filter operations.
    return (mmr);
}

// Dynamically allocate a structure large enough to hold the
// maximum sized filter enabled in the system.
pwfltr = (PWAVEFILTER)LocalAlloc(LPTR, (UINT)cbwfltr);
if (NULL == pwfltr) {
    return (MMSYSERR_NOMEM);
}

// Initialize the ACMFILTERCHOOSE members.
memset(&afc, 0, sizeof(afc));

afc.cbStruct    = sizeof(afc);
afc.fdwStyle    = 0L;               // no special style flags
afc.hwndOwner   = hwnd;             // hwnd of parent window
afc.pwfltr      = pwfltr;           // wfltr to receive selection
afc.cbwfltr     = cbwfltr;          // size of wfltr buffer
afc.pszTitle    = TEXT("Any Filter Selection");

// Call the ACM to bring up the filter-selection dialog box.
mmr = acmFilterChoose(&afc);
if (MMSYSERR_NOERROR == mmr) {
    // The user selected a valid filter. The pwfltr buffer,
    // allocated above, contains the complete filter description.
}

// Clean up and exit.
LocalFree((HLOCAL)pwfltr);
return (mmr);
```

Producing a Dialog Box for Selecting a Specific Type of Format

You might want an application to allow the user to select a format from a restricted list of formats in a dialog box. Restrictions might limit the number of channels, the sampling rate, the waveform-audio format tag, or the number of bits per sample. In all of these cases, the list can be automatically generated by using the **acmFormatChoose** function, setting the **fdwEnum** and **pwfxEnum** members of the **ACMFORMATCHOOSE** structure. The following example illustrates this process.

```
MMRESULT            mmr;
ACMFORMATCHOOSE     afc;
WAVEFORMATEX        wfxSelection;
WAVEFORMATEX        wfxEnum;

// Initialize the ACMFORMATCHOOSE members.
memset(&afc, 0, sizeof(afc));

afc.cbStruct    = sizeof(afc);
afc.fdwStyle    = 0L;               // no special style flags
afc.hwndOwner   = hwnd;             // hwnd of parent window
afc.pwfx        = &wfxSelection;    // wfx to receive selection
afc.cbwfx       = sizeof(wfxSelection);
afc.pszTitle    = TEXT("16 Bit PCM Selection");

//  Request all 16-bit PCM formats be displayed for the user
//  to select from.
memset(&wfxEnum, 0, sizeof(wfxEnum));
wfxEnum.wFormatTag = WAVE_FORMAT_PCM;
wfxEnum.wBitsPerSample = 16;
afc.fdwEnum = ACM_FORMATENUMF_WFORMATTAG |
    ACM_FORMATENUMF_WBITSPERSAMPLE;
afc.pwfxEnum = &wfxEnum;
mmr = acmFormatChoose(&afc);
if ((MMSYSERR_NOERROR != mmr) && (ACMERR_CANCELED != mmr))
{
    // There was a fatal error in bringing up the list
    // dialog box (probably invalid input parameters).
}
```

Producing a Dialog Box for Selecting Restricted Formats

You might want to use the dialog box created by the **acmFormatChoose** function, but limit or control the formats in the dialog box. You can do this by using the ACMFORMATCHOOSE_STYLEF_ENABLEHOOK flag to hook the dialog procedure. The application can then filter the formats by responding to the MM_ACM_FORMATCHOOSE message in the message procedure for the dialog box.

Producing a Dialog Box for Selecting a Format for Saving

You might want an application to save existing waveform-audio data in another format. For example, a waveform-audio editor could save a waveform-audio file as a compressed file. To list only the suggested destination formats for a given source format in the dialog box created by the **acmFormatChoose** function, specify the source format in the **pwfxEnum** member and the ACM_FORMATENUMF_SUGGEST flag in the **fdwEnum** member of the **ACMFORMATCHOOSE** structure.

Similarly, to list valid destination formats for a given format, use the ACM_FORMATENUMF_CONVERT flag instead of the ACM_FORMATENUMF_SUGGEST flag.

Producing a Dialog Box for Selecting a Format for Recording

An application can allow the user to select a format for which an installed waveform-audio device provides native support. For example, you might want a waveform-audio editor to record new waveform-audio data without using an ACM compressor or decompressor to provide a translation layer. To do this, use the **acmFormatChoose** function, specifying the ACM_FORMATENUMF_HARDWARE and ACM_FORMATENUMF_INPUT flags in the **fdwEnum** member of the **ACMFORMATCHOOSE** structure.

Converting Data from One Format to Another

The ACM uses stream functions to support data format conversion. Converters in the ACM change the format, but not the data type. For example, a converter module can change 44-kHz, 16-bit data to 44-kHz, 8-bit data.

The following ACM functions support data format conversion. They are listed in the order you would typically use them.

- The **acmStreamOpen** function opens a conversion stream.
- The **acmStreamSize** function calculates the appropriate size of the source or destination buffer.
- The **acmStreamPrepareHeader** function prepares source and destination buffers to be used in a conversion.
- The **acmStreamConvert** function converts data in a source buffer into the destination format, writing the converted data into the destination buffer.
- The **acmStreamUnprepareHeader** function cleans up the preparation performed by **acmStreamPrepareHeader**. You must call this function before freeing the source and destination buffers.
- The **acmStreamClose** function closes a conversion stream.

When converting data, first identify the source format, and then choose the destination format. The easiest way to do this is by using the **acmFormatChoose** function, which displays a format-selection dialog box and returns the user's choice.

When you know the source and destination formats, you can use **acmStreamOpen** to open a conversion stream. Then you can use the **acmStreamSize** function to determine the appropriate buffer sizes.

The next step is to prepare the buffers to be used in the conversion by using **acmStreamPrepareHeader**.

To perform the conversion, use **acmStreamConvert** until all the buffers have been processed. When the conversion is complete, use **acmStreamUnprepareHeader** to clean up the buffers and then use **acmStreamClose** to close the conversion stream.

Multistep Format Conversion

Sometimes the ACM cannot convert data from one format to another in a single step. For example, an application might need to convert 16-bit, 44-kHz stereo data to 11-kHz mono ADPCM. If the compressor or decompressor cannot do this conversion directly, the application might attempt it in two steps. This usually means making one conversion between two PCM formats and then another to the final format type.

To convert in two steps, use the **acmFormatSuggest** function to find a PCM format that matches the ADPCM format. Then use two conversion streams to perform the conversion. For example, perform one conversion from 16-bit, 44-kHz stereo PCM to 16-bit, 11-kHz mono and then convert from 16-bit, 11-kHz mono to 11-kHz mono ADPCM.

Multistep conversion also happens when either the source or the destination format is not PCM. If the source format is not PCM, it should be changed to a PCM format before conversion. If the destination format is not PCM, the source must be converted to an intermediate PCM format and then converted to the final destination format.

The most straightforward conversions occur when the source and destination formats are both PCM formats. When either the source or destination format is not PCM, the conversion might require an additional step. If both source and destination formats are not PCM, the conversion will usually require more than one step, and, in some instances, conversion might not be possible.

Finding a Specific Format

An application might have only a partial specification for a format when it needs the full specification. For example, the specification might stipulate an 11-kHz mono, 4-bit ADPCM format, but not the average bytes per second. The application can get the full format without user intervention by using the **acmFormatEnum** function and specifying flags in the *fdwEnum* parameter.

Finding a Specific Driver

You might want your application to send a message directly to a specific driver or to identify certain drivers from the list. For example, you might want your application to identify those drivers which support filters and then query each driver to determine which filter tags it supports. You can use the **acmDriverEnum** function to obtain a handle of the desired driver or drivers; this handle can then be used to communicate with that driver.

Note that when an application installs a local driver for its own use, the **acmDriverAdd** function returns a driver handle, which can be used to communicate with the driver. It is not necessary to use **acmDriverEnum** in this case.

Adding Drivers Within an Application

If you need your application to implement its own compression routines internally, the application can add drivers to the ACM by calling the **acmDriverAdd** function. The application implements the driver by providing a function that conforms to the **acmDriverProc** prototype. Once the application has added the driver, the application can use the driver through the ACM like any other driver.

The ACM treats drivers as either global or local. An application specifies whether a driver should be added as global or local when it calls **acmDriverAdd**. There are two differences between global and local drivers:

- For Win32 platforms, unlike the Microsoft Windows operating system versions 3.*x*, drivers added as global drivers are not shared with other applications.
- An application can directly alter the priority of a global driver (but not a local driver) by calling the **acmDriverPriority** function. The ACM conducts a prioritized search when seeking an appropriate driver to satisfy a function call. The ACM always gives local drivers higher priority than global drivers. The most recently added local driver has highest priority.

Generating a Nonstandard Format

Sometimes an application needs a nonstandard format. For example, an application might need a 16-kHz ADPCM-format file. Because 16 kHz is nonstandard, the enumeration functions will not generate this format. In fact, short of custom coding the format algorithms into the application, there is no reliable way to generate a nonstandard format. It is sometimes possible, however, to generate an analogous format by setting up a valid PCM format with all the required information and then using the **acmFormatSuggest** function. Because compressors and decompressors try to suggest a format that is closest to the desired format, the number of channels and frequency are usually preserved.

Audio Compression Manager Reference

This section describes the functions, messages, and structures associated with the ACM. These elements are grouped as follows.

Drivers

acmDriverAdd
acmDriverClose
ACMDRIVERDETAILS
ACMDRIVERDETAILS
acmDriverEnum
acmDriverEnumCallback
acmDriverID
acmDriverMessage
acmDriverOpen
acmDriverPriority
acmDriverProc
acmDriverRemove

Filters

ACMFILTERCHOOSE
ACMFILTERCHOOSE
acmFilterChooseHookProc
ACMFILTERDETAILS
ACMFILTERDETAILS
acmFilterEnum
acmFilterEnumCallback
ACMFILTERTAGDETAILS
ACMFILTERTAGDETAILS
acmFilterTagEnum
acmFilterTagEnumCallback
WAVEFILTER

Formats

ACMFORMATCHOOSE
ACMFORMATCHOOSE
acmFormatChooseHookProc
ACMFORMATDETAILS
ACMFORMATDETAILS
acmFormatEnum
acmFormatEnumCallback
acmFormatSuggest
ACMFORMATTAGDETAILS
ACMFORMATTAGDETAILS
acmFormatTagEnum
acmFormatTagEnumCallback

Messages

MM_ACM_FILTERCHOOSE
MM_ACM_FORMATCHOOSE

Miscellaneous

acmGetVersion
acmMetrics

Streams

acmDriverAdd

```
MMRESULT acmDriverAdd(LPHACMDRIVERID phadid, HINSTANCE hinstModule,
    LPARAM lParam, DWORD dwPriority, DWORD fdwAdd);
```

Adds a driver to the list of available ACM drivers. The driver type and location are dependent on the flags used to add ACM drivers. After a driver is successfully added, the driver entry function will receive ACM driver messages.

- Returns zero if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.
MMSYSERR_NOMEM	The system is unable to allocate resources.

phadid
Address that is filled with a handle identifying the installed driver. This handle is used to identify the driver in calls to other ACM functions.

hinstModule
Handle of the instance of the module whose executable or dynamic-link library (DLL) contains the driver entry function.

lParam
Driver function address or a notification window handle, depending on the *fdwAdd* flags.

dwPriority
Window message to send for notification broadcasts. This parameter is used only with the ACM_DRIVERADDF_NOTIFYHWND flag. All other flags require this member to be set to zero.

fdwAdd
Flags for adding ACM drivers. The following values are defined:

ACM_DRIVERADDF_FUNCTION
The *lParam* parameter is a driver function address conforming to the **acmDriverProc** prototype. The function may reside in either an executable or DLL file.

ACM_DRIVERADDF_GLOBAL
Provided for compatibility with 16-bit applications. For the Win32 API, ACM drivers added by the **acmDriverAdd** function can be used only by the application that added the driver. This is true whether or not ACM_DRIVERADDF_GLOBAL is specified. For more information, see "Adding Drivers Within an Application" earlier in this chapter.

ACM_DRIVERADDF_LOCAL
The ACM automatically gives a local driver higher priority than a global driver when searching for a driver to satisfy a function call. For more information, see "Adding Drivers Within an Application" earlier in this chapter.

ACM_DRIVERADDF_NOTIFYHWND
The *lParam* parameter is a handle of a notification window that receives messages when changes to global driver priorities and states are made. The window message to receive is defined by the application and must be passed in *dwPriority*. The *wParam* and *lParam* parameters passed with the window message are reserved for future use and should be ignored. ACM_DRIVERADDF_GLOBAL cannot be specified in conjunction with this flag. For more information about driver priorities, see the description for the **acmDriverPriority** function.

acmDriverClose

```
MMRESULT acmDriverClose(HACMDRIVER had, DWORD fdwClose);
```

Closes a previously opened ACM driver instance. If the function is successful, the handle is invalidated.

- Returns zero if successful or an error otherwise. Possible error values include the following:

ACMERR_BUSY	The driver is in use and cannot be closed.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.

had
Handle of the open driver instance to be closed.

fdwClose
Reserved; must be zero.

acmDriverDetails

```
MMRESULT acmDriverDetails(HACMDRIVERID hadid, LPACMDRIVERDETAILS padd,
    DWORD fdwDetails);
```

Queries a specified ACM driver to determine its capabilities.

- Returns zero if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.

hadid
Handle of the driver identifier of an installed ACM driver. Disabled drivers can be queried for details.

padd
Address of an **ACMDRIVERDETAILS** structure that will receive the driver details. The **cbStruct** member must be initialized to the size, in bytes, of the structure.

fdwDetails
Reserved; must be zero.

acmDriverEnum

```
MMRESULT acmDriverEnum(ACMDRIVERENUMCB fnCallback, DWORD dwInstance,
    DWORD fdwEnum);
```

Enumerates the available ACM drivers, continuing until there are no more drivers or the callback function returns FALSE.

- Returns zero if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.

fnCallback
Procedure instance address of the application-defined callback function.

dwInstance
A 32-bit application-defined value that is passed to the callback function along with ACM driver information.

fdwEnum
Flags for enumerating ACM drivers. The following values are defined:

ACM_DRIVERENUMF_DISABLED
Disabled ACM drivers should be included in the enumeration. Drivers can be disabled by the user through the Control Panel or by an application using the **acmDriverPriority** function. If a driver is disabled, the *fdwSupport* parameter to the callback function will have the ACMDRIVERDETAILS_SUPPORTF_DISABLED flag set.

ACM_DRIVERENUMF_NOLOCAL
Only global drivers should be included in the enumeration.

The **acmDriverEnum** function will return MMSYSERR_NOERROR (zero) if no ACM drivers are installed. Moreover, the callback function will not be called.

acmDriverEnumCallback

```
BOOL ACMDRIVERENUMCB acmDriverEnumCallback(HACMDRIVERID hadid,
    DWORD dwInstance, DWORD fdwSupport);
```

Specifies a callback function used with the **acmDriverEnum** function. The **acmDriverEnumCallback** name is a placeholder for an application-defined function name.

- The callback function must return TRUE to continue enumeration or FALSE to stop enumeration.

hadid
Handle of an ACM driver identifier.

dwInstance
Application-defined value specified in **acmDriverEnum**.

fdwSupport
Driver-support flags specific to the driver specified by *hadid*. These flags are identical to the **fdwSupport** flags of the **ACMDRIVERDETAILS** structure. This parameter can be a combination of the following values:

ACMDRIVERDETAILS_SUPPORTF_ASYNC
Driver supports asynchronous conversions.

ACMDRIVERDETAILS_SUPPORTF_CODEC
Driver supports conversion between two different format tags. For example, if a driver supports compression from WAVE_FORMAT_PCM to WAVE_FORMAT_ADPCM, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_CONVERTER
Driver supports conversion between two different formats of the same format tag. For example, if a driver supports resampling of WAVE_FORMAT_PCM, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_DISABLED
Driver has been disabled. An application must specify the ACM_DRIVERENUMF_DISABLED flag with **acmDriverEnum** to include disabled drivers in the enumeration.

ACMDRIVERDETAILS_SUPPORTF_FILTER
Driver supports a filter (modification of the data without changing any of the format attributes). For example, if a driver supports volume or echo operations on WAVE_FORMAT_PCM, this flag is set.

The **acmDriverEnum** function will return MMSYSERR_NOERROR (zero) if no ACM drivers are installed. Moreover, the callback function will not be called.

The following functions should not be called from within the callback function: **acmDriverAdd**, **acmDriverRemove**, and **acmDriverPriority**.

acmDriverID

```
MMRESULT acmDriverID(HACMOBJ hao, LPHACMDRIVERID phadid,
    DWORD fdwDriverID);
```

Returns the handle of an ACM driver identifier associated with an open ACM driver instance or stream handle.

- Returns zero if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.

hao
Handle of the open driver instance or stream handle. This is the handle of an ACM object, such **HACMDRIVER** or **HACMSTREAM**.

phadid
Address that is filled with a handle identifying the installed driver that is associated with *hao*.

fdwDriverID
Reserved; must be zero.

acmDriverMessage

```
LRESULT acmDriverMessage(HACMDRIVER had, UINT uMsg, LPARAM lParam1,
    LPARAM lParam2);
```

Sends a user-defined message to a given ACM driver instance.

- The return value is specific to the user-defined ACM driver message specified by the *uMsg* parameter. However, possible error values include the following:

MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	The *uMsg* parameter is not in the ACMDM_USER range.
MMSYSERR_NOTSUPPORTED	The ACM driver did not process the message.

had
Handle of the ACM driver instance to which the message will be sent.

uMsg
Message that the ACM driver must process. This message must be in the ACMDM_USER message range (above or equal to ACMDM_USER and less than ACMDM_RESERVED_LOW). The exceptions to this restriction are the ACMDM_DRIVER_ABOUT, DRV_QUERYCONFIGURE, and DRV_CONFIGURE messages.

lParam1 and *lParam2*
Message parameters.

To display a custom About dialog box from an ACM driver, an application must send the ACMDM_DRIVER_ABOUT message to the driver. The *lParam1* parameter should be the handle of the owner window for the custom About dialog box, and *lParam2* must be set to zero. If the driver does not support a custom About dialog box, MMSYSERR_NOTSUPPORTED will be returned and it is the application's responsibility to display its own dialog box. For example, the Control Panel Sound Mapper option will display a default About dialog box based on the **ACMDRIVERDETAILS** structure when an ACM driver returns MMSYSERR_NOTSUPPORTED. An application can query a driver for custom About dialog box support without the dialog box being displayed by setting *lParam1* to - 1L. If the driver supports a custom About dialog box, MMSYSERR_NOERROR will be returned. Otherwise, the return value is MMSYSERR_NOTSUPPORTED.

User-defined messages must be sent only to an ACM driver that specifically supports the messages. The caller should verify that the ACM driver is the correct driver by retrieving the driver details and checking the **wMid**, **wPid**, and **vdwDriver** members of the **ACMDRIVERDETAILS** structure.

Never send user-defined messages to an unknown ACM driver.

acmDriverOpen

```
MMRESULT acmDriverOpen(LPHACMDRIVER phad, HACMDRIVERID hadid,
    DWORD fdwOpen);
```

Opens the specified ACM driver and returns a driver instance handle that can be used to communicate with the driver.

- Returns zero if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.
MMSYSERR_NOMEM	The system is unable to allocate resources.
MMSYSERR_NOTENABLED	The driver is not enabled.

phad
Address that will receive the new driver instance handle that can be used to communicate with the driver.

hadid
Handle of the driver identifier of an installed and enabled ACM driver.

fdwOpen
Reserved; must be zero.

acmDriverPriority

```
MMRESULT acmDriverPriority(HACMDRIVERID hadid, DWORD dwPriority,
    DWORD fdwPriority);
```

Modifies the priority and state of an ACM driver.

- Returns zero if successful or an error otherwise. Possible error values include the following:

MMSYSERR_ALLOCATED	The deferred broadcast lock is owned by a different task.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.
MMSYSERR_NOTSUPPORTED	The requested operation is not supported for the specified driver. For example, local and notify driver identifiers do not support priorities (but can be enabled and disabled). If an application specifies a nonzero value for *dwPriority* for local and notify driver identifiers, this error will be returned.

hadid
Handle of the driver identifier of an installed ACM driver. If the ACM_DRIVERPRIORITYF_BEGIN and ACM_DRIVERPRIORITYF_END flags are specified, this parameter must be NULL.

dwPriority
New priority for a global ACM driver identifier. A zero value specifies that the priority of the driver identifier should remain unchanged. A value of 1 specifies that the driver should be placed as the highest search priority driver. A value of - 1 specifies that the driver should be placed as the lowest search priority driver. Priorities are used only for global drivers.

fdwPriority
Flags for setting priorities of ACM drivers. The following values are defined:

ACM_DRIVERPRIORITYF_BEGIN
Change notification broadcasts should be deferred. An application must reenable notification broadcasts as soon as possible with the ACM_DRIVERPRIORITYF_END flag. Note that *hadid* must be NULL, *dwPriority* must be zero, and only the ACM_DRIVERPRIORITYF_BEGIN flag can be set.

ACM_DRIVERPRIORITYF_DISABLE
ACM driver should be disabled if it is currently enabled. Disabling a disabled driver does nothing.

ACM_DRIVERPRIORITYF_ENABLE
ACM driver should be enabled if it is currently disabled. Enabling an enabled driver does nothing.

ACM_DRIVERPRIORITYF_END
Calling task wants to reenable change notification broadcasts. An application must call **acmDriverPriority** with ACM_DRIVERPRIORITYF_END for each successful call with the ACM_DRIVERPRIORITYF_BEGIN flag. Note that *hadid* must be NULL, *dwPriority* must be zero, and only the ACM_DRIVERPRIORITYF_END flag can be set.

All driver identifiers can be enabled and disabled, including global, local and notification driver identifiers.

If more than one global driver identifier needs to be enabled, disabled or shifted in priority, an application should defer change notification broadcasts by using the

ACM_DRIVERPRIORITYF_BEGIN flag. A single change notification will be broadcast when the ACM_DRIVERPRIORITYF_END flag is specified.

An application can use the **acmMetrics** function with the ACM_METRIC_DRIVER_PRIORITY metric index to retrieve the current priority of a global driver. Drivers are always enumerated from highest to lowest priority by the **acmDriverEnum** function.

All enabled driver identifiers will receive change notifications. An application can register a notification message by using the **acmDriverAdd** function in conjunction with the ACM_DRIVERADDF_NOTIFYHWND flag. Changes to nonglobal driver identifiers will not be broadcast.

Priorities are simply used for the search order when an application does not specify a driver. Boosting the priority of a driver will have no effect on the performance of a driver.

acmDriverProc

```
LRESULT CALLBACK acmDriverProc(DWORD dwID, HDRIVER hdrvr, UINT uMsg,
    LPARAM lParam1, LPARAM lParam2);
```

Specifies a callback function used with the ACM driver. The **acmDriverProc** name is a placeholder for an application-defined function name. The actual name must be exported by including it in the module-definition file of the executable or DLL file.

dwID
Identifier of the installable ACM driver.

hdrvr
Handle of the installable ACM driver. This parameter is a unique handle the ACM assigns to the driver.

uMsg
ACM driver message.

lParam1 and *lParam2*
Message parameters.

Applications should not call any system-defined functions from inside a callback function, except for **PostMessage**, **timeGetSystemTime**, **timeGetTime**, **timeSetEvent**, **timeKillEvent**, **midiOutShortMsg**, **midiOutLongMsg**, and **OutputDebugStr**.

acmDriverRemove

```
MMRESULT acmDriverRemove(HACMDRIVERID hadid, DWORD fdwRemove);
```

Removes an ACM driver from the list of available ACM drivers. The driver will be removed for the calling application only. If the driver is globally installed, other applications will still be able to use it.

- Returns zero if successful or an error otherwise. Possible error values include the following:

ACMERR_BUSY	The driver is in use and cannot be removed.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.

hadid
Handle of the driver identifier to be removed.

fdwRemove
Reserved; must be zero.

acmFilterChoose

```
MMRESULT acmFilterChoose(LPACMFILTERCHOOSE pafltrc);
```

Creates an ACM-defined dialog box that enables the user to select a waveform-audio filter.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

ACMERR_CANCELED	The user chose the Cancel button or the Close command on the System menu to close the dialog box.
ACMERR_NOTPOSSIBLE	The buffer identified by the **pwfltr** member of the **ACMFILTERCHOOSE** structure is too small to contain the selected filter.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.
MMSYSERR_NODRIVER	A suitable driver is not available to provide valid filter selections.

pafltrc

Address of an **ACMFILTERCHOOSE** structure that contains information used to initialize the dialog box. When **acmFilterChoose** returns, this structure contains information about the user's filter selection.

The **pwfltr** member of this structure must contain a valid pointer to a memory location that will contain the returned filter header structure. The **cbwfltr** member must be filled in with the size, in bytes, of this memory buffer.

acmFilterChooseHookProc

```
UINT ACMFILTERCHOOSEHOOKPROC acmFilterChooseHookProc(HWND hwnd,
    UINT uMsg, WPARAM wParam, LPARAM lParam);
```

Specifies a user-defined function that hooks the **acmFilterChoose** dialog box.

hwnd
Window handle for the dialog box.

uMsg
Window message.

wParam and *lParam*
Message parameters.

To customize the dialog box selections, a hook function can optionally process the MM_ACM_FILTERCHOOSE message.

You should use this function the same way as you use the Common Dialog hook functions for customizing common dialog boxes.

acmFilterDetails

```
MMRESULT acmFilterDetails(HACMDRIVER had, LPACMFILTERDETAILS pafd,
    DWORD fdwDetails);
```

Queries the ACM for details about a filter with a specific waveform-audio filter tag.

- Returns zero if successful or an error otherwise. Possible error values include the following:

ACMERR_NOTPOSSIBLE	The details requested are not available.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.

had
Handle of the ACM driver to query for waveform-audio filter details for a filter tag. If this parameter is NULL, the ACM uses the details from the first suitable ACM driver.

pafd
Address of the **ACMFILTERDETAILS** structure that is to receive the filter details for the given filter tag.

fdwDetails
Flags for getting the details. The following values are defined:

ACM_FILTERDETAILSF_FILTER
A **WAVEFILTER** structure pointed to by the **pwfltr** member of the **ACMFILTERDETAILS** structure was given and the remaining details should be returned. The **dwFilterTag** member of the **ACMFILTERDETAILS** structure must be initialized to the same filter tag **pwfltr** specifies. This query type can be used to get a string description of an arbitrary filter structure. If an application specifies an ACM driver handle for *had*, details on the filter will be returned for that driver. If an application specifies NULL for *had*, the ACM finds the first acceptable driver to return the details.

ACM_FILTERDETAILSF_INDEX
A filter index for the filter tag was given in the **dwFilterIndex** member of the **ACMFILTERDETAILS** structure. The filter details will be returned in the structure defined by *pafd*. The index ranges from zero to one less than the **cStandardFilters** member returned in the **ACMFILTERTAGDETAILS** structure for a filter tag. An application must specify a driver handle for *had* when retrieving filter details with this flag. For information about what members should be initialized before calling this function, see the **ACMFILTERDETAILS** structure.

acmFilterEnum

```
MMRESULT acmFilterEnum(HACMDRIVER had, LPACMFILTERDETAILS pafd,
    ACMFILTERENUMCB fnCallback, DWORD dwInstance, DWORD fdwEnum);
```

Enumerates waveform-audio filters available for a given filter tag from an ACM driver. This function continues enumerating until there are no more suitable filters for the filter tag or the callback function returns FALSE.

- Returns zero if successful or an error otherwise. Possible error values include the following:

ACMERR_NOTPOSSIBLE	The details for the filter cannot be returned.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.

had
Handle of the ACM driver to query for waveform-audio filter details. If this parameter is NULL, the ACM uses the details from the first suitable ACM driver.

pafd
Address of the **ACMFILTERDETAILS** structure that contains the filter details when it is passed to the function specified by *fnCallback*. When your application calls **acmFilterEnum**, the **cbStruct**, **pwfltr**, and **cbwfltr** members of this structure must be initialized. The **dwFilterTag** member must also be initialized to either WAVE_FILTER_UNKNOWN or a valid filter tag.

fnCallback
Procedure-instance address of the application-defined callback function.

dwInstance
A 32-bit, application-defined value that is passed to the callback function along with ACM filter details.

fdwEnum
Flags for enumerating the filters for a given filter tag. The following values are defined:

ACM_FILTERENUMF_DWFILTERTAG
The **dwFilterTag** member of the **WAVEFILTER** structure pointed to by the **pwfltr** member of the **ACMFILTERDETAILS** structure is valid. The enumerator will enumerate only a filter that conforms to this attribute. The **dwFilterTag** member of the **ACMFILTERDETAILS** structure must be equal to the **dwFilterTag** member of the **WAVEFILTER** structure.

The **acmFilterEnum** function will return MMSYSERR_NOERROR (zero) if no suitable ACM drivers are installed. Moreover, the callback function will not be called.

The following functions should not be called from within the callback function: **acmDriverAdd**, **acmDriverRemove**, **acmDriverPriority**.

acmFilterEnumCallback

```
BOOL ACMFILTERENUMCB acmFilterEnumCallback(HACMDRIVERID hadid,
    LPACMFILTERDETAILS pafd, DWORD dwInstance, DWORD fdwSupport);
```

Specifies a callback function used with the **acmFilterEnum** function. The **acmFilterEnumCallback** name is a placeholder for an application-defined function name.

- The callback function must return TRUE to continue enumeration or FALSE to stop enumeration.

hadid
Handle of the ACM driver identifier.

pafd
Address of an **ACMFILTERDETAILS** structure that contains the enumerated filter details for a filter tag.

dwInstance
Application-defined value specified in **acmFilterEnum**.

fdwSupport
Driver-support flags specific to the driver identified by *hadid* for the specified filter. These flags are identical to the **fdwSupport** flags of the **ACMDRIVERDETAILS** structure, but they are specific to the filter that is being enumerated. This parameter can be a combination of the following values and identifies which operations the driver supports for the filter tag:

ACMDRIVERDETAILS_SUPPORTF_ASYNC
Driver supports asynchronous conversions with the specified filter tag.

ACMDRIVERDETAILS_SUPPORTF_CODEC
Driver supports conversion between two different format tags while using the specified filter. For example, if a driver supports compression from WAVE_FORMAT_PCM to WAVE_FORMAT_ADPCM with the specified filter, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_CONVERTER
Driver supports conversion between two different formats of the same format tag while using the specified filter. For example, if a driver supports resampling of WAVE_FORMAT_PCM with the specified filter, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_FILTER
Driver supports a filter (modification of the data without changing any of the format attributes). For example, if a driver supports volume or echo operations on WAVE_FORMAT_PCM, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_HARDWARE
Driver supports hardware input, output, or both with the specified filter through a waveform-audio device. An application should use the **acmMetrics** function with the ACM_METRIC_HARDWARE_WAVE_INPUT and ACM_METRIC_HARDWARE_WAVE_OUTPUT metric indices to get the waveform-audio device identifiers associated with the supporting ACM driver.

The **acmFilterEnum** function will return MMSYSERR_NOERROR (zero) if no filters are to be enumerated. Moreover, the callback function will not be called.

The following functions should not be called from within the callback function: **acmDriverAdd**, **acmDriverRemove**, **acmDriverPriority**.

acmFilterTagDetails

```
MMRESULT acmFilterTagDetails(HACMDRIVER had,
    LPACMFILTERTAGDETAILS paftd, DWORD fdwDetails);
```

Queries the ACM for details about a specific waveform-audio filter tag.

- Returns zero if successful or an error otherwise. Possible error values include the following:

ACMERR_NOTPOSSIBLE	The details requested are not available.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.

had
Handle of the ACM driver to query for waveform-audio filter tag details. If this parameter is NULL, the ACM uses the details from the first suitable ACM driver. An application must specify a valid **HACMDRIVER** or **HACMDRIVERID** identifier when using the ACM_FILTERTAGDETAILSF_INDEX query type. Driver identifiers for disabled drivers are not allowed.

paftd
Address of the **ACMFILTERTAGDETAILS** structure that is to receive the filter tag details.

fdwDetails
Flags for getting the details. The following values are defined:

ACM_FILTERTAGDETAILSF_FILTERTAG
A filter tag was given in the **dwFilterTag** member of the **ACMFILTERTAGDETAILS** structure. The filter tag details will be returned in the structure pointed to by *paftd*. If an application specifies an ACM driver handle for *had*, details on the filter tag will be returned for that driver. If an application specifies NULL for *had*, the ACM finds the first acceptable driver to return the details.

ACM_FILTERTAGDETAILSF_INDEX
A filter tag index was given in the **dwFilterTagIndex** member of the **ACMFILTERTAGDETAILS** structure. The filter tag and details will be returned in the structure pointed to by *paftd*. The index ranges from zero to one less than the **cFilterTags** member returned in the **ACMDRIVERDETAILS** structure for an ACM driver. An application must specify a driver handle for *had* when retrieving filter tag details with this flag.

ACM_FILTERTAGDETAILSF_LARGESTSIZE
Details on the filter tag with the largest filter size, in bytes, are to be returned. The **dwFilterTag** member must either be WAVE_FILTER_UNKNOWN or the filter tag to find the largest size for. If an application specifies an ACM driver handle for *had*, details on the largest filter tag will be returned for that driver. If an application specifies NULL for *had*, the ACM finds an acceptable driver with the largest filter tag requested to return the details.

acmFilterTagEnum

```
MMRESULT acmFilterTagEnum(HACMDRIVER had, LPACMFILTERTAGDETAILS paftd,
    ACMFILTERTAGENUMCB fnCallback, DWORD dwInstance, DWORD fdwEnum);
```

Enumerates waveform-audio filter tags available from an ACM driver. This function continues enumerating until there are no more suitable filter tags or the callback function returns FALSE.

- Returns zero if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.

had
Handle of the ACM driver to query for waveform-audio filter tag details. If this parameter is NULL, the ACM uses the details from the first suitable ACM driver.

paftd
Address of the **ACMFILTERTAGDETAILS** structure that contains the filter tag details when it is passed to the *fnCallback* function. When your application calls **acmFilterTagEnum**, the **cbStruct** member of this structure must be initialized.

fnCallback
Procedure instance address of the application-defined callback function.

dwInstance
A 32-bit application-defined value that is passed to the callback function along with ACM filter tag details.

fdwEnum
Reserved; must be zero.

This function will return MMSYSERR_NOERROR (zero) if no suitable ACM drivers are installed. Moreover, the callback function will not be called.

acmFilterTagEnumCallback

```
BOOL ACMFILTERTAGENUMCB acmFilterTagEnumCallback(HACMDRIVERID hadid,
    LPACMFILTERTAGDETAILS paftd, DWORD dwInstance, DWORD fdwSupport);
```

Specifies a callback function used with the **acmFilterTagEnum** function. The **acmFilterTagEnumCallback** function name is a placeholder for an application-defined function name.

- The callback function must return TRUE to continue enumeration or FALSE to stop enumeration.

hadid
Handle of the ACM driver identifier.

paftd
Address of an **ACMFILTERTAGDETAILS** structure that contains the enumerated filter tag details.

dwInstance
Application-defined value specified in **acmFilterTagEnum**.

fdwSupport
Driver-support flags specific to the driver identifier *hadid*. These flags are identical to the **fdwSupport** flags of the **ACMDRIVERDETAILS** structure. This parameter can be a combination of the following values and identifies which operations the driver supports with the filter tag:

ACMSDRIVERDETAILS_SUPPORTF_ASYNC
Driver supports asynchronous conversions with the specified filter tag.

ACMDRIVERDETAILS_SUPPORTF_CODEC
Driver supports conversion between two different format tags while using the specified filter tag. For example, if a driver supports compression from WAVE_FORMAT_PCM to WAVE_FORMAT_ADPCM with the specified filter tag, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_CONVERTER
Driver supports conversion between two different formats of the same format tag while using the specified filter tag. For example, if a driver supports resampling of WAVE_FORMAT_PCM with the specified filter tag, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_FILTER
Driver supports a filter (modification of the data without changing any of the format attributes). For example, if a driver supports volume or echo operations on WAVE_FORMAT_PCM, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_HARDWARE
Driver supports hardware input, output, or both with the specified filter tag through a waveform-audio device. An application should use the **acmMetrics** function with the ACM_METRIC_HARDWARE_WAVE_INPUT and ACM_METRIC_HARDWARE_WAVE_OUTPUT metric indices to get the waveform-audio device identifiers associated with the supporting ACM driver.

The **acmFilterTagEnum** function will return MMSYSERR_NOERROR (zero) if no filter tags are to be enumerated. Moreover, the callback function will not be called.

The following functions should not be called from within the callback function: **acmDriverAdd**, **acmDriverRemove**, and **acmDriverPriority**.

acmFormatChoose

```
MMRESULT acmFormatChoose(LPACMFORMATCHOOSE pfmtc);
```

Creates an ACM-defined dialog box that enables the user to select a waveform-audio format.

- Returns MMSYSERR_NOERROR if the function was successful or an error otherwise. Possible error values include the following:

ACMERR_CANCELED	The user chose the Cancel button or the Close command on the System menu to close the dialog box.
ACMERR_NOTPOSSIBLE	The buffer identified by the **pwfx** member of the **ACMFORMATCHOOSE** structure is too small to contain the selected format.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.
MMSYSERR_NODRIVER	A suitable driver is not available to provide valid format selections.

pfmtc

Address of an **ACMFORMATCHOOSE** structure that contains information used to initialize the dialog box. When this function returns, this structure contains information about the user's format selection.

The **pwfx** member of this structure must contain a valid pointer to a memory location that will contain the returned format header structure. Moreover, the **cbwfx** member must be filled in with the size, in bytes, of this memory buffer.

acmFormatChooseHookProc

```
UINT ACMFORMATCHOOSEHOOKPROC acmFormatChooseHookProc(HWND hwnd,
    UINT uMsg, WPARAM wParam, LPARAM lParam);
```

Specifies a user-defined function that hooks the **acmFormatChoose** dialog box. The **acmFormatChooseHookProc** name is a placeholder for an application-defined name.

hwnd
Window handle for the dialog box.

uMsg
Window message.

wParam and *lParam*
Message parameters.

If the hook function processes one of the WM_CTLCOLOR messages, this function must return a handle of the brush that should be used to paint the control background.

A hook function can optionally process the MM_ACM_FORMATCHOOSE message.

You should use this function the same way as you use the Common Dialog hook functions for customizing common dialog boxes.

acmFormatDetails

```
MMRESULT acmFormatDetails(HACMDRIVER had, LPACMFORMATDETAILS pafd,
    DWORD fdwDetails);
```

Queries the ACM for format details for a specific waveform-audio format tag.

- Returns zero if successful or an error otherwise. Possible error values include the following:

ACMERR_NOTPOSSIBLE	The details requested are not available.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.

had
Handle of the ACM driver to query for waveform-audio format details for a format tag. If this parameter is NULL, the ACM uses the details from the first suitable ACM driver.

pafd
Address of an **ACMFORMATDETAILS** structure to contain the format details for the given format tag.

fdwDetails
Flags for getting the waveform-audio format tag details. The following values are defined:

ACM_FORMATDETAILSF_FORMAT
A **WAVEFORMATEX** structure pointed to by the **pwfx** member of the **ACMFORMATDETAILS** structure was given and the remaining details should be returned. The **dwFormatTag** member of the **ACMFORMATDETAILS** structure must be initialized to the same format tag as **pwfx** specifies. This query type can be used to get a string description of an arbitrary format structure. If an application specifies an ACM driver handle for *had*, details on the format will be returned for that driver. If an application specifies NULL for *had*, the ACM finds the first acceptable driver to return the details.

ACM_FORMATDETAILSF_INDEX
A format index for the format tag was given in the **dwFormatIndex** member of the **ACMFORMATDETAILS** structure. The format details will be returned in the structure defined by *pafd*. The index ranges from zero to one less than the **cStandardFormats** member returned in the **ACMFORMATTAGDETAILS** structure for a format tag. An application must specify a driver handle for *had* when retrieving format details with this flag. For information about which members should be initialized before calling this function, see the **ACMFORMATDETAILS** structure.

acmFormatEnum

```
MMRESULT acmFormatEnum(HACMDRIVER had, LPACMFORMATDETAILS pafd,
    ACMFORMATENUMCB fnCallback, DWORD dwInstance, DWORD fdwEnum);
```

Enumerates waveform-audio formats available for a given format tag from an ACM driver. This function continues enumerating until there are no more suitable formats for the format tag or the callback function returns FALSE.

- Returns zero if successful or an error otherwise. Possible error values include the following:

ACMERR_NOTPOSSIBLE	The details for the format cannot be returned.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.

had
Handle of the ACM driver to query for waveform-audio format details. If this parameter is NULL, the ACM uses the details from the first suitable ACM driver.

pafd
Address of an **ACMFORMATDETAILS** structure to contain the format details passed to the *fnCallback* function. This structure must have the **cbStruct**, **pwfx**, and **cbwfx** members of the **ACMFORMATDETAILS** structure initialized. The **dwFormatTag** member must also be initialized to either WAVE_FORMAT_UNKNOWN or a valid format tag.

fnCallback
Procedure instance address of the application-defined callback function.

dwInstance
A 32-bit application-defined value that is passed to the callback function along with ACM format details.

fdwEnum
Flags for enumerating the formats for a given format tag. The following values are defined:

ACM_FORMATENUMF_CONVERT
The **WAVEFORMATEX** structure pointed to by the **pwfx** member of the **ACMFORMATDETAILS** structure is valid. The enumerator will only enumerate destination formats that can be converted from the given **pwfx** format.

ACM_FORMATENUMF_HARDWARE
The enumerator should only enumerate formats that are supported as native input or output formats on one or more of the installed waveform-audio devices. This flag provides a way for an application to choose only formats native to an installed waveform-audio device. This flag must be used with one or both of the ACM_FORMATENUMF_INPUT and ACM_FORMATENUMF_OUTPUT flags. Specifying both ACM_FORMATENUMF_INPUT and ACM_FORMATENUMF_OUTPUT will enumerate only formats that can be opened for input or output. This is true regardless of whether this flag is specified.

ACM_FORMATENUMF_INPUT
Enumerator should enumerate only formats that are supported for input (recording).

ACM_FORMATENUMF_NCHANNELS
The **nChannels** member of the **WAVEFORMATEX** structure pointed to by the **pwfx** member of the **ACMFORMATDETAILS** structure is valid. The enumerator will enumerate only a format that conforms to this attribute.

ACM_FORMATENUMF_NSAMPLESPERSEC
The **nSamplesPerSec** member of the **WAVEFORMATEX** structure pointed to by the **pwfx** member of the **ACMFORMATDETAILS** structure is valid. The enumerator will enumerate only a

format that conforms to this attribute.

ACM_FORMATENUMF_OUTPUT
Enumerator should enumerate only formats that are supported for output (playback).

ACM_FORMATENUMF_SUGGEST
The **WAVEFORMATEX** structure pointed to by the **pwfx** member of the **ACMFORMATDETAILS** structure is valid. The enumerator will enumerate all suggested destination formats for the given **pwfx** format. This mechanism can be used instead of the **acmFormatSuggest** function to allow an application to choose the best suggested format for conversion. The **dwFormatIndex** member will always be set to zero on return.

ACM_FORMATENUMF_WBITSPERSAMPLE
The **wBitsPerSample** member of the **WAVEFORMATEX** structure pointed to by the **pwfx** member of the **ACMFORMATDETAILS** structure is valid. The enumerator will enumerate only a format that conforms to this attribute.

ACM_FORMATENUMF_WFORMATTAG
The **wFormatTag** member of the **WAVEFORMATEX** structure pointed to by the **pwfx** member of the **ACMFORMATDETAILS** structure is valid. The enumerator will enumerate only a format that conforms to this attribute. The **dwFormatTag** member of the **ACMFORMATDETAILS** structure must be equal to the **wFormatTag** member.

This function will return MMSYSERR_NOERROR (zero) if no suitable ACM drivers are installed. Moreover, the callback function will not be called.

acmFormatEnumCallback

```
BOOL ACMFORMATENUMCB acmFormatEnumCallback(HACMDRIVERID hadid,
    LPACMFORMATDETAILS pafd, DWORD dwInstance, DWORD fdwSupport);
```

Specifies a callback function used with the **acmFormatEnum** function. The **acmFormatEnumCallback** name is a placeholder for the application-defined function name.

- The callback function must return TRUE to continue enumeration or FALSE to stop enumeration.

hadid
Handle of the ACM driver identifier.

pafd
Address of an **ACMFORMATDETAILS** structure that contains the enumerated format details for a format tag.

dwInstance
Application-defined value specified in the **acmFormatEnum** function.

fdwSupport
Driver support flags specific to the driver identified by *hadid* for the specified format. These flags are identical to the **fdwSupport** flags of the **ACMDRIVERDETAILS** structure, but they are specific to the format that is being enumerated. This parameter can be a combination of the following values and indicates which operations the driver supports for the format tag:

ACMDRIVERDETAILS_SUPPORTF_ASYNC
Driver supports asynchronous conversions with the specified filter tag.

ACMDRIVERDETAILS_SUPPORTF_CODEC
Driver supports conversion between two different format tags for the specified format. For example, if a driver supports compression from WAVE_FORMAT_PCM to WAVE_FORMAT_ADPCM with the specified format, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_CONVERTER
Driver supports conversion between two different formats of the same format tag while using the specified format. For example, if a driver supports resampling of WAVE_FORMAT_PCM to the specified format, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_FILTER
Driver supports a filter (modification of the data without changing any of the format attributes) with the specified format. For example, if a driver supports volume or echo operations on WAVE_FORMAT_PCM, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_HARDWARE
Driver supports hardware input, output, or both of the specified format tags through a waveform-audio device. An application should use the **acmMetrics** function with the ACM_METRIC_HARDWARE_WAVE_INPUT and ACM_METRIC_HARDWARE_WAVE_OUTPUT metric indexes to get the waveform-audio device identifiers associated with the supporting ACM driver.

The **acmFormatEnum** function will return MMSYSERR_NOERROR (zero) if no formats are to be enumerated. Moreover, the callback function will not be called.

The following functions should not be called from within the callback function: **acmDriverAdd**, **acmDriverRemove**, and **acmDriverPriority**.

acmFormatSuggest

```
MMRESULT acmFormatSuggest(HACMDRIVER had, LPWAVEFORMATEX pwfxSrc,
    LPWAVEFORMATEX pwfxDst, DWORD cbwfxDst, DWORD fdwSuggest);
```

Queries the ACM or a specified ACM driver to suggest a destination format for the supplied source format. For example, an application can use this function to determine one or more valid PCM formats to which a compressed format can be decompressed.

- Returns zero if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.

had
Handle of an open instance of a driver to query for a suggested destination format. If this parameter is NULL, the ACM attempts to find the best driver to suggest a destination format.

pwfxSrc
Address of a **WAVEFORMATEX** structure that identifies the source format for which a destination format will be suggested by the ACM or specified driver.

pwfxDst
Address of a **WAVEFORMATEX** structure that will receive the suggested destination format for the *pwfxSrc* format. Depending on the *fdwSuggest* parameter, some members of the structure pointed to by *pwfxDst* may require initialization.

cbwfxDst
Size, in bytes, available for the destination format. The **acmMetrics** and **acmFormatTagDetails** functions can be used to determine the maximum size required for any format available for the specified driver (or for all installed ACM drivers).

fdwSuggest
Flags for matching the desired destination format. The following values are defined:

ACM_FORMATSUGGESTF_NCHANNELS
The **nChannels** member of the structure pointed to by *pwfxDst* is valid. The ACM will query acceptable installed drivers that can suggest a destination format matching **nChannels** or fail.

ACM_FORMATSUGGESTF_NSAMPLESPERSEC
The **nSamplesPerSec** member of the structure pointed to by *pwfxDst* is valid. The ACM will query acceptable installed drivers that can suggest a destination format matching **nSamplesPerSec** or fail.

ACM_FORMATSUGGESTF_WBITSPERSAMPLE
The **wBitsPerSample** member of the structure pointed to by *pwfxDst* is valid. The ACM will query acceptable installed drivers that can suggest a destination format matching **wBitsPerSample** or fail.

ACM_FORMATSUGGESTF_WFORMATTAG
The **wFormatTag** member of the structure pointed to by *pwfxDst* is valid. The ACM will query acceptable installed drivers that can suggest a destination format matching **wFormatTag** or fail.

acmFormatTagDetails

```
MMRESULT acmFormatTagDetails(HACMDRIVER had,
    LPACMFORMATTAGDETAILS paftd, DWORD fdwDetails);
```

Queries the ACM for details on a specific waveform-audio format tag.

- Returns zero if successful or an error otherwise. Possible error values include the following:

ACMERR_NOTPOSSIBLE	The details requested are not available.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.

had
Handle of the ACM driver to query for waveform-audio format tag details. If this parameter is NULL, the ACM uses the details from the first suitable ACM driver. An application must specify a valid handle or driver identifier when using the ACM_FORMATTAGDETAILSF_INDEX query type. Driver identifiers for disabled drivers are not allowed.

paftd
Address of the **ACMFORMATTAGDETAILS** structure that is to receive the format tag details.

fdwDetails
Flags for getting the details. The following values are defined:

ACM_FORMATTAGDETAILSF_FORMATTAG
A format tag was given in the **dwFormatTag** member of the **ACMFORMATTAGDETAILS** structure. The format tag details will be returned in the structure pointed to by *paftd*. If an application specifies an ACM driver handle for *had*, details on the format tag will be returned for that driver. If an application specifies NULL for *had*, the ACM finds the first acceptable driver to return the details.

ACM_FORMATTAGDETAILSF_INDEX
A format tag index was given in the **dwFormatTagIndex** member of the **ACMFORMATTAGDETAILS** structure. The format tag and details will be returned in the structure defined by *paftd*. The index ranges from zero to one less than the **cFormatTags** member returned in the **ACMDRIVERDETAILS** structure for an ACM driver. An application must specify a driver handle for *had* when retrieving format tag details with this flag.

ACM_FORMATTAGDETAILSF_LARGESTSIZE
Details on the format tag with the largest format size, in bytes, are to be returned. The **dwFormatTag** member of the **ACMFORMATTAGDETAILS** structure must either be WAVE_FORMAT_UNKNOWN or the format tag to find the largest size for. If an application specifies an ACM driver handle for *had*, details on the largest format tag will be returned for that driver. If an application specifies NULL for *had*, the ACM finds an acceptable driver with the largest format tag requested to return the details.

acmFormatTagEnum

```
MMRESULT acmFormatTagEnum(HACMDRIVER had, LPACMFORMATTAGDETAILS paftd,
    ACMFORMATTAGENUMCB fnCallback, DWORD dwInstance, DWORD fdwEnum);
```

Enumerates waveform-audio format tags available from an ACM driver. This function continues enumerating until there are no more suitable format tags or the callback function returns FALSE.

- Returns zero if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.

had
Handle of the ACM driver to query for waveform-audio format tag details. If this parameter is NULL, the ACM uses the details from the first suitable ACM driver.

paftd
Address of the **ACMFORMATTAGDETAILS** structure that is to receive the format tag details passed to the function specified in *fnCallback*. This structure must have the **cbStruct** member of the **ACMFORMATTAGDETAILS** structure initialized.

fnCallback
Procedure instance address of the application-defined callback function.

dwInstance
A 32-bit application-defined value that is passed to the callback function along with ACM format tag details.

fdwEnum
Reserved; must be zero.

This function will return MMSYSERR_NOERROR (zero) if no suitable ACM drivers are installed. Moreover, the callback function will not be called.

acmFormatTagEnumCallback

```
BOOL ACMFORMATTAGENUMCB acmFormatTagEnumCallback(HACMDRIVERID hadid,
    LPACMFORMATTAGDETAILS paftd, DWORD dwInstance, DWORD fdwSupport);
```

Specifies a callback function used with the **acmFormatTagEnum** function. The **acmFormatTagEnumCallback** name is a placeholder for an application-defined function name.

- The callback function must return TRUE to continue enumeration or FALSE to stop enumeration.

hadid
Handle of the ACM driver identifier.

paftd
Address of an **ACMFORMATTAGDETAILS** structure that contains the enumerated format tag details.

dwInstance
Application-defined value specified in the **acmFormatTagEnum** function.

fdwSupport
Driver-support flags specific to the format tag. These flags are identical to the **fdwSupport** flags of the **ACMDRIVERDETAILS** structure. This parameter can be a combination of the following values and indicates which operations the driver supports with the format tag:

ACMDRIVERDETAILS_SUPPORTF_ASYNC
Driver supports asynchronous conversions with the specified filter tag.

ACMDRIVERDETAILS_SUPPORTF_CODEC
Driver supports conversion between two different format tags where one of the tags is the specified format tag. For example, if a driver supports compression from WAVE_FORMAT_PCM to WAVE_FORMAT_ADPCM, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_CONVERTER
Driver supports conversion between two different formats of the specified format tag. For example, if a driver supports resampling of WAVE_FORMAT_PCM, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_FILTER
Driver supports a filter (modification of the data without changing any of the format attributes). For example, if a driver supports volume or echo operations on the specified format tag, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_HARDWARE
Driver supports hardware input, output, or both of the specified format tag through a waveform-audio device. An application should use **acmMetrics** with the ACM_METRIC_HARDWARE_WAVE_INPUT and ACM_METRIC_HARDWARE_WAVE_OUTPUT metric indexes to get the waveform-audio device identifiers associated with the supporting ACM driver.

The **acmFormatTagEnum** function will return MMSYSERR_NOERROR (zero) if no format tags are to be enumerated. Moreover, the callback function will not be called.

The following functions should not be called from within the callback function: **acmDriverAdd**, **acmDriverRemove**, and **acmDriverPriority**.

acmGetVersion

```
DWORD acmGetVersion(VOID);
```

Returns the version number of the ACM.

- The version number is returned as a hexadecimal number of the form 0xAABBCCCC, where AA is the major version number, BB is the minor version number, and CCCC is the build number.

Win32 applications must verify that the ACM version is at least 0x03320000 (version 3.50) or greater before attempting to use any other ACM functions. The build number (CCCC) is always zero for the retail (non-debug) version of the ACM.

To display the ACM version for a user, an application should use the following format (note that the values should be printed as unsigned decimals):

```
{
    DWORD   dw;
    TCHAR   ach[10];

    dw = acmGetVersion();
    wsprintf(ach, "%u.%.02u", HIWORD(dw) >> 8, HIWORD(dw) & 0x00FF);
}
```

acmMetrics

```
MMRESULT acmMetrics(HACMOBJ hao, UINT uMetric, LPVOID pMetric);
```

Returns various metrics for the ACM or related ACM objects.

- Returns zero if successful or an error otherwise. Possible error values include the following:

ACMERR_NOTPOSSIBLE	The index specified in *uMetric* cannot be returned for the specified *hao*.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.
MMSYSERR_NOTSUPPORTED	The index specified in *uMetric* is not supported.

hao
Handle of the ACM object to query for the metric specified in *uMetric*. For some queries, this parameter can be NULL.

uMetric
Metric index to be returned in *pMetric*.

ACM_METRIC_COUNT_CODECS
Returned value is the number of global ACM compressor or decompressor drivers in the system. The *hao* parameter must be NULL for this metric index. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value.

ACM_METRIC_COUNT_CONVERTERS
Returned value is the number of global ACM converter drivers in the system. The *hao* parameter must be NULL for this metric index. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value.

ACM_METRIC_COUNT_DISABLED
Returned value is the total number of global disabled ACM drivers (of all support types) in the system. The *hao* parameter must be NULL for this metric index. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value. The sum of the ACM_METRIC_COUNT_DRIVERS and ACM_METRIC_COUNT_DISABLED metric indices is the total number of globally installed ACM drivers.

ACM_METRIC_COUNT_DRIVERS
Returned value is the total number of enabled global ACM drivers (of all support types) in the system. The *hao* parameter must be NULL for this metric index. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value.

ACM_METRIC_COUNT_FILTERS
Returned value is the number of global ACM filter drivers in the system. The *hao* parameter must be NULL for this metric index. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value.

ACM_METRIC_COUNT_HARDWARE
Returned value is the number of global ACM hardware drivers in the system. The *hao* parameter must be NULL for this metric index. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value.

ACM_METRIC_COUNT_LOCAL_CODECS
Returned value is the number of local ACM compressor drivers, ACM decompressor drivers, or both for the calling task. The *hao* parameter must be NULL for this metric index. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value.

ACM_METRIC_COUNT_LOCAL_CONVERTERS
Returned value is the number of local ACM converter drivers for the calling task. The *hao*

parameter must be NULL for this metric index. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value.

ACM_METRIC_COUNT_LOCAL_DISABLED
Returned value is the total number of local disabled ACM drivers, of all support types, for the calling task. The *hao* parameter must be NULL for this metric index. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value. The sum of the ACM_METRIC_COUNT_LOCAL_DRIVERS and ACM_METRIC_COUNT_LOCAL_DISABLED metric indices is the total number of locally installed ACM drivers.

ACM_METRIC_COUNT_LOCAL_DRIVERS
Returned value is the total number of enabled local ACM drivers (of all support types) for the calling task. The *hao* parameter must be NULL for this metric index. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value.

ACM_METRIC_COUNT_LOCAL_FILTERS
Returned value is the number of local ACM filter drivers for the calling task. The *hao* parameter must be NULL for this metric index. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value.

ACM_METRIC_DRIVER_PRIORITY
Returned value is the current priority for the specified driver. The *hao* parameter must be a valid ACM driver identifier of the **HACMDRIVERID** data type. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value.

ACM_METRIC_DRIVER_SUPPORT
Returned value is the **fdwSupport** flags for the specified driver. The *hao* parameter must be a valid ACM driver identifier of the **HACMDRIVERID** data type. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value.

ACM_METRIC_HARDWARE_WAVE_INPUT
Returned value is the waveform-audio input device identifier associated with the specified driver. The *hao* parameter must be a valid ACM driver identifier of the **HACMDRIVERID** data type that supports the ACMDRIVERDETAILS_SUPPORTF_HARDWARE flag. If no waveform-audio input device is associated with the driver, MMSYSERR_NOTSUPPORTED is returned. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value.

ACM_METRIC_HARDWARE_WAVE_OUTPUT
Returned value is the waveform-audio output device identifier associated with the specified driver. The *hao* parameter must be a valid ACM driver identifier of the **HACMDRIVERID** data type that supports the ACMDRIVERDETAILS_SUPPORTF_HARDWARE flag. If no waveform-audio output device is associated with the driver, MMSYSERR_NOTSUPPORTED is returned. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value.

ACM_METRIC_MAX_SIZE_FILTER
Returned value is the size of the largest **WAVEFILTER** structure. If *hao* is NULL, the return value is the largest **WAVEFILTER** structure in the system. If *hao* identifies an open instance of an ACM driver of the **HACMDRIVER** data type or an ACM driver identifier of the **HACMDRIVERID** data type, the largest **WAVEFILTER** structure for that driver is returned. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value. This metric is not allowed for an ACM stream handle of the **HACMSTREAM** data type.

ACM_METRIC_MAX_SIZE_FORMAT
Returned value is the size of the largest **WAVEFORMATEX** structure. If *hao* is NULL, the return value is the largest **WAVEFORMATEX** structure in the system. If *hao* identifies an open instance of an ACM driver of the **HACMDRIVER** data type or an ACM driver identifier of the **HACMDRIVERID** data type, the largest **WAVEFORMATEX** structure for that driver is returned. The *pMetric* parameter must point to a buffer of a size equal to a doubleword value. This metric is not allowed for an ACM stream handle of the **HACMSTREAM** data type.

pMetric
Address of the buffer to receive the metric details. The exact definition depends on the *uMetric*

index.

acmStreamClose

```
MMRESULT acmStreamClose(HACMSTREAM has, DWORD fdwClose);
```

Closes an ACM conversion stream. If the function is successful, the handle is invalidated.

- Returns zero if successful or an error otherwise. Possible error values include the following:

ACMERR_BUSY	The conversion stream cannot be closed because an asynchronous conversion is still in progress.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.

has
Handle of the open conversion stream to be closed.

fdwClose
Reserved; must be zero.

acmStreamConvert

```
MMRESULT acmStreamConvert(HACMSTREAM has, LPACMSTREAMHEADER pash,
    DWORD fdwConvert);
```

Requests the ACM to perform a conversion on the specified conversion stream. A conversion may be synchronous or asynchronous, depending on how the stream was opened.

- Returns zero if successful or an error otherwise. Possible error values include the following:

ACMERR_BUSY	The stream header specified in *pash* is currently in use and cannot be reused.
ACMERR_UNPREPARED	The stream header specified in *pash* is currently not prepared by the **acmStreamPrepareHeader** function.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.

has
Handle of the open conversion stream.

pash
Address of a stream header that describes source and destination buffers for a conversion. This header must have been prepared previously by using the **acmStreamPrepareHeader** function.

fdwConvert
Flags for doing the conversion. The following values are defined:

ACM_STREAMCONVERTF_BLOCKALIGN
Only integral numbers of blocks will be converted. Converted data will end on block-aligned boundaries. An application should use this flag for all conversions on a stream until there is not enough source data to convert to a block-aligned destination. In this case, the last conversion should be specified without this flag.

ACM_STREAMCONVERTF_END
ACM conversion stream should begin returning pending instance data. For example, if a conversion stream holds instance data, such as the end of an echo filter operation, this flag will cause the stream to start returning this remaining data with optional source data. This flag can be specified with the ACM_STREAMCONVERTF_START flag.

ACM_STREAMCONVERTF_START
ACM conversion stream should reinitialize its instance data. For example, if a conversion stream holds instance data, such as delta or predictor information, this flag will restore the stream to starting defaults. This flag can be specified with the ACM_STREAMCONVERTF_END flag.

You must use the **acmStreamPrepareHeader** function to prepare the source and destination buffers before they are passed to **acmStreamConvert**.

If an asynchronous conversion request is successfully queued by the ACM or driver and the conversion is later determined to be impossible, the **ACMSTREAMHEADER** structure is posted back to the application's callback function with the **cbDstLengthUsed** member set to zero.

acmStreamConvertCallback

```
void CALLBACK acmStreamConvertCallback(HACMSTREAM has, UINT uMsg,
    DWORD dwInstance, LPARAM lParam1, LPARAM lParam2);
```

Specifies an application-provided callback function to be used when the **acmStreamOpen** function specifies the CALLBACK_FUNCTION flag. The **acmStreamConvertCallback** name is a placeholder for an application-defined function name.

has
Handle of the ACM conversion stream associated with the callback function.

uMsg
ACM conversion stream message. The following values are defined:

MM_ACM_CLOSE
ACM has successfully closed the conversion stream identified by *has*. The handle specified by *has* is no longer valid after receiving this message.

MM_ACM_DONE
ACM has successfully converted the buffer identified by *lParam1* (which is a pointer to the **ACMSTREAMHEADER** structure) for the stream handle identified by *has*.

MM_ACM_OPEN
ACM has successfully opened the conversion stream identified by *has*.

dwInstance
User-instance data given as the *dwInstance* parameter of the **acmStreamOpen** function.

lParam1 and *lParam2*
Message parameters.

The following functions should not be called from within the callback function: **acmDriverAdd**, **acmDriverRemove**, and **acmDriverPriority**.

acmStreamMessage

```
MMRESULT ACMAPI acmStreamMessage(HACMSTREAM has, UINT uMsg,
    LPARAM lParam1, LPARAM lParam2);
```

Sends a driver-specific message to an ACM driver.

- Returns the value returned by the ACM device driver.

has
Handle of an open conversion stream.

uMsg
Message to send.

lParam1 and *lParam2*
Message parameters.

acmStreamOpen

```
MMRESULT acmStreamOpen(LPHACMSTREAM phas, HACMDRIVER had,
    LPWAVEFORMATEX pwfxSrc, LPWAVEFORMATEX pwfxDst, LPWAVEFILTER pwfltr,
    DWORD dwCallback, DWORD dwInstance, DWORD fdwOpen);
```

Opens an ACM conversion stream. Conversion streams are used to convert data from one specified audio format to another.

- Returns zero if successful or an error otherwise. Possible error values include the following:

ACMERR_NOTPOSSIBLE	The requested operation cannot be performed.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.
MMSYSERR_NOMEM	The system is unable to allocate resources.

phas
Address of a handle that will receive the new stream handle that can be used to perform conversions. This handle is used to identify the stream in calls to other ACM stream conversion functions. If the ACM_STREAMOPENF_QUERY flag is specified, this parameter should be NULL.

had
Handle of an ACM driver. If this handle is specified, it identifies a specific driver to be used for a conversion stream. If this parameter is NULL, all suitable installed ACM drivers are queried until a match is found.

pwfxSrc
Address of a **WAVEFORMATEX** structure that identifies the desired source format for the conversion.

pwfxDst
Address of a **WAVEFORMATEX** structure that identifies the desired destination format for the conversion.

pwfltr
Address of a **WAVEFILTER** structure that identifies the desired filtering operation to perform on the conversion stream. If no filtering operation is desired, this parameter can be NULL. If a filter is specified, the source (*pwfxSrc*) and destination (*pwfxDst*) formats must be the same.

dwCallback
Address of a callback function, a handle of a window, or a handle of an event. A callback function will be called only if the conversion stream is opened with the ACM_STREAMOPENF_ASYNC flag. A callback function is notified when the conversion stream is opened or closed and after each buffer is converted. If the conversion stream is opened without the ACM_STREAMOPENF_ASYNC flag, this parameter should be set to zero.

dwInstance
User-instance data passed to the callback function specified by the *dwCallback* parameter. This parameter is not used with window and event callbacks. If the conversion stream is opened without the ACM_STREAMOPENF_ASYNC flag, this parameter should be set to zero.

fdwOpen
Flags for opening the conversion stream. The following values are defined:

ACM_STREAMOPENF_ASYNC
Stream conversion should be performed asynchronously. If this flag is specified, the application can use a callback function to be notified when the conversion stream is opened and closed and after each buffer is converted. In addition to using a callback function, an application can examine

the **fdwStatus** member of the **ACMSTREAMHEADER** structure for the ACMSTREAMHEADER_STATUSF_DONE flag.

ACM_STREAMOPENF_NONREALTIME
: ACM will not consider time constraints when converting the data. By default, the driver will attempt to convert the data in real time. For some formats, specifying this flag might improve the audio quality or other characteristics.

ACM_STREAMOPENF_QUERY
: ACM will be queried to determine whether it supports the given conversion. A conversion stream will not be opened, and no handle will be returned in the *phas* parameter.

CALLBACK_EVENT
: The *dwCallback* parameter is a handle of an event.

CALLBACK_FUNCTION
: The *dwCallback* parameter is a callback procedure address. The function prototype must conform to the **acmStreamConvertCallback** prototype.

CALLBACK_WINDOW
: The *dwCallback* parameter is a window handle.

If an ACM driver cannot perform real-time conversions and the ACM_STREAMOPENF_NONREALTIME flag is not specified for the *fdwOpen* parameter, the open operation will fail returning an ACMERR_NOTPOSSIBLE error code. An application can use the ACM_STREAMOPENF_QUERY flag to determine if real-time conversions are supported for input.

If an application uses a window to receive callback information, the MM_ACM_OPEN, MM_ACM_CLOSE, and MM_ACM_DONE messages are sent to the window procedure function to indicate the progress of the conversion stream. In this case, the *wParam* parameter identifies the **HACMSTREAM** handle. The *lParam* parameter identifies the **ACMSTREAMHEADER** structure for MM_ACM_DONE, but it is not used for MM_ACM_OPEN and MM_ACM_CLOSE.

If an application uses a function to receive callback information, the MM_ACM_OPEN, MM_ACM_CLOSE, and MM_ACM_DONE messages are sent to the function to indicate the progress of waveform-audio output. The callback function must reside in a dynamic-link library (DLL).

If an application uses an event for callback notification, the event is signaled to indicate the progress of the conversion stream. The event will be signaled when a stream is opened, after each buffer is converted, and when the stream is closed.

acmStreamPrepareHeader

```
MMRESULT acmStreamPrepareHeader(HACMSTREAM has, LPACMSTREAMHEADER pash,
    DWORD fdwPrepare);
```

Prepares an **ACMSTREAMHEADER** structure for an ACM stream conversion. This function must be called for every stream header before it can be used in a conversion stream. An application needs to prepare a stream header only once for the life of a given stream. The stream header can be reused as long as the sizes of the source and destination buffers do not exceed the sizes used when the stream header was originally prepared.

- Returns zero if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.
MMSYSERR_NOMEM	The system is unable to allocate resources.

has
Handle of the conversion steam.

pash
Address of an **ACMSTREAMHEADER** structure that identifies the source and destination buffers to be prepared.

fdwPrepare
Reserved; must be zero.

Preparing a stream header that has already been prepared has no effect, and the function returns zero. Nevertheless, you should ensure your application does not prepare a stream header multiple times.

acmStreamReset

```
MMRESULT acmStreamReset(HACMSTREAM has, DWORD fdwReset);
```

Stops conversions for a given ACM stream. All pending buffers are marked as done and returned to the application.

- Returns zero if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.

has
Handle of the conversion stream.

fdwReset
Reserved; must be zero.

Resetting an ACM conversion stream is necessary only for asynchronous conversion streams. Resetting a synchronous conversion stream will succeed, but no action will be taken.

acmStreamSize

```
MMRESULT acmStreamSize(HACMSTREAM has, DWORD cbInput,
    LPDWORD pdwOutputBytes, DWORD fdwSize);
```

Returns a recommended size for a source or destination buffer on an ACM stream.

- Returns zero if successful or an error otherwise. Possible error values include the following:

ACMERR_NOTPOSSIBLE	The requested operation cannot be performed.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.

has
Handle of the conversion stream.

cbInput
Size, in bytes, of the source or destination buffer. The *fdwSize* flags specify what the input parameter defines. This parameter must be nonzero.

pdwOutputBytes
Address of a variable that contains the size, in bytes, of the source or destination buffer. The *fdwSize* flags specify what the output parameter defines. If the **acmStreamSize** function succeeds, this location will always be filled with a nonzero value.

fdwSize
Flags for the stream size query. The following values are defined:

ACM_STREAMSIZEF_DESTINATION
The *cbInput* parameter contains the size of the destination buffer. The *pdwOutputBytes* parameter will receive the recommended source buffer size, in bytes.

ACM_STREAMSIZEF_SOURCE
The *cbInput* parameter contains the size of the source buffer. The *pdwOutputBytes* parameter will receive the recommended destination buffer size, in bytes.

An application can use this function to determine suggested buffer sizes for either source or destination buffers. The buffer sizes returned might be only an estimation of the actual sizes required for conversion. Because actual conversion sizes cannot always be determined without performing the conversion, the sizes returned will usually be overestimated.

In the event of an error, the location pointed to by *pdwOutputBytes* will receive zero. This assumes that the pointer specified by *pdwOutputBytes* is valid.

acmStreamUnprepareHeader

```
MMRESULT acmStreamUnprepareHeader(HACMSTREAM has,
    LPACMSTREAMHEADER pash, DWORD fdwUnprepare);
```

Cleans up the preparation performed by the **acmStreamPrepareHeader** function for an ACM stream. This function must be called after the ACM is finished with the given buffers. An application must call this function before freeing the source and destination buffers.

- Returns zero if successful or an error otherwise. Possible error values include the following:

ACMERR_BUSY	The stream header specified in *pash* is currently in use and cannot be unprepared.
ACMERR_UNPREPARED	The stream header specified in *pash* is currently not prepared by the **acmStreamPrepareHeader** function.
MMSYSERR_INVALFLAG	At least one flag is invalid.
MMSYSERR_INVALHANDLE	The specified handle is invalid.
MMSYSERR_INVALPARAM	At least one parameter is invalid.

has
Handle of the conversion steam.

pash
Address of an **ACMSTREAMHEADER** structure that identifies the source and destination buffers to be unprepared.

fdwUnprepare
Reserved; must be zero.

Unpreparing a stream header that has already been unprepared is an error. An application must specify the source and destination buffer lengths (**cbSrcLength** and **cbDstLength**, respectively) that were used during a call to the corresponding **acmStreamPrepareHeader**. Failing to reset these member values will cause **acmStreamUnprepareHeader** to fail with an MMSYSERR_INVALPARAM error.

The ACM can recover from some errors. The ACM will return a nonzero error, yet the stream header will be properly unprepared. To determine whether the stream header was actually unprepared, an application can examine the ACMSTREAMHEADER_STATUSF_PREPARED flag. If **acmStreamUnprepareHeader** returns success, the header will always be unprepared.

MM_ACM_FILTERCHOOSE

```
MM_ACM_FILTERCHOOSE
wParam = (WPARAM) wDropDown
lParam = (LONG) lCustom
```

Notifies an **acmFilterChoose** dialog box hook function before adding an element to one of the three drop-down list boxes. This message allows an application to further customize the selections available through the user interface.

- Returns TRUE if an application handles this message or FALSE otherwise.

wDropDown

Drop-down list box being initialized and a verify or add operation.

FILTERCHOOSE_CUSTOM_VERIFY

The *lParam* parameter is a pointer to a **WAVEFILTER** structure to be added to the custom Name drop-down list box.

FILTERCHOOSE_FILTER_ADD

The *lParam* parameter is a pointer to a buffer that will accept a **WAVEFILTER** structure to be added to the Filter drop-down list box. The application must copy the filter structure to be added into this buffer.

FILTERCHOOSE_FILTER_VERIFY

The *lParam* parameter is a pointer to a **WAVEFILTER** structure to be added to the Filter drop-down list box.

FILTERCHOOSE_FILTERTAG_ADD

The *lParam* parameter is a pointer to a **DWORD** that will accept a waveform-audio filter tag to be added to the Filter Tag drop-down list box.

FILTERCHOOSE_FILTERTAG_VERIFY

The *lParam* parameter is a waveform-audio filter tag to be listed in the Filter Tag drop-down list box.

lCustom

Value defined by the listbox specified in the *wParam* parameter.

If the application processes the FILTERCHOOSE_FILTER_ADD operation, the size of the memory buffer supplied in *lParam* will be determined from the **acmMetrics** function.

If the application processes a verify operation, the application must precede the return value with **SetWindowLong**(hwnd, DWL_MSGRESULT, (LONG) FALSE) to prevent the dialog box from listing this selection or with **SetWindowLong**(hwnd, DWL_MSGRESULT, (LONG)TRUE) to allow the dialog box to list this selection. If processing an add operation, the application must precede the return with **SetWindowLong**(hwnd, DWL_MSGRESULT, (LONG)FALSE) to indicate that no more additions are required or with **SetWindowLong**(hwnd, DWL_MSGRESULT, (LONG)TRUE) if more additions are required.

MM_ACM_FORMATCHOOSE

```
MM_ACM_FORMATCHOOSE
wParam = (WPARAM) wDropDown
lParam = (LONG) lCustom
```

Notifies an **acmFormatChoose** dialog hook function before adding an element to one of the three drop-down list boxes. This message allows an application to further customize the selections available through the user interface.

- Returns TRUE if an application handles this message or FALSE otherwise.

wDropDown

Drop-down listbox being initialized and a verify or add operation.

FORMATCHOOSE_CUSTOM_VERIFY

The *lParam* parameter is a pointer to a **WAVEFORMATEX** structure to be added to the custom Name drop-down list box.

FORMATCHOOSE_FORMAT_ADD

The *lParam* parameter is a pointer to a buffer that will accept a **WAVEFORMATEX** structure to be added to the Format drop-down list box. The application must copy the format structure to be added into this buffer.

FORMATCHOOSE_FORMAT_VERIFY

The *lParam* parameter is a pointer to a **WAVEFORMATEX** structure to be added to the Format drop-down list box.

FORMATCHOOSE_FORMATTAG_ADD

The *lParam* parameter is a pointer to a variable that will accept a waveform-audio format tag to be added to the Format Tag drop-down list box.

FORMATCHOOSE_FORMATTAG_VERIFY

The *lParam* parameter is a waveform-audio format tag to be listed in the Format Tag drop-down list box.

lCustom

Value defined by the listbox specified in the *wParam* parameter.

If the application processes the FILTERCHOOSE_FORMAT_ADD operation, the size of the memory buffer supplied in *lParam* will be determined from the **acmMetrics** function.

If your application is processing a verify operation, it can prevent the dialog box from listing this selection by calling the **SetWindowLong** function with *nIndex* set to DWL_MSGRESULT and *lNewLong* set to FALSE (cast to a **LONG** data type). To allow the dialog box to list this selection, call this function with *lNewLong* set to TRUE.

If your application is processing an add operation, it can indicate that no more additions are required by calling the **SetWindowLong** function with *nIndex* set to DWL_MSGRESULT and *lNewLong* set to FALSE (cast to a **LONG** data type). To indicate more additions are required, call this function with *lNewLong* set to TRUE.

ACMDRIVERDETAILS

```
typedef struct {
    DWORD  cbStruct;      // see below
    FOURCC fccType;       // see below
    FOURCC fccComp;       // see below
    WORD   wMid;          // manufacturer identifier
    WORD   wPid;          // product identifier
    DWORD  vdwACM;        // see below
    DWORD  vdwDriver;     // see below
    DWORD  fdwSupport;    // see below
    DWORD  cFormatTags;   // see below
    DWORD  cFilterTags;   // see below
    HICON  hicon;         // see below
    char   szShortName[ACMDRIVERDETAILS_SHORTNAME_CHARS];  // see below
    char   szLongName[ACMDRIVERDETAILS_LONGNAME_CHARS];    // see below
    char   szCopyright[ACMDRIVERDETAILS_COPYRIGHT_CHARS];  // see below
    char   szLicensing[ACMDRIVERDETAILS_LICENSING_CHARS];  // see below
    char   szFeatures[ACMDRIVERDETAILS_FEATURES_CHARS];    // see below
} ACMDRIVERDETAILS;
```

Describes the features of an ACM driver.

cbStruct
Size, in bytes, of the valid information contained in the **ACMDRIVERDETAILS** structure. An application should initialize this member to the size, in bytes, of the desired information. The size specified in this member must be large enough to contain the **cbStruct** member of the **ACMDRIVERDETAILS** structure. When the **acmDriverDetails** function returns, this member contains the actual size of the information returned. The returned information will never exceed the requested size.

fccType
Type of the driver. For ACM drivers, set this member to ACMDRIVERDETAILS_FCCTYPE_AUDIOCODEC.

fccComp
Subtype of the driver. This member is currently set to ACMDRIVERDETAILS_FCCCOMP_UNDEFINED (zero).

vdwACM
Version of the ACM for which this driver was compiled. The version number is a hexadecimal number in the format 0xAABBCCCC, where AA is the major version number, BB is the minor version number, and CCCC is the build number. The version parts (major, minor, and build) should be displayed as decimal numbers.

vdwDriver
Version of the driver. The version number is a hexadecimal number in the format 0xAABBCCCC, where AA is the major version number, BB is the minor version number, and CCCC is the build number. The version parts (major, minor, and build) should be displayed as decimal numbers.

fdwSupport
Support flags for the driver. The following values are defined:

ACMDRIVERDETAILS_SUPPORTF_ASYNC
Driver supports asynchronous conversions.

ACMDRIVERDETAILS_SUPPORTF_CODEC
Driver supports conversion between two different format tags. For example, if a driver supports compression from WAVE_FORMAT_PCM to WAVE_FORMAT_ADPCM, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_CONVERTER

Driver supports conversion between two different formats of the same format tag. For example, if a driver supports resampling of WAVE_FORMAT_PCM, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_DISABLED
Driver has been disabled. This flag is set by the ACM for a driver when it has been disabled for any of a number of reasons. Disabled drivers cannot be opened and can be used only under very limited circumstances.

ACMDRIVERDETAILS_SUPPORTF_FILTER
Driver supports a filter (modification of the data without changing any of the format attributes). For example, if a driver supports volume or echo operations on WAVE_FORMAT_PCM, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_HARDWARE
Driver supports hardware input, output, or both through a waveform-audio device. An application should use the **acmMetrics** function with the ACM_METRIC_HARDWARE_WAVE_INPUT and ACM_METRIC_HARDWARE_WAVE_OUTPUT metric indexes to get the waveform-audio device identifiers associated with the supporting ACM driver.

ACMDRIVERDETAILS_SUPPORTF_LOCAL
The driver has been installed locally with respect to the current task.

cFormatTags
Number of unique format tags supported by this driver.

cFilterTags
Number of unique filter tags supported by this driver.

hicon
Handle of a custom icon for this driver. An application can use this icon for referencing the driver visually. This member can be NULL.

szShortName
Null-terminated string that describes the name of the driver. This string is intended to be displayed in small spaces.

szLongName
Null-terminated string that describes the full name of the driver. This string is intended to be displayed in large (descriptive) spaces.

szCopyright
Null-terminated string that provides copyright information for the driver.

szLicensing
Null-terminated string that provides special licensing information for the driver.

szFeatures
Null-terminated string that provides special feature information for the driver.

ACMFILTERCHOOSE

```
typedef struct {
    DWORD                   cbStruct;
    DWORD                   fdwStyle;
    HWND                    hwndOwner;
    LPWAVEFILTER            pwfltr;
    DWORD                   cbwfltr;
    LPCSTR                  pszTitle;
    char szFilterTag[ACMFILTERTAGDETAILS_FILTERTAG_CHARS];
    char szFilter[ACMFILTERDETAILS_FILTER_CHARS];
    LPSTR                   pszName;
    DWORD                   cchName;
    DWORD                   fdwEnum;
    LPWAVEFILTER            pwfltrEnum;
    HINSTANCE               hInstance;
    LPCSTR                  pszTemplateName;
    LPARAM                  lCustData;
    ACMFILTERCHOOSEHOOKPROC pfnHook;
} ACMFILTERCHOOSE;
```

Contains information the ACM uses to initialize the system-defined waveform-audio filter selection dialog box. After the user closes the dialog box, the system returns information about the user's selection in this structure.

cbStruct
Size, in bytes, of the **ACMFILTERCHOOSE** structure. This member must be initialized before an application calls the **acmFilterChoose** function. The size specified in this member must be large enough to contain the base **ACMFILTERCHOOSE** structure.

fdwStyle
Optional style flags for the **acmFilterChoose** function. This member must be initialized to a valid combination of the following flags before an application calls the **acmFilterChoose** function. The following values are defined:

ACMFILTERCHOOSE_STYLEF_CONTEXTHELP
Context-sensitive help will be available in the dialog box. To use this feature, an application must register the ACMHELPMSGCONTEXTMENU and ACMHELPMSGCONTEXTHELP constants, using the **RegisterWindowMessage** function. When the user invokes help, the registered message will be posted to the owning window. The message will contain the *wParam* and *lParam* parameters from the original WM_CONTEXTMENU or WM_CONTEXTHELP message.

ACMFILTERCHOOSE_STYLEF_ENABLEHOOK
Enables the hook function specified in the **pfnHook** member. An application can use hook functions for a variety of customizations, including answering the MM_ACM_FILTERCHOOSE message.

ACMFILTERCHOOSE_STYLEF_ENABLETEMPLATE
Causes the ACM to create the dialog box template identified by the **hInstance** and **pszTemplateName** members.

ACMFILTERCHOOSE_STYLEF_ENABLETEMPLATEHANDLE
The **hInstance** member identifies a data block that contains a preloaded dialog box template. If this flag is specified, the ACM ignores the **pszTemplateName** member.

ACMFILTERCHOOSE_STYLEF_INITTOFILTERSTRUCT
The buffer pointed to by **pwfltr** contains a valid **WAVEFILTER** structure that the dialog box will use as the initial selection.

ACMFILTERCHOOSE_STYLEF_SHOWHELP

A help button will appear in the dialog box. To use a custom Help file, an application must register the ACMHELPMSGSTRING value with the **RegisterWindowMessage** function. When the user presses the help button, the registered message is posted to the owner.

hwndOwner

Handle of the window that owns the dialog box. This member can be any valid window handle or NULL if the dialog box has no owner. This member must be initialized before calling the **acmFilterChoose** function.

pwfltr

Address of a **WAVEFILTER** structure. If the ACMFILTERCHOOSE_STYLEF_INITTOFILTERSTRUCT flag is specified in the **fdwStyle** member, this structure must be initialized to a valid filter. When the **acmFilterChoose** function returns, this buffer contains the selected filter. If the user cancels the dialog box, no changes will be made to this buffer.

cbwfltr

Size, in bytes, of the buffer pointed to by the **pwfltr** member. The **acmFilterChoose** function returns ACMERR_NOTPOSSIBLE if the buffer is too small to contain the filter information; the ACM also copies the required size into this member. An application can use the **acmMetrics** and **acmFilterTagDetails** functions to determine the largest size required for this buffer.

pszTitle

Address of a string to be placed in the title bar of the dialog box. If this member is NULL, the ACM uses the default title (that is, "Filter Selection").

szFilterTag

Buffer containing a null-terminated string describing the filter tag of the filter selection when the **acmFilterChoose** function returns. This string is equivalent to the **szFilterTag** member of the **ACMFILTERTAGDETAILS** structure returned by **acmFilterTagDetails**. If the user cancels the dialog box, this member will contain a null-terminated string.

szFilter

Buffer containing a null-terminated string describing the filter attributes of the filter selection when the **acmFilterChoose** function returns. This string is equivalent to the **szFilter** member of the **ACMFILTERDETAILS** structure returned by **acmFilterDetails**. If the user cancels the dialog box, this member will contain a null-terminated string.

pszName

Address of a string for a user-defined filter name. If this is a non-null-terminated string, the ACM attempts to match the name with a previously saved user-defined filter name. If a match is found, the dialog box is initialized to that filter. If a match is not found or this member is a null-terminated string, this member is ignored for input. When the **acmFilterChoose** function returns, this buffer contains a null-terminated string describing the user-defined filter. If the filter name is untitled (that is, the user has not given a name for the filter), this member will be a null-terminated string on return. If the user cancels the dialog box, no changes will be made to this buffer.

If the ACMFILTERCHOOSE_STYLEF_INITTOFILTERSTRUCT flag is specified by the **fdwStyle** member, the **pszName** member is ignored as an input member.

cchName

Size, in characters, of the buffer identified by the **pszName** member. This buffer should be at least 128 characters long. If **pszName** is NULL, this member is ignored.

fdwEnum

Optional flags for restricting the type of filters listed in the dialog box. These flags are identical to the *fdwEnum* flags for the **acmFilterEnum** function. If **pwfltrEnum** is NULL, this member should be zero.

ACM_FILTERENUMF_DWFILTERTAG

The **dwFilterTag** member of the **WAVEFILTER** structure pointed to by the **pwfltrEnum** member is valid. The enumerator will only enumerate a filter that conforms to this attribute.

pwfltrEnum

Address of a **WAVEFILTER** structure that will be used to restrict the filters listed in the dialog box. The **fdwEnum** member defines which members of this structure should be used for the enumeration restrictions. The **cbStruct** member of this **WAVEFILTER** structure must be initialized to the size of the **WAVEFILTER** structure. If no special restrictions are desired, this member can be NULL.

hInstance

Handle of a data block that contains a dialog box template specified by the **pszTemplateName** member. This member is used only if the **fdwStyle** member specifies the ACMFILTERCHOOSE_STYLEF_ENABLETEMPLATE or ACMFILTERCHOOSE_STYLEF_ENABLETEMPLATEHANDLE flag; otherwise, this member should be NULL on input.

pszTemplateName

Address of a null-terminated string that specifies the name of the resource file for the dialog box template that is to be substituted for the dialog box template in the ACM. An application can use the **MAKEINTRESOURCE** macro for numbered dialog box resources. This member is used only if the **fdwStyle** member specifies the ACMFILTERCHOOSE_STYLEF_ENABLETEMPLATE flag; otherwise, this member should be NULL on input.

lCustData

Application-defined data that the ACM passes to the hook function identified by the **pfnHook** member. The system passes the data in the *lParam* parameter of the WM_INITDIALOG message.

pfnHook

Address of a hook function that processes messages intended for the dialog box. An application must specify the ACMFILTERCHOOSE_STYLEF_ENABLEHOOK flag in the **fdwStyle** member to enable the hook; otherwise, this member should be NULL. The hook function should return FALSE to pass a message to the standard dialog box procedure or TRUE to discard the message.

ACMFILTERDETAILS

```
typedef struct {
    DWORD        cbStruct;
    DWORD        dwFilterIndex;
    DWORD        dwFilterTag;
    DWORD        fdwSupport;
    LPWAVEFILTER pwfltr;
    DWORD        cbwfltr;
    char        szFilter[ACMFILTERDETAILS_FILTER_CHARS];
} ACMFILTERDETAILS;
```

Details a waveform-audio filter for a specific filter tag for an ACM driver.

cbStruct

Size, in bytes, of the **ACMFILTERDETAILS** structure. This member must be initialized before calling the **acmFilterDetails** or **acmFilterEnum** functions. The size specified in this member must be large enough to contain the base **ACMFILTERDETAILS** structure. When the **acmFilterDetails** function returns, this member contains the actual size of the information returned. The returned information will never exceed the requested size.

dwFilterIndex

Index of the filter about which details will be retrieved. The index ranges from zero to one less than the number of standard filters supported by an ACM driver for a filter tag. The number of standard filters supported by a driver for a filter tag is contained in the **cStandardFilters** member of the **ACMFILTERTAGDETAILS** structure. The **dwFilterIndex** member is used only when querying standard filter details about a driver by index; otherwise, this member should be zero. Also, this member will be set to zero by the ACM when an application queries for details on a filter; in other words, this member is used only for input and is never returned by the ACM or an ACM driver.

dwFilterTag

Waveform-audio filter tag that the **ACMFILTERDETAILS** structure describes. This member is used as an input for the ACM_FILTERDETAILSF_INDEX query flag. For the ACM_FILTERDETAILSF_FORMAT query flag, this member must be initialized to the same filter tag as the **pwfltr** member specifies. If the **acmFilterDetails** function is successful, this member is always returned. This member should be set to WAVE_FILTER_UNKNOWN for all other query flags.

fdwSupport

Driver-support flags specific to the specified filter. These flags are identical to the **fdwSupport** flags of the **ACMDRIVERDETAILS** structure, but they are specific to the filter that is being queried. This member can be a combination of the following values and identifies which operations the driver supports for the filter tag:

ACMDRIVERDETAILS_SUPPORTF_ASYNC
Driver supports asynchronous conversions.

ACMDRIVERDETAILS_SUPPORTF_CODEC
Driver supports conversion between two different format tags while using the specified filter. For example, if a driver supports compression from WAVE_FORMAT_PCM to WAVE_FORMAT_ADPCM with the specified filter, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_CONVERTER
Driver supports conversion between two different formats of the same format tag while using the specified filter. For example, if a driver supports resampling of WAVE_FORMAT_PCM with the specified filter, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_FILTER
Driver supports a filter (modification of the data without changing any of the format attributes). For example, if a driver supports volume or echo operations on WAVE_FORMAT_PCM, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_HARDWARE

Driver supports hardware input, output, or both with the specified filter through a waveform-audio device. An application should use the **acmMetrics** function with the ACM_METRIC_HARDWARE_WAVE_INPUT and ACM_METRIC_HARDWARE_WAVE_OUTPUT metric indexes to retrieve the waveform-audio device identifiers associated with the supporting ACM driver.

pwfltr

Address of a **WAVEFILTER** structure that will receive the filter details. This structure requires no initialization by the application unless the ACM_FILTERDETAILSF_FILTER flag is specified with the **acmFilterDetails** function. In this case, the **dwFilterTag** member of the **WAVEFILTER** structure must be equal to the **dwFilterTag** member of the **ACMFILTERDETAILS** structure.

cbwfltr

Size, in bytes, available for **pwfltr** to receive the filter details. The **acmMetrics** and **acmFilterTagDetails** functions can be used to determine the maximum size required for any filter available for the specified driver (or for all installed ACM drivers).

szFilter

String that describes the filter for the **dwFilterTag** type. If the **acmFilterDetails** function is successful, this string is always returned.

ACMFILTERTAGDETAILS

```
typedef struct {
    DWORD cbStruct;
    DWORD dwFilterTagIndex;
    DWORD dwFilterTag;
    DWORD cbFilterSize;
    DWORD fdwSupport;
    DWORD cStandardFilters;
    char szFilterTag[ACMFILTERTAGDETAILS_FILTERTAG_CHARS];
} ACMFILTERTAGDETAILS;
```

Details a waveform-audio filter tag for an ACM filter driver.

cbStruct
Size, in bytes, of the **ACMFILTERTAGDETAILS** structure. This member must be initialized before an application calls the **acmFilterTagDetails** or **acmFilterTagEnum** function. The size specified in this member must be large enough to contain the base **ACMFILTERTAGDETAILS** structure. When the **acmFilterTagDetails** function returns, this member contains the actual size of the information returned. The returned information will never exceed the requested size.

dwFilterTagIndex
Index of the filter tag to retrieve details for. The index ranges from zero to one less than the number of filter tags supported by an ACM driver. The number of filter tags supported by a driver is contained in the **cFilterTags** member of the **ACMDRIVERDETAILS** structure. The **dwFilterTagIndex** member is used only when querying filter tag details about a driver by index; otherwise, this member should be zero.

dwFilterTag
Waveform-audio filter tag that the **ACMFILTERTAGDETAILS** structure describes. This member is used as an input for the ACM_FILTERTAGDETAILSF_FILTERTAG and ACM_FILTERTAGDETAILSF_LARGESTSIZE query flags. This member is always returned if the **acmFilterTagDetails** function is successful. This member should be set to WAVE_FILTER_UNKNOWN for all other query flags.

cbFilterSize
Largest total size, in bytes, of a waveform-audio filter of the **dwFilterTag** type. For example, this member will be 40 for WAVE_FILTER_ECHO and 36 for WAVE_FILTER_VOLUME.

fdwSupport
Driver-support flags specific to the filter tag. These flags are identical to the **fdwSupport** flags of the **ACMDRIVERDETAILS** structure. This member can be a combination of the following values and identifies which operations the driver supports with the filter tag:

ACMDRIVERDETAILS_SUPPORTF_ASYNC
Driver supports asynchronous conversions.

ACMDRIVERDETAILS_SUPPORTF_CODEC
Driver supports conversion between two different format tags while using the specified filter tag. For example, if a driver supports compression from WAVE_FORMAT_PCM to WAVE_FORMAT_ADPCM with the specified filter tag, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_CONVERTER
Driver supports conversion between two different formats of the same format tag while using the specified filter tag. For example, if a driver supports resampling of WAVE_FORMAT_PCM with the specified filter tag, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_FILTER
Driver supports a filter (modification of the data without changing any of the format attributes). For example, if a driver supports volume or echo operations on WAVE_FORMAT_PCM, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_HARDWARE

Driver supports hardware input, output, or both with the specified filter tag through a waveform-audio device. An application should use the **acmMetrics** function with the ACM_METRIC_HARDWARE_WAVE_INPUT and ACM_METRIC_HARDWARE_WAVE_OUTPUT metric indexes to get the waveform-audio device identifiers associated with the supporting ACM driver.

cStandardFilters

Number of standard filters of the **dwFilterTag** type (that is, the combination of all filter characteristics). This value cannot specify all filters supported by the driver.

szFilterTag

String that describes the **dwFilterTag** type. This string is always returned if the **acmFilterTagDetails** function is successful.

ACMFORMATCHOOSE

```
typedef struct {
    DWORD                   cbStruct;
    DWORD                   fdwStyle;
    HWND                    hwndOwner;
    LPWAVEFORMATEX          pwfx;
    DWORD                   cbwfx;
    LPCSTR                  pszTitle;
    char szFormatTag[ACMFORMATTAGDETAILS_FORMATTAG_CHARS];
    char szFormat[ACMFORMATDETAILS_FORMAT_CHARS];
    LPSTR                   pszName;
    DWORD                   cchName;
    DWORD                   fdwEnum;
    LPWAVEFORMATEX          pwfxEnum;
    HINSTANCE               hInstance;
    LPCSTR                  pszTemplateName;
    LPARAM                  lCustData;
    ACMFORMATCHOOSEHOOKPROC pfnHook;
} ACMFORMATCHOOSE;
```

Contains information the ACM uses to initialize the system-defined waveform-audio format selection dialog box. After the user closes the dialog box, the system returns information about the user's selection in this structure.

cbStruct

Size, in bytes, of the **ACMFORMATCHOOSE** structure. This member must be initialized before an application calls the **acmFormatChoose** function. The size specified in this member must be large enough to contain the base **ACMFORMATCHOOSE** structure.

fdwStyle

Optional style flags for the **acmFormatChoose** function. This member must be initialized to a valid combination of the following flags before an application calls the **acmFormatChoose** function:

ACMFORMATCHOOSE_STYLEF_CONTEXTHELP

Context-sensitive help will be available in the dialog box. To use this feature, an application must register the ACMHELPMSGCONTEXTMENU and ACMHELPMSGCONTEXTHELP constants, using the **RegisterWindowMessage** function. When the user invokes help, the registered message will be posted to the owning window. The message will contain the *wParam* and *lParam* parameters from the original WM_CONTEXTMENU or WM_CONTEXTHELP message.

ACMFORMATCHOOSE_STYLEF_ENABLEHOOK

Enables the hook function pointed to by the **pfnHook** member. An application can use hook functions for a variety of customizations, including answering the MM_ACM_FORMATCHOOSE message.

ACMFORMATCHOOSE_STYLEF_ENABLETEMPLATE

Causes the ACM to create the dialog box template identified by **hInstance** and **pszTemplateName**.

ACMFORMATCHOOSE_STYLEF_ENABLETEMPLATEHANDLE

The **hInstance** member identifies a data block that contains a preloaded dialog box template. If this flag is specified, the ACM ignores the **pszTemplateName** member.

ACMFORMATCHOOSE_STYLEF_INITTOWFXSTRUCT

The buffer pointed to by **pwfx** contains a valid **WAVEFORMATEX** structure that the dialog box will use as the initial selection.

ACMFORMATCHOOSE_STYLEF_SHOWHELP

A help button will appear in the dialog box. To use a custom Help file, an application must register

the ACMHELPMSGSTRING constant with the **RegisterWindowMessage** function. When the user presses the help button, the registered message will be posted to the owner.

hwndOwner

Handle of the window that owns the dialog box. This member can be any valid window handle, or NULL if the dialog box has no owner. This member must be initialized before calling the **acmFormatChoose** function.

pwfx

Address of a **WAVEFORMATEX** structure. If the ACMFORMATCHOOSE_STYLEF_INITTOWFXSTRUCT flag is specified in the **fdwStyle** member, this structure must be initialized to a valid format. When the **acmFormatChoose** function returns, this buffer contains the selected format. If the user cancels the dialog box, no changes will be made to this buffer.

cbwfx

Size, in bytes, of the buffer pointed to by **pwfx**. If the buffer is too small to contain the format information, the **acmFormatChoose** function returns ACMERR_NOTPOSSIBLE. Also, the ACM copies the required size into this member. An application can use the **acmMetrics** and **acmFormatTagDetails** functions to determine the largest size required for this buffer.

pszTitle

Address of a string to be placed in the title bar of the dialog box. If this member is NULL, the ACM uses the default title (that is, "Sound Selection").

szFormatTag

Buffer containing a null-terminated string describing the format tag of the format selection when the **acmFormatChoose** function returns. This string is equivalent to the **szFormatTag** member of the **ACMFORMATTAGDETAILS** structure returned by the **acmFormatTagDetails** function. If the user cancels the dialog box, this member will contain a null-terminated string.

szFormat

Buffer containing a null-terminated string describing the format attributes of the format selection when the **acmFormatChoose** function returns. This string is equivalent to the **szFormat** member of the **ACMFORMATDETAILS** structure returned by the **acmFormatDetails** function. If the user cancels the dialog box, this member will contain a null-terminated string.

pszName

Address of a string for a user-defined format name. If this is a non-null-terminated string, the ACM will attempt to match the name with a previously saved user-defined format name. If a match is found, the dialog box is initialized to that format. If a match is not found or this member is a null-terminated string, this member is ignored on input. When the **acmFormatChoose** function returns, this buffer contains a null-terminated string describing the user-defined format. If the format name is untitled (that is, the user has not given a name for the format), this member will be a null-terminated string on return. If the user cancels the dialog box, no changes will be made to this buffer.

If the ACMFORMATCHOOSE_STYLEF_INITTOWFXSTRUCT flag is specified in the **fdwStyle** member, the **pszName** member is ignored for input.

cchName

Size, in characters, of the buffer identified by the **pszName** member. This buffer should be at least 128 characters long. If the **pszName** member is NULL, this member is ignored.

fdwEnum

Optional flags for restricting the type of formats listed in the dialog box. These flags are identical to the *fdwEnum* flags for the **acmFormatEnum** function. If **pwfxEnum** is NULL, this member should be zero. The following values are defined:

ACM_FORMATENUMF_CONVERT

The **WAVEFORMATEX** structure pointed to by the **pwfxEnum** member is valid. The enumerator will enumerate only destination formats that can be converted from the given **pwfxEnum** format.

ACM_FORMATENUMF_HARDWARE

The enumerator should enumerate only formats that are supported in hardware by one or more of

the installed waveform-audio devices. This flag provides a way for an application to choose only formats native to an installed waveform-audio device.

ACM_FORMATENUMF_INPUT
The enumerator should enumerate only formats that are supported for input (recording).

ACM_FORMATENUMF_NCHANNELS
The **nChannels** member of the **WAVEFORMATEX** structure pointed to by the **pwfxEnum** member is valid. The enumerator will enumerate only a format that conforms to this attribute.

ACM_FORMATENUMF_NSAMPLESPERSEC
The **nSamplesPerSec** member of the **WAVEFORMATEX** structure pointed to by the **pwfxEnum** member is valid. The enumerator will enumerate only a format that conforms to this attribute.

ACM_FORMATENUMF_OUTPUT
The enumerator should enumerate only formats that are supported for output (playback).

ACM_FORMATENUMF_SUGGEST
The **WAVEFORMATEX** structure pointed to by the **pwfxEnum** member is valid. The enumerator will enumerate all suggested destination formats for the given **pwfxEnum** format.

ACM_FORMATENUMF_WBITSPERSAMPLE
The **wBitsPerSample** member of the **WAVEFORMATEX** structure pointed to by the **pwfxEnum** member is valid. The enumerator will enumerate only a format that conforms to this attribute.

ACM_FORMATENUMF_WFORMATTAG
The **wFormatTag** member of the **WAVEFORMATEX** structure pointed to by the **pwfxEnum** member is valid. The enumerator will enumerate only a format that conforms to this attribute.

pwfxEnum
Address of a **WAVEFORMATEX** structure that will be used to restrict the formats listed in the dialog box. The **fdwEnum** member defines the members of the structure pointed to by **pwfxEnum** that should be used for the enumeration restrictions. If no special restrictions are desired, this member can be NULL. For other requirements associated with the **pwfxEnum** member, see the description for the **acmFormatEnum** function.

hInstance
Handle of a data block that contains a dialog box template specified by the **pszTemplateName** member. This member is used only if the **fdwStyle** member specifies the ACMFORMATCHOOSE_STYLEF_ENABLETEMPLATE or ACMFORMATCHOOSE_STYLEF_ENABLETEMPLATEHANDLE flag; otherwise, this member should be NULL on input.

pszTemplateName
Address of a null-terminated string that specifies the name of the resource file for the dialog box template that is to be substituted for the dialog box template in the ACM. An application can use the **MAKEINTRESOURCE** macro for numbered dialog box resources. This member is used only if the **fdwStyle** member specifies the ACMFORMATCHOOSE_STYLEF_ENABLETEMPLATE flag; otherwise, this member should be NULL on input.

lCustData
Application-defined data that the ACM passes to the hook function identified by the **pfnHook** member. The system passes the data in the *lParam* parameter of the WM_INITDIALOG message.

pfnHook
Address of a hook function that processes messages intended for the dialog box. An application must specify the ACMFORMATCHOOSE_STYLEF_ENABLEHOOK flag in the **fdwStyle** member to enable the hook; otherwise, this member should be NULL. The hook function should return FALSE to pass a message to the standard dialog box procedure or TRUE to discard the message.

ACMFORMATDETAILS

```
typedef struct {
    DWORD          cbStruct;
    DWORD          dwFormatIndex;
    DWORD          dwFormatTag;
    DWORD          fdwSupport;
    LPWAVEFORMATEX pwfx;
    DWORD          cbwfx;
    char szFormat[ACMFORMATDETAILS_FORMAT_CHARS];
} ACMFORMATDETAILS;
```

Details a waveform-audio format for a specific format tag for an ACM driver.

cbStruct
Size, in bytes, of the **ACMFORMATDETAILS** structure. This member must be initialized before an application calls the **acmFormatDetails** or **acmFormatEnum** function. The size specified by this member must be large enough to contain the base **ACMFORMATDETAILS** structure. When the **acmFormatDetails** function returns, this member contains the actual size of the information returned. The returned information will never exceed the requested size.

dwFormatIndex
Index of the format to retrieve details for. The index ranges from zero to one less than the number of standard formats supported by an ACM driver for a format tag. The number of standard formats supported by a driver for a format tag is contained in the **cStandardFormats** member of the **ACMFORMATTAGDETAILS** structure. The **dwFormatIndex** member is used only when an application queries standard format details about a driver by index; otherwise, this member should be zero. Also, this member will be set to zero by the ACM when an application queries for details on a format; in other words, this member is used only for input and is never returned by the ACM or an ACM driver.

dwFormatTag
Waveform-audio format tag that the **ACMFORMATDETAILS** structure describes. This member is used for input for the ACM_FORMATDETAILSF_INDEX query flag. For the ACM_FORMATDETAILSF_FORMAT query flag, this member must be initialized to the same format tag as the **pwfx** member specifies. If a call to the **acmFormatDetails** function is successful, this member is always returned. This member should be set to WAVE_FORMAT_UNKNOWN for all other query flags.

fdwSupport
Driver-support flags specific to the specified format. These flags are identical to the **fdwSupport** flags of the **ACMDRIVERDETAILS** structure. This member can be a combination of the following values and indicates which operations the driver supports for the format tag:

ACMDRIVERDETAILS_SUPPORTF_ASYNC
Driver supports asynchronous conversions with the specified format tag.

ACMDRIVERDETAILS_SUPPORTF_CODEC
Driver supports conversion between two different format tags for the specified format. For example, if a driver supports compression from WAVE_FORMAT_PCM to WAVE_FORMAT_ADPCM with the specified format, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_CONVERTER
Driver supports conversion between two different formats of the same format tag while using the specified format. For example, if a driver supports resampling of WAVE_FORMAT_PCM to the specified format, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_FILTER
Driver supports a filter (which modifies data without changing any format attributes) with the specified format. For example, if a driver supports volume or echo operations on

WAVE_FORMAT_PCM, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_HARDWARE
Driver supports hardware input and/or output of the specified format through a waveform-audio device. An application should use **acmMetrics** with the ACM_METRIC_HARDWARE_WAVE_INPUT and ACM_METRIC_HARDWARE_WAVE_OUTPUT metric indexes to get the waveform-audio device identifiers associated with the supporting ACM driver.

pwfx
Address of a **WAVEFORMATEX** structure that will receive the format details. This structure requires no initialization by the application unless the ACM_FORMATDETAILSF_FORMAT flag is specified in the **acmFormatDetails** function. In this case, the **wFormatTag** member of the **WAVEFORMATEX** structure must be equal to the **dwFormatTag** of the **ACMFORMATDETAILS** structure.

cbwfx
Size, in bytes, available for **pwfx** to receive the format details. The **acmMetrics** and **acmFormatTagDetails** functions can be used to determine the maximum size required for any format available for the specified driver (or for all installed ACM drivers).

szFormat
String that describes the format for the **dwFormatTag** type. If the **acmFormatDetails** function is successful, this string is always returned.

ACMFORMATTAGDETAILS

```
typedef struct {
    DWORD cbStruct;
    DWORD dwFormatTagIndex;
    DWORD dwFormatTag;
    DWORD cbFormatSize;
    DWORD fdwSupport;
    DWORD cStandardFormats;
    char szFormatTag[ACMFORMATTAGDETAILS_FORMATTAG_CHARS];
} ACMFORMATTAGDETAILS;
```

Details a waveform-audio format tag for an ACM driver.

cbStruct
Size, in bytes, of the **ACMFORMATTAGDETAILS** structure. This member must be initialized before an application calls the **acmFormatTagDetails** or **acmFormatTagEnum** function. The size specified by this member must be large enough to contain the base **ACMFORMATTAGDETAILS** structure. When the **acmFormatTagDetails** function returns, this member contains the actual size of the information returned. The returned information will never exceed the requested size.

dwFormatTagIndex
Index of the format tag for which details will be retrieved. The index ranges from zero to one less than the number of format tags supported by an ACM driver. The number of format tags supported by a driver is contained in the **cFormatTags** member of the **ACMDRIVERDETAILS** structure. The **dwFormatTagIndex** member is used only when querying format tag details on a driver by index; otherwise, this member should be zero.

dwFormatTag
Waveform-audio format tag that the **ACMFORMATTAGDETAILS** structure describes. This member is used for input for the ACM_FORMATTAGDETAILSF_FORMATTAG and ACM_FORMATTAGDETAILSF_LARGESTSIZE query flags. If the **acmFormatTagDetails** function is successful, this member is always returned. This member should be set to WAVE_FORMAT_UNKNOWN for all other query flags.

cbFormatSize
Largest total size, in bytes, of a waveform-audio format of the **dwFormatTag** type. For example, this member will be 16 for WAVE_FORMAT_PCM and 50 for WAVE_FORMAT_ADPCM.

fdwSupport
Driver-support flags specific to the format tag. These flags are identical to the **fdwSupport** flags of the **ACMDRIVERDETAILS** structure. This member may be some combination of the following values and refer to what operations the driver supports with the format tag:

ACMDRIVERDETAILS_SUPPORTF_ASYNC
Driver supports asynchronous conversions with the specified format tag.

ACMDRIVERDETAILS_SUPPORTF_CODEC
Driver supports conversion between two different format tags where one of the tags is the specified format tag. For example, if a driver supports compression from WAVE_FORMAT_PCM to WAVE_FORMAT_ADPCM, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_CONVERTER
Driver supports conversion between two different formats of the specified format tag. For example, if a driver supports resampling of WAVE_FORMAT_PCM, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_FILTER
Driver supports a filter (modification of the data without changing any of the format attributes). For example, if a driver supports volume or echo operations on the specified format tag, this flag is set.

ACMDRIVERDETAILS_SUPPORTF_HARDWARE

Driver supports hardware input, output, or both of the specified format tag through a waveform-audio device. An application should use the **acmMetrics** function with the ACM_METRIC_HARDWARE_WAVE_INPUT and ACM_METRIC_HARDWARE_WAVE_OUTPUT metric indexes to get the waveform-audio device identifiers associated with the supporting ACM driver.

cStandardFormats

Number of standard formats of the **dwFormatTag** type; that is, the combination of all sample rates, bits per sample, channels, and so on. This value can specify all formats supported by the driver, but not necessarily.

szFormatTag

String that describes the **dwFormatTag** type. If the **acmFormatTagDetails** function is successful, this string is always returned.

ACMSTREAMHEADER

```
typedef struct {
    DWORD  cbStruct;
    DWORD  fdwStatus;
    DWORD  dwUser;
    LPBYTE pbSrc;
    DWORD  cbSrcLength;
    DWORD  cbSrcLengthUsed;
    DWORD  dwSrcUser;
    LPBYTE pbDst;
    DWORD  cbDstLength;
    DWORD  cbDstLengthUsed;
    DWORD  dwDstUser;
    DWORD  dwReservedDriver[10];
} ACMSTREAMHEADER;
```

Defines the header used to identify an ACM conversion source and destination buffer pair for a conversion stream.

cbStruct
Size, in bytes, of the **ACMSTREAMHEADER** structure. This member must be initialized before the application calls any ACM stream functions using this structure. The size specified in this member must be large enough to contain the base **ACMSTREAMHEADER** structure.

fdwStatus
Flags giving information about the conversion buffers. This member must be initialized to zero before the application calls the **acmStreamPrepareHeader** function and should not be modified by the application while the stream header remains prepared.

ACMSTREAMHEADER_STATUSF_DONE
Set by the ACM or driver to indicate that it is finished with the conversion and is returning the buffers to the application.

ACMSTREAMHEADER_STATUSF_INQUEUE
Set by the ACM or driver to indicate that the buffers are queued for conversion.

ACMSTREAMHEADER_STATUSF_PREPARED
Set by the ACM to indicate that the buffers have been prepared by using the **acmStreamPrepareHeader** function.

dwUser
User data. This can be any instance data specified by the application.

pbSrc
Address of the source buffer. This pointer must always refer to the same location while the stream header remains prepared. If an application needs to change the source location, it must unprepare the header and reprepare it with the alternate location.

cbSrcLength
Length, in bytes, of the source buffer pointed to by **pbSrc**. When the header is prepared, this member must specify the maximum size that will be used in the source buffer. Conversions can be performed on source lengths less than or equal to the original prepared size. However, this member must be reset to the original size when an application unprepares the header.

cbSrcLengthUsed
Amount of data, in bytes, used for the conversion. This member is not valid until the conversion is complete. This value can be less than or equal to **cbSrcLength**. An application must use the **cbSrcLengthUsed** member when advancing to the next piece of source data for the conversion stream.

dwSrcUser

User data. This can be any instance data specified by the application.

pbDst
Address of the destination buffer. This pointer must always refer to the same location while the stream header remains prepared. If an application needs to change the destination location, it must unprepare the header and reprepare it with the alternate location.

cbDstLength
Length, in bytes, of the destination buffer pointed to by **pbDst**. When the header is prepared, this member must specify the maximum size that will be used in the destination buffer.

cbDstLengthUsed
Amount of data, in bytes, returned by a conversion. This member is not valid until the conversion is complete. This value can be less than or equal to **cbDstLength**. An application must use the **cbDstLengthUsed** member when advancing to the next destination location for the conversion stream.

dwDstUser
User data. This can be any instance data specified by the application.

dwReservedDriver
Reserved; do not use. This member requires no initialization by the application and should never be modified while the header remains prepared.

Before an **ACMSTREAMHEADER** structure can be used for a conversion, it must be prepared by using the **acmStreamPrepareHeader** function. When an application is finished with an **ACMSTREAMHEADER** structure, it must call the **acmStreamUnprepareHeader** function before freeing the source and destination buffers.

WAVEFILTER

```
typedef struct {
    DWORD cbStruct;
    DWORD dwFilterTag;
    DWORD fdwFilter;
    DWORD dwReserved[5];
} WAVEFILTER;
```

Defines a filter for waveform-audio data. Only filter information common to all waveform-audio data filters is included in this structure. For filters that require additional information, this structure is included as the first member in another structure along with the additional information.

cbStruct
Size, in bytes, of the **WAVEFILTER** structure. The size specified in this member must be large enough to contain the base **WAVEFILTER** structure.

dwFilterTag
Waveform-audio filter type. Filter tags are registered with Microsoft Corporation for various filter algorithms.

fdwFilter
Flags for the **dwFilterTag** member. The flags defined for this member are universal to all filters. Currently, no flags are defined.

dwReserved
Reserved for system use; should not be examined or modified by an application.

Musical Instrument Digital Interface (MIDI)

The Musical Instrument Digital Interface (MIDI) is a protocol and set of commands for storing and transmitting information about music. MIDI output devices interpret this information and use it to synthesize music.

The Microsoft Win32 application programming interface (API) provides the following methods for applications to work with MIDI data:

- The Media Control Interface (MCI). Although the simplest way to play a MIDI file is to use the MCIWnd window class, you can also use MCI commands to create a customized interface to a MIDI device. For more information about the MCIWnd window class, see Chapter 2, "Getting Started Using MCIWnd." For more information about MCI, see Chapter 3, "MCI Overview," or the next section, "Media Control Interface (MCI)".
- Stream buffers. This format allows an application to manipulate buffers of time-stamped MIDI data for playback. Stream buffers are useful to applications that require more precise control over output than MCI offers.
- Low-level MIDI services. Applications that need the most precise control of MIDI data typically use these low-level services.

This chapter discusses each of these methods, provides an overview of the MIDI services of the Microsoft Windows operating system, and contains a detailed reference to the MIDI-related functions, structures, and messages.

Media Control Interface (MCI)

The MCI system component that plays MIDI files is the MCI MIDI sequencer. Applications can play MIDI files easily using MCI, but MCI imposes the following restrictions on MIDI capabilities:

- MCI supports MIDI output only.
- MCI does not allow close synchronization between MIDI events and other real-time events (such as video).

If you need accurate MIDI synchronization, you must use the stream buffers or the low-level MIDI services. If you need MIDI input capabilities, you must use the low-level MIDI services.

The MCI MIDI sequencer plays standard MIDI files and resource interchange file format (RIFF) MIDI files, known as RMID files. Standard MIDI files conform to the *Standard MIDI Files 1.0* specification. Because RMID files are standard MIDI files with a RIFF header, information about standard MIDI files also applies to RMID files. For more information about RIFF files, see Chapter 15, "File Input and Output."

Although there are currently three kinds of standard MIDI files, the MCI sequencer plays only two of them: Format 0 and Format 1 MIDI files.

For more information about controlling multimedia devices (including sequencers) using MCI commands, see Chapter 3, "MCI Overview."

Stream Buffers

Applications can use stream buffers to send streams of MIDI events to a device. Each stream buffer is a block of memory pointed to by a **MIDIHDR** structure. This block of memory contains data for one or more MIDI events, each of which is defined by a **MIDIEVENT** structure. An application controls the buffer by calling the stream-manipulation functions, such as **midiStreamOpen**, **midiStreamOut**, and **midiStreamClose**.

Stream Buffer Format

The **lpData** member of the **MIDIHDR** structure points to a stream buffer, and the **dwBufferLength** member specifies the actual size of this buffer. The **dwBytesRecorded** member of **MIDIHDR** specifies the number of bytes in the buffer that are actually used by the MIDI events; this value must be less than or equal to the value specified by **dwBufferLength**.

Each of the MIDI events in the stream buffer is specified by a **MIDIEVENT** structure, which contains the time for the event, a stream identifier, an event code, and, when appropriate, parameters for the event. Each of these **MIDIEVENT** structures must begin on a doubleword boundary. If necessary, pad bytes must be added to the end of the structure to ensure that the next one starts on a doubleword boundary.

Timing Information

Timing information for a MIDI event is stored in the **dwDeltaTime** member of the **MIDIEVENT** structure. Time is given in ticks, as defined in the *Standard MIDI Files 1.0* specification. The length of a tick is defined by the time format and possibly the tempo associated with the stream. For more information about streams, see MIDI Streams.

A tick is expressed either as microseconds per quarter note or as ticks of SMPTE (Society of Motion Picture and Television Engineers) time. Applications that send MIDI messages individually or use unprocessed MIDI messages use quarter note time and tempo information to determine the duration of a tick. Applications that preprocess MIDI messages can store the elapsed time as a count of the SMPTE units being used.

Quarter note time is indicated with a 0 in the high-word bit (bit 15) of the time-division word. The remainder of the word contains the ticks per quarter note. A tempo associated with a stream of MIDI data is kept in units (microseconds per quarter note) consistent with the *Standard MIDI Files 1.0* specification. For example, a quarter note in 4/4 time that uses a tempo of 500,000 microseconds per quarter note plays at the rate of 120 beats per minute.

SMPTE time division formats completely specify the length of a tick without the need for tempo information. Using SMPTE time formats, MIDI sequences can be synchronized exactly with other SMPTE events, such as video or striped audio. SMPTE time is indicated with a 1 in the high-order bit (bit 15) of the time-division word. The rest of the most-significant byte specifies the SMPTE format in use as negative values. The supported SMPTE formats and their corresponding values (in parentheses) are 24 (-24), 25 (-25), 30 (-30), and 30 drop (-29). The low byte of the time-division word specifies the number of ticks per SMPTE frame.

Event Types

The **dwEvent** member of the **MIDIEVENT** structure describes the MIDI event that is to take place. Short events fit entirely into this member. Long events require one or more doubleword values in addition to the **dwEvent** member to store the event descriptions.

The high byte of the **dwEvent** member contains information about whether the event is long or short and about whether a callback is generated along with the event. In addition, this byte is used to describe the event type. The remaining 24 bits of the **dwEvent** member are used either to contain the event parameters (for short messages) or to contain the length of the event parameters (for long messages). To extract information from the **dwEvent** member, use the **MEVT_EVENTTYPE** and **MEVT_EVENTPARM** macros .

For a description of the predefined event types, see the reference material for the **MIDIEVENT** structure.

MIDI Streams

MIDI events occur in the context of a stream of MIDI data. Although an application can use several streams to define musical data, the MIDI mapper does not recognize multiple streams. Most applications that use streams will use a single MIDI stream.

The following functions work with streams:

midiStreamClose	Closes a MIDI stream.
midiStreamOpen	Opens a MIDI stream and retrieves a handle of it.
midiStreamOut	Plays or queues a stream (buffer) of MIDI data to a MIDI output device.
midiStreamPause	Pauses playback of a specified MIDI stream.
midiStreamPosition	Retrieves the current position in a MIDI stream.
midiStreamProperty	Sets and retrieves stream properties.
midiStreamRestart	Restarts playback of a paused MIDI stream.
midiStreamStop	Turns off all notes on all MIDI channels for the specified MIDI stream.

Low-Level MIDI Services

Most applications will be able to use the MCI MIDI sequencer or stream buffers (and the **midiStreamOut** function) to implement all the MIDI functionality they need. Serious MIDI developers – those producing MIDI authoring or sequencing tools – can use either a combination of the stream capabilities and the low-level MIDI services or use only the low-level services. This section presents general information about using the low-level MIDI services.

Querying MIDI Devices

Before playing or recording MIDI data, you must determine the capabilities of the MIDI hardware present in the system. MIDI capability can vary from one multimedia computer to the next; applications should not make assumptions about the hardware present in a given system.

Windows provides the following functions to determine how many MIDI devices are available for input or output in a given system:

midiInGetNumDevs	Retrieves the number of MIDI input devices present in the system.
midiOutGetNumDevs	Retrieves the number of MIDI output devices present in the system.

Like other audio devices, MIDI devices are identified by a device identifier, which is determined implicitly from the number of devices present in a given system. Device identifiers range from zero to one less than the number of devices present. For example, if there are two MIDI output devices in a system, valid device identifiers are 0 and 1.

After you determine how many MIDI input or output devices are present in a system, you can inquire about the capabilities of each device. Windows provides the following functions to determine the capabilities of audio devices:

midiInGetDevCaps	Retrieves the capabilities of a given MIDI input device and places this information in the **MIDIINCAPS** structure.
midiOutGetDevCaps	Retrieves the capabilities of a given MIDI output device and places this information in the **MIDIOUTCAPS** structure.

Each of these functions has a parameter specifying the address of a structure that the function fills with information about the capabilities of a specified device.

Opening and Closing Device Drivers

You must open a MIDI device before using it, and you should close the device as soon as you finish using it. Windows provides the following functions to open and close different types of MIDI devices:

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

Each function that opens a MIDI device takes as parameters a device identifier, an address of a memory location, and some parameters unique to MIDI devices. The memory location is filled with a device handle, which is used to identify the open audio device in calls to other audio functions.

Many MIDI functions can accept either a device handle or a device identifier. Although you can use a device handle wherever you would use a device identifier, you cannot always use a device identifier when a handle is called for.

Note MIDI devices are not guaranteed to be shareable, so a particular device might not be available when a user requests it. If this happens, the applications should notify the user and allow the user to try to open the device again.

Allocating and Preparing MIDI Data Blocks

The **midiOutLongMsg**, **midiInAddBuffer**, and **midiStreamOut** functions require applications to allocate data blocks to pass to the device drivers for playback or recording purposes. Each of these functions uses a **MIDIHDR** structure to describe its data block.

Before you use one of these functions to pass a data block to a device driver, you must allocate memory for the buffer and the header structure that describes the data block.

Windows provides the following functions for preparing and cleaning up MIDI data blocks:

midiInPrepareHeader	Prepares a MIDI input data block.
midiInUnprepareHeader	Cleans up the preparation of a MIDI input data block.
midiOutPrepareHeader	Prepares a MIDI output data block.
midiOutUnprepareHeader	Cleans up the preparation of a MIDI output data block.

Before you pass a MIDI data block to a device driver, you must prepare the buffer by passing it to the **midiInPrepareHeader** or **midiOutPrepareHeader** function. When the device driver is finished with the buffer and returns it, you must clean up this preparation by passing the buffer to the **midiInUnprepareHeader** or **midiOutUnprepareHeader** function before any allocated memory can be freed.

Managing MIDI Data Blocks

Applications that use data blocks for passing system-exclusive messages (using the **midiOutLongMsg** and **midiInAddBuffer** functions) and stream buffers (using the **midiStreamOut** function) must continually supply the device driver with data blocks until playback or recording is complete.

Even if a single data block is used, applications must be able to determine when a device driver is finished with the data block so that the application can free the memory associated with the data block and header structure. Three methods can be used to determine when a device driver is finished with a data block:

- Specify a callback function to receive a message sent by the driver when it is finished with a data block. To get time-stamped MIDI input data, you must use a callback function.
- Use an event callback (for output only).
- Use a window or thread callback to receive a message sent by the driver when it is finished with a data block.

If an application does not get a data block to the device driver when it is needed, there can be an audible gap in playback or a loss of incoming recorded information. An application should use at least a double-buffering scheme to stay at least one data block ahead of the device driver.

Using a Callback Function to Process Driver Messages

You can write your own callback function to process messages sent by the device driver. To use a callback function, specify the CALLBACK_FUNCTION flag in the *dwFlags* parameter and the address of the callback function in the *dwCallback* parameter of the **midiInOpen** or **midiOutOpen** function.

Messages sent to a callback function are similar to messages sent to a window, except they have two doubleword parameters instead of a unsigned integer parameter and a doubleword parameter. For more information about these messages, see "Sending System-Exclusive Messages" later in this chapter and "Managing MIDI Recording" later in this chapter.

Use one of the following techniques to pass instance data from an application to a callback function:

- Use the *dwCallbackInstance* parameter of the function that opens the device driver.
- Use the **dwUser** member of the **MIDIHDR** structure that identifies a data block being sent to a MIDI device driver.

If you need more than 32 bits of instance data, pass an address of a structure containing the additional information.

Using an Event Callback to Process Driver Messages

To use an event callback, use the **CreateEvent** function to retrieve the handle of an event and specify CALLBACK_EVENT in the call to the **midiOutOpen** function.

An event callback is set by anything that might cause a function callback. Unlike callback functions and window or thread callbacks, event callbacks do not receive specific close, done, or open notifications. An application might, therefore, have to check the status of the process it is waiting for after the event occurs.

For more information about event callbacks, see "Using an Event Callback to Manage Buffered Playback" later in this chapter.

Using a Window or Thread Callback to Process Driver Messages

To use a window callback, specify the CALLBACK_WINDOW flag in the *dwFlags* parameter and a window handle in the low-order word of the *dwCallback* parameter of the **midiInOpen** or **midiOutOpen** function. Driver messages will be sent to the window procedure function for the window identified by the handle in *dwCallback*.

Similarly, to use a thread callback, specify the CALLBACK_THREAD flag and a thread identifier in the call to **midiInOpen** or **midiOutOpen**. In this case, messages will be posted to the specified thread instead of to a window.

Messages sent to a window or thread callback are specific to the MIDI device used. For more information about these messages, see "Sending System-Exclusive Messages" later in this chapter and "Managing MIDI Recording" later in this chapter.

Requesting Time Formats

Windows uses the **MMTIME** structure to represent time in one or more different formats, including milliseconds, samples, SMPTE, and MIDI song pointer formats. The **wType** member specifies the time format.

The **midiStreamPosition** function uses the **MMTIME** structure. Before calling this function, you must set the **wType** member to indicate your requested time format. To see if the requested time format is supported, check **wType** after the call. If the requested time format is not supported, the time is specified in an alternate time format selected by the device driver and the **wType** member is changed to indicate the selected time format.

For more information about the **MMTIME** structure, see Chapter 17, "Timers."

Handling Errors with MIDI Functions

Low-level audio functions return a nonzero error code. For MIDI-associated errors, the **midiInGetErrorText** and **midiOutGetErrorText** functions retrieve textual descriptions for the error codes. The application must still look at the error value itself to determine how to proceed, but it can use the error descriptions in dialog boxes to inform users of the error conditions.

The only low-level MIDI functions that do not return error codes are the **midiInGetNumDevs** and **midiOutGetNumDevs** functions. These functions return a value of zero if no devices are present in a system or if any errors are encountered by the function.

Playing MIDI Files

You should use the MCI MIDI sequencer to play MIDI files whenever you can. If the sequencer services do not meet the needs of your application, you can manage MIDI playback by using stream buffers or the low-level MIDI services.

MIDI Output Data Types

Windows defines the following data types for low-level MIDI output functions:

HMIDIOUT	Handle of a MIDI output device.
MIDIHDR	Header for a block of MIDI system-exclusive or stream data.
MIDIOUTCAPS	Structure used to inquire about the capabilities of a particular MIDI output device.

Querying MIDI Output Devices

Before playing a MIDI file, you should use the **midiOutGetDevCaps** function to determine the capabilities of the MIDI output device that is present in the system. This function takes an address of a **MIDIOUTCAPS** structure, which it fills with information about the capabilities of the given device. This information includes the manufacturer and product identifiers, a product name for the device, and the version number of the device driver (specified in the **wMid**, **wPid**, **szPname**, and **vDriverVersion** members, respectively).

MIDI output devices can be either internal synthesizers or external MIDI output ports. The **wTechnology** member of the **MIDIOUTCAPS** structure specifies the technology of the device.

If the device is an internal synthesizer, additional device information is available in the **wVoices**, **wNotes**, and **wChannelMask** members. The **wVoices** member specifies the number of voices that the device supports. Each voice can have a different sound or timbre. Voices are organized into MIDI channels. The **wNotes** member specifies the *polyphony* of the device – that is, the maximum number of notes that can be played simultaneously. The **wChannelMask** member is a bit representation of the MIDI channels that the device responds to. For example, if the device responds to the first eight MIDI channels, **wChannelMask** is 0x00FF. If the device is an external output port, **wVoices** and **wNotes** are unused, and **wChannelMask** is set to 0xFFFF.

The **dwSupport** member of the **MIDIOUTCAPS** structure indicates whether the device driver supports volume changes, patch caching, and streaming. Volume changes are supported only by internal synthesizer devices. External MIDI output ports do not support volume changes. For information about changing volume, see "Changing Internal MIDI Synthesizer Volume" later in this chapter.

Opening MIDI Output Devices

To open a MIDI output device for playback, use the **midiOutOpen** function. This function opens the device associated with the specified device identifier and returns a handle of the open device by writing the handle to a specified memory location.

One of the parameters of **midiOutOpen** is a doubleword value that specifies a window or thread handle, an event handle, or the address of a callback function that is used to monitor the progress of the playback of MIDI system-exclusive data and stream buffers. Monitoring enables the application to determine when to send additional data blocks and when to free data blocks that have been sent. For more information about these methods, see "Managing MIDI Data Blocks" earlier in this chapter.

Sending MIDI Messages with Stream Buffers

When your application works with stream buffers, it uses the **midiStreamOut** function to send all (short and long) MIDI messages to the device. To specify stream data blocks, use the **MIDIHDR** and **MIDIEVENT** structures. The **MIDIHDR** structure contains an address of a locked data block, the data-block length, and some assorted flags. The data is stored in the form of **MIDIEVENT** structures. The system imposes a size limit of 64K on stream buffers.

After you use **midiStreamOut** to send a stream buffer of data, you must wait until the device driver is finished with the data block before freeing it. If you are sending multiple data blocks, you must monitor the completion of each data block so that you know when to send additional blocks. For information about different techniques for monitoring data-block completion, see "Managing MIDI Data Blocks" earlier in this chapter.

Sending Individual MIDI Messages

You can work with individual MIDI messages by using the following functions:

midiOutLongMsg	Sends a buffer of MIDI data to the specified MIDI output device. Use this function to send system-exclusive messages to a MIDI device.
midiOutReset	Turns off all notes on all channels for a specified MIDI output device. Any pending system-exclusive buffers and stream buffers are marked as done and returned to the application.
midiOutShortMsg	Sends a MIDI message to a specified MIDI output device.

To send any MIDI message (except for system-exclusive messages), use **midiOutShortMsg**.

Sending System-Exclusive Messages

MIDI system-exclusive messages are the only MIDI messages that will not fit into a single doubleword value. System-exclusive messages can be any length. Windows provides the **midiOutLongMsg** function for sending system-exclusive messages to MIDI output devices. To specify MIDI system-exclusive data blocks, use the **MIDIHDR** structure.

After you send a system-exclusive data block using **midiOutLongMsg**, you must wait until the device driver is finished with the data block before freeing it. If you are sending multiple data blocks, you must monitor the completion of each data block so that you know when to send additional blocks. For information about different techniques for monitoring data-block completion, see "Managing MIDI Data Blocks" earlier in this chapter.

Note Any MIDI status byte other than a system - real-time message will terminate a system-exclusive message. If you are using multiple data blocks to send a single system-exclusive message, do not send any MIDI messages other than system - real-time messages between data blocks.

Using a Window or Thread to Manage Buffered Playback

The following messages can be sent to a window or thread for managing playback of MIDI system-exclusive messages or stream buffers:

MM_MOM_CLOSE	Sent when the device is closed by using the **midiOutClose** function.
MM_MOM_DONE	Sent when the device driver is finished with a data block sent by using the **midiOutLongMsg** or **midiStreamOut** function.
MM_MOM_OPEN	Sent when the device is opened by using the **midiOutOpen** function.

A *wParam* parameter and an *lParam* parameter are associated with each of these messages. The *wParam* parameter always specifies the handle of an open MIDI device. For MM_MOM_DONE, *lParam* specifies an address of a **MIDIHDR** structure identifying the completed data block. The *lParam* parameter is unused for MM_MOM_CLOSE and MM_MOM_OPEN.

The most useful message is probably MM_MOM_DONE. Unless you need to allocate memory or initialize variables, you probably do not need to process MM_MOM_OPEN and MM_MOM_CLOSE. When playback of a data block is complete, you can clean up and free the data block.

Using a Callback Function to Manage Buffered Playback

You can define your own callback function to manage buffered playback of MIDI output devices. The callback function is documented as **MidiOutProc** in the Reference section of this chapter.

The following messages can be sent to the *wMsg* parameter of the **MidiOutProc** callback function.

MOM_CLOSE	Sent when the device is closed by using the **midiOutClose** function.
MOM_DONE	Sent when the device driver is finished with a data block sent by using the **midiOutLongMsg** or **midiStreamOut** function.
MOM_OPEN	Sent when the device is opened by using the **midiOutOpen** function.

These messages are similar to those sent to window procedure functions, but the parameters are different. A handle of the open MIDI device is passed as a parameter to the callback function, along with the doubleword of instance data passed by using **midiOutOpen**.

After the driver is finished with a data block, you can clean up and free the data block.

Using an Event Callback to Manage Buffered Playback

To use an event callback, use the **CreateEvent** function to retrieve the handle of an event. In a call to the **midiOutOpen** function, specify CALLBACK_EVENT for the *dwFlags* parameter. After using the **midiOutPrepareHeader** function but before sending MIDI events to the device, create a nonsignaled event by calling the **ResetEvent** function, specifying the event handle retrieved by **CreateEvent**. Then, inside a loop that checks whether the MHDR_DONE bit is set in the **dwFlags** member of the **MIDIHDR** structure, use the **WaitForSingleObject** function, specifying the event handle and a time-out value of INFINITE as parameters.

An event callback is set by anything that might cause a function callback.

Because event callbacks do not receive specific close, done, or open notifications, an application might have to check the status of the process it is waiting for after the event occurs. It is possible that a number of tasks could be completed by the time **WaitForSingleObject** returns.

Resetting MIDI Output

The **midiOutReset** function turns off all notes on all MIDI channels for a specified MIDI device. Then the function marks any pending system-exclusive buffers as done and returns them to the application. This function can be useful in an application that uses MIDI output to provide the user with the ability to reset MIDI output.

Note Terminating a system-exclusive message without sending an EOX (end-of-exclusive) byte might cause problems for the receiving device. The **midiOutReset** function does not send an EOX byte when it terminates a system-exclusive message – applications are responsible for doing this.

Changing Internal MIDI Synthesizer Volume

Windows provides the following functions to retrieve and set the volume level of internal MIDI synthesizer devices:

midiOutGetVolume	Retrieves the volume level of the specified internal MIDI synthesizer device.
midiOutSetVolume	Sets the volume level of the specified internal MIDI synthesizer device.

Not all MIDI output devices support volume changes. Some devices can support individual volume changes on both the left and right channels. For information about how to determine if a particular device supports volume changes, see "Querying MIDI Output Devices" earlier in this chapter.

Unless your application is designed to be a master volume-control application providing the user with volume control for all audio devices in a system, you should open an audio device before changing its volume. You should also check the volume level before changing it and restore the volume level to its previous level as soon as possible.

Volume is specified as a doubleword value. The upper 16 bits specify the relative volume of the right channel, and the lower 16 bits specify the relative volume of the left channel.

For devices that do not support individual volume changes on both the left and right channels, the lower 16 bits specify the volume level and the upper 16 bits are ignored. Values for the volume level range from 0x0 (silence) to 0xFFFF (maximum volume) and are interpreted logarithmically. The perceived volume increase is the same when increasing the volume level from 0x5000 to 0x6000 as it is from 0x4000 to 0x5000.

Preloading Patches with Internal MIDI Synthesizers

Some internal MIDI synthesizer devices cannot keep all of their patches loaded simultaneously. These devices must preload their patch data.

Windows provides the following functions to request that a synthesizer preload and cache specified patches:

midiOutCachePatches	Requests that an internal MIDI synthesizer device preload and cache specified melodic patches.
midiOutCacheDrumPatches	Requests that an internal MIDI synthesizer device preload and cache specified key-based percussion patches.

For information about how to determine if a particular device supports preloading patches, see "Querying MIDI Output Devices" earlier in this chapter.

Recording MIDI Audio

To record MIDI audio data, you must use low-level MIDI input functions. MCI does not provide a device handler for recording MIDI audio.

MIDI Input Data Types

Windows defines the following data types for low-level MIDI input functions:

HMIDIIN	Handle of a MIDI input device.
MIDIHDR	Header for a stream buffer or a block of MIDI system-exclusive data. For input applications, this structure records only system-exclusive data (streaming is not supported for MIDI input).
MIDIINCAPS	Structure used to inquire about the capabilities of a MIDI input device.

Querying MIDI Input Devices

Before recording MIDI audio, you should use the **midiInGetDevCaps** function to determine the capabilities of the MIDI input device that is present in the system. This function takes an address of a **MIDIINCAPS** structure, which it fills with information about the capabilities of the given device. This information includes the manufacturer and product identifiers, a product name for the device, and the version number of the device driver.

Opening MIDI Input Devices

To open a MIDI input device for recording, use the **midiInOpen** function. This function opens the device associated with the specified device identifier and returns a handle of the open device by writing the handle to a specified memory location.

If you use the MIDI_IO_STATUS flag with **midiInOpen**, the system uses the MIM_MOREDATA message to alert your application's callback function whenever it is not processing MIDI data fast enough to keep up with the input device driver. (The MM_MIM_MOREDATA message does the same job with window callbacks, but most applications will use callback functions instead of window callbacks, for performance reasons.) If your application processes MIDI data in a separate thread, boosting the thread's priority can have a significant impact on the application's ability to keep up with the data flow.

Managing MIDI Recording

After you open a MIDI device, you can begin recording MIDI data. Windows provides the following functions for managing MIDI recording:

midiInAddBuffer	Sends a buffer to the device driver so that it can be filled with recorded system-exclusive MIDI data.
midiInReset	Stops MIDI recording and marks all pending buffers as done.
midiInStart	Starts MIDI recording and resets the time stamp to zero.
midiInStop	Stops MIDI recording.

To send buffers to the device driver for recording system-exclusive messages, use **midiInAddBuffer**. The application is notified as the buffers are filled with system-exclusive recorded data. For more information about the notification techniques, see "Managing MIDI Data Blocks" earlier in this chapter.

The **midiInStart** function begins the recording process. When recording system-exclusive messages, send at least one buffer to the driver before starting recording. To stop recording, use **midiInStop**. Before closing the device by using the **midiInClose** function, mark any pending data blocks as being done by calling **midiInReset**.

Applications that require time-stamped data use a callback function to receive MIDI data. If your timing requirements are not strict, you can use a window or thread callback. You cannot use an event callback to receive MIDI data, however.

To record system-exclusive messages with applications that do not use stream buffers, you must supply the device driver with buffers. These buffers are specified by using a **MIDIHDR** structure.

Managing MIDI Thru

It is possible to connect a MIDI input device directly to a MIDI output device so that whenever the input device receives an MIM_DATA message, the system sends a message with the same MIDI event data to the output device driver. To connect a MIDI output device to a MIDI input device, use the **midiConnect** function.

To achieve the best possible performance with multiple outputs, an application can choose to supply a special form of MIDI output driver, called a *thru driver*. Although the system allows only one MIDI output device to be connected to a MIDI input device, multiple MIDI output devices can be connected to a thruing driver. An application on such a system could connect the MIDI input device to this thru device and connect the MIDI thru device to as many MIDI output devices as needed. For more information about thru drivers, see the Windows device-driver documentation.

Using Messages to Manage MIDI Recording

The following messages can be sent to a window or thread callback procedure for managing MIDI recording:

MM_MIM_CLOSE	Sent when a MIDI input device is closed by using the **midiInClose** function.
MM_MIM_DATA	Sent when a complete MIDI message is received. (This message is used for all MIDI messages except system-exclusive messages.)
MM_MIM_ERROR	Sent when an invalid MIDI message is received. (This message is used for all MIDI messages except system-exclusive messages.)
MM_MIM_LONGDATA	Sent when either a complete MIDI system-exclusive message is received or when a buffer has been filled with system-exclusive data.
MM_MIM_LONGERROR	Sent when an invalid MIDI system-exclusive message is received.
MM_MIM_MOREDATA	Sent when an application is not processing MIM_DATA messages fast enough to keep up with the input device driver.
MM_MIM_OPEN	Sent when a MIDI input device is opened by using the **midiInOpen** function.

A *wParam* parameter and an *lParam* parameter are associated with each of these messages. The *wParam* parameter always specifies the handle of an open MIDI device. The *lParam* parameter is unused for the MM_MIM_CLOSE and MM_MIM_OPEN messages.

For the MM_MIM_LONGDATA message, *lpMidiHdr* specifies an address of a **MIDIHDR** structure that identifies the buffer for system-exclusive messages. The buffer might not be completely filled, because you usually do not know the size of the system-exclusive messages before recording them and must allocate buffers whose total size can contain the largest expected message. To determine the amount of valid data present in the buffer, use the **dwBytesRecorded** member of the **MIDIHDR** structure.

Using a Callback Function to Manage MIDI Recording

You can define your own callback function to manage recording for MIDI input devices. The callback function is documented as **MidiInProc** in the Reference section of this chapter.

The following messages can be sent to the *wMsg* parameter of the **MidiInProc** callback function:

MIM_CLOSE	Sent when the device is closed by using the **midiInClose** function.

MIM_DATA	Sent when a complete MIDI message is received (this message is used for all MIDI messages except system-exclusive messages).
MIM_ERROR	Sent when an invalid MIDI message is received (this message is used for all MIDI messages except system-exclusive messages).
MIM_LONGDATA	Sent when either a complete MIDI system-exclusive message is received or when a buffer has been filled with system-exclusive data.
MIM_LONGERROR	Sent when an invalid MIDI system-exclusive message is received.
MIM_MOREDATA	Sent when an application is not processing MIM_DATA messages fast enough to keep up with the input device driver.
MIM_OPEN	Sent when the MIDI input device is opened by using the **midiInOpen** function.

These messages are similar to those sent to window procedure functions, but the parameters are different. A handle of the open MIDI device is passed as a parameter to the callback function, along with the doubleword of instance data that was passed by using **midiInOpen**.

For the MIM_LONGDATA message, *lpMidiHdr* specifies an address of a **MIDIHDR** structure that identifies the buffer for system-exclusive messages. The buffer might not be completely filled. To determine the amount of valid data present in the buffer, use the **dwBytesRecorded** member of the **MIDIHDR** structure.

After the device driver is finished with a data block, you can clean up and free the data block.

Receiving Time-Stamped MIDI Messages

Because of the delay between when the device driver receives a MIDI message and the time the application receives the message, MIDI input device drivers time stamp the MIDI message with the time that the message was received. MIDI time stamps, which are defined as the time the first byte of the message was received, are specified in milliseconds. The **midiInStart** function resets the time stamps for a device to zero.

As stated previously, to receive time stamps with MIDI input, you must use a callback function. The *dwParam2* parameter of the callback function specifies the time stamp for data associated with the MIM_DATA and MIM_LONGDATA messages.

Receiving Running-Status Messages

The *Standard MIDI Files 1.0* specification allows the use of *running status* when a message has the same status byte as the previous message. When running status is used, the status byte of subsequent messages can be omitted. All MIDI input device drivers are required to expand messages using running status into complete messages, so that you always receive complete MIDI messages from a MIDI input device driver.

Processing MIDI Data from Two MIDI Sources

The MIDI subsystem can route MIDI messages from two data sources to a single MIDI output device for concurrent playback. For example, one source could be background music or a bass line that has been pre-recorded and stored in a file. The second source could be live data from a MIDI instrument, such as a keyboard or guitar.

Both data sources send MIDI data to a single MIDI device that is identified with one handle. You send one data stream by using the **midiStreamOut** function and one or more stream buffers. This data stream typically contains prerecorded data that is packed into the buffer.

You send the second data stream (typically from a MIDI instrument) asynchronously by using the **midiOutShortMsg** function. The running status of a stream buffer will not be adversely affected by the asynchronous calls made by the second data stream.

Each short message sent with **midiOutShortMsg** must be a complete MIDI message, with a status byte and the appropriate number of data bytes. If the status byte is omitted, **midiOutShortMsg** returns an error. (There is no running status with stream output, however.)

Creating MIDI Files

When creating MIDI files, you should use the guidelines set forth in the General MIDI Level 1.0 specification published by the International MIDI Association. This document defines standard voice and percussive patch assignments for MIDI instruments.

For more information about the general MIDI specification, contact the International MIDI Association at the following address:

The International MIDI Association
23634 Emelita Street
Woodland Hills, CA 91367
Phone: (818) 598-0088

Using MIDI

This section contains examples demonstrating how to perform the following tasks:

- Use the MCI MIDI sequencer.
- Send individual MIDI messages.

Using the MCI MIDI Sequencer

Like all MCI devices, the MCI MIDI sequencer responds to standard MCI commands. This section discusses how to retrieve a sequence division type and how to retrieve and set a tempo. For more information about MCI, see Chapter 3, "MCI Overview."

Retrieving the Sequence Division Type

The *division type* of a MIDI sequence determines the amount of time between MIDI events in the sequence. To determine the division type of a sequence, use the **MCI_STATUS** command and set the **dwItem** member of the **MCI_STATUS_PARMS** structure to MCI_SEQ_STATUS_DIVTYPE .

If the **MCI_STATUS** command is successful, the **dwReturn** member of the **MCI_STATUS_PARMS** structure contains one of the following values to indicate the division type.

Value	Division type
MCI_SEQ_DIV_PPQN	PPQN (parts-per-quarter note)
MCI_SEQ_DIV_SMPTE_24	SMPTE, 24 fps (frames per second)
MCI_SEQ_DIV_SMPTE_25	SMPTE, 25 fps
MCI_SEQ_DIV_SMPTE_30	SMPTE, 30 fps
MCI_SEQ_DIV_SMPTE_30DROP	SMPTE, 30 fps drop frame

You must know the division type of a sequence to change or query its tempo. You cannot change the division type of a sequence by using the MCI sequencer.

Querying and Setting the Tempo

To retrieve the tempo of a sequence, use the **MCI_STATUS** command and set the **dwItem** member of the **MCI_STATUS_PARMS** structure to MCI_SEQ_STATUS_TEMPO. If the **MCI_STATUS** command is successful, the **dwReturn** member of the **MCI_STATUS_PARMS** structure contains the current tempo.

To change tempo, use the **MCI_SET** command with the **MCI_SEQ_SET_PARMS** structure, specifying the MCI_SEQ_SET_TEMPO flag and setting the **dwTempo** member of the structure to the desired tempo.

The way tempo is represented depends on the division type of the sequence. If the division type is PPQN, the tempo is represented in beats per minute. If the division type is one of the SMPTE division types, the tempo is represented in frames per second. For information about determining the division type of a sequence, see "Retrieving the Sequence Division Type" earlier in this chapter.

Using midiOutShortMsg to Send Individual MIDI Messages

The following example uses the **midiOutShortMsg** function to send a specified MIDI event to a given MIDI output device:

```
UINT sendMIDIEvent(HMIDIOUT hmo, BYTE bStatus, BYTE bData1,
    BYTE bData2)
{
    union {
        DWORD dwData;
        BYTE bData[4];
    } u;

    // Construct the MIDI message.

    u.bData[0] = bStatus;  // MIDI status byte
    u.bData[1] = bData1;   // first MIDI data byte
    u.bData[2] = bData2;   // second MIDI data byte
    u.bData[3] = 0;

    // Send the message.
    return midiOutShortMsg(hmo, u.dwData);
}
```

Note MIDI output drivers are not required to verify data before sending it to an output port. Applications must ensure that only valid data is sent.

MIDI Reference

This section describes the functions, macros, messages, and structures associated with the Musical Instrument Digital Interface (MIDI). These elements are grouped as follows.

Allocating and Managing Buffers

MIDIHDR
midiInAddBuffer
midiInPrepareHeader
midiInUnprepareHeader
midiOutPrepareHeader
midiOutUnprepareHeader

Callback Functions

MidiInProc
MidiOutProc

Device Capabilities

MIDIINCAPS
midiInGetDevCaps
midiInGetID
midiInGetNumDevs
MIDIOUTCAPS
midiOutGetDevCaps
midiOutGetID
midiOutGetNumDevs
MIDISTRMBUFFVER

Error Processing

midiInGetErrorText
midiOutGetErrorText
MIM_ERROR
MIM_LONGERROR
MM_MIM_ERROR
MM_MIM_LONGERROR

Managing MIDI Streams

midiStreamClose
midiStreamOpen
midiStreamOut
midiStreamPause
midiStreamPosition
midiStreamProperty
midiStreamRestart
midiStreamStop

Opening and Closing Devices

midiInClose
midiInOpen
midiOutClose
midiOutOpen
MIM_CLOSE
MIM_OPEN
MM_MIM_CLOSE
MM_MIM_OPEN
MM_MOM_CLOSE

MM_MOM_OPEN
MOM_CLOSE
MOM_OPEN

Output Devices

KEYARRAY
midiOutCacheDrumPatches
midiOutCachePatches
midiOutGetVolume
midiOutSetVolume
PATCHARRAY

Playing a Message or Messages

MEVT_EVENTPARM
MEVT_EVENTTYPE
MIDIEVENT
midiOutLongMsg
midiOutReset
midiOutShortMsg
midiStreamOut
midiStreamPause
midiStreamRestart
midiStreamStop
MM_MOM_DONE
MM_MOM_POSITIONCB
MOM_DONE
MOM_POSITIONCB

Recording

midiConnect
midiDisconnect
midiInReset
midiInStart
midiInStop
MIDIPROPTEMPO
MIDIPROPTIMEDIV
MIM_DATA
MIM_LONGDATA
MIM_MOREDATA
MM_MIM_DATA
MM_MIM_MOREDATA
MM_MIM_LONGDATA

Sending Messages to Devices

midiInMessage
midiOutMessage

midiConnect

```
MMRESULT midiConnect(HMIDI hMidi, HMIDIOUT hmo,
    LPVOID pReserved);
```

Connects a MIDI input device to a MIDI thru or output device, or connects a MIDI thru device to a MIDI output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MIDIERR_NOTREADY	Specified input device is already connected to an output device.
MMSYSERR_INVALHANDLE	Specified device handle is invalid.

hMidi
Handle of a MIDI input device or a MIDI thru device. (For thru devices, this handle must have been returned by a call to the **midiOutOpen** function.)

hmo
Handle of the MIDI output or thru device.

pReserved
Reserved; must be NULL.

After calling this function, the MIDI input device receives event data in an MIM_DATA message whenever a message with the same event data is sent to the output device driver.

A thru driver is a special form of MIDI output driver. The system will allow only one MIDI output device to be connected to a MIDI input device, but multiple MIDI output devices can be connected to a MIDI thru device. Whenever the given MIDI input device receives event data in an MIM_DATA message, a message with the same event data is sent to the given output device driver (or through the thru driver to the output drivers).

midiDisconnect

```
MMRESULT midiDisconnect(HMIDI hMidi, HMIDIOUT hmo,
    LPVOID pReserved);
```

Disconnects a MIDI input device from a MIDI thru or output device, or disconnects a MIDI thru device from a MIDI output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

  MMSYSERR_INVALHANDLE Specified device handle is invalid.

hMidi
Handle of a MIDI input device or a MIDI thru device.

hmo
Handle of the MIDI output device to be disconnected.

pReserved
Reserved; must be NULL.

MIDI input, output, and thru devices can be connected by using the **midiConnect** function. Thereafter, whenever the MIDI input device receives event data in an MIM_DATA message, a message with the same event data is sent to the output device driver (or through the thru driver to the output drivers).

midiInAddBuffer

```
MMRESULT midiInAddBuffer(HMIDIIN hMidiIn, LPMIDIHDR lpMidiInHdr,
    UINT cbMidiInHdr);
```

Sends an input buffer to a specified opened MIDI input device. This function is used for system-exclusive messages.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MIDIERR_STILLPLAYING	The buffer pointed to by *lpMidiInHdr* is still in the queue.
MIDIERR_UNPREPARED	The buffer pointed to by *lpMidiInHdr* has not been prepared.
MMSYSERR_INVALHANDLE	The specified device handle is invalid.
MMSYSERR_INVALPARAM	The specified pointer or structure is invalid.
MMSYSERR_NOMEM	The system is unable to allocate or lock memory.

hMidiIn
Handle of the MIDI input device.

lpMidiInHdr
Address of a **MIDIHDR** structure that identifies the buffer.

cbMidiInHdr
Size, in bytes, of the **MIDIHDR** structure.

When the buffer is filled, it is sent back to the application.

The buffer must be prepared by using the **midiInPrepareHeader** function before it is passed to the **midiInAddBuffer** function.

midiInClose

```
MMRESULT midiInClose(HMIDIIN hMidiIn);
```

Closes the specified MIDI input device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MIDIERR_STILLPLAYING	Buffers are still in the queue.
MMSYSERR_INVALHANDLE	The specified device handle is invalid.
MMSYSERR_NOMEM	The system is unable to allocate or lock memory.

hMidiIn
Handle of the MIDI input device. If the function is successful, the handle is no longer valid after the call to this function.

If there are input buffers that have been sent by using the **midiInAddBuffer** function and have not been returned to the application, the close operation will fail. To return all pending buffers through the callback function, use the **midiInReset** function.

midiInGetDevCaps

```
MMRESULT midiInGetDevCaps(UINT uDeviceID, LPMIDIINCAPS lpMidiInCaps,
    UINT cbMidiInCaps);
```

Determines the capabilities of a specified MIDI input device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_BADDEVICEID	The specified device identifier is out of range.
MMSYSERR_INVALPARAM	The specified pointer or structure is invalid.
MMSYSERR_NODRIVER	The driver is not installed.
MMSYSERR_NOMEM	The system is unable to allocate or lock memory.

uDeviceID
Identifier of the MIDI input device. The device identifier varies from zero to one less than the number of devices present. This parameter can also be a properly cast device handle.

lpMidiInCaps
Address of a **MIDIINCAPS** structure that is filled with information about the capabilities of the device.

cbMidiInCaps
Size, in bytes, of the **MIDIINCAPS** structure. Only *cbMidiInCaps* bytes (or less) of information is copied to the location pointed to by *lpMidiInCaps*. If *cbMidiInCaps* is zero, nothing is copied, and the function returns MMSYSERR_NOERROR.

To determine the number of MIDI input devices present on the system, use the **midiInGetNumDevs** function.

midiInGetErrorText

```
MMRESULT midiInGetErrorText(MMRESULT wError, LPSTR lpText,
    UINT cchText);
```

Retrieves a textual description for an error identified by the specified error code.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_BADERRNUM	The specified error number is out of range.
MMSYSERR_INVALPARAM	The specified pointer or structure is invalid.
MMSYSERR_NOMEM	The system is unable to allocate or lock memory.

wError
Error code.

lpText
Address of the buffer to be filled with the textual error description.

cchText
Length, in characters, of the buffer pointed to by *lpText*.

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

midiInGetID

```
MMRESULT midiInGetID(HMIDIIN hmi, LPUINT puDeviceID);
```

Gets the device identifier for the given MIDI input device.

This function is supported for backward compatibility. New applications can cast a handle of the device rather than retrieving the device identifier.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	The *hwi* parameter specifies an invalid handle.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

hmi
Handle of the MIDI input device.

puDeviceID
Address of a variable to be filled with the device identifier.

midiInGetNumDevs

```
UINT midiInGetNumDevs(VOID);
```

Retrieves the number of MIDI input devices in the system.

- Returns the number of MIDI input devices present in the system. A return value of zero means that there are no devices (not that there is no error).

midiInMessage

```
DWORD midiInMessage(HMIDIIN hMidiIn, UINT msg, DWORD dw1, DWORD dw2);
```

Sends a message to the MIDI device driver.

- Returns the value returned by the audio device driver.

hMidiIn
: Handle of the MIDI device.

msg
: Message to send.

dw1 and *dw2*
: Message parameters.

This function is used only for driver-specific messages that are not supported by the MIDI API.

midiInOpen

```
MMRESULT midiInOpen(LPHMIDIIN lphMidiIn, UINT uDeviceID,
    DWORD dwCallback, DWORD dwCallbackInstance, DWORD dwFlags);
```

Opens a specified MIDI input device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_ALLOCATED	The specified resource is already allocated.
MMSYSERR_BADDEVICEID	The specified device identifier is out of range.
MMSYSERR_INVALFLAG	The flags specified by *dwFlags* are invalid.
MMSYSERR_INVALPARAM	The specified pointer or structure is invalid.
MMSYSERR_NOMEM	The system is unable to allocate or lock memory.

lphMidiIn
Address of an **HMIDIIN** handle. This location is filled with a handle identifying the opened MIDI input device. The handle is used to identify the device in calls to other MIDI input functions.

uDeviceID
Identifier of the MIDI input device to be opened.

dwCallback
Address of a callback function, a thread identifier, or a handle of a window called with information about incoming MIDI messages.

dwCallbackInstance
User instance data passed to the callback function. This parameter is not used with window callback functions or threads.

dwFlags
Callback flag for opening the device and, optionally, a status flag that helps regulate rapid data transfers. It can be the following values:

CALLBACK_FUNCTION
The *dwCallback* parameter is a callback procedure address.

CALLBACK_NULL
There is no callback mechanism. This value is the default setting.

CALLBACK_THREAD
The *dwCallback* parameter is a thread identifier.

CALLBACK_WINDOW
The *dwCallback* parameter is a window handle.

MIDI_IO_STATUS
When this parameter also specifies CALLBACK_FUNCTION, MIM_MOREDATA messages are sent to the callback function as well as MIM_DATA messages. Or, if this parameter also specifies CALLBACK_WINDOW, MM_MIM_MOREDATA messages are sent to the window as well as MM_MIM_DATA messages. This flag does not affect event or thread callbacks.

Most applications that use a callback mechanism will specify CALLBACK_FUNCTION for this parameter.

To determine the number of MIDI input devices present in the system, use the **midiInGetNumDevs** function. The device identifier specified by *wDeviceID* varies from zero to one less than the number of

devices present.

If a window or thread is chosen to receive callback information, the following messages are sent to the window procedure or thread 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, and MM_MIM_MOREDATA.

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, and MIM_MOREDATA.

midiInPrepareHeader

```
MMRESULT midiInPrepareHeader(HMIDIIN hMidiIn, LPMIDIHDR lpMidiInHdr,
    UINT cbMidiInHdr);
```

Prepares a buffer for MIDI input.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	The specified device handle is invalid.
MMSYSERR_INVALPARAM	The specified address is invalid.
MMSYSERR_NOMEM	The system is unable to allocate or lock memory.

hMidiIn
Handle of the MIDI input device.

lpMidiInHdr
Address of a **MIDIHDR** structure that identifies the buffer to be prepared.

cbMidiInHdr
Size, in bytes, of the **MIDIHDR** structure.

Preparing a header that has already been prepared has no effect, and the function returns zero.

Before using this function, you must set the **lpData**, **dwBufferLength**, and **dwFlags** members of the **MIDIHDR** structure. The **dwFlags** member must be set to zero.

MidiInProc

```
void CALLBACK MidiInProc(HMIDIIN hMidiIn, UINT wMsg, DWORD dwInstance,
    DWORD dwParam1, DWORD dwParam2);
```

Callback function for handling incoming MIDI messages. **MidiInProc** is a placeholder for the application-supplied function name.

hMidiIn
Handle of the MIDI input device.

wMsg
MIDI input message.

dwInstance
Instance data supplied with the **midiInOpen** function.

dwParam1 and *dwParam2*
Message parameters.

Applications should not call any system-defined functions from inside a callback function, except for **PostMessage**, **timeGetSystemTime**, **timeGetTime**, **timeSetEvent**, **timeKillEvent**, **midiOutShortMsg**, **midiOutLongMsg**, and **OutputDebugStr**.

midiInReset

```
MMRESULT midiInReset(HMIDIIN hMidiIn);
```

Stops input on a given MIDI input device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

  MMSYSERR_INVALHAND LE — The specified device handle is invalid.

hMidiIn
Handle of the MIDI input device.

This function returns all pending input buffers to the callback function and sets the MHDR_DONE flag in the **dwFlags** member of the **MIDIHDR** structure.

midiInStart

```
MMRESULT midiInStart(HMIDIIN hMidiIn);
```

Starts MIDI input on the specified MIDI input device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

  MMSYSERR_INVALHAND The specified device handle is invalid.
  LE

hMidiIn
  Handle of the MIDI input device.

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

All messages except system-exclusive messages are sent directly to the client when they are received. System-exclusive messages are placed in the buffers supplied by the **midiInAddBuffer** function. If there are no buffers in the queue, the system-exclusive data is thrown away without notification to the client and input continues. Buffers are returned to the client when they are full, when a complete system-exclusive message has been received, or when the **midiInReset** function is used. The **dwBytesRecorded** member of the **MIDIHDR** structure will contain the actual length of data received.

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

midiInStop

```
MMRESULT midiInStop(HMIDIIN hMidiIn);
```

Stops MIDI input on the specified MIDI input device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

  MMSYSERR_INVALHAND The specified device handle is invalid.
  LE

hMidiIn
Handle of the MIDI input device.

If there are any system-exclusive messages or stream buffers in the queue, the current buffer is marked as done (the **dwBytesRecorded** member of the **MIDIHDR** structure will contain the actual length of data), but any empty buffers in the queue remain there and are not marked as done.

Calling this function when input is not started has no effect, and the function returns zero.

midiInUnprepareHeader

```
MMRESULT midiInUnprepareHeader(HMIDIIN hMidiIn, LPMIDIHDR lpMidiInHdr,
    UINT cbMidiInHdr);
```

Cleans up the preparation performed by the **midiInPrepareHeader** function.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MIDIERR_STILLPLAYING	The buffer pointed to by *lpMidiInHdr* is still in the queue.
MMSYSERR_INVALPARAM	The specified pointer or structure is invalid.
MMSYSERR_INVALHANDLE	The specified device handle is invalid.

hMidiIn
Handle of the MIDI input device.

lpMidiInHdr
Address of a **MIDIHDR** structure identifying the buffer to be cleaned up.

cbMidiInHdr
Size of the **MIDIHDR** structure.

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

midiOutCacheDrumPatches

```
MMRESULT midiOutCacheDrumPatches(HMIDIOUT hmo, UINT wPatch,
    WORD FAR* lpKeyArray, UINT wFlags);
```

Requests that an internal MIDI synthesizer device preload and cache a specified set of key-based percussion patches.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALFLAG	The flag specified by *wFlags* is invalid.
MMSYSERR_INVALHANDLE	The specified device handle is invalid.
MMSYSERR_INVALPARAM	The array pointed to by the *lpKeyArray* array is invalid.
MMSYSERR_NOMEM	The device does not have enough memory to cache all of the requested patches.
MMSYSERR_NOTSUPPORTED	The specified device does not support patch caching.

hmo
Handle of the opened MIDI output device. This device should be an internal MIDI synthesizer. This parameter can also be the handle of a MIDI stream, cast to **HMIDIOUT**.

wPatch
Drum patch number that should be used. This parameter should be set to zero to cache the default drum patch.

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

wFlags
Options for the cache operation. It can be one of the following flags:

MIDI_CACHE_ALL
Caches all of the specified patches. If they cannot all be cached, it caches none, clears the **KEYARRAY** array, and returns MMSYSERR_NOMEM.

MIDI_CACHE_BESTFIT
Caches all of the specified patches. If they cannot all be cached, it caches as many patches as possible, changes the **KEYARRAY** array to reflect which patches were cached, and returns MMSYSERR_NOMEM.

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

MIDI_UNCACHE
Uncaches the specified patches and clears the **KEYARRAY** array.

Some synthesizers are not capable of keeping all percussion patches loaded simultaneously. Caching patches ensures that the specified patches are available.

Each element of the **KEYARRAY** array represents one of the 128 key-based percussion patches and has bits set for each of the 16 MIDI channels that use the particular patch. The least-significant bit represents physical channel 0, and 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. To see if a device supports patch caching, use the MIDICAPS_CACHE flag to test the **dwSupport** member of the **MIDIOUTCAPS** structure filled by the **midiOutGetDevCaps** function.

midiOutCachePatches

```
MMRESULT midiOutCachePatches(HMIDIOUT hmo, UINT wBank,
    WORD FAR* lpPatchArray, UINT wFlags);
```

Requests that an internal MIDI synthesizer device preload and cache a specified set of patches.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALFLAG	The flag specified by *wFlags* is invalid.
MMSYSERR_INVALHANDLE	The specified device handle is invalid.
MMSYSERR_INVALPARAM	The array pointed to by *lpPatchArray* is invalid.
MMSYSERR_NOMEM	The device does not have enough memory to cache all of the requested patches.
MMSYSERR_NOTSUPPORTED	The specified device does not support patch caching.

hmo
Handle of the opened MIDI output device. This device must be an internal MIDI synthesizer. This parameter can also be the handle of a MIDI stream, cast to **HMIDIOUT**.

wBank
Bank of patches that should be used. This parameter should be set to zero to cache the default patch bank.

lpPatchArray
Address of a **PATCHARRAY** array indicating the patches to be cached or uncached.

wFlags
Options for the cache operation. It can be one of the following flags:

MIDI_CACHE_ALL
Caches all of the specified patches. If they cannot all be cached, it caches none, clears the **PATCHARRAY** array, and returns MMSYSERR_NOMEM.

MIDI_CACHE_BESTFIT
Caches all of the specified patches. If they cannot all be cached, it caches as many patches as possible, changes the **PATCHARRAY** array to reflect which patches were cached, and returns MMSYSERR_NOMEM.

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

MIDI_UNCACHE
Uncaches the specified patches and clears the **PATCHARRAY** array.

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 that the specified patches are immediately available.

Each element of the **PATCHARRAY** array represents one of the 128 patches and has bits set for each of the 16 MIDI channels that use the particular patch. The least-significant bit represents physical channel 0, and 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 applies only to internal MIDI synthesizer devices. Not all internal synthesizers support patch caching. To see if a device supports patch caching, use the MIDICAPS_CACHE flag to test the

dwSupport member of the **MIDIOUTCAPS** structure filled by the **midiOutGetDevCaps** function.

midiOutClose

```
MMRESULT midiOutClose(HMIDIOUT hmo);
```

Closes the specified MIDI output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MIDIERR_STILLPLAYING	Buffers are still in the queue.
MMSYSERR_INVALHANDLE	The specified device handle is invalid.
MMSYSERR_NOMEM	The system is unable to load mapper string description.

hmo
Handle of the MIDI output device. If the function is successful, the handle is no longer valid after the call to this function.

If there are output buffers that have been sent by using the **midiOutLongMsg** function and have not been returned to the application, the close operation will fail. To mark all pending buffers as being done, use the **midiOutReset** function.

midiOutGetDevCaps

```
MMRESULT midiOutGetDevCaps(UINT uDeviceID, LPMIDIOUTCAPS lpMidiOutCaps,
    UINT cbMidiOutCaps);
```

Queries a specified MIDI output device to determine its capabilities.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_BADDEVICEID	The specified device identifier is out of range.
MMSYSERR_INVALPARAM	The specified pointer or structure is invalid.
MMSYSERR_NODRIVER	The driver is not installed.
MMSYSERR_NOMEM	The system is unable to load mapper string description.

uDeviceID
Identifier of the MIDI output device. The device identifier specified by this parameter varies from zero to one less than the number of devices present. The MIDI_MAPPER constant is also a valid device identifier.

This parameter can also be a properly cast device handle.

lpMidiOutCaps
Address of a **MIDIOUTCAPS** structure. This structure is filled with information about the capabilities of the device.

cbMidiOutCaps
Size, in bytes, of the **MIDIOUTCAPS** structure. Only *cbMidiOutCaps* bytes (or less) of information is copied to the location pointed to by *lpMidiOutCaps*. If *cbMidiOutCaps* is zero, nothing is copied, and the function returns MMSYSERR_NOERROR.

To determine the number of MIDI output devices present in the system, use the **midiOutGetNumDevs** function.

midiOutGetErrorText

```
UINT midiOutGetErrorText(MMRESULT mmrError, LPSTR lpText, UINT cchText);
```

Retrieves a textual description for an error identified by the specified error code.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_BADERRNUM	The specified error number is out of range.
MMSYSERR_INVALPARAM	The specified pointer or structure is invalid.

mmrError
Error code.

lpText
Address of a buffer to be filled with the textual error description.

cchText
Length, in characters, of the buffer pointed to by *lpText*.

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

midiOutGetID

```
MMRESULT midiOutGetID(HMIDIOUT hmo, LPUINT puDeviceID);
```

Retrieves the device identifier for the given MIDI output device.

This function is supported for backward compatibility. New applications can cast a handle of the device rather than retrieving the device identifier.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	The *hwo* parameter specifies an invalid handle.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

hmo
Handle of the MIDI output device.

puDeviceID
Address of a variable to be filled with the device identifier.

midiOutGetNumDevs

```
UINT midiOutGetNumDevs(VOID);
```

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

- Returns the number of MIDI output devices. A return value of zero means that there are no devices (not that there is no error).

midiOutGetVolume

```
MMRESULT midiOutGetVolume(HMIDIOUT hmo, LPDWORD lpdwVolume);
```

Retrieves the current volume setting of a MIDI output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	The specified device handle is invalid.
MMSYSERR_INVALPARAM	The specified pointer or structure is invalid.
MMSYSERR_NOMEM	The system is unable to allocate or lock memory.
MMSYSERR_NOTSUPPORTED	The function is not supported.

hmo
Handle of an open MIDI output device. This parameter can also contain the handle of a MIDI stream, as long as it is cast to **HMIDIOUT**.

lpdwVolume
Address of the location to contain 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 mono volume level.

Any value set by using the **midiOutSetVolume** function is returned, regardless of whether the device supports that value.

Not all devices support volume control. You can determine whether a device supports volume control by querying the device by using the **midiOutGetDevCaps** function and specifying the MIDICAPS_VOLUME flag.

You can also determine whether the device supports volume control on both the left and right channels by querying the device by using the **midiOutGetDevCaps** function and specifying the MIDICAPS_LRVOLUME flag.

midiOutLongMsg

```
MMRESULT midiOutLongMsg(HMIDIOUT hmo, LPMIDIHDR lpMidiOutHdr,
    UINT cbMidiOutHdr);
```

Sends a system-exclusive MIDI message to the specified MIDI output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MIDIERR_NOTREADY	The hardware is busy with other data.
MIDIERR_UNPREPARED	The buffer pointed to by *lpMidiOutHdr* has not been prepared.
MMSYSERR_INVALHANDLE	The specified device handle is invalid.
MMSYSERR_INVALPARAM	The specified pointer or structure is invalid.

hmo
Handle of the MIDI output device. This parameter can also be the handle of a MIDI stream cast to **HMIDIOUT**.

lpMidiOutHdr
Address of a **MIDIHDR** structure that identifies the MIDI buffer.

cbMidiOutHdr
Size, in bytes, of the **MIDIHDR** structure.

Before the buffer is passed to **midiOutLongMsg**, it must be prepared by using the **midiOutPrepareHeader** function. The MIDI output device driver determines whether the data is sent synchronously or asynchronously.

midiOutMessage

```
DWORD midiOutMessage(HMIDIOUT hmo, UINT msg, DWORD dw1, DWORD dw2);
```

Sends a message to the MIDI device drivers. This function is used only for driver-specific messages that are not supported by the MIDI API.

- Returns the value returned by the audio device driver.

hmo
Handle of the MIDI device. This parameter can also be the handle of a MIDI stream cast to **HMIDIOUT**.

msg
Message to send.

dw1 and *dw2*
Message parameters.

midiOutOpen

```
UINT midiOutOpen(LPHMIDIOUT lphmo, UINT uDeviceID,
    DWORD dwCallback, DWORD dwCallbackInstance, DWORD dwFlags);
```

Opens a MIDI output device for playback.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MIDIERR_NODEVICE	No MIDI port was found. This error occurs only when the mapper is opened.
MMSYSERR_ALLOCATED	The specified resource is already allocated.
MMSYSERR_BADDEVICEID	The specified device identifier is out of range.
MMSYSERR_INVALPARAM	The specified pointer or structure is invalid.
MMSYSERR_NOMEM	The system is unable to allocate or lock memory.

lphmo
Address of an **HMIDIOUT** handle. This location is filled with a handle identifying the opened MIDI output device. The handle is used to identify the device in calls to other MIDI output functions.

uDeviceID
Identifier of the MIDI output device that is to be opened.

dwCallback
Address of a callback function, an event handle, a thread identifier, or a handle of a window or thread called during MIDI playback to process messages related to the progress of the playback. If no callback is desired, specify NULL for this parameter.

dwCallbackInstance
User instance data passed to the callback. This parameter is not used with window callbacks or threads.

dwFlags
Callback flag for opening the device. It can be the following values:

CALLBACK_EVENT
The *dwCallback* parameter is an event handle. This callback mechanism is for output only.

CALLBACK_FUNCTION
The *dwCallback* parameter is a callback function address.

CALLBACK_NULL
There is no callback mechanism. This value is the default setting.

CALLBACK_THREAD
The *dwCallback* parameter is a thread identifier.

CALLBACK_WINDOW
The *dwCallback* parameter is a window handle.

To determine the number of MIDI output devices present in the system, use the **midiOutGetNumDevs** function. The device identifier specified by *wDeviceID* varies from zero to one less than the number of devices present. MIDI_MAPPER can also be used as the device identifier.

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

MM_MOM_CLOSE, and 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, and MOM_DONE.

midiOutPrepareHeader

```
MMRESULT midiOutPrepareHeader(HMIDIOUT hmo, LPMIDIHDR lpMidiOutHdr,
    UINT cbMidiOutHdr);
```

Prepares a MIDI system-exclusive or stream buffer for output.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	The specified device handle is invalid.
MMSYSERR_INVALPARAM	The specified address is invalid or the given stream buffer is greater than 64K.
MMSYSERR_NOMEM	The system is unable to allocate or lock memory.

hmo
Handle of the MIDI output device. This parameter can also be the handle of a MIDI stream cast to **HMIDIOUT**.

lpMidiOutHdr
Address of a **MIDIHDR** structure that identifies the buffer to be prepared.

cbMidiOutHdr
Size, in bytes, of the **MIDIHDR** structure.

A stream buffer cannot be larger than 64K.

Preparing a header that has already been prepared has no effect, and the function returns MMSYSERR_NOERROR.

Before using this function, you must set the **lpData**, **dwBufferLength**, and **dwFlags** members of the **MIDIHDR** structure. The **dwFlags** member must be set to zero.

MidiOutProc

```
void CALLBACK MidiOutProc(HMIDIOUT hmo, UINT wMsg,
    DWORD dwInstance, DWORD dwParam1, DWORD dwParam2);
```

Callback function for handling outgoing MIDI messages. **MidiOutProc** is a placeholder for the application-supplied function name.

hmo
Handle of the MIDI device associated with the callback function.

wMsg
MIDI output message.

dwInstance
Instance data supplied by using the **midiOutOpen** function.

dwParam1 and *dwParam2*
Message parameters.

Applications should not call any system-defined functions from inside a callback function, except for **PostMessage**, **timeGetSystemTime**, **timeGetTime**, **timeSetEvent**, **timeKillEvent**, **midiOutShortMsg**, **midiOutLongMsg**, and **OutputDebugStr**.

midiOutReset

```
MMRESULT midiOutReset(HMIDIOUT hmo);
```

Turns off all notes on all MIDI channels for the specified MIDI output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

  MMSYSERR_INVALHAND The specified device handle is invalid.
  LE

hmo
Handle of the MIDI output device. This parameter can also be the handle of a MIDI stream cast to **HMIDIOUT**.

Any pending system-exclusive or stream output buffers are returned to the callback function and the MHDR_DONE flag is set in the **dwFlags** member of the **MIDIHDR** structure.

Terminating a system-exclusive message without sending an EOX (end-of-exclusive) byte might cause problems for the receiving device. The **midiOutReset** function does not send an EOX byte when it terminates a system-exclusive message – applications are responsible for doing this.

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

midiOutSetVolume

```
MMRESULT midiOutSetVolume(HMIDIOUT hmo, DWORD dwVolume);
```

Sets the volume of a MIDI output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	The specified device handle is invalid.
MMSYSERR_NOMEM	The system is unable to allocate or lock memory.
MMSYSERR_NOTSUPPORTED	The function is not supported.

hmo
Handle of an open MIDI output device. This parameter can also contain the handle of a MIDI stream, as long as it is cast to **HMIDIOUT**.

dwVolume
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 mono volume level, and the high-order word is ignored.

Changing the volume on a handle affects only one instance of the device. It does not change the default volume for the device (and affect all instances of the device).

Not all devices support volume changes. You can determine whether a device supports it by querying the device using the **midiOutGetDevCaps** function and the MIDICAPS_VOLUME flag.

You can also determine whether the device supports volume control on both the left and right channels by querying the device using the **midiOutGetDevCaps** function and the MIDICAPS_LRVOLUME flag.

Devices that do not support a full 16 bits of volume-level control use the high-order bits of the requested volume setting. For example, a device that supports 4 bits of volume control produces the same volume setting for the following volume-level values: 0x4000, 0x43be, and 0x4fff. The **midiOutGetVolume** function returns the full 16-bit value, as set by **midiOutSetVolume**, irrespective of the device's capabilities.

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

midiOutShortMsg

```
MMRESULT midiOutShortMsg(HMIDIOUT hmo, DWORD dwMsg);
```

Sends a short MIDI message to the specified MIDI output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MIDIERR_BADOPENMODE	The application sent a message without a status byte to a stream handle.
MIDIERR_NOTREADY	The hardware is busy with other data.
MMSYSERR_INVALHANDLE	The specified device handle is invalid.

hmo
Handle of the MIDI output device. This parameter can also be the handle of a MIDI stream cast to **HMIDIOUT**.

dwMsg
MIDI message. The message is packed into a doubleword value with the first byte of the message in the low-order byte. The message is packed into this parameter as follows:

High word	High-order byte	Not used.
	Low-order byte	Contains a second byte of MIDI data (when needed).
Low word	High-order byte	Contains the first byte of MIDI data (when needed).
	Low-order byte	Contains the MIDI status.

The two MIDI data bytes are optional, depending on the MIDI status byte. When a series of messages have the same status byte, the status byte can be omitted from messages after the first one in the series, creating a running status. Pack a message for running status as follows:

High word	High-order byte	Not used.
	Low-order byte	Not used.
Low word	High-order byte	Contains a second byte of MIDI data (when needed).
	Low-order byte	Contains the first byte of MIDI data.

This function is used to send any MIDI message except for system-exclusive or stream messages.

This function might not return until the message has been sent to the output device. You can send short messages while streams are playing on the same device (although you cannot use a running status in this case).

midiOutUnprepareHeader

```
MMRESULT midiOutUnprepareHeader(HMIDIOUT hmo,
    LPMIDIHDR lpMidiOutHdr, UINT cbMidiOutHdr);
```

Cleans up the preparation performed by the **midiOutPrepareHeader** function.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MIDIERR_STILLPLAYING	The buffer pointed to by *lpMidiOutHdr* is still in the queue.
MMSYSERR_INVALHANDLE	The specified device handle is invalid.
MMSYSERR_INVALPARAM	The specified pointer or structure is invalid.

hmo
Handle of the MIDI output device. This parameter can also be the handle of a MIDI stream cast to **HMIDIOUT**.

lpMidiOutHdr
Address of a **MIDIHDR** structure identifying the buffer to be cleaned up.

cbMidiOutHdr
Size, in bytes, of the **MIDIHDR** structure.

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

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

midiStreamClose

[New - Windows 95]

```
MMRESULT midiStreamClose(HMIDISTRM hStream);
```

Closes an open MIDI stream.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. You can use either the **midiInGetErrorText** or **midiOutGetErrorText** function to interpret error values for the stream functions. A possible error value is:

  MMSYSERR_INVALHAND The specified device handle is invalid.
  LE

hStream
Handle of a MIDI stream, as retrieved by using the **midiStreamOpen** function.

midiStreamOpen

[New - Windows 95]

```
MMRESULT midiStreamOpen(LPHMIDISTRM lphStream, LPUINT puDeviceID,
    DWORD cMidi, DWORD dwCallback, DWORD dwInstance, DWORD fdwOpen);
```

Opens a MIDI stream for output. By default, the device is opened in paused mode. The stream handle retrieved by this function must be used in all subsequent references to the stream.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. You can use the **midiOutGetErrorText** function to interpret error values for the stream functions. Possible error values include the following:

MMSYSERR_BADDEVICEID	The specified device identifier is out of range.
MMSYSERR_INVALPARAM	The given handle or flags parameter is invalid.
MMSYSERR_NOMEM	The system is unable to allocate or lock memory.

lphStream
Address of a variable to contain the stream handle when the function returns.

puDeviceID
Address of a device identifier. The device is opened on behalf of the stream and closed again when the stream is closed.

cMidi
Reserved; must be 1.

dwCallback
Address of a callback function, an event handle, a thread identifier, or a handle of a window or thread called during MIDI playback to process messages related to the progress of the playback. If no callback mechanism is desired, specify NULL for this parameter.

dwInstance
Application-specific instance data that is returned to the application with every callback function.

fdwOpen
Callback flag for opening the device. One of the following callback flags must be specified:

CALLBACK_EVENT
The *dwCallback* parameter is an event handle. This callback mechanism is for output only.

CALLBACK_FUNCTION
The *dwCallback* parameter is a callback procedure address.

CALLBACK_NULL
There is no callback mechanism. This is the default setting.

CALLBACK_THREAD
The *dwCallback* parameter is a thread identifier.

CALLBACK_WINDOW
The *dwCallback* parameter is a window handle.

midiStreamOut

[New - Windows 95]

```
MMRESULT midiStreamOut(HMIDISTRM hMidiStream, LPMIDIHDR lpMidiHdr,
    UINT cbMidiHdr);
```

Plays or queues a stream (buffer) of MIDI data to a MIDI output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. You can use the **midiOutGetErrorText** function to interpret error values for the stream functions. Possible error values include the following:

MMSYSERR_NOMEM	The system is unable to allocate or lock memory.
MIDIERR_STILLPLAYING	The output buffer pointed to by *lpMidiHdr* is still playing or is queued from a previous call to **midiStreamOut**.
MIDIERR_UNPREPARED	The header pointed to by *lpMidiHdr* has not been prepared.
MMSYSERR_INVALHANDLE	The specified device handle is invalid.
MMSYSERR_INVALPARAM	The pointer specified by *lpMidiHdr* is invalid.

hMidiStream
Handle of a MIDI stream. This handle must have been returned by a call to the **midiStreamOpen** function. This handle identifies the output device.

lpMidiHdr
Address of a **MIDIHDR** structure that identifies the MIDI buffer.

cbMidiHdr
Size, in bytes, of the **MIDIHDR** structure.

Because the **midiStreamOpen** function opens the output device in paused mode, you must call the **midiStreamRestart** function before you can use **midiStreamOut** to start the playback.

For the current implementation of this function, the buffer must be smaller than 64K.

The buffer pointed to by the **MIDIHDR** structure contains one or more MIDI events, each of which is defined by a **MIDIEVENT** structure.

midiStreamPause

[New - Windows 95]

```
MMRESULT midiStreamPause(HMIDISTRM hms);
```

Pauses playback of a specified MIDI stream.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. You can use the **midiOutGetErrorText** function to interpret error values for the stream functions. Possible error values include the following:

MMSYSERR_INVALHANDLE	The specified device handle is invalid.

hms
Handle of a MIDI stream. This handle must have been returned by a call to the **midiStreamOpen** function. This handle identifies the output device.

The current playback position is saved when playback is paused. To resume playback from the current position, use the **midiStreamRestart** function.

Calling this function when the output is already paused has no effect, and the function returns MMSYSERR_NOERROR.

midiStreamPosition

[New - Windows 95]

```
MMRESULT midiStreamPosition(HMIDISTRM hms, LPMMTIME pmmt, UINT cbmmt);
```

Retrieves the current position in a MIDI stream.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. You can use the **midiOutGetErrorText** function to interpret error values for the stream functions. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_INVALPARAM	Specified pointer or structure is invalid.

hms
Handle of a MIDI stream. This handle must have been returned by a call to the **midiStreamOpen** function. This handle identifies the output device.

pmmt
Address of an **MMTIME** structure.

cbmmt
Size, in bytes, of the **MMTIME** structure.

Before calling **midiStreamPosition**, set the **wType** member of the **MMTIME** structure to indicate the time format you desire. After calling **midiStreamPosition**, check the **wType** member 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.

midiStreamProperty

[New - Windows 95]

```
MMRESULT midiStreamProperty(HMIDISTRM hm, LPBYTE lppropdata,
    DWORD dwProperty);
```

Sets or retrieves properties of a MIDI data stream associated with a MIDI output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. You can use the **midiOutGetErrorText** function to interpret error values for the stream functions. Possible error values include the following:

MMSYSERR_INVALHANDLE	The specified handle is not a stream handle.
MMSYSERR_INVALPARAM	The given handle or flags parameter is invalid.

hm
Handle of the MIDI device that the property is associated with.

lppropdata
Address of the property data.

dwProperty
Flags that specify the action to perform and identify the appropriate property of the MIDI data stream. The **midiStreamProperty** function requires setting two flags in each use. One flag (either MIDIPROP_GET or MIDIPROP_SET) specifies an action, and the other identifies a specific property to examine or edit:

MIDIPROP_GET
Retrieves the current setting of the given property.

MIDIPROP_SET
Sets the given property.

MIDIPROP_TEMPO
Retrieves the tempo property. The *lppropdata* parameter points to a **MIDIPROPTEMPO** structure. The current tempo value can be retrieved at any time. Output devices set the tempo by inserting MEVT_TEMPO events into the MIDI data.

MIDIPROP_TIMEDIV
Specifies the time division property. You can get or set this property. The *lppropdata* parameter points to a **MIDIPROPTIMEDIV** structure. This property can be set only when the device is stopped.

These properties are the default properties defined by the system. Driver writers can implement and document their own properties.

midiStreamRestart

[New - Windows 95]

```
MMRESULT midiStreamRestart(HMIDISTRM hms);
```

Restarts a paused MIDI stream.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. You can use the **midiOutGetErrorText** function to interpret error values for the stream functions. Possible error values include the following:

MMSYSERR_INVALHANDLE	The specified device handle is invalid.

hms
Handle of a MIDI stream. This handle must have been returned by a call to the **midiStreamOpen** function. This handle identifies the output device.

Calling this function when the output is not paused has no effect, and the function returns MMSYSERR_NOERROR.

midiStreamStop

[New - Windows 95]

```
MMRESULT midiStreamStop(HMIDISTRM hms);
```

Turns off all notes on all MIDI channels for the specified MIDI output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. You can use the **midiOutGetErrorText** function to interpret error values for the stream functions. Possible error values include the following:

  MMSYSERR_INVALHANDLE The specified device handle is invalid.

hms
Handle of a MIDI stream. This handle must have been returned by a call to the **midiStreamOpen** function. This handle identifies the output device.

When you call this function, any pending system-exclusive or stream output buffers are returned to the callback mechanism and the MHDR_DONE bit is set in the **dwFlags** member of the **MIDIHDR** structure.

While the **midiOutReset** function turns off all notes, **midiStreamStop** turns off only those notes that have been turned on by a MIDI note-on message.

MEVT_EVENTPARM

```
DWORD MEVT_EVENTPARM(DWORD dwEvent)
```

Retrieves the event parameters or length from the value specified in the **dwEvent** member of a **MIDIEVENT** structure.

dwEvent
Code for the MIDI event and the event parameters or length, as specified in the **dwEvent** member of the **MIDIEVENT** structure.

The **MEVT_EVENTPARM** macro is defined as follows:

```
#define MEVT_EVENTPARM(x) ((DWORD) ((x)&0x00FFFFFFL))
```

MEVT_EVENTTYPE

```
BYTE MEVT_EVENTTYPE(DWORD dwEvent)
```

Retrieves the event type from the value specified in the **dwEvent** member of a **MIDIEVENT** structure.

dwEvent
Code for the MIDI event and the event parameters or length, as specified in the **dwEvent** member of the **MIDIEVENT** structure.

The **MEVT_EVENTTYPE** macro is defined as follows:

```
#define MEVT_EVENTTYPE(x) ((BYTE) (((x)>>24)&0xFF))
```

KEYARRAY

```
typedef WORD KEYARRAY[MIDIPATCHSIZE];
```

Specifies a type used to define an array of keys. Each element in the array corresponds to a key-based percussion patch with each of the 16 bits representing one of the 16 MIDI channels. Bits are set for each of the channels that use that particular patch. For example, if the percussion patch for key number 60 is used by physical MIDI channels 9 and 15, element 60 of the array should be set to 0x8200.

MIDIEVENT

```
typedef struct {
    DWORD dwDeltaTime;  // see below
    DWORD dwStreamID;   // reserved; must be zero
    DWORD dwEvent;      // see below
    DWORD dwParms[];    // see below
} MIDIEVENT;
```

Describes a MIDI event in a stream buffer.

dwDeltaTime

Time, in MIDI ticks, between the previous event and the current event. The length of a tick is defined by the time format and possibly the tempo associated with the stream. (The definition is identical to the specification for a tick in a standard MIDI file.)

dwEvent

Event code and event parameters or length. To parse this information, use the **MEVT_EVENTTYPE** and **MEVT_EVENTPARM** macros.

The high byte of this member contains one or more of the following flags and an event code:

MEVT_F_CALLBACK
: The system generates a callback when the event is about to be executed.

MEVT_F_LONG
: The event is a long event. The low 24 bits of **dwEvent** contain the length of the event parameters.

MEVT_F_SHORT
: The event is a short event. The event parameters are contained in the low 24 bits of **dwEvent**.

Either MEVT_F_LONG or MEVT_F_SHORT must be specified, but MEVT_F_CALLBACK is optional.

The remainder of the high byte contains one of the following event codes:

MEVT_COMMENT
: Long event. The event data will be ignored. This event is intended to store commentary information about the stream that might be useful to authoring programs or sequencers if the stream data were to be stored in a file in stream format. In a buffer of this data, the zero byte identifies the comment class and subsequent bytes contain the comment data.

MEVT_LONGMSG
: Long event. The event data is transmitted verbatim. The event data is assumed to be system-exclusive data; that is, running status will be cleared when the event is executed and running status from any previous events will not be applied to any channel events in the event data. Using this event to send a group of channel messages at the same time is not recommended; a set of MEVT_SHORTMSG events with zero delta times should be used instead.

MEVT_NOP
: Short event. This event is a placeholder; it does nothing. The low 24 bits are ignored. This event will still generate a callback if MEVT_F_CALLBACK is set in **dwEvent**.

MEVT_SHORTMSG
: Short event. The data in the low 24 bits of **dwEvent** is a MIDI short message. (For a description of how a short message is packed into a doubleword value, see the **midiOutShortMsg** function.)

MEVT_TEMPO
: Short event. The data in the low 24 bits of **dwEvent** contain the new tempo for following events. The tempo is specified in the same format as it is for the tempo change meta-event in a MIDI file – that is, in microseconds per quarter note. (This event will have no affect if the time format specified for the stream is SMPTE time.)

MEVT_VERSION
: Long event. The event data must contain a **MIDISTRMBUFFVER** structure.

dwParms

Parameters for the event, if **dwEvent** specifies MEVT_F_LONG and the length of the buffer. This parameter data must be padded with zeros so that an integral number of doubleword values are stored. For example, if the event data is five bytes long, three pad bytes must follow the data for a total of eight bytes. In this case, the low 24 bits of **dwEvent** would contain the value 5.

MIDIHDR

```
typedef struct {
    LPSTR  lpData;                  // address of MIDI data
    DWORD  dwBufferLength;          // size of the buffer
    DWORD  dwBytesRecorded;         // see below
    DWORD  dwUser;                  // custom user data
    DWORD  dwFlags;                 // see below
    struct midihdr_tag far * lpNext;  // reserved; do not use
    DWORD  reserved;                // reserved; do not use
    DWORD  dwOffset;                // see below
    DWORD  dwReserved[4];           // reserved; do not use
} MIDIHDR;
```

Defines the header used to identify a MIDI system-exclusive or stream buffer.

dwBytesRecorded
Actual amount of data in the buffer. This value should be less than or equal to the value given in the **dwBufferLength** member.

dwFlags
Flags giving information about the buffer.

MHDR_DONE
Set by the device driver to indicate that it is finished with the buffer and is returning it to the application.

MHDR_INQUEUE
Set by Windows to indicate that the buffer is queued for playback.

MHDR_ISSTRM
Set to indicate that the buffer is a stream buffer.

MHDR_PREPARED
Set by Windows to indicate that the buffer has been prepared by using the **midiInPrepareHeader** or **midiOutPrepareHeader** function.

dwOffset
Offset into the buffer when a callback is performed. (This callback is generated because the MEVT_F_CALLBACK flag is set in the **dwEvent** member of the **MIDIEVENT** structure.) This offset enables an application to determine which event caused the callback.

MIDIINCAPS

```
typedef struct {
    WORD      wMid;                 // see below
    WORD      wPid;                 // see below
    MMVERSION vDriverVersion;       // see below
    CHAR      szPname[MAXPNAMELEN]; // see below
    DWORD     dwSupport;            // reserved; must be zero
} MIDIINCAPS;
```

Describes the capabilities of a MIDI input device.

wMid
Manufacturer identifier of the device driver for the MIDI input device. For a list of identifiers, see Manufacturer and Product Identifiers.

wPid
Product identifier of the MIDI input device. For a list of identifiers, see Manufacturer and Product Identifiers.

vDriverVersion
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
Product name in a null-terminated string.

MIDIOUTCAPS

```
typedef struct {
    WORD      wMid;
    WORD      wPid;
    MMVERSION vDriverVersion;
    CHAR      szPname[MAXPNAMELEN];
    WORD      wTechnology;
    WORD      wVoices;
    WORD      wNotes;
    WORD      wChannelMask;
    DWORD     dwSupport;
} MIDIOUTCAPS;
```

Describes the capabilities of a MIDI output device.

wMid
Manufacturer identifier of the device driver for the MIDI output device. For a list of identifiers, see Chapter 0, "Manufacturer and Product Identifiers."

wPid
Product identifier of the MIDI output device. For a list of identifiers, see Chapter 0, "Manufacturer and Product Identifiers."

vDriverVersion
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
Product name in a null-terminated string.

wTechnology
Flags describing the type of the MIDI output device. It can be one of the following:

MOD_FMSYNTH
The device is an FM synthesizer.

MOD_MAPPER
The device is the Microsoft MIDI mapper.

MOD_MIDIPORT
The device is a MIDI hardware port.

MOD_SQSYNTH
The device is a square wave synthesizer.

MOD_SYNTH
The device is a synthesizer.

wVoices
Number of voices supported by an internal synthesizer device. If the device is a port, this member is not meaningful and is set to 0.

wNotes
Maximum number of simultaneous notes that can be played by an internal synthesizer device. If the device is a port, this member is not meaningful and is set to 0.

wChannelMask
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 that transmit on all channels set this member to 0xFFFF.

dwSupport
Optional functionality supported by the device. It can be one or more of the following:

MIDICAPS_CACHE
Supports patch caching.

MIDICAPS_LRVOLUME
Supports separate left and right volume control.

MIDICAPS_STREAM
Provides direct support for the **midiStreamOut** function.

MIDICAPS_VOLUME
Supports volume control.

If a device supports volume changes, the MIDICAPS_VOLUME flag will be set for the **dwSupport** member. 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 member.

MIDIPROPTEMPO

```
typedef struct {
    DWORD cbStruct;
    DWORD dwTempo;
} MIDIPROPTEMPO;
```

Contains the tempo property for a stream.

cbStruct
Length, in bytes, of this structure. This member must be filled in for both the MIDIPROP_SET and MIDIPROP_GET operations of the **midiStreamProperty** function.

dwTempo
Tempo of the stream, in microseconds per quarter note. The tempo is honored only if the time division for the stream is specified in quarter note format. This member is set in a MIDIPROP_SET operation and is filled on return from a MIDIPROP_GET operation.

The tempo property is read or written by the **midiStreamProperty** function.

MIDIPROPTIMEDIV

```
typedef struct {
    DWORD cbStruct;
    DWORD dwTimeDiv;
} MIDIPROPTIMEDIV;
```

Contains the time division property for a stream.

cbStruct
Length, in bytes, of this structure. This member must be filled in for both the MIDIPROP_SET and MIDIPROP_GET operations of the **midiStreamProperty** function.

dwTimeDiv
Time division for this stream, in the format specified in the *Standard MIDI Files 1.0* specification. The low 16 bits of this doubleword value contain the time division. This member is set in a MIDIPROP_SET operation and is filled on return from a MIDIPROP_GET operation.

The time division property is read or written by the **midiStreamProperty** function.

MIDISTRMBUFFVER

```
typedef struct {
    DWORD dwVersion;
    DWORD dwMid;
    DWORD dwOEMVersion;
} MIDISTRMBUFFVER;
```

Contains version information for a long MIDI event of the MEVT_VERSION type.

dwVersion
Version of the stream. The high 16 bits contain the major version, and the low 16 bits contain the minor version. The version number for the first implementation of MIDI streams should be 1.0.

dwMid
Manufacturer identifier. For a list of manufacturer identifiers, see Chapter 0, "Manufacturer and Product Identifiers."

dwOEMVersion
OEM version of the stream. Original equipment manufacturers can use this field to version-stamp any custom events they have specified. If a custom event is specified, it must be the first event sent after the stream is opened.

PATCHARRAY

```
typedef WORD PATCHARRAY[MIDIPATCHSIZE];
```

Specifies a type used to define an array of MIDI patches. Each element in the array corresponds to a patch with each of the 16 bits representing one of the 16 MIDI channels. Bits are set for each of the channels that use that particular patch. For example, if patch number 0 is used by physical MIDI channels 0 and 8, element 0 of the array should be set to 0x0101.

MIM_CLOSE

```
MIM_CLOSE
dwParam1 = reserved
dwParam2 = reserved
```

Sent to a MIDI input callback function when a MIDI input device is closed.

- No return value.

dwParam1 and *dwParam2*
Reserved; do not use.

The device handle is no longer valid after this message has been sent.

MIM_DATA

```
MIM_DATA
dwParam1 = dwMidiMessage
dwParam2 = dwTimestamp
```

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

- No return value.

dwMidiMessage

MIDI message that was received. The message is packed into a doubleword value as follows:

High word	High-order byte	Not used.
	Low-order byte	Contains a second byte of MIDI data (when needed).
Low word	High-order byte	Contains the first byte of MIDI data (when needed).
	Low-order byte	Contains the MIDI status.

The two MIDI data bytes are optional, depending on the MIDI status byte.

dwTimestamp

Time that the message was received by the input device driver. The time stamp is specified in milliseconds, beginning at zero when the **midiInStart** function was called.

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.

MIM_ERROR

```
MIM_ERROR
dwParam1 = dwMidiMessage
dwParam2 = dwTimestamp
```

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

- No return value.

dwMidiMessage
Invalid MIDI message that was received. The message is packed into a doubleword value with the first byte of the message in the low-order byte.

dwTimestamp
Time that the message was received by the input device driver. The time stamp is specified in milliseconds, beginning at zero when the **midiInStart** function was called.

MIM_LONGDATA

```
MIM_LONGDATA
dwParam1 = (DWORD) lpMidiHdr
dwParam2 = dwTimestamp
```

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

- No return value.

lpMidiHdr
Address of a **MIDIHDR** structure identifying the input buffer.

dwTimestamp
Time that the data was received by the input device driver. The time stamp is specified in milliseconds, beginning at zero when the **midiInStart** function was called.

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

MIM_LONGERROR

```
MIM_LONGERROR
dwParam1 = (DWORD) lpMidiHdr
dwParam2 = dwTimestamp
```

Sent to a MIDI input callback function when an invalid or incomplete MIDI system-exclusive message is received.

- No return value.

lpMidiHdr
Address of a **MIDIHDR** structure identifying the buffer containing the invalid message.

dwTimestamp
Time that the data was received by the input device driver. The time stamp is specified in milliseconds, beginning at zero when the **midiInStart** function was called.

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

MIM_MOREDATA

```
MIM_MOREDATA
dwParam1 = dwMidiMessage
dwParam2 = dwTimestamp
```

Sent to a MIDI input callback function when a MIDI message is received by a MIDI input device but the application is not processing MIM_DATA messages fast enough to keep up with the input device driver. The callback function receives this message only when the application specifies MIDI_IO_STATUS in the call to the **midiInOpen** function.

- No return value.

dwMidiMessage

Specifies the MIDI message that was received. The message is packed into a doubleword value as follows:

High word	High-order byte	Not used.
	Low-order byte	Contains a second byte of MIDI data (when needed).
Low word	High-order byte	Contains the first byte of MIDI data (when needed).
	Low-order byte	Contains the MIDI status.

The two MIDI data bytes are optional, depending on the MIDI status byte.

dwTimestamp

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

An application should do only a minimal amount of processing of MIM_MOREDATA messages. (In particular, applications should not call the **PostMessage** function while processing MIM_MOREDATA.) Instead, the application should place the event data into a buffer and then return.

When an application receives an MIM_DATA message after a series of MIM_MOREDATA messages, it has caught up with incoming MIDI events and can safely call time-intensive functions.

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.

MIM_OPEN

```
MIM_OPEN
dwParam1 = reserved
dwParam2 = reserved
```

Sent to a MIDI input callback function when a MIDI input device is opened.

- No return value.

dwParam1 and *dwParam2*
Reserved; do not use.

MM_MIM_CLOSE

```
MM_MIM_CLOSE
wParam = (WPARAM) hInput
lParam = reserved
```

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

- No return value.

hInput
Handle of the MIDI input device that was closed.

lParam
Reserved; do not use.

The device handle is no longer valid after this message has been sent.

MM_MIM_DATA

```
MM_MIM_DATA
wParam = (WPARAM) hInput
lParam = (LPARAM) (DWORD) lMidiMessage
```

Sent to a window when a complete MIDI message is received by a MIDI input device.

- No return value.

hInput
Handle of the MIDI input device that received the MIDI message.

lMidiMessage
MIDI message that was received. The message is packed into a doubleword value as follows:

High word	High-order byte	Not used.
	Low-order byte	Contains a second byte of MIDI data (when needed).
Low word	High-order byte	Contains the first byte of MIDI data (when needed).
	Low-order byte	Contains the MIDI status.

The two MIDI data bytes are optional, depending on the MIDI status byte.

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 time stamp is available with this message. For time-stamped input data, you must use the messages that are sent to callback functions.

MM_MIM_ERROR

```
MM_MIM_ERROR
wParam = (WPARAM) hInput
lParam = (LPARAM) (DWORD) lMidiMessage
```

Sent to a window when an invalid MIDI message is received.

- No return value.

hInput
Handle of the MIDI input device that received the invalid message.

lMidiMessage
Invalid MIDI message. The message is packed into a doubleword value with the first byte of the message in the low-order byte.

MM_MIM_LONGDATA

```
MM_MIM_LONGDATA
wParam = (WPARAM) hInput
lParam = (LPARAM) lpMidiHdr
```

Sent to a window when either a complete MIDI system-exclusive message is received or when a buffer has been filled with system-exclusive data.

- No return value.

hInput
Handle of the MIDI input device that received the data.

lpMidiHdr
Address of a **MIDIHDR** structure identifying the buffer.

The returned buffer might not be full. To determine the number of bytes recorded into the returned buffer, use the **dwBytesRecorded** member of the **MIDIHDR** structure pointed to by *lpMidiHdr*.

No time stamp is available with this message. For time-stamped input data, you must use the messages that are sent to callback functions.

MM_MIM_LONGERROR

```
MM_MIM_LONGERROR
wParam = (WPARAM) hInput
lParam = (LPARAM) lpMidiHdr
```

Sent to a window when an invalid or incomplete MIDI system-exclusive message is received.

- No return value.

hInput
Handle of the MIDI input device that received the invalid message.

lpMidiHdr
Address of a **MIDIHDR** structure identifying the buffer containing the invalid message.

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

MM_MIM_MOREDATA

```
MM_MIM_MOREDATA
wParam = (WPARAM) hInput
lParam = (LPARAM) (DWORD) lMidiMessage
```

Sent to a callback window when a MIDI message is received by a MIDI input device but the application is not processing MIM_DATA messages fast enough to keep up with the input device driver. The window receives this message only when the application specifies MIDI_IO_STATUS in the call to the **midiInOpen** function.

- No return value.

hInput
Handle of the MIDI input device that received the MIDI message.

lMidiMessage
Specifies the MIDI message that was received. The message is packed into a doubleword value as follows:

High word	High-order byte	Not used.
	Low-order byte	Contains a second byte of MIDI data (when needed).
Low word	High-order byte	Contains the first byte of MIDI data (when needed).
	Low-order byte	Contains the MIDI status.

The two MIDI data bytes are optional, depending on the MIDI status byte.

If your application will receive MIDI data faster than it can process it, you should not use a window callback mechanism. To maximize speed, use a callback function, and use the MIM_MOREDATA message instead of MM_MIM_MOREDATA.

An application should do only a minimal amount of processing of MM_MIM_MOREDATA messages. (In particular, applications should not call the **PostMessage** function while processing MM_MIM_MOREDATA.) Instead, the application should place the event data into a buffer and then return.

When an application receives an MM_MIM_DATA message after a series of MM_MIM_MOREDATA messages, it has caught up with incoming MIDI events and can safely call time-intensive functions.

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 time stamp is available with this message. For time-stamped input data, you must use the messages that are sent to callback functions.

MM_MIM_OPEN

```
MM_MIM_OPEN
wParam = (WPARAM) hInput
lParam = reserved
```

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

- No return value.

hInput
Handle of the MIDI input device that was opened.

lParam
Reserved; do not use.

MM_MOM_CLOSE

```
MM_MOM_CLOSE
wParam = (WPARAM) hOutput
lParam = reserved
```

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

- No return value.

hOutput
Handle of the MIDI output device.

lParam
Reserved; do not use.

The device handle is no longer valid after this message has been sent.

MM_MOM_DONE

```
MM_MOM_DONE
wParam = (WPARAM) hOutput
lParam = (LPARAM) lpMidiHdr
```

Sent to a window when the specified MIDI system-exclusive or stream buffer has been played and is being returned to the application.

- No return value.

hOutput
Handle of the MIDI output device that played the buffer.

lpMidiHdr
Address of a **MIDIHDR** structure identifying the buffer.

MM_MOM_OPEN

```
MM_MOM_OPEN
wParam = (WPARAM) hOutput
lParam = reserved
```

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

- No return value.

hOutput
Handle of the MIDI output device.

lParam
Reserved; do not use.

MM_MOM_POSITIONCB

```
MM_MOM_POSITIONCB
wParam = (WPARAM) lpMidiHdr
lParam = reserved
```

Sent to a window when an MEVT_F_CALLBACK event is reached in the MIDI output stream.

- No return value.

lpMidiHdr
Address of a **MIDIHDR** structure that identifies the event that caused the callback. The **dwOffset** member gives the offset of the event.

lParam
Reserved; do not use.

Playback of the stream buffer continues even while the callback function is executing. Any events after the MEVT_F_CALLBACK event in the buffer will be scheduled and sent on time regardless of how much time is spent in the callback function.

If position callbacks are being generated more quickly than your application can process them, the **dwOffset** member of the **MIDIHDR** structure might refer to an event your application has not yet processed.

MOM_CLOSE

```
MOM_CLOSE
dwParam1 = reserved
dwParam2 = reserved
```

Sent to a MIDI output callback function when a MIDI output device is closed.

- No return value.

dwParam1 and *dwParam2*
Reserved; do not use.

The device handle is no longer valid after this message has been sent.

MOM_DONE

```
MOM_DONE
dwParam1 = (DWORD) lpMidiHdr
dwParam2 = reserved
```

Sent to a MIDI output callback function when the specified system-exclusive or stream buffer has been played and is being returned to the application.

- No return value.

lpMidiHdr
Address of a **MIDIHDR** structure identifying the buffer.

dwParam2
Reserved; do not use.

MOM_OPEN

```
MOM_OPEN
dwParam1 = reserved
dwParam2 = reserved
```

Sent to a MIDI output callback function when a MIDI output device is opened.

- No return value.

dwParam1 and *dwParam2*
Reserved; do not use.

MOM_POSITIONCB

```
MOM_POSITIONCB
dwParam1 = (DWORD) hOutput
dwParam2 = (DWORD) lpMidiHdr
```

Sent to a window when a MEVT_F_CALLBACK event is reached in the MIDI output stream.

- No return value.

hOutput
Handle of the MIDI output device.

lpMidiHdr
Address of a **MIDIHDR** structure that identifies the event that caused the callback function. The **dwOffset** member gives the offset of the event.

Playback of the stream buffer continues even while the callback function is executing. Any events after the MEVT_F_CALLBACK event in the buffer will be scheduled and sent on time regardless of how much time is spent in the callback function.

If position callbacks are being generated more quickly than your application can process them, the **dwOffset** member of the **MIDIHDR** structure might refer to an event your application has not yet processed.

Audio Mixers

Audio mixer services control the routing of audio lines to a destination device for playing or recording. These services can also control volume and other effects. Many of the techniques required to use these services are similar to those for audio devices discussed in other chapters. This section presents general information about using audio mixer services.

Mixer Architecture

The basic element of the mixer architecture is an *audio line*. An audio line consists of one or more channels of data originating from a single source or a system resource. For example, a stereo audio line has two data channels, but it is considered a single audio line because it originates from a single source.

The mixer architecture provides routing services to manage audio lines on a computer. You can use the routing services if you have adequate hardware devices and software drivers installed. The mixer architecture allows several audio source lines to map to a single destination audio line.

Each audio line can have mixer controls associated with it. A mixer control can perform any number of functions (such as control volume), depending on the characteristics of the associated audio line.

Control Types

The mixer services include the following classes of standard controls to associate with audio lines:

- Custom controls
- Faders
- Lists
- Meters
- Numbers
- Sliders
- Switches
- Time controls

Audio Mixer Custom Controls

Custom controls are the most generic. This control allows a mixer driver to define the control's characteristics, and by implication, the visual representation of the control.

Faders

The fader controls are typically vertical controls that can be adjusted up or down. These controls have a linear scale and use the **MIXERCONTROLDETAILS_UNSIGNED** structure to retrieve and set control details. Types of faders include the following:

Control	Description
Fader	General fade control. The range of acceptable values is 0 through 65,535.
Volume	General volume fade control. The range of acceptable values is 0 through 65,535. For information about changing this range, see the documentation for your mixer device.
Bass	Bass volume fade control. The range of acceptable values is 0 through 65,535. The limits of the bass frequency band are hardware specific. For information about band limits, see the documentation for your mixer device.
Treble	Treble volume fade control. The range of acceptable values is 0 through 65,535. The limits of the treble frequency band are hardware specific. For information about the band limits, see the documentation for your mixer device.
Equalizer	Graphic equalizer control. The range of acceptable values for one band of the equalizer is 0 through 65,535. The number of equalizer bands and their limits are hardware specific. For information about the equalizer, see the documentation for your mixer device. You can use the **MIXERCONTROLDETAILS_LISTTEXT** structure to retrieve text labels for the equalizer.

Lists

The list controls provide single-select or multiple-select states for complex audio lines. These controls use the **MIXERCONTROLDETAILS_BOOLEAN** structure to retrieve and set control properties. The **MIXERCONTROLDETAILS_LISTTEXT** structure is also used to retrieve all text descriptions of a multiple-item control. Types of list controls include the following:

Control	Description
Single-select	Restricts the control selection to one item at a time. Unlike the multiplexer control, this control can be used to control more than audio source lines. For example, you could use this control to select a low-pass filter from a list of filters supported by a mixer device.
Multiplexer (MUX)	Restricts the line selection to one source line at a time.
Multiple-select	Allows the user to select multiple items from a list simultaneously. Unlike the mixer control, the multiple-select control can be used to control more than audio source lines.
Mixer	Allows the user to select source lines from a list simultaneously.

Meters

The meter controls measure data passing through an audio line. These controls use the **MIXERCONTROLDETAILS_BOOLEAN**, **MIXERCONTROLDETAILS_SIGNED**, and **MIXERCONTROLDETAILS_UNSIGNED** structures to retrieve and set control properties. Types of meters include the following:

Control	Description
Boolean	Measures whether an integer value is FALSE/OFF (zero) or TRUE/ON (nonzero).
Peak	Measures the deflection from 0 in both the positive and negative directions. The range of integer values for the peak meter is - 32,768 through 32,767.
Signed	Measures integer values in the range of - $2^{(31)}$ through ($2^{(31)}$ - 1). The mixer driver defines the limits of this meter.
Unsigned	Measures integer values in the range of 0 through ($2^{(32)}$ - 1). The mixer driver defines the limits of this meter.

Numbers

The number controls allow the user to enter numerical data associated with a line. The numerical data is expressed as signed integers, unsigned integers, or integer decibel values. These controls use the **MIXERCONTROLDETAILS_SIGNED** and **MIXERCONTROLDETAILS_UNSIGNED** structures to retrieve and set control properties. Types of number controls include the following:

Control	Description
Signed	Allows integer values entered in the range of $-2^{(31)}$ through $(2^{(31)} - 1)$.
Unsigned	Allows integer values entered in the range of 0 through $(2^{(32)} - 1)$.
Decibel	Allows integer decibel values to be entered, in tenths of decibels. The range of values for this control is -32,768 through 32,767.
Percent	Allows values to be entered as percentages.

Sliders

The slider controls are typically horizontal controls that can be adjusted to the left or right. These controls use the **MIXERCONTROLDETAILS_SIGNED** structure to retrieve and set control properties. Types of sliders include the following:

Control	Description
Slider	Has a range of - 32,768 through 32,767. The mixer driver defines the limits of this control.
Pan	Has a range of -32,768 through 32,767. The mixer driver defines the limits of this control, with 0 as the midrange value.
QSound® Pan	Provides expanded sound control through QSound. This control has a range of -15 through 15.

Switches

The switch controls are two-state switches. These controls use the **MIXERCONTROLDETAILS_BOOLEAN** structure to retrieve and set control properties. Types of switches include the following:

Control	Description
Boolean	The generic switch. It can be set to TRUE or FALSE.
Button	Set to TRUE for all buttons in an application that the driver should handle as though they had been pressed. If the value is FALSE, no action is taken.
On/Off	An alternative switch that is represented by a different graphic than the Boolean switch. It can be set to ON or OFF.
Mute	Mutes an audio line (suppressing the data flow of the line) or allows the audio data to play. This switch is frequently used to help control the lines feeding into the mixer.
Mono	Switches between mono and stereo output for a stereo audio line. Set to OFF to play stereo data as separate channels. Set to ON to combine data from both channels into a mono audio line.
Loudness	Boosts low-volume bass for an audio line. Set to ON to boost low-volume bass. Set to OFF to set volume levels to normal. The amount of boost is hardware specific. For more information, see the documentation for your mixer device.
Stereo Enhanced	Increases stereo separation. Set to ON to increase stereo separation. Set to OFF for no enhancement.

Time Controls

The time controls allow the user to enter timing-related data, such as an echo delay or reverberation. The time data is expressed as positive integers. Types of time controls include the following:

Control	Description
Microsecond	Supports timing data expressed in microseconds. The range of acceptable values is 0 through ($2^{(32)} - 1$).
Millisecond	Supports timing data expressed in milliseconds. The range of acceptable values is 0 through ($2^{(32)} - 1$).

Mixer Device Queries

The mixer services provide functions for determining the number of mixer devices present in the system and the capabilities of the devices. You can also use a mixer services function to determine the device identifier for a mixer device.

You can use the **mixerGetNumDevs** function to determine how many mixer devices are present in the system. Mixer devices are identified by a device identifier. Device identifiers are determined implicitly from the number of devices present in a given system. They range from zero through one less than the number of devices present.

Before using a mixer device, you must determine its capabilities. Audio capabilities can vary from one multimedia computer to another, so applications should not make assumptions about audio hardware.

You can use the **mixerGetDevCaps** function to determine the capabilities of mixer devices. This function fills a **MIXERCAPS** structure with information about the capabilities of a specified device.

The **mixerGetID** function retrieves the audio mixer device identifier associated with a specified device handle. For example, you could use this function to retrieve the device identifier for an audio mixer and then use the device identifier to adjust the volume or to display another control.

Opening and Closing Mixer Devices

When you want to use a mixer device, you can either simply begin using it or you can explicitly open the device before using it. Explicitly opening a mixer device offers two main benefits:

- It guarantees the continued existence of that mixer device.
- It lets you receive notification of audio line and control changes.

You can use the **mixerOpen** function to explicitly open a mixer device. This function takes as parameters a device identifier, a pointer to a memory location, and other values unique to each type of device. The memory location is filled with a device handle. Use this device handle to identify the open mixer device when calling other audio mixer functions. As long as a handle of a mixer device exists, the device continues to exist in the system. If a configuration change occurs to the mixer device and it hasn't been explicitly opened, your application might suddenly be unable to access it.

Note The difference between device identifiers and device handles is important. Device handles are returned when you open a device driver by using **mixerOpen**. Device identifiers are determined implicitly from the number of devices present in a system, which is obtained by using the **mixerGetNumDevs** function.

You can use the **mixerClose** function to close a mixer device. You should close the device after you finish using it.

Window Callback Services

The mixer services provide window callback services so that your application can process messages from mixer drivers. To use these services, specify the CALLBACK_WINDOW flag in the *fdwOpen* parameter and a window handle in the *dwCallback* parameter of the **mixerOpen** function. Driver messages are sent to the window procedure function for the window identified by the handle in *dwCallback*. The messages are specific to the audio device type.

Audio Line and Control Queries

The mixer services provide functions for determining information about audio lines, audio line controls, and control details. The services also provide functions for setting control details.

You can use the **mixerGetLineInfo** function to retrieve information about a specified audio line. This function fills the **MIXERLINE** structure for a specified source audio line, destination audio line, or line identifier. The structure includes the destination line number, the number of channels in the audio line, as well as a short and a long name for the audio line.

The **mixerGetLineControls** function retrieves general information about one or more controls associated with an audio line. This function fills the **MIXERLINECONTROLS** structure with information about the specified control or controls. You can use **mixerGetLineControls** to retrieve control properties for one of the following**:**

- All controls for a specified source line
- Specified control for a specified source line
- First control of a specific class for a specified source line

The **mixerGetControlDetails** function retrieves properties of a single control associated with an audio line. This function fills the **MIXERCONTROLDETAILS** structure with the current values for a control.

The **mixerSetControlDetails** function uses the contents of the **MIXERCONTROLDETAILS** structure to set the properties of the specified control. You must ensure all members of this structure are filled before you call **mixerSetControlDetails**.

Audio Mixer Reference

This section describes the functions, structures, and messages associated with audio mixers. These elements are grouped as follows.

Querying Devices

MIXERCAPS
mixerGetDevCaps
mixerGetNumDevs

Opening and Closing

mixerClose
mixerOpen

Retrieving Mixer Identifiers

mixerGetID

Retrieving Line Controls

MIXERCONTROL
mixerGetLineControls
MIXERLINECONTROLS

Changing Control Attributes

MIXERCONTROLDETAILS
MIXERCONTROLDETAILS_BOOLEAN
MIXERCONTROLDETAILS_LISTTEXT
MIXERCONTROLDETAILS_SIGNED
MIXERCONTROLDETAILS_UNSIGNED
mixerGetControlDetails
mixerSetControlDetails

Retrieving Line Information

mixerGetLineInfo
MIXERLINE
MM_MIXM_CONTROL_CHANGE
MM_MIXM_LINE_CHANGE

Sending User-Defined Messages

mixerMessage

mixerClose

```
MMRESULT mixerClose(HMIXER hmx);
```

Closes the specified mixer device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

  MMSYSERR_INVALHANDLE Specified device handle is invalid.

hmx
Handle of the mixer device. This handle must have been returned successfully by the **mixerOpen** function. If **mixerClose** is successful, *hmx* is no longer valid.

mixerGetControlDetails

```
MMRESULT mixerGetControlDetails(HMIXEROBJ hmxobj,
    LPMIXERCONTROLDETAILS pmxcd, DWORD fdwDetails);
```

Retrieves details about a single control associated with an audio line.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MIXERR_INVALCONTROL	The control reference is invalid.
MMSYSERR_BADDEVICEID	The *hmxobj* parameter specifies an invalid device identifier.
MMSYSERR_INVALFLAG	One or more flags are invalid.
MMSYSERR_INVALHANDLE	The *hmxobj* parameter specifies an invalid handle.
MMSYSERR_INVALPARAM	One or more parameters are invalid.
MMSYSERR_NODRIVER	No mixer device is available for the object specified by *hmxobj*.

hmxobj
Handle of the mixer device object being queried.

pmxcd
Address of a **MIXERCONTROLDETAILS** structure, which is filled with state information about the control.

fdwDetails
Flags for retrieving control details. The following values are defined:

MIXER_GETCONTROLDETAILSF_LISTTEXT
The **paDetails** member of the **MIXERCONTROLDETAILS** structure points to one or more **MIXERCONTROLDETAILS_LISTTEXT** structures to receive text labels for multiple-item controls. An application must get all list text items for a multiple-item control at once. This flag cannot be used with MIXERCONTROL_CONTROLTYPE_CUSTOM controls.

MIXER_GETCONTROLDETAILSF_VALUE
Current values for a control are retrieved. The **paDetails** member of the **MIXERCONTROLDETAILS** structure points to one or more details structures appropriate for the control class.

MIXER_OBJECTF_AUX
The *hmxobj* parameter is an auxiliary device identifier in the range of zero to one less than the number of devices returned by the **auxGetNumDevs** function.

MIXER_OBJECTF_HMIDIIN
The *hmxobj* parameter is the handle of a MIDI (Musical Instrument Digital Interface) input device. This handle must have been returned by the **midiInOpen** function.

MIXER_OBJECTF_HMIDIOUT
The *hmxobj* parameter is the handle of a MIDI output device. This handle must have been returned by the **midiOutOpen** function.

MIXER_OBJECTF_HMIXER
The *hmxobj* parameter is a mixer device handle returned by the **mixerOpen** function. This flag is optional.

MIXER_OBJECTF_HWAVEIN
The *hmxobj* parameter is a waveform-audio input handle returned by the **waveInOpen** function.

MIXER_OBJECTF_HWAVEOUT
The *hmxobj* parameter is a waveform-audio output handle returned by the **waveOutOpen**

function.

MIXER_OBJECTF_MIDIIN
: The *hmxobj* parameter is the identifier of a MIDI input device. This identifier must be in the range of zero to one less than the number of devices returned by the **midiInGetNumDevs** function.

MIXER_OBJECTF_MIDIOUT
: The *hmxobj* parameter is the identifier of a MIDI output device. This identifier must be in the range of zero to one less than the number of devices returned by the **midiOutGetNumDevs** function.

MIXER_OBJECTF_MIXER
: The *hmxobj* parameter is the identifier of a mixer device in the range of zero to one less than the number of devices returned by the **mixerGetNumDevs** function. This flag is optional.

MIXER_OBJECTF_WAVEIN
: The *hmxobj* parameter is the identifier of a waveform-audio input device in the range of zero to one less than the number of devices returned by the **waveInGetNumDevs** function.

MIXER_OBJECTF_WAVEOUT
: The *hmxobj* parameter is the identifier of a waveform-audio output device in the range of zero to one less than the number of devices returned by the **waveOutGetNumDevs** function.

All members of the **MIXERCONTROLDETAILS** structure must be initialized before calling this function.

mixerGetDevCaps

```
MMRESULT mixerGetDevCaps(UINT uMxId, LPMIXERCAPS pmxcaps,
    UINT cbmxcaps);
```

Queries a specified mixer device to determine its capabilities.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_BADDEVICEID	The specified device identifier is out of range.
MMSYSERR_INVALHANDLE	The mixer device handle is invalid.
MMSYSERR_INVALPARAM	One or more parameters are invalid.

uMxId
Identifier or handle of an open mixer device.

pmxcaps
Address of a **MIXERCAPS** structure that receives information about the capabilities of the device.

cbmxcaps
Size, in bytes, of the **MIXERCAPS** structure.

Use the **mixerGetNumDevs** function to determine the number of mixer devices present in the system. The device identifier specified by *uMxId* varies from zero to one less than the number of mixer devices present.

Only the number of bytes (or less) of information specified in *cbmxcaps* is copied to the location pointed to by *pmxcaps*. If *cbmxcaps* is zero, nothing is copied, and the function returns successfully.

This function also accepts a mixer device handle returned by the **mixerOpen** function as the *uMxId* parameter. The application should cast the **HMIXER** handle to a **UINT**.

mixerGetID

```
MMRESULT mixerGetID(HMIXEROBJ hmxobj, UINT FAR * puMxId, DWORD fdwId);
```

Retrieves the device identifier for a mixer device associated with a specified device handle.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_BADDEVICEID	The *hmxobj* parameter specifies an invalid device identifier.
MMSYSERR_INVALFLAG	One or more flags are invalid.
MMSYSERR_INVALHANDLE	The *hmxobj* parameter specifies an invalid handle.
MMSYSERR_INVALPARAM	One or more parameters are invalid.
MMSYSERR_NODRIVER	No audio mixer device is available for the object specified by *hmxobj*. The location referenced by *puMxId* also contains the value -1.

hmxobj
Handle of the audio mixer object to map to a mixer device identifier.

puMxId
Address of a variable that receives the mixer device identifier. If no mixer device is available for the *hmxobj* object, the value - 1 is placed in this location and the MMSYSERR_NODRIVER error value is returned.

fdwId
Flags for mapping the mixer object *hmxobj*. The following values are defined:

MIXER_OBJECTF_AUX
The *hmxobj* parameter is an auxiliary device identifier in the range of zero to one less than the number of devices returned by the **auxGetNumDevs** function.

MIXER_OBJECTF_HMIDIIN
The *hmxobj* parameter is the handle of a MIDI input device. This handle must have been returned by the **midiInOpen** function.

MIXER_OBJECTF_HMIDIOUT
The *hmxobj* parameter is the handle of a MIDI output device. This handle must have been returned by the **midiOutOpen** function.

MIXER_OBJECTF_HMIXER
The *hmxobj* parameter is a mixer device handle returned by the **mixerOpen** function. This flag is optional.

MIXER_OBJECTF_HWAVEIN
The *hmxobj* parameter is a waveform-audio input handle returned by the **waveInOpen** function.

MIXER_OBJECTF_HWAVEOUT
The *hmxobj* parameter is a waveform-audio output handle returned by the **waveOutOpen** function.

MIXER_OBJECTF_MIDIIN
The *hmxobj* parameter is the identifier of a MIDI input device. This identifier must be in the range of zero to one less than the number of devices returned by the **midiInGetNumDevs** function.

MIXER_OBJECTF_MIDIOUT
The *hmxobj* parameter is the identifier of a MIDI output device. This identifier must be in the range of zero to one less than the number of devices returned by the **midiOutGetNumDevs** function.

MIXER_OBJECTF_MIXER

The *hmxobj* parameter is the identifier of a mixer device in the range of zero to one less than the number of devices returned by the **mixerGetNumDevs** function. This flag is optional.

MIXER_OBJECTF_WAVEIN

The *hmxobj* parameter is the identifier of a waveform-audio input device in the range of zero to one less than the number of devices returned by the **waveInGetNumDevs** function.

MIXER_OBJECTF_WAVEOUT

The *hmxobj* parameter is the identifier of a waveform-audio output device in the range of zero to one less than the number of devices returned by the **waveOutGetNumDevs** function.

mixerGetLineControls

```
MMRESULT mixerGetLineControls(HMIXEROBJ hmxobj,
    LPMIXERLINECONTROLS pmxlc, DWORD fdwControls);
```

Retrieves one or more controls associated with an audio line.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MIXERR_INVALCONTROL	The control reference is invalid.
MIXERR_INVALLINE	The audio line reference is invalid.
MMSYSERR_BADDEVICEID	The *hmxobj* parameter specifies an invalid device identifier.
MMSYSERR_INVALFLAG	One or more flags are invalid.
MMSYSERR_INVALHANDLE	The *hmxobj* parameter specifies an invalid handle.
MMSYSERR_INVALPARAM	One or more parameters are invalid.
MMSYSERR_NODRIVER	No mixer device is available for the object specified by *hmxobj*.

hmxobj
Handle of the mixer device object that is being queried.

pmxlc
Address of a **MIXERLINECONTROLS** structure. This structure is used to reference one or more **MIXERCONTROL** structures to be filled with information about the controls associated with an audio line. The **cbStruct** member of the **MIXERLINECONTROLS** structure must always be initialized to be the size, in bytes, of the **MIXERLINECONTROLS** structure.

fdwControls
Flags for retrieving information about one or more controls associated with an audio line. The following values are defined:

MIXER_GETLINECONTROLSF_ALL
The *pmxlc* parameter references a list of **MIXERCONTROL** structures that will receive information on all controls associated with the audio line identified by the **dwLineID** member of the **MIXERLINECONTROLS** structure. The **cControls** member must be initialized to the number of controls associated with the line. This number is retrieved from the **cControls** member of the **MIXERLINE** structure returned by the **mixerGetLineInfo** function. The **cbmxctrl** member must be initialized to the size, in bytes, of a single **MIXERCONTROL** structure. The **pamxctrl** member must point to the first **MIXERCONTROL** structure to be filled. The **dwControlID** and **dwControlType** members are ignored for this query.

MIXER_GETLINECONTROLSF_ONEBYID
The *pmxlc* parameter references a single **MIXERCONTROL** structure that will receive information on the control identified by the **dwControlID** member of the **MIXERLINECONTROLS** structure. The **cControls** member must be initialized to 1. The **cbmxctrl** member must be initialized to the size, in bytes, of a single **MIXERCONTROL** structure. The **pamxctrl** member must point to a **MIXERCONTROL** structure to be filled. The **dwLineID** and **dwControlType** members are ignored for this query. This query is usually used to refresh a control after receiving a MM_MIXM_CONTROL_CHANGE control change notification message by the user-defined callback (see **mixerOpen**).

MIXER_GETLINECONTROLSF_ONEBYTYPE
The **mixerGetLineControls** function retrieves information about the first control of a specific class for the audio line that is being queried. The *pmxlc* parameter references a single **MIXERCONTROL** structure that will receive information about the specific control. The audio line

is identified by the **dwLineID** member. The control class is specified in the **dwControlType** member of the **MIXERLINECONTROLS** structure.

The **dwControlID** member is ignored for this query. This query can be used by an application to get information on a single control associated with a line. For example, you might want your application to use a peak meter only from a waveform-audio output line.

MIXER_OBJECTF_AUX
The *hmxobj* parameter is an auxiliary device identifier in the range of zero to one less than the number of devices returned by the **auxGetNumDevs** function.

MIXER_OBJECTF_HMIDIIN
The *hmxobj* parameter is the handle of a MIDI input device. This handle must have been returned by the **midiInOpen** function.

MIXER_OBJECTF_HMIDIOUT
The *hmxobj* parameter is the handle of a MIDI output device. This handle must have been returned by the **midiOutOpen** function.

MIXER_OBJECTF_HMIXER
The *hmxobj* parameter is a mixer device handle returned by the **mixerOpen** function. This flag is optional.

MIXER_OBJECTF_HWAVEIN
The *hmxobj* parameter is a waveform-audio input handle returned by the **waveInOpen** function.

MIXER_OBJECTF_HWAVEOUT
The *hmxobj* parameter is a waveform-audio output handle returned by the **waveOutOpen** function.

MIXER_OBJECTF_MIDIIN
The *hmxobj* parameter is the identifier of a MIDI input device. This identifier must be in the range of zero to one less than the number of devices returned by the **midiInGetNumDevs** function.

MIXER_OBJECTF_MIDIOUT
The *hmxobj* parameter is the identifier of a MIDI output device. This identifier must be in the range of zero to one less than the number of devices returned by the **midiOutGetNumDevs** function.

MIXER_OBJECTF_MIXER
The *hmxobj* parameter is the identifier of a mixer device in the range of zero to one less than the number of devices returned by the **mixerGetNumDevs** function. This flag is optional.

MIXER_OBJECTF_WAVEIN
The *hmxobj* parameter is the identifier of a waveform-audio input device in the range of zero to one less than the number of devices returned by the **waveInGetNumDevs** function.

MIXER_OBJECTF_WAVEOUT
The *hmxobj* parameter is the identifier of a waveform-audio output device in the range of zero to one less than the number of devices returned by the **waveOutGetNumDevs** function.

mixerGetLineInfo

```
MMRESULT mixerGetLineInfo(HMIXEROBJ hmxobj, LPMIXERLINE pmxl,
    DWORD fdwInfo);
```

Retrieves information about a specific line of a mixer device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MIXERR_INVALLINE	The audio line reference is invalid.
MMSYSERR_BADDEVICEID	The *hmxobj* parameter specifies an invalid device identifier.
MMSYSERR_INVALFLAG	One or more flags are invalid.
MMSYSERR_INVALHANDLE	The *hmxobj* parameter specifies an invalid handle.
MMSYSERR_INVALPARAM	One or more parameters are invalid.
MMSYSERR_NODRIVER	No mixer device is available for the object specified by *hmxobj*.

hmxobj
Handle of the mixer device object that controls the specific audio line.

pmxl
Address of a **MIXERLINE** structure. This structure is filled with information about the audio line for the mixer device. The **cbStruct** member must always be initialized to be the size, in bytes, of the **MIXERLINE** structure.

fdwInfo
Flags for retrieving information about an audio line. The following values are defined:

MIXER_GETLINEINFOF_COMPONENTTYPE
The *pmxl* parameter will receive information about the first audio line of the type specified in the **dwComponentType** member of the **MIXERLINE** structure. This flag is used to retrieve information about an audio line of a specific component type. Remaining structure members except **cbStruct** require no further initialization.

MIXER_GETLINEINFOF_DESTINATION
The *pmxl* parameter will receive information about the destination audio line specified by the **dwDestination** member of the **MIXERLINE** structure. This index ranges from zero to one less than the value in the **cDestinations** member of the **MIXERCAPS** structure. All remaining structure members except **cbStruct** require no further initialization.

MIXER_GETLINEINFOF_LINEID
The *pmxl* parameter will receive information about the audio line specified by the **dwLineID** member of the **MIXERLINE** structure. This is usually used to retrieve updated information about the state of an audio line. All remaining structure members except **cbStruct** require no further initialization.

MIXER_GETLINEINFOF_SOURCE
The *pmxl* parameter will receive information about the source audio line specified by the **dwDestination** and **dwSource** members of the **MIXERLINE** structure. The index specified by **dwDestination** ranges from zero to one less than the value in the **cDestinations** member of the **MIXERCAPS** structure. The index specified by **dwSource** ranges from zero to one less than the value in the **cConnections** member of the **MIXERLINE** structure returned for the audio line stored in the **dwDestination** member. All remaining structure members except **cbStruct** require no further initialization.

MIXER_GETLINEINFOF_TARGETTYPE
The *pmxl* parameter will receive information about the audio line that is for the **dwType** member

of the **Target** structure, which is a member of the **MIXERLINE** structure. This flag is used to retrieve information about an audio line that handles the target type (for example, MIXERLINE_TARGETTYPE_WAVEOUT). An application must initialize the **dwType**, **wMid**, **wPid**, **vDriverVersion** and **szPname** members of the **MIXERLINE** structure before calling **mixerGetLineInfo**. All of these values can be retrieved from the device capabilities structures for all media devices. Remaining structure members except **cbStruct** require no further initialization.

MIXER_OBJECTF_AUX
The *hmxobj* parameter is an auxiliary device identifier in the range of zero to one less than the number of devices returned by the **auxGetNumDevs** function.

MIXER_OBJECTF_HMIDIIN
The *hmxobj* parameter is the handle of a MIDI input device. This handle must have been returned by the **midiInOpen** function.

MIXER_OBJECTF_HMIDIOUT
The *hmxobj* parameter is the handle of a MIDI output device. This handle must have been returned by the **midiOutOpen** function.

MIXER_OBJECTF_HMIXER
The *hmxobj* parameter is a mixer device handle returned by the **mixerOpen** function. This flag is optional.

MIXER_OBJECTF_HWAVEIN
The *hmxobj* parameter is a waveform-audio input handle returned by the **waveInOpen** function.

MIXER_OBJECTF_HWAVEOUT
The *hmxobj* parameter is a waveform-audio output handle returned by the **waveOutOpen** function.

MIXER_OBJECTF_MIDIIN
The *hmxobj* parameter is the identifier of a MIDI input device. This identifier must be in the range of zero to one less than the number of devices returned by the **midiInGetNumDevs** function.

MIXER_OBJECTF_MIDIOUT
The *hmxobj* parameter is the identifier of a MIDI output device. This identifier must be in the range of zero to one less than the number of devices returned by the **midiOutGetNumDevs** function.

MIXER_OBJECTF_MIXER
The *hmxobj* parameter is a mixer device identifier in the range of zero to one less than the number of devices returned by the **mixerGetNumDevs** function. This flag is optional.

MIXER_OBJECTF_WAVEIN
The *hmxobj* parameter is the identifier of a waveform-audio input device in the range of zero to one less than the number of devices returned by the **waveInGetNumDevs** function.

MIXER_OBJECTF_WAVEOUT
The *hmxobj* parameter is the identifier of a waveform-audio output device in the range of zero to one less than the number of devices returned by the **waveOutGetNumDevs** function.

mixerGetNumDevs

```
UINT mixerGetNumDevs(VOID);
```

Retrieves the number of mixer devices present in the system.

- Returns the number of mixer devices or zero if no mixer devices are available.

mixerMessage

```
DWORD mixerMessage(HMIXER hmx, UINT uMsg, DWORD dwParam1,
    DWORD dwParam2);
```

Sends a custom mixer driver message directly to a mixer driver.

- Returns a value that is specific to the custom mixer driver message in the *uMsg* parameter.

MMSYSERR_INVALHANDLE	The specified device handle is invalid.
MMSYSERR_INVALPARAM	The *uMsg* parameter specified in the MXDM_USER message is invalid.
MMSYSERR_NOTSUPPORTED	The mixer device did not process the message.

hmx
Handle of an open instance of a mixer device. This handle is returned by the **mixerOpen** function.

uMsg
Custom mixer driver message to send to the mixer driver. This message must be above or equal to the MXDM_USER constant.

dwParam1 and *dwParam2*
Arguments associated with the message being sent.

User-defined messages must be sent only to a mixer driver that supports the messages. The application should verify that the mixer driver is the driver that supports the message by retrieving the mixer capabilities and checking the **wMid**, **wPid**, **vDriverVersion**, and **szPname** members of the **MIXERCAPS** structure.

mixerOpen

```
MMRESULT mixerOpen(LPHMIXER phmx, UINT uMxId, DWORD dwCallback,
    DWORD dwInstance, DWORD fdwOpen);
```

Opens a specified mixer device and ensures that the device will not be removed until the application closes the handle.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_ALLOCATED	The specified resource is already allocated by the maximum number of clients possible.
MMSYSERR_BADDEVICEID	The *uMxId* parameter specifies an invalid device identifier.
MMSYSERR_INVALFLAG	One or more flags are invalid.
MMSYSERR_INVALHANDLE	The *uMxId* parameter specifies an invalid handle.
MMSYSERR_INVALPARAM	One or more parameters are invalid.
MMSYSERR_NODRIVER	No mixer device is available for the object specified by *uMxId*. Note that the location referenced by *uMxId* will also contain the value - 1.
MMSYSERR_NOMEM	Unable to allocate resources.

phmx
Address of a variable that will receive a handle identifying the opened mixer device. Use this handle to identify the device when calling other audio mixer functions. This parameter cannot be NULL.

uMxId
Identifier of the mixer device to open. Use a valid device identifier or any **HMIXEROBJ** (see the **mixerGetID** function for a description of mixer object handles). A "mapper" for audio mixer devices does not currently exist, so a mixer device identifier of - 1 is not valid.

dwCallback
Handle of a window called when the state of an audio line and/or control associated with the device being opened is changed. Specify zero for this parameter if no callback mechanism is to be used.

dwInstance
User instance data passed to the callback function. This parameter is not used with window callback functions.

fdwOpen
Flags for opening the device. The following values are defined:

CALLBACK_WINDOW
The *dwCallback* parameter is assumed to be a window handle.

MIXER_OBJECTF_AUX
The *uMxId* parameter is an auxiliary device identifier in the range of zero to one less than the number of devices returned by the **auxGetNumDevs** function.

MIXER_OBJECTF_HMIDIIN
The *uMxId* parameter is the handle of a MIDI input device. This handle must have been returned by the **midiInOpen** function.

MIXER_OBJECTF_HMIDIOUT
The *uMxId* parameter is the handle of a MIDI output device. This handle must have been returned by the **midiOutOpen** function.

MIXER_OBJECTF_HMIXER
The *uMxId* parameter is a mixer device handle returned by the **mixerOpen** function. This flag is optional.

MIXER_OBJECTF_HWAVEIN
The *uMxId* parameter is a waveform-audio input handle returned by the **waveInOpen** function.

MIXER_OBJECTF_HWAVEOUT
The *uMxId* parameter is a waveform-audio output handle returned by the **waveOutOpen** function.

MIXER_OBJECTF_MIDIIN
The *uMxId* parameter is the identifier of a MIDI input device. This identifier must be in the range of zero to one less than the number of devices returned by the **midiInGetNumDevs** function.

MIXER_OBJECTF_MIDIOUT
The *uMxId* parameter is the identifier of a MIDI output device. This identifier must be in the range of zero to one less than the number of devices returned by the **midiOutGetNumDevs** function.

MIXER_OBJECTF_MIXER
The *uMxId* parameter is a mixer device identifier in the range of zero to one less than the number of devices returned by the **mixerGetNumDevs** function. This flag is optional.

MIXER_OBJECTF_WAVEIN
The *uMxId* parameter is the identifier of a waveform-audio input device in the range of zero to one less than the number of devices returned by the **waveInGetNumDevs** function.

MIXER_OBJECTF_WAVEOUT
The *uMxId* parameter is the identifier of a waveform-audio output device in the range of zero to one less than the number of devices returned by the **waveOutGetNumDevs** function.

Use the **mixerGetNumDevs** function to determine the number of audio mixer devices present in the system. The device identifier specified by *uMxId* varies from zero to one less than the number of devices present.

If a window is chosen to receive callback information, the MM_MIXM_LINE_CHANGE and MM_MIXM_CONTROL_CHANGE messages are sent to the window procedure function to indicate when an audio line or control state changes. For both messages, the *wParam* parameter is the handle of the mixer device. The *lParam* parameter is the line identifier for MM_MIXM_LINE_CHANGE or the control identifier for MM_MIXM_CONTROL_CHANGE that changed state.

To query for audio mixer support or a media device, use the **mixerGetID** function.

mixerSetControlDetails

```
MMRESULT mixerSetControlDetails(HMIXEROBJ hmxobj,
    LPMIXERCONTROLDETAILS pmxcd, DWORD fdwDetails);
```

Sets properties of a single control associated with an audio line.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MIXERR_INVALCONTROL	The control reference is invalid.
MMSYSERR_BADDEVICEID	The *hmxobj* parameter specifies an invalid device identifier.
MMSYSERR_INVALFLAG	One or more flags are invalid.
MMSYSERR_INVALHANDLE	The *hmxobj* parameter specifies an invalid handle.
MMSYSERR_INVALPARAM	One or more parameters are invalid.
MMSYSERR_NODRIVER	No mixer device is available for the object specified by *hmxobj*.

hmxobj
Handle of the mixer device object for which properties are being set.

pmxcd
Address of a **MIXERCONTROLDETAILS** structure. This structure is used to reference control detail structures that contain the desired state for the control.

fdwDetails
Flags for setting properties for a control. The following values are defined:

MIXER_OBJECTF_AUX
The *hmxobj* parameter is an auxiliary device identifier in the range of zero to one less than the number of devices returned by the **auxGetNumDevs** function.

MIXER_OBJECTF_HMIDIIN
The *hmxobj* parameter is the handle of a MIDI input device. This handle must have been returned by the **midiInOpen** function.

MIXER_OBJECTF_HMIDIOUT
The *hmxobj* parameter is the handle of a MIDI output device. This handle must have been returned by the **midiOutOpen** function.

MIXER_OBJECTF_HMIXER
The *hmxobj* parameter is a mixer device handle returned by the **mixerOpen** function. This flag is optional.

MIXER_OBJECTF_HWAVEIN
The *hmxobj* parameter is a waveform-audio input handle returned by the **waveInOpen** function.

MIXER_OBJECTF_HWAVEOUT
The *hmxobj* parameter is a waveform-audio output handle returned by the **waveOutOpen** function.

MIXER_OBJECTF_MIDIIN
The *hmxobj* parameter is the identifier of a MIDI input device. This identifier must be in the range of zero to one less than the number of devices returned by the **midiInGetNumDevs** function.

MIXER_OBJECTF_MIDIOUT
The *hmxobj* parameter is the identifier of a MIDI output device. This identifier must be in the range of zero to one less than the number of devices returned by the **midiOutGetNumDevs** function.

MIXER_OBJECTF_MIXER
The *hmxobj* parameter is a mixer device identifier in the range of zero to one less than the

number of devices returned by the **mixerGetNumDevs** function. This flag is optional.

MIXER_OBJECTF_WAVEIN
The *hmxobj* parameter is the identifier of a waveform-audio input device in the range of zero to one less than the number of devices returned by the **waveInGetNumDevs** function.

MIXER_OBJECTF_WAVEOUT
The *hmxobj* parameter is the identifier of a waveform-audio output device in the range of zero to one less than the number of devices returned by the **waveOutGetNumDevs** function.

MIXER_SETCONTROLDETAILSF_CUSTOM
A custom dialog box for the specified custom mixer control is displayed. The mixer device gathers the required information from the user and returns the data in the specified buffer. The handle for the owning window is specified in the **hwndOwner** member of the **MIXERCONTROLDETAILS** structure. (This handle can be set to NULL.) The application can then save the data from the dialog box and use it later to reset the control to the same state by using the MIXER_SETCONTROLDETAILSF_VALUE flag.

MIXER_SETCONTROLDETAILSF_VALUE
The current value(s) for a control are set. The **paDetails** member of the **MIXERCONTROLDETAILS** structure points to one or more mixer-control details structures of the appropriate class for the control.

All members of the **MIXERCONTROLDETAILS** structure must be initialized before calling **mixerSetControlDetails**.

If an application needs to retrieve only the current state of a custom mixer control and not display a dialog box, then **mixerGetControlDetails** can be used with the MIXER_GETCONTROLDETAILSF_VALUE flag.

MIXERCAPS

```
typedef struct {
    WORD    wMid;
    WORD    wPid;
    MMVERSION vDriverVersion;
    CHAR    szPname[MAXPNAMELEN];
    DWORD   fdwSupport;
    DWORD   cDestinations;
} MIXERCAPS;
```

Describes the capabilities of a mixer device.

wMid
A manufacturer identifier for the mixer device driver. For more information about manufacturer identifiers, see Chapter 0, "Manufacturer and Product Identifiers."

wPid
A product identifier for the mixer device driver. For more information about product identifiers, see Chapter 0, "Manufacturer and Product Identifiers."

vDriverVersion
Version number of the mixer device driver. The high-order byte is the major version number, and the low-order byte is the minor version number.

szPname
Name of the product. If the mixer device driver supports multiple cards, this string must uniquely and easily identify (potentially to a user) the specific card.

fdwSupport
Various support information for the mixer device driver. No extended support bits are currently defined.

cDestinations
The number of audio line destinations available through the mixer device. All mixer devices must support at least one destination line, so this member cannot be zero. Destination indexes used in the **dwDestination** member of the **MIXERLINE** structure range from zero to the value specified in the **cDestinations** member minus one.

MIXERCONTROL

```
typedef struct {
    DWORD cbStruct;
    DWORD dwControlID;
    DWORD dwControlType;
    DWORD fdwControl;
    DWORD cMultipleItems;
    CHAR  szShortName[MIXER_SHORT_NAME_CHARS];
    CHAR  szName[MIXER_LONG_NAME_CHARS];
    union {                     // start Bounds union
        struct {
            LONG lMinimum;
            LONG lMaximum;
        };
        struct {
            DWORD dwMinimum;
            DWORD dwMaximum;
        };
        DWORD dwReserved[6];
    } Bounds;
    union {                     // start Metrics union
        DWORD cSteps;
        DWORD cbCustomData;
        DWORD dwReserved[6];    // reserved; do not use.
    } Metrics;
} MIXERCONTROL, *PMIXERCONTROL, FAR *LPMIXERCONTROL;
```

Describes the state and metrics of a single control for an audio line.

cbStruct
Size, in bytes, of the **MIXERCONTROL** structure.

dwControlID
Audio mixer-defined identifier that uniquely refers to the control described by the **MIXERCONTROL** structure. This identifier can be in any format supported by the mixer device. An application should use this identifier only as an abstract handle. No two controls for a single mixer device can ever have the same control identifier.

dwControlType
Class of the control for which the identifier is specified in **dwControlID**. An application must use this information to display the appropriate control for input from the user. An application can also display tailored graphics based on the control class or search for a particular control class on a specific line. If an application does not know about a control class, this control must be ignored. There are eight control class classifications, each with one or more standard control types:

MIXERCONTROL_CT_CLASS_CUSTOM
  MIXERCONTROL_CONTROLTYPE_CUSTOM

MIXERCONTROL_CT_CLASS_FADER
  MIXERCONTROL_CONTROLTYPE_BASS
  MIXERCONTROL_CONTROLTYPE_EQUALIZER
  MIXERCONTROL_CONTROLTYPE_FADER
  MIXERCONTROL_CONTROLTYPE_TREBLE
  MIXERCONTROL_CONTROLTYPE_VOLUME

MIXERCONTROL_CT_CLASS_LIST
  MIXERCONTROL_CONTROLTYPE_MIXER
  MIXERCONTROL_CONTROLTYPE_MULTIPLESELECT

MIXERCONTROL_CONTROLTYPE_MUX
MIXERCONTROL_CONTROLTYPE_SINGLESELECT

MIXERCONTROL_CT_CLASS_METER
MIXERCONTROL_CONTROLTYPE_BOOLEANMETER
MIXERCONTROL_CONTROLTYPE_PEAKMETER
MIXERCONTROL_CONTROLTYPE_SIGNEDMETER
MIXERCONTROL_CONTROLTYPE_UNSIGNEDMETER

MIXERCONTROL_CT_CLASS_NUMBER
MIXERCONTROL_CONTROLTYPE_DECIBELS
MIXERCONTROL_CONTROLTYPE_PERCENT
MIXERCONTROL_CONTROLTYPE_SIGNED
MIXERCONTROL_CONTROLTYPE_UNSIGNED

MIXERCONTROL_CT_CLASS_SLIDER
MIXERCONTROL_CONTROLTYPE_PAN
MIXERCONTROL_CONTROLTYPE_QSOUNDPAN
MIXERCONTROL_CONTROLTYPE_SLIDER

MIXERCONTROL_CT_CLASS_SWITCH
MIXERCONTROL_CONTROLTYPE_BOOLEAN
MIXERCONTROL_CONTROLTYPE_BUTTON
MIXERCONTROL_CONTROLTYPE_LOUDNESS
MIXERCONTROL_CONTROLTYPE_MONO
MIXERCONTROL_CONTROLTYPE_MUTE
MIXERCONTROL_CONTROLTYPE_ONOFF
MIXERCONTROL_CONTROLTYPE_STEREOENH

MIXERCONTROL_CT_CLASS_TIME
MIXERCONTROL_CONTROLTYPE_MICROTIME
MIXERCONTROL_CONTROLTYPE_MILLITIME

fdwControl

Status and support flags for the audio line control. The following values are defined:

MIXERCONTROL_CONTROLF_DISABLED
The control is disabled, perhaps due to other settings for the mixer hardware, and cannot be used. An application can read current settings from a disabled control, but it cannot apply settings.

MIXERCONTROL_CONTROLF_MULTIPLE
The control has two or more settings per channel. An equalizer, for example, requires this flag because each frequency band can be set to a different value. An equalizer that affects both channels of a stereo line in a uniform fashion will also specify the MIXERCONTROL_CONTROLF_UNIFORM flag.

MIXERCONTROL_CONTROLF_UNIFORM
The control acts on all channels of a multichannel line in a uniform fashion. For example, a control that mutes both channels of a stereo line would set this flag. Most MIXERCONTROL_CONTROLTYPE_MUX and MIXERCONTROL_CONTROLTYPE_MIXER controls also specify the MIXERCONTROL_CONTROLF_UNIFORM flag.

cMultipleItems

Number of items per channel that make up a MIXERCONTROL_CONTROLF_MULTIPLE control. This number is always two or greater for multiple-item controls. If the control is not a multiple-item control, do not use this member; it will be zero.

szShortName

Short string that describes the audio line control specified by **dwControlID**. This description should be appropriate to use as a concise label for the control.

szName

String that describes the audio line control specified by **dwControlID**. This description should be

appropriate to use as a complete description for the control.

Bounds
Union of boundary types.

lMinimum
Minimum signed value for a control that has a signed boundary nature. This member cannot be used in conjunction with **dwMinimun**.

lMaximum
Maximum signed value for a control that has a signed boundary nature. This member cannot be used in conjunction with **dwMaximun**.

dwMinimum
Minimum unsigned value for a control that has an unsigned boundary nature. This member cannot be used in conjunction with **lMinimun**.

dwMaximum
Maximum unsigned value for a control that has an unsigned boundary nature. This member cannot be used in conjunction with **lMaximum**.

Metrics
Union of boundary metrics.

cSteps
Number of discrete ranges within the union specified for a control specified by the **Bounds** member. This member overlaps with the other members of the **Metrics** structure member and cannot be used in conjunction with those members.

cbCustomData
Size, in bytes, required to contain the state of a custom control class. This member is appropriate only for the MIXERCONTROL_CONTROLTYPE_CUSTOM control class.

To determine if the **dwMinimum**, **dwMaximum**, **lMinimum**, **lMaximum**, **cSteps**, and **cbCustomData** members are appropriate for a control class, see "Control Types" earlier in this chapter.

The calling application does not need to initialize any members of this structure because the **MIXERCONTROL** structure is passed to the **mixerGetLineControls** function as a receiving buffer that is referenced and described by the **MIXERLINECONTROLS** structure. When **mixerGetLineControls** returns, the **cbStruct** member contains the actual size of the information returned by the mixer device. The returned information will not exceed the requested size, nor will it be smaller than the **MIXERCONTROL** structure.

MIXERCONTROLDETAILS

```
typedef struct {
    DWORD cbStruct;
    DWORD dwControlID;
    DWORD cChannels;
    union {
        HWND  hwndOwner;
        DWORD cMultipleItems;
    };
    DWORD  cbDetails;
    LPVOID paDetails;
} MIXERCONTROLDETAILS;
```

Refers to control-detail structures, retrieving or setting state information of an audio mixer control. All members of this structure must be initialized before calling the **mixerGetControlDetails** and **mixerSetControlDetails** functions.

cbStruct
Size, in bytes, of the **MIXERCONTROLDETAILS** structure. The size must be large enough to contain the base **MIXERCONTROLDETAILS** structure. When **mixerGetControlDetails** returns, this member contains the actual size of the information returned. The returned information will not exceed the requested size, nor will it be smaller than the base **MIXERCONTROLDETAILS** structure.

dwControlID
Control identifier on which to get or set properties.

cChannels
Number of channels on which to get or set control properties. The following values are defined:

0
Use this value when the control is a MIXERCONTROL_CONTROLTYPE_CUSTOM control.

1
Use this value when the control is a MIXERCONTROL_CONTROLF_UNIFORM control or when an application needs to get and set all channels as if they were uniform.

MIXERLINE.cChannels
Use this value when the properties for the control are expected on all channels for a line.

An application cannot specify a value that falls between 1 and the number of channels for the audio line. For example, specifying 2 or 3 for a four-channel line is not valid. This member cannot be 0 for noncustom control types.

hwndOwner
Handle of the window that owns a custom dialog box for a mixer control. This member is used when the MIXER_SETCONTROLDETAILSF_CUSTOM flag is specified in the **mixerSetControlDetails** function.

cMultipleItems
Number of multiple items per channel on which to get or set properties. The following values are defined:

0
Use this value for all controls except for a MIXERCONTROL_CONTROLF_MULTIPLE or a MIXERCONTROL_CONTROLTYPE_CUSTOM control.

MIXERCONTROL.cMultipleItems
Use this value when the control class is MIXERCONTROL_CONTROLF_MULTIPLE.

MIXERCONTROLDETAILS.hwndOwner
Use this value when the control is a MIXERCONTROL_CONTROLTYPE_CUSTOM control and the MIXER_SETCONTROLDETAILSF_CUSTOM flag is specified for the

mixerSetControlDetails function. In this case, the **hwndOwner** member overlaps with **cMultipleItems**, providing the value of the window handle.

When using a MIXERCONTROL_CONTROLTYPE_CUSTOM control without the MIXERCONTROL_CONTROLTYPE_CUSTOM flag, specify zero for this member.

An application cannot specify any value other than the value specified in the **cMultipleItems** member of the **MIXERCONTROL** structure for a MIXERCONTROL_CONTROLF_MULTIPLE control.

cbDetails

Size, in bytes, of one of the following details structures being used:

MIXERCONTROLDETAILS_BOOLEAN
: Boolean value for an audio line control.

MIXERCONTROLDETAILS_LISTTEXT
: List text buffer for an audio line control. For information about the appropriate details structure for a specific control, see "Control Types" earlier in this chapter.

MIXERCONTROLDETAILS_SIGNED
: Signed value for an audio line control.

MIXERCONTROLDETAILS_UNSIGNED
: Unsigned value for an audio line control.

If the control is a MIXERCONTROL_CONTROLTYPE_CUSTOM control, this member must be equal to the **cbCustomData** member of the **MIXERCONTROL** structure.

paDetails

Address of an array of one or more structures in which properties for the specified control are retrieved or set. For MIXERCONTROL_CONTROLF_MULTIPLE controls, the size of this buffer should be the product of the **cChannels**, **cMultipleItems** and **cbDetails** members of the **MIXERCONTROLDETAILS** structure. For controls other than MIXERCONTROL_CONTROLF_MULTIPLE types, the size of this buffer is the product of the **cChannels** and **cbDetails** members of the **MIXERCONTROLDETAILS** structure.

For controls that are MIXERCONTROL_CONTROLF_MULTIPLE types, the array can be treated as a two-dimensional array that is channel major. That is, all multiple items for the left channel are given, then all multiple items for the right channel, and so on.

For controls other than MIXERCONTROL_CONTROLF_MULTIPLE types, each element index is equivalent to the zero-based channel that it affects. That is, **paDetails**[0] is for the left channel and **paDetails**[1] is for the right channel.

If the control is a MIXERCONTROL_CONTROLTYPE_CUSTOM control, this member must point to a buffer that is at least large enough to contain the size, in bytes, specified by the **cbCustomData** member of the **MIXERCONTROL** structure.

MIXERCONTROLDETAILS_BOOLEAN

```
typedef struct {
    LONG fValue;
} MIXERCONTROLDETAILS_BOOLEAN;
```

Retrieves and sets Boolean control properties for an audio mixer control.

fValue
Boolean value for a single item or channel. This value is assumed to be zero for a FALSE state (such as off or disabled), and nonzero for a TRUE state (such as on or enabled).

The following standard control types use this structure for retrieving and setting properties.

Meter controls:
MIXERCONTROL_CONTROLTYPE_BOOLEANMETER

Switch controls:
MIXERCONTROL_CONTROLTYPE_BOOLEAN
MIXERCONTROL_CONTROLTYPE_BUTTON
MIXERCONTROL_CONTROLTYPE_LOUDNESS
MIXERCONTROL_CONTROLTYPE_MONO
MIXERCONTROL_CONTROLTYPE_MUTE
MIXERCONTROL_CONTROLTYPE_ONOFF
MIXERCONTROL_CONTROLTYPE_STEREOENH

List controls:
MIXERCONTROL_CONTROLTYPE_MIXER
MIXERCONTROL_CONTROLTYPE_MULTIPLESELECT
MIXERCONTROL_CONTROLTYPE_MUX
MIXERCONTROL_CONTROLTYPE_SINGLESELECT

MIXERCONTROLDETAILS_LISTTEXT

```
typedef struct {
    DWORD dwParam1;
    DWORD dwParam2;
    CHAR  szName[MIXER_LONG_NAME_CHARS];
} MIXERCONTROLDETAILS_LISTTEXT;
```

Retrieves list text, label text, and/or band-range information for multiple-item controls. This structure is used when the MIXER_GETCONTROLDETAILSF_LISTTEXT flag is specified in the **mixerGetControlDetails** function.

dwParam1 and **dwParam2**
Control class-specific values. The following control types are listed with their corresponding values:

EQUALIZER
MIXERCONTROL.Bounds.dwMinimum

MIXER and MUX
MIXERLINE.dwLineID

MULTIPLESELECT and SINGLESELECT
Undefined; must be zero

szName
Name describing a single item in a multiple-item control. This text can be used as a label or item text, depending on the control class.

The following standard control types use this structure for retrieving the item text descriptions on multiple-item controls:

Fader control:
MIXERCONTROL_CONTROLTYPE_EQUALIZER

List controls:
MIXERCONTROL_CONTROLTYPE_MIXER
MIXERCONTROL_CONTROLTYPE_MULTIPLESELECT
MIXERCONTROL_CONTROLTYPE_MUX
MIXERCONTROL_CONTROLTYPE_SINGLESELECT

MIXERCONTROLDETAILS_SIGNED

```
typedef struct {
    LONG lValue;
} MIXERCONTROLDETAILS_SIGNED;
```

Retrieves and sets signed type control properties for an audio mixer control.

lValue
Signed integer value for a single item or channel. This value must be inclusively within the bounds given in the **Bounds** member of this structure for signed integer controls.

The following standard control types use this structure for retrieving and setting properties:

Meter controls:
MIXERCONTROL_CONTROLTYPE_PEAKMETER
MIXERCONTROL_CONTROLTYPE_SIGNEDMETER

Member controls:
MIXERCONTROL_CONTROLTYPE_SIGNED

Number controls:
MIXERCONTROL_CONTROLTYPE_DECIBELS

Slider controls:
MIXERCONTROL_CONTROLTYPE_PAN
MIXERCONTROL_CONTROLTYPE_QSOUNDPAN
MIXERCONTROL_CONTROLTYPE_SLIDER

MIXERCONTROLDETAILS_UNSIGNED

```
typedef struct {
    DWORD dwValue;
} MIXERCONTROLDETAILS_UNSIGNED;
```

Retrieves and sets unsigned type control properties for an audio mixer control.

dwValue
Unsigned integer value for a single item or channel. This value must be inclusively within the bounds given in the **Bounds** structure member of the **MIXERCONTROL** structure for unsigned integer controls.

The following standard control types use this structure for retrieving and setting properties:

Meter control:
MIXERCONTROL_CONTROLTYPE_UNSIGNEDMETER

Number control:
MIXERCONTROL_CONTROLTYPE_UNSIGNED

Fader controls:
MIXERCONTROL_CONTROLTYPE_BASS
MIXERCONTROL_CONTROLTYPE_EQUALIZER
MIXERCONTROL_CONTROLTYPE_FADER
MIXERCONTROL_CONTROLTYPE_TREBLE
MIXERCONTROL_CONTROLTYPE_VOLUME

Time controls:
MIXERCONTROL_CONTROLTYPE_MICROTIME
MIXERCONTROL_CONTROLTYPE_MILLITIME
MIXERCONTROL_CONTROLTYPE_PERCENT

MIXERLINE

```
typedef struct {
    DWORD cbStruct;
    DWORD dwDestination;
    DWORD dwSource;
    DWORD dwLineID;
    DWORD fdwLine;
    DWORD dwUser;
    DWORD dwComponentType;
    DWORD cChannels;
    DWORD cConnections;
    DWORD cControls;
    CHAR  szShortName[MIXER_SHORT_NAME_CHARS];
    CHAR  szName[MIXER_LONG_NAME_CHARS];
    struct {                    // start Target structure
        DWORD     dwType;
        DWORD     dwDeviceID;
        WORD      wMid;
        WORD      wPid;
        MMVERSION vDriverVersion;
        CHAR      szPname[MAXPNAMELEN];
    } Target;
} MIXERLINE;
```

Describes the state and metrics of an audio line.

cbStruct
Size, in bytes, of the **MIXERLINE** structure. This member must be initialized before calling the **mixerGetLineInfo** function. The size specified in this member must be large enough to contain the **MIXERLINE** structure. When **mixerGetLineInfo** returns, this member contains the actual size of the information returned. The returned information will not exceed the requested size.

dwDestination
Destination line index. This member ranges from zero to one less than the value specified in the **cDestinations** member of the **MIXERCAPS** structure retrieved by the **mixerGetDevCaps** function. When the **mixerGetLineInfo** function is called with the MIXER_GETLINEINFOF_DESTINATION flag, properties for the destination line are returned. (The **dwSource** member must be set to zero in this case.) When called with the MIXER_GETLINEINFOF_SOURCE flag, the properties for the source given by the **dwSource** member that is associated with the **dwDestination** member are returned.

dwSource
Index for the audio source line associated with the **dwDestination** member. That is, this member specifies the *n*th audio source line associated with the specified audio destination line. This member is not used for destination lines and must be set to zero when MIXER_GETLINEINFOF_DESTINATION is specified in the **mixerGetLineInfo** function. When the MIXER_GETLINEINFOF_SOURCE flag is specified, this member ranges from zero to one less than the value specified in the **cConnections** member for the audio destination line given in the **dwDestination** member.

dwLineID
An identifier defined by the mixer device that uniquely refers to the audio line described by the **MIXERLINE** structure. This identifier is unique for each mixer device and can be in any format. An application should use this identifier only as an abstract handle.

fdwLine
Status and support flags for the audio line. This member is always returned to the application and

requires no initialization.

MIXERLINE_LINEF_ACTIVE
Audio line is active. An active line indicates that a signal is probably passing through the line.

MIXERLINE_LINEF_DISCONNECTED
Audio line is disconnected. A disconnected line's associated controls can still be modified, but the changes have no effect until the line is connected.

MIXERLINE_LINEF_SOURCE
Audio line is an audio source line associated with a single audio destination line. If this flag is not set, this line is an audio destination line associated with zero or more audio source lines.

If an application is not using a waveform-audio output device, the audio line associated with that device would not be active (that is, the MIXERLINE_LINEF_ACTIVE flag would not be set). If the waveform-audio output device is opened, then the audio line is considered active and the MIXERLINE_LINEF_ACTIVE flag will be set. A paused or starved waveform-audio output device is still considered active. In other words, if the waveform-audio output device is opened by an application regardless of whether data is being played, the associated audio line is considered active. If a line cannot be strictly defined as active, the mixer device will always set the MIXERLINE_LINEF_ACTIVE flag.

dwUser
Instance data defined by the audio device for the line. This member is intended for custom mixer applications designed specifically for the mixer device returning this information. Other applications should ignore this data.

dwComponentType
Component type for this audio line. An application can use this information to display tailored graphics or to search for a particular component. If an application does not use component types, this member should be ignored. This member can be one of the following values:

MIXERLINE_COMPONENTTYPE_DST_DIGITAL
Audio line is a digital destination (for example, digital input to a DAT or CD audio device).

MIXERLINE_COMPONENTTYPE_DST_HEADPHONES
Audio line is an adjustable (gain and/or attenuation) destination intended to drive headphones. Most audio cards use the same audio destination line for speakers and headphones, in which case the mixer device simply uses the MIXERLINE_COMPONENTTYPE_DST_SPEAKERS type.

MIXERLINE_COMPONENTTYPE_DST_LINE
Audio line is a line level destination (for example, line level input from a CD audio device) that will be the final recording source for the analog-to-digital converter (ADC). Because most audio cards for personal computers provide some sort of gain for the recording audio source line, the mixer device will use the MIXERLINE_COMPONENTTYPE_DST_WAVEIN type.

MIXERLINE_COMPONENTTYPE_DST_MONITOR
Audio line is a destination used for a monitor.

MIXERLINE_COMPONENTTYPE_DST_SPEAKERS
Audio line is an adjustable (gain and/or attenuation) destination intended to drive speakers. This is the typical component type for the audio output of audio cards for personal computers.

MIXERLINE_COMPONENTTYPE_DST_TELEPHONE
Audio line is a destination that will be routed to a telephone line.

MIXERLINE_COMPONENTTYPE_DST_UNDEFINED
Audio line is a destination that cannot be defined by one of the standard component types. A mixer device is required to use this component type for line component types that have not been defined by Microsoft Corporation.

MIXERLINE_COMPONENTTYPE_DST_VOICEIN
Audio line is a destination that will be the final recording source for voice input. This component type is exactly like MIXERLINE_COMPONENTTYPE_DST_WAVEIN but is intended specifically for settings used during voice recording/recognition. Support for this line is optional for a mixer

device. Many mixer devices provide only MIXERLINE_COMPONENTTYPE_DST_WAVEIN.

MIXERLINE_COMPONENTTYPE_DST_WAVEIN
Audio line is a destination that will be the final recording source for the waveform-audio input (ADC). This line typically provides some sort of gain or attenuation. This is the typical component type for the recording line of most audio cards for personal computers.

MIXERLINE_COMPONENTTYPE_SRC_ANALOG
Audio line is an analog source (for example, analog output from a video-cassette tape).

MIXERLINE_COMPONENTTYPE_SRC_AUXILIARY
Audio line is a source originating from the auxiliary audio line. This line type is intended as a source with gain or attenuation that can be routed to the MIXERLINE_COMPONENTTYPE_DST_SPEAKERS destination and/or recorded from the MIXERLINE_COMPONENTTYPE_DST_WAVEIN destination.

MIXERLINE_COMPONENTTYPE_SRC_COMPACTDISC
Audio line is a source originating from the output of an internal audio CD. This component type is provided for audio cards that provide an audio source line intended to be connected to an audio CD (or CD-ROM playing an audio CD).

MIXERLINE_COMPONENTTYPE_SRC_DIGITAL
Audio line is a digital source (for example, digital output from a DAT or audio CD).

MIXERLINE_COMPONENTTYPE_SRC_LINE
Audio line is a line-level source (for example, line-level input from an external stereo) that can be used as an optional recording source. Because most audio cards for personal computers provide some sort of gain for the recording source line, the mixer device will use the MIXERLINE_COMPONENTTYPE_SRC_AUXILIARY type.

MIXERLINE_COMPONENTTYPE_SRC_MICROPHONE
Audio line is a microphone recording source. Most audio cards for personal computers provide at least two types of recording sources: an auxiliary audio line and microphone input. A microphone audio line typically provides some sort of gain. Audio cards that use a single input for use with a microphone or auxiliary audio line should use the MIXERLINE_COMPONENTTYPE_SRC_MICROPHONE component type.

MIXERLINE_COMPONENTTYPE_SRC_PCSPEAKER
Audio line is a source originating from personal computer speaker. Several audio cards for personal computers provide the ability to mix what would typically be played on the internal speaker with the output of an audio card. Some audio cards support the ability to use this output as a recording source.

MIXERLINE_COMPONENTTYPE_SRC_SYNTHESIZER
Audio line is a source originating from the output of an internal synthesizer. Most audio cards for personal computers provide some sort of MIDI synthesizer (for example, an Adlib®-compatible or OPL/3 FM synthesizer).

MIXERLINE_COMPONENTTYPE_SRC_TELEPHONE
Audio line is a source originating from an incoming telephone line.

MIXERLINE_COMPONENTTYPE_SRC_UNDEFINED
Audio line is a source that cannot be defined by one of the standard component types. A mixer device is required to use this component type for line component types that have not been defined by Microsoft Corporation.

MIXERLINE_COMPONENTTYPE_SRC_WAVEOUT
Audio line is a source originating from the waveform-audio output digital-to-analog converter (DAC). Most audio cards for personal computers provide this component type as a source to the MIXERLINE_COMPONENTTYPE_DST_SPEAKERS destination. Some cards also allow this source to be routed to the MIXERLINE_COMPONENTTYPE_DST_WAVEIN destination.

cChannels
Maximum number of separate channels that can be manipulated independently for the audio line. The minimum value for this field is 1 because a line must have at least one channel. Most modern

audio cards for personal computers are stereo devices; for them, the value of this member is 2.

Channel 1 is assumed to be the left channel; channel 2 is assumed to be the right channel. A multichannel line might have one or more uniform controls (controls that affect all channels of a line uniformly) associated with it.

cConnections

Number of connections that are associated with the audio line. This member is used only for audio destination lines and specifies the number of audio source lines that are associated with it. This member is always zero for source lines and for destination lines that do not have any audio source lines associated with them.

cControls

Number of controls associated with the audio line. This value can be zero. If no controls are associated with the line, the line is likely to be a source that might be selected in a MIXERCONTROL_CONTROLTYPE_MUX or MIXERCONTROL_CONTROLTYPE_MIXER but allows no manipulation of the signal.

szShortName

Short string that describes the audio mixer line specified in the **dwLineID** member. This description should be appropriate as a concise label for the line.

szName

String that describes the audio mixer line specified in the **dwLineID** member. This description should be appropriate as a complete description for the line.

Target

Target media information.

dwType

Target media device type associated with the audio line described in the **MIXERLINE** structure. An application must ignore target information for media device types it does not use. The following values are defined:

MIXERLINE_TARGETTYPE_AUX

The audio line described by the **MIXERLINE** structure is strictly bound to the auxiliary device detailed in the remaining members of the **Target** structure member of the **MIXERLINE** structure.

MIXERLINE_TARGETTYPE_MIDIIN

The audio line described by the **MIXERLINE** structure is strictly bound to the MIDI input device detailed in the remaining members of the **Target** structure member of the **MIXERLINE** structure.

MIXERLINE_TARGETTYPE_MIDIOUT

The audio line described by the **MIXERLINE** structure is strictly bound to the MIDI output device detailed in the remaining members of the **Target** structure member of the **MIXERLINE** structure.

MIXERLINE_TARGETTYPE_UNDEFINED

The audio line described by the **MIXERLINE** structure is not strictly bound to a defined media type. All remaining **Target** structure members of the **MIXERLINE** structure should be ignored. An application cannot use the MIXERLINE_TARGETTYPE_UNDEFINED target type when calling the **mixerGetLineInfo** function with the MIXER_GETLINEINFOF_TARGETTYPE flag.

MIXERLINE_TARGETTYPE_WAVEIN

The audio line described by the **MIXERLINE** structure is strictly bound to the waveform-audio input device detailed in the remaining members of the **Target** structure member of the **MIXERLINE** structure.

MIXERLINE_TARGETTYPE_WAVEOUT

The audio line described by the **MIXERLINE** structure is strictly bound to the waveform-audio output device detailed in the remaining members of the **Target** structure member of the **MIXERLINE** structure.

dwDeviceID

Current device identifier of the target media device when the **dwType** member is a target type other than MIXERLINE_TARGETTYPE_UNDEFINED. This identifier is identical to the current media

device index of the associated media device. When calling the **mixerGetLineInfo** function with the MIXER_GETLINEINFOF_TARGETTYPE flag, this member is ignored on input and will be returned to the caller by the audio mixer manager.

wMid

Manufacturer identifier of the target media device when the **dwType** member is a target type other than MIXERLINE_TARGETTYPE_UNDEFINED. This identifier is identical to the **wMid** member of the device-capabilities structure for the associated media.

wPid

Product identifier of the target media device when the **dwType** member is a target type other than MIXERLINE_TARGETTYPE_UNDEFINED. This identifier is identical to the **wPid** member of the device-capabilities structure for the associated media.

vDriverVersion

Driver version of the target media device when the **dwType** member is a target type other than MIXERLINE_TARGETTYPE_UNDEFINED. This version is identical to the **vDriverVersion** member of the device-capabilities structure for the associated media.

szPname

Product name of the target media device when the **dwType** member is a target type other than MIXERLINE_TARGETTYPE_UNDEFINED. This name is identical to the **szPname** member of the device-capabilities structure for the associated media.

MIXERLINECONTROLS

```
typedef struct {
    DWORD cbStruct;
    DWORD dwLineID;
    union {
        DWORD dwControlID;
        DWORD dwControlType;
    };
    DWORD          cControls;
    DWORD          cbmxctrl;
    LPMIXERCONTROL pamxctrl;
} MIXERLINECONTROLS;
```

Contains information about the controls of an audio line.

cbStruct
Size, in bytes, of the **MIXERLINECONTROLS** structure. This member must be initialized before calling the **mixerGetLineControls** function. The size specified in this member must be large enough to contain the **MIXERLINECONTROLS** structure. When **mixerGetLineControls** returns, this member contains the actual size of the information returned. The returned information will not exceed the requested size, nor will it be smaller than the **MIXERLINECONTROLS** structure.

dwLineID
Line identifier for which controls are being queried. This member is not used if the MIXER_GETLINECONTROLSF_ONEBYID flag is specified for the **mixerGetLineControls** function, but the mixer device still returns this member in this case. The **dwControlID** and **dwControlType** members are not used when MIXER_GETLINECONTROLSF_ALL is specified.

dwControlID
Control identifier of the desired control. This member is used with the MIXER_GETLINECONTROLSF_ONEBYID flag for the **mixerGetLineControls** function to retrieve the control information of the specified control. Note that the **dwLineID** member of the **MIXERLINECONTROLS** structure will be returned by the mixer device and is not required as an input parameter. This member overlaps with the **dwControlType** member and cannot be used in conjunction with the MIXER_GETLINECONTROLSF_ONEBYTYPE query type.

dwControlType
Class of the desired control. This member is used with the MIXER_GETLINECONTROLSF_ONEBYTYPE flag for the **mixerGetLineControls** function to retrieve the first control of the specified class on the line specified by the **dwLineID** member of the **MIXERLINECONTROLS** structure. This member overlaps with the **dwControlID** member and cannot be used in conjunction with the MIXER_GETLINECONTROLSF_ONEBYID query type.

cControls
Number of **MIXERCONTROL** structure elements to retrieve. This member must be initialized by the application before calling the **mixerGetLineControls** function. This member can be 1 only if MIXER_GETLINECONTROLSF_ONEBYID or MIXER_GETLINECONTROLSF_ONEBYTYPE is specified or the value returned in the **cControls** member of the **MIXERLINE** structure returned for an audio line. This member cannot be zero. If an audio line specifies that it has no controls, **mixerGetLineControls** should not be called.

cbmxctrl
Size, in bytes, of a single **MIXERCONTROL** structure. The size specified in this member must be at least large enough to contain the base **MIXERCONTROL** structure. The total size, in bytes, required for the buffer pointed to by the **pamxctrl** member is the product of the **cbmxctrl** and **cControls** members of the **MIXERLINECONTROLS** structure.

pamxctrl

Address of one or more **MIXERCONTROL** structures to receive the properties of the requested audio line controls. This member cannot be NULL and must be initialized before calling the **mixerGetLineControls** function. Each element of the array of controls must be at least large enough to contain a base **MIXERCONTROL** structure. The **cbmxctrl** member must specify the size, in bytes, of each element in this array. No initialization of the buffer pointed to by this member is required by the application. All members are filled in by the mixer device (including the **cbStruct** member of each **MIXERCONTROL** structure) upon returning successfully.

MM_MIXM_CONTROL_CHANGE

```
MM_MIXM_CONTROL_CHANGE
wParam = (WPARAM) hMixer
lParam = (LPARAM) dwControlID
```

Sent by a mixer device to notify an application that the state of a control associated with an audio line has changed. The application should refresh its display and cached values for the specified control.

hMixer
Handle of the mixer device (**HMIXER**) that sent the notification message.

dwControlID
Control identifier for the mixer control that has changed state. This identifier is the same as the **dwControlID** member of the **MIXERCONTROL** structure returned by the **mixerGetLineControls** function.

An application must open a mixer device and specify a callback window to receive the MM_MIXM_CONTROL_CHANGE message.

MM_MIXM_LINE_CHANGE

```
MM_MIXM_LINE_CHANGE
wParam = (WPARAM) hMixer
lParam = (LPARAM) dwLineID
```

Sent by a mixer device to notify an application that the state of an audio line on the specified device has changed. The application should refresh its display and cached values for the specified audio line.

hMixer
Handle of the mixer device that sent the notification message.

dwLineID
Line identifier for the audio line that has changed state. This identifier is the same as the **dwLineID** member of the **MIXERLINE** structure returned by the **mixerGetLineInfo** function.

An application must open a mixer device and specify a callback window to receive the MM_MIXM_LINE_CHANGE message.

Waveform Audio

This chapter explains how to use the waveform and auxiliary audio services of the Microsoft Win32 application programming interface (API) to add sound to applications.

Adding sound to your application can make it more efficient and more fun to use. You can improve your users' efficiency by using sounds to get their attention at critical points, to help them avoid mistakes, or to let them know that a time-consuming operation has finished. You can help them have more fun by adding music or sound effects.

This chapter explains how to do the following things with sound:

- Play waveform audio.
- Use low-level audio services.
- Record waveform audio.
- Use auxiliary audio devices.
- Use audio clipboard formats.

This chapter documents several methods for adding sound to your application. The simplest method documented here is using the **PlaySound** function. Most of the other waveform-audio API elements documented in this chapter are relatively low-level, however. Part Two of this volume, "Media Control Interface," documents a mid-level interface to multimedia programming that offers a simpler and faster method of adding sound to your application than using the low-level sound API.

The PlaySound Function

You can use the **PlaySound** function to play waveform audio, as long as the sound fits into available memory. (The **sndPlaySound** function offers a subset of the capabilities of **PlaySound**. To maximize the portability of your Win32 application, use **PlaySound**, not **sndPlaySound**.)

PlaySound allows you to specify a sound in one of three ways:

- As a system alert, using the alias stored in the WIN.INI file or the registry
- As a filename
- As a resource identifier

One **PlaySound** capability allows you to play a sound in a continuous loop, ending only when you call **PlaySound** again, specifying either NULL or the sound identifier of another sound for the *pszSound* parameter.

You can use **PlaySound** to play the sound synchronously or asynchronously, and to control the behavior of the function in other ways when it must share system resources.

For examples of how to use **PlaySound** in your Win32 applications, see "Playing WAVE Resources" later in this chapter.

Waveform-Audio Files

In the Microsoft Windows operating system, most waveform-audio files use the .WAV filename extension.

The following statement plays the C:\SOUNDS\BELLS.WAV file:

```
PlaySound("C:\\SOUNDS\\BELLS.WAV", NULL, SND_SYNC);
```

If the specified file does not exist, or if the file does not fit into the available memory, **PlaySound** plays the default system sound. If no default system sound has been defined, **PlaySound** fails without producing any sound. If you do not want the default system sound to play, specify the SND_NODEFAULT flag, as shown in the following example:

```
PlaySound("C:\\SOUNDS\\BELLS.WAV", NULL, SND_SYNC | SND_NODEFAULT);
```

Looping Sounds

If you specify the SND_LOOP and SND_ASYNC flags for the *fdwSound* parameter of the **PlaySound** function, the sound will continue to play repeatedly as shown in the following example:

```
PlaySound("C:\\SOUNDS\\BELLS.WAV", NULL, SND_LOOP | SND_ASYNC);
```

If you want to loop a sound, you must play it asynchronously; you cannot use the SND_SYNC flag with the SND_LOOP flag. A looped sound will continue to play until **PlaySound** is called to play another sound. To stop playing a sound (looped or asynchronous) without playing another sound, use the following statement:

```
PlaySound(NULL, NULL, 0);
```

Playing Sounds Specified in the Registry

The **PlaySound** function will also play sounds referred to by a keyname in the registry. This allows users to assign their own sounds to system alerts and warnings, or to user actions, such as a mouse button click. Sounds that are associated with system alerts and warnings are called *sound events*.

To play a sound event, call **PlaySound** with the *pszSound* parameter pointing to a string containing the name of the registry entry that identifies the sound. For example, to play the sound associated with the "MouseClick" entry and to wait for the sound to complete before returning, use the following statement:

```
PlaySound("MouseClick", NULL, SND_SYNC);
```

If the specified registry entry or the waveform-audio file it identifies does not exist, or if the file does not fit into the available memory, **PlaySound** plays the default system sound.

The sound events that are predefined by the system can vary with the Win32 platform. The following list gives the sound events that are defined for all Win32 implementations:

SystemAsterisk

SystemExclamation

SystemExit

SystemHand

SystemQuestion

SystemStart

If an application registers its own sound events, the user can configure the sound event by using the standard Control Panel interface. The application should register the sound event by using the standard registry functions; for more information about the registry, see Chapter 52, "Registry." The entries belong at the same position in the registry hierarchy as the rest of the sound events. This position varies with the Win32 implementation. The appropriate data value also varies with the implementation.

The **sndPlaySound** function always searches the registry for a keyname matching *lpszSound* before attempting to load a file with this name. **PlaySound** accepts flags that specify the location of the sound.

Low-Level Audio Interface

This section documents the low-level audio interface, which is used by applications that need the finest possible control over audio devices. The functions and structures of this interface are named with the prefix "wave".

Devices and Data Types

This section discusses working with waveform-audio devices, such as how to open, close and query them for their capabilities. It also describes how to keep track of the devices in a system by using device handles and device identifiers.

Opening Waveform-Audio Output Devices

Use the **waveOutOpen** function to open a waveform-audio output device for playback. This function opens the device associated with the given device identifier and returns a handle of the open device by writing the handle of a specified memory location.

Some multimedia computers have multiple waveform-audio output devices. Unless you know you want to open a specific waveform-audio output device in a system, you should use the WAVE_MAPPER flag for the device identifier when you open a device. The **waveOutOpen** function chooses the device in the system that is best able to play the given data format.

Querying Audio Devices

Windows provides the following functions to determine how many devices of a certain type are available in a given system.

Function	Description
auxGetNumDevs	Retrieves the number of auxiliary output devices present in the system.
waveInGetNumDevs	Retrieves the number of waveform-audio input devices present in the system.
waveOutGetNumDevs	Retrieves the number of waveform-audio output devices present in the system.

Audio devices are identified by a device identifier. The device identifier is determined implicitly from the number of devices present in a given system. Device identifiers range from zero to one less than the number of devices present. For example, if there are two waveform-audio output devices in a system, valid device identifiers are 0 and 1.

After you determine how many devices of a certain type are present in a system, you can use one of the following functions to query the capabilities of each device.

Function	Description
auxGetDevCaps	Retrieves the capabilities of a given auxiliary output device.
waveInGetDevCaps	Retrieves the capabilities of a given waveform-audio input device.
waveOutGetDevCaps	Retrieves the capabilities of a given waveform-audio output device.

Each of these functions fills a structure with information about the capabilities of a specified device. The following table lists the structures that correspond to each of these functions.

Function	Structure
auxGetDevCaps	AUXCAPS
waveInGetDevCaps	WAVEINCAPS
waveOutGetDevCaps	WAVEOUTCAPS

Waveform-audio devices can support nonstandard formats. (Standard formats are listed in the

dwFormats member of the **WAVEOUTCAPS** structure, later in this chapter.) To determine whether a particular format (standard or nonstandard) is supported by a device, you can call the **waveOutOpen** function with the WAVE_FORMAT_QUERY flag. This flag does not open the device. You specify the format in question in the **WAVEFORMATEX** structure pointed to by the *pwfx* parameter passed to **waveOutOpen**. For information about setting up this structure, see "Devices and Data Types" earlier in this chapter.

Waveform-audio output devices vary in the capabilities they support. The **dwSupport** member of the **WAVEOUTCAPS** structure indicates whether a given device supports such capabilities as volume and pitch changes.

Device Handles and Device Identifiers

Each function that opens an audio device specifies a device identifier, a pointer to a memory location, and some parameters that are unique to each type of device. The memory location is filled with a device handle. Use this device handle to identify the open audio device when calling other audio functions.

The difference between identifiers and handles for audio devices is subtle but important:

- Device identifiers are determined implicitly from the number of devices present in a system. This number is obtained by using the **auxGetNumDevs**, **waveInGetNumDevs**, or **waveOutGetNumDevs** function.
- Device handles are returned when device drivers are opened by using the **waveInOpen** or **waveOutOpen** function.

There are no functions for opening and closing auxiliary audio devices. Auxiliary audio devices need not be opened and closed like waveform-audio devices because there is no continuous data transfer associated with them. All auxiliary audio functions use device identifiers to identify devices.

Waveform-Audio Output Data Types

The following data types are defined for waveform-audio output functions.

Type	Description
HWAVEOUT	Handle of an open waveform-audio output device.
WAVEFORMATEX	Structure that specifies the data formats supported by a particular waveform-audio input device. This structure is used also for waveform-audio input devices.
WAVEHDR	Structure used as a header for a block of waveform-audio input data. This structure is also used for waveform-audio input devices.
WAVEOUTCAPS	Structure used to query the capabilities of a particular waveform-audio output device.

Specifying Waveform-Audio Data Formats

When you call the **waveOutOpen** function to open a device driver for playback or to query whether the driver supports a particular data format, use the *pwfx* parameter to specify a pointer to a **WAVEFORMATEX** structure containing the requested waveform-audio data format. The **WAVEFORMATEX** structure is an extended version of the **WAVEFORMAT** structure. It contains all the members of **WAVEFORMAT**, and adds two more: a **wBitsPerSample** member, which contains extra information required for the PCM (Pulse Code Modulation) format, and a **cbSize** member at the end. You can append data to the structure following **cbSize** as long as you fill **cbSize** with the size of the data. You can use the **WAVEFORMATEX** structure to describe PCM data, although you could also use the **PCMWAVEFORMAT** structure. When the waveform-audio format type is not PCM, you must use **WAVEFORMATEX** instead of **WAVEFORMAT**.

The outmoded **WAVEFORMAT** structure does not contain all the information required to describe the PCM format. The **PCMWAVEFORMAT** structure includes a **WAVEFORMAT** structure along with an additional member containing PCM-specific information. The **PCMWAVEFORMAT** structure has also been superseded by the **WAVEFORMATEX** structure.

There are also two clipboard formats you can use to represent audio data: CF_WAVE and CF_RIFF. Use the CF_WAVE format to represent data in one of the standard formats, such as 11 kHz or 22 kHz PCM. Use the CF_RIFF format to represent more complex data formats that cannot be represented as standard waveform-audio files.

Writing Waveform-Audio Data

After successfully opening a waveform-audio output device driver, you can begin playing a sound. Windows provides the **waveOutWrite** function for sending data blocks to waveform-audio output devices.

Use the **WAVEHDR** structure to specify the waveform-audio data block you are sending using **waveOutWrite**. This structure contains a pointer to a locked data block, the length of the data block, and some flags. This data block must be prepared before you use it; for information about preparing a data block, see "Audio Data Blocks" later in this chapter.

After you send a data block to an output device by using **waveOutWrite**, you must wait until the device driver is finished with the data block before freeing it. If you are sending multiple data blocks, you must monitor the completion of data blocks to know when to send additional blocks. For more information about data blocks, see "Audio Data Blocks" later in this chapter.

PCM Waveform-Audio Data Format

The **lpData** member of the **WAVEHDR** structure points to the waveform-audio data samples. For 8-bit PCM data, each sample is represented by a single unsigned data byte. For 16-bit PCM data, each sample is represented by a 16-bit signed value. The following table summarizes the maximum, minimum, and midpoint values for PCM waveform-audio data.

Data format	Maximum value	Minimum value	Midpoint value
8-bit PCM	255 (0xFF)	0	128 (0x80)
16-bit PCM	32,767 (0x7FFF)	- 32,768 (0x8000)	0

PCM Data Packing

The order of the data bytes varies between 8-bit and 16-bit formats and between mono and stereo formats. The following list describes data packing for the different PCM waveform-audio data formats.

PCM waveform-audio format	Description
8-bit mono	Each sample is 1 byte that corresponds to a single audio channel. Sample 1 is followed by samples 2, 3, 4, and so on.
8-bit stereo	Each sample is 2 bytes. Sample 1 is followed by samples 2, 3, 4, and so on. For each sample, the first byte is channel 0 (the left channel) and the second byte is channel 1 (the right channel).
16-bit mono	Each sample is 2 bytes. Sample 1 is followed by samples 2, 3, 4, and so on. For each sample, the first byte is the low-order byte of channel 0 and the second byte is the high-order byte of channel 0.

16-bit stereo	Each sample is 4 bytes. Sample 1 is followed by samples 2, 3, 4, and so on. For each sample, the first byte is the low-order byte of channel 0 (left channel); the second byte is the high-order byte of channel 0; the third byte is the low-order byte of channel 1 (right channel); and the fourth byte is the high-order byte of channel 1.

Closing Waveform-Audio Output Devices

After waveform-audio playback is complete, call **waveOutClose** to close the output device. If **waveOutClose** is called while a waveform-audio file is playing, the close operation fails and the function returns an error code indicating that the device was not closed. If you do not want to wait for playback to end before closing the device, call the **waveOutReset** function before closing. This ends playback and allows the device to be closed. Be sure to use the **waveOutUnprepareHeader** function to clean up the preparation on all data blocks before closing the device.

Playing Waveform-Audio Files

It's easy to play sounds in your application by using the functions, macros, and messages discussed in this chapter. The techniques and API elements documented here operate only on waveform audio; that is, digitized representations of a sound's physical shape. If you want to add music to your application, and you do not care about other kinds of sounds, you might want to use MIDI. For a discussion of a simple playback MIDI implementation, see Chapter 2, "Getting Started Using MCIWnd." For a discussion of the low-level MIDI interface, see Chapter 13, "Musical Instrument Digital Interface (MIDI)."

You can use the following functions to play waveform audio in your application in a single function call:

Function	Description
MessageBeep	Plays the sound that corresponds to a given system-alert level.
sndPlaySound	Plays the sound that corresponds to the system sound entered in the registry or the contents of the given filename.
PlaySound	Provides all of the functionality of **sndPlaySound** and can directly access resources.

The **MessageBeep** function is a standard part of the Win32 API; because its capabilities are very limited and it is documented elsewhere, it is not discussed here.

The functions in this list provide the following methods of playing waveform audio:

- Playing waveform-audio files associated with system-alert levels
- Playing waveform-audio files specified by entries in the registry
- Playing in-memory WAVE resources
- Playing waveform-audio files stored on a hard disk or compact disc - read-only memory (CD-ROM)

The **sndPlaySound** and **PlaySound** functions load an entire waveform-audio file into memory and, in effect, limit the size of file they can play. Use **sndPlaySound** and **PlaySound** to play waveform-audio files that are relatively small − up to about 100K. These two functions also require the sound data to be in a format that is playable by one of the installed waveform-audio drivers, including the wave mapper.

For larger sound files, use the Media Control Interface (MCI) services or the low-level audio API. For information on using MCI, see Chapter 3, "MCI Overview."

Using Window Messages to Manage Waveform-Audio Playback

The following messages can be sent to a window procedure function for managing waveform-audio playback.

Message	Description
MM_WOM_CLOSE	Sent when the device is closed by using the **waveOutClose** function.
MM_WOM_DONE	Sent when the device driver is finished with a data block sent by using the **waveOutWrite** function.
MM_WOM_OPEN	Sent when the device is opened by using the **waveOutOpen** function.

A *wParam* and *lParam* parameter is associated with each of these messages. The *wParam* parameter always specifies a handle of the open waveform-audio device. For the MM_WOM_DONE message, *lParam* specifies a pointer to a **WAVEHDR** structure that identifies the completed data block. The *lParam* parameter is unused for the MM_WOM_CLOSE and MM_WOM_OPEN messages.

The most useful message is probably MM_WOM_DONE. When this message signals that playback of a data block is complete, you can clean up and free the data block. Unless you need to allocate memory or initialize variables, you probably do not need to process the MM_WOM_OPEN and MM_WOM_CLOSE messages.

The callback function for waveform-audio output devices is supplied by the application. For information about this callback function, see the **waveOutProc** function, later in this chapter.

Retrieving the Current Playback Position

You can monitor the current playback position within the file while waveform audio is playing by using the **waveOutGetPosition** function.

For waveform-audio devices, samples are the preferred time format in which to represent the current position. Thus, the current position of a waveform-audio device is specified as the number of samples for one channel from the beginning of the waveform-audio file. To query the current position of a waveform-audio device, set the **wType** member of the **MMTIME** structure to TIME_SAMPLES and pass this structure to **waveOutGetPosition**.

The **MMTIME** structure can represent time in one or more different formats, including milliseconds, samples, SMPTE (Society of Motion Picture and Television Engineers), and MIDI song pointer formats. The **wType** member specifies the format used to represent time. Before calling a function that uses the **MMTIME** structure, you must set **wType** to indicate your requested time format. Be sure to check **wType** after the call to see whether the requested time format is supported. If the requested time format is not supported, the device driver specifies the time in an alternate time format and changes the **wType** member to the selected time format.

For more information about the **MMTIME** structure, see Chapter 17, "Timers."

Stopping, Pausing, and Restarting Playback

You can stop or pause playback while waveform audio is playing. After playback has been paused, you can restart it. Windows provides the following functions for controlling waveform-audio playback.

Function	Description
waveOutPause	Pauses playback on a waveform-audio output device.
waveOutReset	Stops playback on a waveform-audio output device and marks all pending data blocks as done.
waveOutRestart	Resumes playback on a paused waveform-audio output device.

Pausing a waveform-audio device by using **waveOutPause** might not be instantaneous; the driver can finish playing the current block before pausing playback.

Generally, as soon as the first waveform-audio data block is sent by using the **waveOutWrite** function, the waveform-audio device begins playing. If you do not want the sound to start playing immediately, call **waveOutPause** before calling **waveOutWrite**. Then, when you want to begin playing the waveform, call **waveOutRestart**.

You cannot use **waveOutRestart** to restart a device that has been stopped with **waveOutReset**; you must use **waveOutWrite** to send the first data block to resume playback on the device.

Looping Playback

Looping a sound is controlled by the **dwLoops** and **dwFlags** members in the **WAVEHDR** structures passed to the device with the **waveOutWrite** function. Use the WHDR_BEGINLOOP and WHDR_ENDLOOP flags in the **dwFlags** member to specify the beginning and ending data blocks for looping.

To loop a single data block, specify both flags for the same block. To specify the number of loops, use

the **dwLoops** member in the **WAVEHDR** structure for the first block in the loop.

You can call the **waveOutBreakLoop** function to stop a looping sound.

Changing the Volume of Waveform-Audio Playback

Windows provides the following functions to query and set the volume level of waveform-audio output devices.

Function	Description
waveOutGetVolume	Retrieves the current volume level of the specified waveform-audio output device.
waveOutSetVolume	Sets the volume level of the specified waveform-audio output device.

Not all waveform-audio devices support volume changes. Some devices support individual volume control on both the left and right channels. For information about how to determine the volume-control capabilities of waveform-audio devices, see "Devices and Data Types" earlier in this chapter.

Some applications allow the user to control the volume for all audio devices in a system. (Many applications of this type use the audio mixer services; for more information, see Chapter 14, "Audio Mixers.") Unless your application is capable of this kind of master volume control, you should open an audio device before changing its volume. You should also query the volume level before changing it and restore the volume level to its previous level as soon as possible.

Volume is specified in a doubleword value. When the audio format is stereo, the upper 16 bits specify the relative volume of the right channel and the lower 16 bits specify the relative volume of the left channel. For devices that do not support left- and right-channel volume control, the lower 16 bits specify the volume level, and the upper 16 bits are ignored.

Volume-level values range from 0x0 (silence) to 0xFFFF (maximum volume) and are interpreted logarithmically. The perceived volume increase is the same when increasing the volume level from 0x5000 to 0x6000 as it is from 0x4000 to 0x5000.

Changing Pitch and Playback Rate

Some waveform-audio output devices can vary the pitch and the playback rate of waveform-audio data. Not all waveform-audio devices support pitch and playback-rate changes. For information about how to determine whether a particular waveform-audio device supports pitch and playback rate changes, see "Devices and Data Types" earlier in this chapter.

The differences between changing pitch and playback rate are as follows:

- Changing the playback rate is performed by the device driver and does not require specialized hardware. The sample rate is not changed, but the driver interpolates by skipping or synthesizing samples. For example, if the playback rate is changed by a factor of two, the driver skips every other sample.
- Changing the pitch requires specialized hardware. The playback rate and sample rate are not changed.

Windows provides the following functions to query and set waveform-audio pitch and playback rates.

Function	Description
waveOutGetPitch	Retrieves the pitch for the specified waveform-audio output device.
waveOutGetPlaybackRate	Retrieves the playback rate for the specified waveform-audio output device.
waveOutSetPitch	Sets the pitch for the specified waveform-audio output device.

waveOutSetPlaybackRate	Sets the playback rate for the specified waveform-audio output device.

The pitch and playback rates are changed by a factor specified with a fixed-point number packed into a doubleword value. The upper 16 bits specify the integer part of the number; the lower 16 bits specify the fractional part. For example, the value 1.5 is represented as 0x00018000L. The value 0.75 is represented as 0x0000C000L. A value of 1.0 (0x00010000) means the pitch or playback rate is unchanged.

Recording Waveform Audio

If the MCI waveform-audio recording services do not meet the needs of your application, you can handle waveform-audio recording using the low-level waveform-audio services. For more information about MCI, see Chapter 3, "MCI Overview."

Waveform-Audio Input Data Types

The following data types are defined for waveform-audio input functions:

Type	Description
HWAVEIN	Handle of an open waveform-audio input device.
WAVEFORMATEX	Structure that specifies the data formats supported by a particular waveform-audio input device. This structure is also used for waveform-audio output devices.
WAVEHDR	Structure used as a header for a block of waveform-audio input data. This structure is also used for waveform-audio output devices.
WAVEINCAPS	Structure used to inquire about the capabilities of a particular waveform-audio input device.

Querying Waveform-Audio Input Devices

Before recording waveform audio, you should call the **waveInGetDevCaps** function to determine the waveform-audio input capabilities of the system. This function fills a **WAVEINCAPS** structure with information about the capabilities of a given device. This information includes the manufacturer and product identifiers, a product name for the device, and the version number of the device driver. In addition, the **WAVEINCAPS** structure provides information about the standard waveform-audio formats that the device supports.

Opening Waveform-Audio Input Devices

Use the **waveInOpen** function to open a waveform-audio input device for recording. This function opens the device associated with the given device identifier and returns a handle of the open device by writing the handle of a specified memory location.

Some multimedia computers have multiple waveform-audio input devices. Unless you know you want to open a specific waveform-audio input device in a system, you should use the WAVE_MAPPER constant for the device identifier when you open a device. The **waveInOpen** function will choose the device in the system best able to record in the given data format.

Managing Waveform-Audio Recording

After you open a waveform-audio input device, you can begin recording waveform-audio data. Waveform-audio data is recorded into application-supplied buffers specified by a **WAVEHDR** structure. These data blocks must be prepared before they are used; for more information, see "Audio Data Blocks" later in this chapter.

Windows provides the following functions to manage waveform-audio recording.

Function	Description
waveInAddBuffer	Sends a buffer to the device driver so it can be filled with recorded waveform-audio data.
waveInReset	Stops waveform-audio recording and marks all pending buffers as done.
waveInStart	Starts waveform-audio recording.

waveInStop Stops waveform-audio recording.

Use the **waveInAddBuffer** function to send buffers to the device driver. As the buffers are filled with recorded waveform-audio data, the application is notified with a window message, callback message, thread message, or event, depending on the flag specified when the device was opened.

Before you begin recording by using **waveInStart**, you should send at least one buffer to the driver, or incoming data might be lost.

Before closing the device using **waveInClose**, call **waveInReset** to mark any pending data blocks as being done.

Using Window Messages to Manage Waveform-Audio Recording

The following messages can be sent to a window procedure function for managing waveform-audio recording.

Message	Description
MM_WIM_CLOSE	Sent when the device is closed by using the **waveInClose** function.
MM_WIM_DATA	Sent when the device driver is finished with a buffer sent by using the **waveInAddBuffer** function.
MM_WIM_OPEN	Sent when the device is opened by using the **waveInOpen** function.

The *lParam* parameter of MM_WIM_DATA specifies a pointer to a **WAVEHDR** structure that identifies the buffer. This buffer might not be completely filled with waveform-audio data; recording can stop before the buffer is filled. Use the **dwBytesRecorded** member of the **WAVEHDR** structure to determine the amount of valid data present in the buffer.

The most useful message is probably MM_WIM_DATA. When your application finishes using the data block sent by the device driver, you can clean up and free the data block. Unless you need to allocate memory or initialize variables, you probably do not need to use the MM_WIM_OPEN and MM_WIM_CLOSE messages.

The callback function for waveform-audio input devices is supplied by the application. For information about this callback function, see the **waveInProc** function, later in this chapter.

Auxiliary Audio Interface

Auxiliary audio devices are audio devices whose output is mixed with the MIDI and waveform-audio output devices in a multimedia computer. An example of an auxiliary audio device is the CD audio output from a CD-ROM drive.

Querying Auxiliary Audio Devices

Not all multimedia systems have auxiliary audio support. You can use the **auxGetNumDevs** function to determine the number of controllable auxiliary devices present in a system.

To get information about a particular auxiliary audio device, use the **auxGetDevCaps** function. This function fills an **AUXCAPS** structure with information about the capabilities of a given device. This information includes the manufacturer and product identifiers, a product name for the device, and the device-driver version number.

Changing the Volume of Auxiliary Audio-Devices

Windows provides the following functions to query and set the volume for auxiliary audio devices.

Function	Description
auxGetVolume	Retrieves the current volume setting of the specified auxiliary output device.
auxSetVolume	Sets the volume of the specified auxiliary output device.

Not all auxiliary audio devices support volume changes. Some devices can support individual volume changes on both the left and the right channels.

Volume is specified in a doubleword value, as with the waveform-audio and MIDI volume-control functions . When the audio format is stereo, the upper 16 bits specify the relative volume of the right channel and the lower 16 bits specify the relative volume of the left channel. For devices that do not support left- and right-channel volume control, the lower 16 bits specify the volume level, and the upper 16 bits are ignored.

Volume-level values range from 0x0 (silence) to 0xFFFF (maximum volume) and are interpreted logarithmically. The perceived volume increase is the same when increasing the volume level from 0x5000 to 0x6000 as it is from 0x4000 to 0x5000.

Audio Data Blocks

The **waveInAddBuffer** and **waveOutWrite** functions require applications to allocate data blocks to pass to the device drivers for recording or playback purposes. Each of these functions uses the **WAVEHDR** structure to describe its data block.

Before using one of these functions to pass a data block to a device driver, you must allocate memory for the data block and the header structure that describes the data block. The headers can be prepared and unprepared by using the following functions.

Function	Description
waveInPrepareHeader	Prepares a waveform-audio input data block.
waveInUnprepareHeader	Cleans up the preparation on a waveform-audio input data block.
waveOutPrepareHeader	Prepares a waveform-audio output data block.
waveOutUnprepareHeader	Cleans up the preparation on a waveform-audio output data block.

Before you pass an audio data block to a device driver, you must prepare the data block by passing it to either **waveInPrepareHeader** or **waveOutPrepareHeader**. When the device driver is finished with the data block and returns it, you must clean up this preparation by passing the data block to either **waveInUnprepareHeader** or **waveOutUnprepareHeader** before any allocated memory can be freed.

Unless the waveform-audio input and output data is small enough to be contained in a single data block, applications must continually supply the device driver with data blocks until playback or recording is complete.

Even if a single data block is used, an application must be able to determine when a device driver is finished with the data block so the application can free the memory associated with the data block and header structure. There are several ways to determine when a device driver is finished with a data block:

- By specifying a callback function to receive a message sent by the driver when it is finished with a data block
- By using an event callback
- By specifying a window or thread to receive a message sent by the driver when it is finished with a data block
- By polling the WHDR_DONE bit in the **dwFlags** member of the **WAVEHDR** structure sent with each data block

If an application does not get a data block to the device driver when needed, there can be an audible gap in playback or a loss of incoming recorded information. This requires at least a double-buffering scheme – staying at least one data block ahead of the device driver.

The following sections describe ways to determine when a device driver is finished with a data block.

Using a Callback Function to Process Driver Messages

You can write your own callback function to process messages sent by the device driver. To use a callback function, specify the CALLBACK_FUNCTION flag in the *fdwOpen* parameter and the address of the callback in the *dwCallback* parameter of the **waveInOpen** or **waveOutOpen** function.

Messages sent to a callback function are similar to messages sent to a window, except they have two **DWORD** parameters instead of a **UINT** and a **DWORD** parameter. For details on these messages, see "Playing Waveform-Audio Files" earlier in this chapter.

To pass instance data from an application to a callback function, use one of the following techniques:

- Pass the instance data using the *dwInstance* parameter of the function that opens the device driver.
- Pass the instance data using the **dwUser** member of the **WAVEHDR** structure that identifies an audio data block being sent to a device driver.

If you need more than 32 bits of instance data, pass a pointer to a structure containing the additional information.

Using an Event Callback to Process Driver Messages

To use an event callback, use the **CreateEvent** function to retrieve the handle of an event. In the call to the **waveOutOpen** function, specify CALLBACK_EVENT for the *fdwOpen* parameter. After calling the **waveOutPrepareHeader** function but before sending waveform-audio data to the device, create a nonsignalled event by calling the **ResetEvent** function, specifying the event handle retrieved by **CreateEvent**. Then, inside a loop that checks whether the WHDR_DONE bit is set in the **dwFlags** member of the **WAVEHDR** structure, call the **WaitForSingleObject** function, specifying as parameters the event handle and a time-out value of INFINITE.

An event callback is set by anything that might cause a function callback.

Because event callbacks do not receive specific close, done, or open notifications, an application might have to check the status of the process it is waiting for after the event occurs. It is possible that a number of tasks could have been completed by the time **WaitForSingleObject** returns.

Using a Window or Thread to Process Driver Messages

To use a window callback function, specify the CALLBACK_WINDOW flag in the *fdwOpen* parameter and a window handle in the low-order word of the *dwCallback* parameter of the **waveInOpen** or **waveOutOpen** function. Driver messages will be sent to the window procedure for the window identified by the handle in *dwCallback*.

Similarly, to use a thread callback, specify CALLBACK_THREAD and a thread handle in the call to **waveInOpen** or **waveOutOpen**. In this case, messages are posted to the specified thread instead of to a window.

Messages sent to the window or thread callback are specific to the audio device type used. For more information about these messages, see "Playing Waveform-Audio Files" earlier in this chapter.

Managing Data Blocks by Polling

In addition to using a callback function, you can poll the **dwFlags** member of a **WAVEHDR** structure to determine when an audio device is finished with a data block. Sometimes it is better to poll **dwFlags** than to wait for another mechanism to receive messages from the drivers. For example, after you call the **waveOutReset** function to release pending data blocks, you can immediately poll to be sure that the data blocks are done before calling **waveOutUnprepareHeader** and freeing the memory for the data block.

You can use the WHDR_DONE flag to test the **dwFlags** member. As soon as the WHDR_DONE flag is set in the **dwFlags** member of the **WAVEHDR** structure, the driver is finished with the data block.

Handling Errors with Audio Functions

The waveform-audio and auxiliary-audio functions return a nonzero value when an error occurs. Windows provides functions that convert these error values into textual descriptions of the errors. The application must still examine the error values to determine how to proceed, but textual descriptions of errors can be used in dialog boxes that describe errors to users.

You can use the following functions to retrieve textual descriptions of audio error values:

Function	Description
waveInGetErrorText	Retrieves a textual description of a specified waveform-audio input error.
waveOutGetErrorText	Retrieves a textual description of a specified waveform-audio output error.

The only audio functions that do not return error values are **auxGetNumDevs**, **waveInGetNumDevs**, and **waveOutGetNumDevs**. These functions return zero if no devices are present in a system or if they encounter any errors.

Using Waveform and Auxiliary Audio

This section demonstrates implementing waveform and auxiliary audio in your application. The following topics are discussed:

- Playing WAVE resources
- Determining support for nonstandard data formats
- Processing the MM_WOM_DONE message

Playing WAVE Resources

You can use the **PlaySound** function to play a sound that is stored as a resource. Although this is also possible using the **sndPlaySound** function, **sndPlaySound** requires you to find, load, lock, unlock, and free the resource; **PlaySound** achieves all of this with a single line of code.

```
PlaySound("SoundName", hInst, SND_RESOURCE | SND_ASYNC);
```

Determining Nonstandard Format Support

To see whether a device supports a particular format (standard or nonstandard), you can call the **waveOutOpen** function with the WAVE_FORMAT_QUERY flag. The following example uses this technique to determine whether a given waveform-audio device supports a given format.

```
// Determines whether the given waveform-audio output device supports a
// given waveform-audio format. Returns MMSYSERR_NOERROR if the format
// is supported, WAVEERR_BADFORMAT if the format is not supported, and
// one of the other MMSYSERR_ error codes if there are other errors
// encountered in opening the given waveform-audio device.

MMRESULT IsFormatSupported(LPWAVEFORMATEX pwfx, UINT uDeviceID) {
    return (waveOutOpen(
        NULL,                 // ptr can be NULL for query
        uDeviceID,            // the device identifier
        pwfx,                 // defines requested format
        NULL,                 // no callback
        NULL,                 // no instance data
        WAVE_FORMAT_QUERY));  // query only, do not open device
}
```

This technique for determining nonstandard format support also applies to waveform-audio input devices. The only difference is that the **waveInOpen** function is used in place of **waveOutOpen** to query for format support.

To determine whether a particular waveform-audio data format is supported by any of the waveform-audio devices in a system, use the technique illustrated in the previous example, but specify the WAVE_MAPPER constant for the *uDeviceID* parameter.

Processing the MM_WOM_DONE Message

The following example shows how to process the MM_WOM_DONE message. This example assumes the application does not play multiple data blocks, so it can close the output device after playing a single data block.

```
// WndProc--Main window procedure.
LRESULT FAR PASCAL WndProc(HWND hWnd, UINT msg, WPARAM wParam,
    LPARAM lParam)
{
    switch (msg)
    {
    .
    .
    .

case MM_WOM_DONE:

    // A waveform-audio data block has been played and
    // can now be freed.
    waveOutUnprepareHeader((HWAVEOUT) wParam,
        (LPWAVEHDR) lParam, sizeof(WAVEHDR) );
    .
    . // Free hData memory.
    .
    waveOutClose((HWAVEOUT) wParam);
    break;
    }
    return DefWindowProc(hWnd, msg, wParam, lParam);
}
```

Waveform Audio Reference

This section describes the functions, messages, and structures associated with waveform audio. These elements are grouped as follows.

Auxiliary Devices

AUXCAPS
auxGetDevCaps
auxGetNumDevs
auxGetVolume
auxOutMessage
auxSetVolume

Easy Playback

PlaySound
sndPlaySound

Errors

waveInGetErrorText
waveOutGetErrorText

Opening and Closing

PCMWAVEFORMAT
MM_WIM_CLOSE
MM_WIM_OPEN
MM_WOM_CLOSE
MM_WOM_OPEN
WAVEFORMAT
WAVEFORMATEX
waveInClose
waveInProc
waveInOpen
waveOutClose
waveOutProc
waveOutOpen
WIM_CLOSE
WIM_OPEN
WOM_CLOSE
WOM_OPEN

Pitch

waveOutGetPitch
waveOutSetPitch

Playback Rate

waveOutGetPlaybackRate
waveOutSetPlaybackRate

Playback Progress

MM_WOM_DONE
waveOutBreakLoop
waveOutPause
waveOutReset
waveOutRestart
WOM_DONE

Playing

MM_WOM_DONE
WAVEHDR
waveOutPrepareHeader
waveOutUnprepareHeader
waveOutWrite
WOM_DONE

Querying a Device

WAVEINCAPS
waveInGetDevCaps
waveInGetNumDevs
WAVEOUTCAPS
waveOutGetDevCaps
waveOutGetNumDevs

Recording

MM_WIM_DATA
waveInAddBuffer
waveInPrepareHeader
waveInReset
waveInStart
waveInStop
waveInUnprepareHeader
WIM_DATA

Retrieving Device Identifiers

waveInGetID
waveOutGetID

Retrieving the Current Position

waveInGetPosition
waveOutGetPosition

Sending Custom Messages

waveInMessage
waveOutMessage

Volume

waveOutGetVolume
waveOutSetVolume

auxGetDevCaps

```
MMRESULT auxGetDevCaps(UINT uDeviceID, LPAUXCAPS lpCaps, UINT cbCaps);
```

Retrieves the capabilities of a given auxiliary output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

  MMSYSERR_BADDEVICEID Specified device identifier is out of range.

uDeviceID
Identifier of the auxiliary output device to be queried. Specify a valid device identifier (see the following comments section), or use the following constant:

AUX_MAPPER
Auxiliary audio mapper. The function returns an error if no auxiliary audio mapper is installed.

lpCaps
Address of an **AUXCAPS** structure to be filled with information about the capabilities of the device.

cbCaps
Size, in bytes, of the **AUXCAPS** structure.

The device identifier in *uDeviceID* varies from zero to one less than the number of devices present. AUX_MAPPER may also be used. Use the **auxGetNumDevs** function to determine the number of auxiliary output devices present in the system.

auxGetNumDevs

```
UINT auxGetNumDevs(VOID);
```

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

- Returns the number of devices. A return value of zero means that no devices are present or that an error occurred.

auxGetVolume

```
MMRESULT auxGetVolume(UINT uDeviceID, LPDWORD lpdwVolume);
```

Retrieves the current volume setting of the specified auxiliary output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_BADDEVICEID	Specified device identifier is out of range.

uDeviceID
Identifier of the auxiliary output device to be queried.

lpdwVolume
Address of a variable 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 the **auxSetVolume** function are returned, regardless of whether the device supports the full 16 bits of volume-level control.

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

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

auxOutMessage

```
DWORD auxOutMessage(UINT uDeviceID, UINT uMsg, DWORD dwParam1,
    DWORD dwParam2);
```

Sends a message to the given auxiliary output device. This function also performs error checking on the device identifier passed as part of the message.

- Returns the message return value.

uDeviceID
Identifier of the auxiliary output device to receive the message.

uMsg
Message to send.

dwParam1 and *dwParam2*
Message parameters.

auxSetVolume

```
MMRESULT auxSetVolume(UINT uDeviceID, DWORD dwVolume);
```

Sets the volume of the specified auxiliary output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_BADDEVICEID	Specified device identifier is out of range.

uDeviceID
Identifier of the auxiliary output device to be queried. Device identifiers are determined implicitly from the number of devices present in the system. Device identifier values range from zero to one less than the number of devices present. Use the **auxGetNumDevs** function to determine the number of auxiliary devices in the system.

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.

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

To determine whether the device supports volume control on both the left and right channels, use the AUXCAPS_LRVOLUME flag to test the **dwSupport** member 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 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.

PlaySound

```
BOOL PlaySound(LPCSTR pszSound, HMODULE hmod, DWORD fdwSound);
```

Plays a sound specified by the given filename, resource, or system event. (A system event may be associated with a sound in the registry or in the WIN.INI file.)

- Returns TRUE if successful or FALSE otherwise.

pszSound
A string that specifies the sound to play. If this parameter is NULL, any currently playing waveform sound is stopped. To stop a non-waveform sound, specify SND_PURGE in the *fdwSound* parameter.

Three flags in *fdwSound* (SND_ALIAS, SND_FILENAME, and SND_RESOURCE) determine whether the name is interpreted as an alias for a system event, a filename, or a resource identifier. If none of these flags are specified, **PlaySound** searches the registry or the WIN.INI file for an association with the specified sound name. If an association is found, the sound event is played. If no association is found in the registry, the name is interpreted as a filename.

hmod
Handle of the executable file that contains the resource to be loaded. This parameter must be NULL unless SND_RESOURCE is specified in *fdwSound*.

fdwSound
Flags for playing the sound. The following values are defined:

SND_APPLICATION
The sound is played using an application-specific association.

SND_ALIAS
The *pszSound* parameter is a system-event alias in the registry or the WIN.INI file. Do not use with either SND_FILENAME or SND_RESOURCE.

SND_ALIAS_ID
The *pszSound* parameter is a predefined sound identifier.

SND_ASYNC
The sound is played asynchronously and **PlaySound** returns immediately after beginning the sound. To terminate an asynchronously played waveform sound, call **PlaySound** with *pszSound* set to NULL.

SND_FILENAME
The *pszSound* parameter is a filename.

SND_LOOP
The sound plays repeatedly until **PlaySound** is called again with the *pszSound* parameter set to NULL. You must also specify the SND_ASYNC flag to indicate an asynchronous sound event.

SND_MEMORY
A sound event's file is loaded in RAM. The parameter specified by *pszSound* must point to an image of a sound in memory.

SND_NODEFAULT
No default sound event is used. If the sound cannot be found, **PlaySound** returns silently without playing the default sound.

SND_NOSTOP
The specified sound event will yield to another sound event that is already playing. If a sound cannot be played because the resource needed to generate that sound is busy playing another sound, the function immediately returns FALSE without playing the requested sound.

If this flag is not specified, **PlaySound** attempts to stop the currently playing sound so that the device can be used to play the new sound.

SND_NOWAIT

If the driver is busy, return immediately without playing the sound.

SND_PURGE
: Sounds are to be stopped for the calling task. If *pszSound* is not NULL, all instances of the specified sound are stopped. If *pszSound* is NULL, all sounds that are playing on behalf of the calling task are stopped.

  You must also specify the instance handle to stop SND_RESOURCE events.

SND_RESOURCE
: The *pszSound* parameter is a resource identifier; *hmod* must identify the instance that contains the resource.

SND_SYNC
: Synchronous playback of a sound event. **PlaySound** returns after the sound event completes.

The sound specified by *pszSound* must fit into available physical memory and be playable by an installed waveform-audio device driver. **PlaySound** searches the following directories for sound files: the current directory; the Windows directory; the Windows system directory; directories listed in the PATH environment variable; and the list of directories mapped in a network. For more information about the directory search order, see the documentation for the **OpenFile** function.

If it cannot find the specified sound, **PlaySound** uses the default system event sound entry instead. If the function can find neither the system default entry nor the default sound, it makes no sound and returns FALSE.

sndAlias

```
DWORD sndAlias(ch0, ch1)
```

Creates an alias identifier from two characters, for use with the **PlaySound** function.

- Returns an alias identifier corresponding to the two supplied characters.

ch0 and *ch1*
Characters describing the sound alias.

This macro is defined as follows:

```
sndAlias(ch0, ch1)     (SND_ALIAS_START + (DWORD)(BYTE)(ch0) |
    ((DWORD)(BYTE)(ch1) << 8))
```

sndPlaySound

```
BOOL sndPlaySound(LPCSTR lpszSound, UINT fuSound);
```

Plays a waveform sound specified either by a filename or by an entry in the registry or the WIN.INI file. This function offers a subset of the functionality of the **PlaySound** function; **sndPlaySound** is being maintained for backward compatibility.

- Returns TRUE if successful or FALSE otherwise.

lpszSound
A string that specifies the sound to play. This parameter can be either an entry in the registry or in WIN.INI that identifies a system sound, or it can be the name of a waveform-audio file. (If the function does not find the entry, the parameter is treated as a filename.) If this parameter is NULL, any currently playing sound is stopped.

fuSound
Flags for playing the sound. The following values are defined:

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_LOOP
The sound plays 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_MEMORY
The parameter specified by *lpszSoundName* points to an image of a waveform sound in memory.

SND_NODEFAULT
If the sound cannot be found, the function returns silently without playing the default sound.

SND_NOSTOP
If a sound is currently playing, the function immediately returns FALSE, without playing the requested sound.

SND_SYNC
The sound is played synchronously and the function does not return until the sound ends.

If the specified sound cannot be found, **sndPlaySound** plays the system default sound. If there is no system default entry in the registry or WIN.INI file, or if the default sound cannot be found, the function makes no sound and returns FALSE.

The specified sound must fit in available physical memory and be playable by an installed waveform-audio device driver. If **sndPlaySound** does not find the sound in the current directory, the function searches for it using the standard directory-search order.

waveInAddBuffer

```
MMRESULT waveInAddBuffer(HWAVEIN hwi, LPWAVEHDR pwh, UINT cbwh);
```

Sends an input buffer to the given waveform-audio input device. When the buffer is filled, the application is notified.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
WAVERR_UNPREPARED	The buffer pointed to by the *pwh* parameter hasn't been prepared.

hwi
Handle of the waveform-audio input device.

pwh
Address of a **WAVEHDR** structure that identifies the buffer.

cbwh
Size, in bytes, of the **WAVEHDR** structure.

When the buffer is filled, the WHDR_DONE bit is set in the **dwFlags** member of the **WAVEHDR** structure.

The buffer must be prepared with the **waveInPrepareHeader** function before it is passed to this function.

waveInClose

```
MMRESULT waveInClose(HWAVEIN hwi);
```

Closes the given waveform-audio input device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
WAVERR_STILLPLAYING	There are still buffers in the queue.

hwi
Handle of the waveform-audio input device. If the function succeeds, the handle is no longer valid after this call.

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

waveInGetDevCaps

```
MMRESULT waveInGetDevCaps(UINT uDeviceID, LPWAVEINCAPS pwic,
    UINT cbwic);
```

Retrieves the capabilities of a given waveform-audio input device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error include the following:

MMSYSERR_BADDEVICEID	Specified device identifier is out of range.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

uDeviceID
Identifier of the waveform-audio output device. It can be either a device identifier or a handle of an open waveform-audio input device.

pwic
Address of a **WAVEINCAPS** structure to be filled with information about the capabilities of the device.

cbwic
Size, in bytes, of the **WAVEINCAPS** structure.

Use this function to determine the number of waveform-audio input devices present in the system. If the value specified by the *uDeviceID* parameter is a device identifier, it can vary from zero to one less than the number of devices present. The WAVE_MAPPER constant can also be used as a device identifier. Only *cbwic* bytes (or less) of information is copied to the location pointed to by *pwic*. If *cbwic* is zero, nothing is copied and the function returns zero.

waveInGetErrorText

```
MMRESULT waveInGetErrorText(MMRESULT mmrError, LPSTR pszText,
    UINT cchText);
```

Retrieves a textual description of the error identified by the given error number.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_BADERRNUM	Specified error number is out of range.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

mmrError
Error number.

pszText
Address of the buffer to be filled with the textual error description.

cchText
Size, in characters, of the buffer pointed to by *pszText*.

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

waveInGetID

```
MMRESULT waveInGetID(HWAVEIN hwi, LPUINT puDeviceID);
```

Gets the device identifier for the given waveform-audio input device.

This function is supported for backward compatibility. New applications can cast a handle of the device rather than retrieving the device identifier.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	The *hwi* parameter specifies an invalid handle.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

hwi
Handle of the waveform-audio input device.

puDeviceID
Address of a variable to be filled with the device identifier.

waveInGetNumDevs

```
UINT waveInGetNumDevs(VOID);
```

Returns the number of waveform-audio input devices present in the system.

- Returns the number of devices. A return value of zero means that no devices are present or that an error occurred.

waveInGetPosition

```
MMRESULT waveInGetPosition(HWAVEIN hwi, LPMMTIME pmmt, UINT cbmmt);
```

Retrieves the current input position of the given waveform-audio input device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

hwi
Handle of the waveform-audio input device.

pmmt
Address of an **MMTIME** structure.

cbmmt
Size, in bytes, of the **MMTIME** structure.

Before calling this function, set the **wType** member of the **MMTIME** structure to indicate the time format you want. After calling this function, check **wType** to determine whether the desired time format is supported. If the format is not supported, the member will specify an alternative format.

The position is set to zero when the device is opened or reset.

waveInMessage

```
DWORD waveInMessage(HWAVEIN hwi, UINT uMsg, DWORD dwParam1,
    DWORD dwParam2);
```

Sends messages to the waveform-audio input device drivers.

- Returns the value returned from the driver.

hwi
Handle of the waveform-audio input device.

uMsg
Message to send.

dwParam1 and *dwParam2*
Message parameters.

waveInOpen

```
MMRESULT waveInOpen(LPHWAVEIN phwi, UINT uDeviceID, LPWAVEFORMATEX pwfx,
    DWORD dwCallback, DWORD dwCallbackInstance, DWORD fdwOpen);
```

Opens the given waveform-audio input device for recording.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_ALLOCATED	Specified resource is already allocated.
MMSYSERR_BADDEVICEID	Specified device identifier is out of range.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
WAVERR_BADFORMAT	Attempted to open with an unsupported waveform-audio format.

phwi
Address filled with a handle identifying the open waveform-audio input device. Use this handle to identify the device when calling other waveform-audio input functions. This parameter can be NULL if WAVE_FORMAT_QUERY is specified for *fdwOpen*.

uDeviceID
Identifier of the waveform-audio input device to open. It can be either a device identifier or a handle of an open waveform-audio input device.You can use the following flag instead of a device identifier:

WAVE_MAPPER
The function selects a waveform-audio input device capable of recording in the specified format.

pwfx
Address of a **WAVEFORMATEX** structure that identifies the desired format for recording waveform-audio data. You can free this structure immediately after **waveInOpen** returns.

dwCallback
Address of a fixed callback function, an event handle, or a handle of a window or thread called during waveform-audio recording to process messages related to the progress of recording. If no callback function is required, this value can be zero.

dwCallbackInstance
User-instance data passed to the callback mechanism. This parameter is not used with the window callback mechanism.

fdwOpen
Flags for opening the device. The following values are defined:

CALLBACK_EVENT
The *dwCallback* parameter is an event handle.

CALLBACK_FUNCTION
The *dwCallback* parameter is a callback procedure address.

CALLBACK_NULL
No callback mechanism. This is the default setting.

CALLBACK_THREAD
The *dwCallback* parameter is a thread handle.

CALLBACK_WINDOW
The *dwCallback* parameter is a window handle.

WAVE_FORMAT_QUERY

The function queries the device to determine whether it supports the given format, but it does not open the device.

WAVE_MAPPED
The *uDeviceID* parameter specifies a waveform-audio device to be mapped to by the wave mapper.

Use the **waveInGetNumDevs** function to determine the number of waveform-audio input devices present on the system. The device identifier specified by *uDeviceID* varies from zero to one less than the number of devices present. The WAVE_MAPPER constant can also be used as a device identifier.

If you choose to have a window or thread receive callback information, the following messages are sent to the window procedure or thread to indicate the progress of waveform-audio input: MM_WIM_OPEN, MM_WIM_CLOSE, and MM_WIM_DATA.

If you choose to have a function receive callback information, the following messages are sent to the function to indicate the progress of waveform-audio input: WIM_OPEN, WIM_CLOSE, and WIM_DATA.

waveInPrepareHeader

```
MMRESULT waveInPrepareHeader(HWAVEIN hwi, LPWAVEHDR pwh, UINT cbwh);
```

Prepares a buffer for waveform-audio input.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

hwi
Handle of the waveform-audio input device.

pwh
Address of a **WAVEHDR** structure that identifies the buffer to be prepared.

cbwh
Size, in bytes, of the **WAVEHDR** structure.

The **lpData**, **dwBufferLength**, and **dwFlags** members of the **WAVEHDR** structure must be set before calling this function (**dwFlags** must be zero).

waveInProc

```
void CALLBACK waveInProc(HWAVEIN hwi, UINT uMsg, DWORD dwInstance,
    DWORD dwParam1, DWORD dwParam2);
```

Callback function used with the waveform-audio input device. This function is a placeholder for the application-defined function name.

hwi
Handle of the waveform-audio device associated with the callback function.

uMsg
Waveform-audio input message. It can be one of the following messages:

WIM_CLOSE
Sent when the device is closed using the **waveInClose** function.

WIM_DATA
Sent when the device driver is finished with a data block sent using the **waveInAddBuffer** function.

WIM_OPEN
Sent when the device is opened using the **waveInOpen** function.

dwInstance
User instance data specified with **waveInOpen**.

dwParam1 and *dwParam2*
Message parameters.

Applications should not call any system-defined functions from inside a callback function, except for **PostMessage**, **timeGetSystemTime**, **timeGetTime**, **timeSetEvent**, **timeKillEvent**, **midiOutShortMsg**, **midiOutLongMsg**, and **OutputDebugStr**.

waveInReset

```
MMRESULT waveInReset(HWAVEIN hwi);
```

Stops input on the given waveform-audio input device and resets the current position to zero. All pending buffers are marked as done and returned to the application.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

hwi
Handle of the waveform-audio input device.

waveInStart

```
MMRESULT waveInStart(HWAVEIN hwi);
```

Starts input on the given waveform-audio input device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

hwi
Handle of the waveform-audio input device.

Buffers are returned to the application when full or when the **waveInReset** function is called (the **dwBytesRecorded** member in the header will contain the length of data). If there are no buffers in the queue, the data is thrown away without notifying the application, and input continues.

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

waveInStop

```
MMRESULT waveInStop(HWAVEIN hwi);
```

Stops waveform-audio input.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

hwi
Handle of the waveform-audio input device.

If there are any buffers in the queue, the current buffer will be marked as done (the **dwBytesRecorded** member in the header will contain the 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.

waveInUnprepareHeader

```
MMRESULT waveInUnprepareHeader(HWAVEIN hwi, LPWAVEHDR pwh, UINT cbwh);
```

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

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
WAVERR_STILLPLAYING	The buffer pointed to by the *pwh* parameter is still in the queue.

hwi
Handle of the waveform-audio input device.

pwh
Address of a **WAVEHDR** structure identifying the buffer to be cleaned up.

cbwh
Size, in bytes, of the **WAVEHDR** structure.

This function complements the **waveInPrepareHeader** function.

You must call this function before freeing the buffer. After passing a buffer to the device driver with the **waveInAddBuffer** function, 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.

waveOutBreakLoop

```
MMRESULT waveOutBreakLoop(HWAVEOUT hwo);
```

Breaks a loop on the given waveform-audio output device and allows playback to continue with the next block in the driver list.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

hwo
Handle of the waveform-audio output device.

The blocks making up the loop are played to the end before the loop is terminated.

Calling this function when nothing is playing or looping has no effect, and the function returns zero.

waveOutClose

```
MMRESULT waveOutClose(HWAVEOUT hwo);
```

Closes the given waveform-audio output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
WAVERR_STILLPLAYING	There are still buffers in the queue.

hwo
Handle of the waveform-audio output device. If the function succeeds, the handle is no longer valid after this call.

If the device is still playing a waveform-audio file, the close operation fails. Use the **waveOutReset** function to terminate playback before calling **waveOutClose**.

waveOutGetDevCaps

```
MMRESULT waveOutGetDevCaps(UINT uDeviceID, LPWAVEOUTCAPS pwoc,
    UINT cbwoc);
```

Retrieves the capabilities of a given waveform-audio output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_BADDEVICEID	Specified device identifier is out of range.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

uDeviceID
Identifier of the waveform-audio output device. It can be either a device identifier or a handle of an open waveform-audio output device.

pwoc
Address of a **WAVEOUTCAPS** structure to be filled with information about the capabilities of the device.

cbwoc
Size, in bytes, of the **WAVEOUTCAPS** structure.

Use the **waveOutGetNumDevs** function to determine the number of waveform-audio output devices present in the system. If the value specified by the *uDeviceID* parameter is a device identifier, it can vary from zero to one less than the number of devices present. The WAVE_MAPPER constant can also be used as a device identifier. Only *cbwoc* bytes (or less) of information is copied to the location pointed to by *pwoc*. If *cbwoc* is zero, nothing is copied and the function returns zero.

waveOutGetErrorText

```
MMRESULT waveOutGetErrorText(MMRESULT mmrError, LPSTR pszText,
    UINT cchText);
```

Retrieves a textual description of the error identified by the given error number.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_BADERRNUM	Specified error number is out of range.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

mmrError
Error number.

pszText
Address of a buffer to be filled with the textual error description.

cchText
Size, in characters, of the buffer pointed to by *pszText*.

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

waveOutGetID

```
MMRESULT waveOutGetID(HWAVEOUT hwo, LPUINT puDeviceID);
```

Retrieves the device identifier for the given waveform-audio output device.

This function is supported for backward compatibility. New applications can cast a handle of the device rather than retrieving the device identifier.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	The *hwo* parameter specifies an invalid handle.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

hwo
Handle of the waveform-audio output device.

puDeviceID
Address of a variable to be filled with the device identifier.

waveOutGetNumDevs

```
UINT waveOutGetNumDevs(VOID);
```

Retrieves the number of waveform-audio output devices present in the system.

- Returns the number of devices. A return value of zero means that no devices are present or that an error occurred.

waveOutGetPitch

```
MMRESULT waveOutGetPitch(HWAVEOUT hwo, LPDWORD pdwPitch);
```

Retrieves the current pitch setting for the specified waveform-audio output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
MMSYSERR_NOTSUPPORTED	Function isn't supported.

hwo
Handle of the waveform-audio output device.

pdwPitch
Address of a variable 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 variable contains the signed integer part of the number, and the low-order word contains the fractional part. A value of 0x8000 in the low-order word 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.

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 **dwSupport** member of the **WAVEOUTCAPS** structure (filled by the **waveOutGetDevCaps** function).

waveOutGetPlaybackRate

```
MMRESULT waveOutGetPlaybackRate(HWAVEOUT hwo, LPDWORD pdwRate);
```

Retrieves the current playback rate for the specified waveform-audio output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
MMSYSERR_NOTSUPPORTED	Function isn't supported.

hwo
Handle of the waveform-audio output device.

pdwRate
Address of a variable 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 variable contains the signed integer part of the number, and the low-order word contains the fractional part. A value of 0x8000 in the low-order word 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.

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** member of the **WAVEOUTCAPS** structure (filled by the **waveOutGetDevCaps** function).

waveOutGetPosition

```
MMRESULT waveOutGetPosition(HWAVEOUT hwo, LPMMTIME pmmt, UINT cbmmt);
```

Retrieves the current playback position of the given waveform-audio output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

hwo
Handle of the waveform-audio output device.

pmmt
Address of an **MMTIME** structure.

cbmmt
Size, in bytes, of the **MMTIME** structure.

Before calling this function, set the **wType** member of the **MMTIME** structure to indicate the time format you want. After calling this function, check **wType** to determine whether the time format is supported. If the format is not supported, **wType** will specify an alternative format.

The position is set to zero when the device is opened or reset.

waveOutGetVolume

```
MMRESULT waveOutGetVolume(HWAVEOUT hwo, LPDWORD pdwVolume);
```

Retrieves the current volume level of the specified waveform-audio output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
MMSYSERR_NOTSUPPORTED	Function isn't supported.

hwo
Handle of an open waveform-audio output device.

pdwVolume
Address of a variable 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 mono volume level.

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

Not all devices support volume changes. To determine whether the device supports volume control, use the WAVECAPS_VOLUME flag to test the **dwSupport** member of the **WAVEOUTCAPS** structure (filled by the **waveOutGetDevCaps** function).

To determine whether the device supports left- and right-channel volume control, use the WAVECAPS_LRVOLUME flag to test the **dwSupport** member of the **WAVEOUTCAPS** structure (filled by **waveOutGetDevCaps**).

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.

waveOutMessage

```
DWORD waveOutMessage(HWAVEOUT hwo, UINT uMsg, DWORD dwParam1,
    DWORD dwParam2);
```

Sends messages to the waveform-audio output device drivers.

- Returns the value returned from the driver.

hwo
Handle of the waveform-audio output device.

uMsg
Message to send.

dwParam1 and *dwParam2*
Message parameters.

waveOutOpen

```
MMRESULT waveOutOpen(LPHWAVEOUT phwo, UINT uDeviceID,
    LPWAVEFORMATEX pwfx, DWORD dwCallback, DWORD dwCallbackInstance,
    DWORD fdwOpen);
```

Opens the given waveform-audio output device for playback.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_ALLOCATED	Specified resource is already allocated.
MMSYSERR_BADDEVICEID	Specified device identifier is out of range.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
WAVERR_BADFORMAT	Attempted to open with an unsupported waveform-audio format.
WAVERR_SYNC	The device is synchronous but **waveOutOpen** was called without using the WAVE_ALLOWSYNC flag.

phwo
Address filled with a handle identifying the open waveform-audio output device. Use the handle to identify the device when calling other waveform-audio output functions. This parameter might be NULL if the WAVE_FORMAT_QUERY flag is specified for *fdwOpen*.

uDeviceID
Identifier of the waveform-audio output device to open. It can be either a device identifier or a handle of an open waveform-audio input device.You can use the following flag instead of a device identifier:

WAVE_MAPPER
The function selects a waveform-audio output device capable of playing the given format.

pwfx
Address of a **WAVEFORMATEX** structure that identifies the format of the waveform-audio data to be sent to the device. You can free this structure immediately after passing it to **waveOutOpen**.

dwCallback
Address of a fixed callback function, an event handle, or a handle of a window or thread called during waveform-audio playback to process messages related to the progress of the playback. If no callback function is required, this value can be zero.

dwCallbackInstance
User-instance data passed to the callback mechanism. This parameter is not used with the window callback mechanism.

fdwOpen
Flags for opening the device. The following values are defined:

CALLBACK_EVENT
The *dwCallback* parameter is an event handle.

CALLBACK_FUNCTION
The *dwCallback* parameter is a callback procedure address.

CALLBACK_NULL
No callback mechanism. This is the default setting.

CALLBACK_THREAD
The *dwCallback* parameter is a thread handle.

CALLBACK_WINDOW
The *dwCallback* parameter is a window handle.

WAVE_ALLOWSYNC
If this flag is specified, a synchronous waveform-audio device can be opened. If this flag is not specified while opening a synchronous driver, the device will fail to open.

WAVE_FORMAT_QUERY
If this flag is specified, **waveOutOpen** queries the device to determine if it supports the given format, but the device is not actually opened.

WAVE_MAPPED
If this flag is specified, the *uDeviceID* parameter specifies a waveform-audio device to be mapped to by the wave mapper.

Use the **waveOutGetNumDevs** function to determine the number of waveform-audio output devices present in the system. If the value specified by the *uDeviceID* parameter is a device identifier, it can vary from zero to one less than the number of devices present. The WAVE_MAPPER constant can also be used as a device identifier.

The structure pointed to by *pwfx* can 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. For all other waveform-audio formats, use the **WAVEFORMATEX** structure to specify the length of the additional data.

If you choose to have a window or thread receive callback information, the following messages are sent to the window procedure function to indicate the progress of waveform-audio output: MM_WOM_OPEN, MM_WOM_CLOSE, and MM_WOM_DONE.

If you choose to have a function receive callback information, the following messages are sent to the function to indicate the progress of waveform-audio output: WOM_OPEN, WOM_CLOSE, and WOM_DONE.

waveOutPause

```
MMRESULT waveOutPause(HWAVEOUT hwo);
```

Pauses playback on the given waveform-audio output device. The current position is saved. Use the **waveOutRestart** function to resume playback from the current position.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
MMSYSERR_NOTSUPPORTED	Specified device is synchronous and does not support pausing.

hwo
Handle of the waveform-audio output device.

Calling this function when the output is already paused has no effect, and the function returns zero.

waveOutPrepareHeader

```
MMRESULT waveOutPrepareHeader(HWAVEOUT hwo, LPWAVEHDR pwh, UINT cbwh);
```

Prepares a waveform-audio data block for playback.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.

hwo
Handle of the waveform-audio output device.

pwh
Address of a **WAVEHDR** structure that identifies the data block to be prepared.

cbwh
Size, in bytes, of the **WAVEHDR** structure.

The **lpData**, **dwBufferLength**, and **dwFlags** members of the **WAVEHDR** structure must be set before calling this function (**dwFlags** must be zero).

The **dwFlags**, **dwBufferLength**, and **dwLoops** members of the **WAVEHDR** structure can change between calls to this function and the **waveOutWrite** function. (The only flags that can change in this interval for the **dwFlags** member are WHDR_BEGINLOOP and WHDR_ENDLOOP.) If you change the size specified by **dwBufferLength** before the call to **waveOutWrite**, the new value must be less than the prepared value.

Preparing a header that has already been prepared has no effect, and the function returns zero.

waveOutProc

```
void CALLBACK waveOutProc(HWAVEOUT hwo, UINT uMsg, DWORD dwInstance,
    DWORD dwParam1, DWORD dwParam2);
```

Callback function used with the waveform-audio output device. The **waveOutProc** function is a placeholder for the application-defined function name.

hwo
Handle of the waveform-audio device associated with the callback.

uMsg
Waveform-audio output message. It can be one of the following values:

WOM_CLOSE
Sent when the device is closed using the **waveOutClose** function.

WOM_DONE
Sent when the device driver is finished with a data block sent using the **waveOutWrite** function.

WOM_OPEN
Sent when the device is opened using the **waveOutOpen** function.

dwInstance
User-instance data specified with **waveOutOpen**.

dwParam1 and *dwParam2*
Message parameters.

Applications should not call any system-defined functions from inside a callback function, except for **PostMessage**, **timeGetSystemTime**, **timeGetTime**, **timeSetEvent**, **timeKillEvent**, **midiOutShortMsg**, **midiOutLongMsg**, and **OutputDebugStr**.

waveOutReset

```
MMRESULT waveOutReset(HWAVEOUT hwo);
```

Stops playback on the given waveform-audio output device and resets the current position to zero. All pending playback buffers are marked as done and returned to the application.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
MMSYSERR_NOTSUPPORTED	Specified device is synchronous and does not support pausing.

hwo
Handle of the waveform-audio output device.

waveOutRestart

```
MMRESULT waveOutRestart(HWAVEOUT hwo);
```

Resumes playback on a paused waveform-audio output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
MMSYSERR_NOTSUPPORTED	Specified device is synchronous and does not support pausing.

hwo
Handle of the waveform-audio output device.

Calling this function when the output is not paused has no effect, and the function returns zero.

waveOutSetPitch

```
MMRESULT waveOutSetPitch(HWAVEOUT hwo, DWORD dwPitch);
```

Sets the pitch for the specified waveform-audio output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
MMSYSERR_NOTSUPPORTED	Function isn't supported.

hwo
Handle of the waveform-audio output device.

dwPitch
New pitch multiplier setting. This 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 contains the signed integer part of the number, and the low-order word contains the fractional part. A value of 0x8000 in the low-order word 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.

Changing the pitch does not change the playback rate or the sample rate, nor does it change the playback time. Not all devices support pitch changes. To determine whether the device supports pitch control, use the WAVECAPS_PITCH flag to test the **dwSupport** member of the **WAVEOUTCAPS** structure (filled by the **waveOutGetDevCaps** function).

waveOutSetPlaybackRate

```
MMRESULT waveOutSetPlaybackRate(HWAVEOUT hwo, DWORD dwRate);
```

Sets the playback rate for the specified waveform-audio output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
MMSYSERR_NOTSUPPORTED	Function isn't supported.

hwo
Handle of the waveform-audio output device.

dwRate
New playback rate setting. This 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. A value of 0x8000 in the low-order word 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.

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** member of the **WAVEOUTCAPS** structure (filled by the **waveOutGetDevCaps** function).

waveOutSetVolume

```
MMRESULT waveOutSetVolume(HWAVEOUT hwo, DWORD dwVolume);
```

Sets the volume level of the specified waveform-audio output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
MMSYSERR_NOTSUPPORTED	Function is not supported.

hwo
Handle of an open waveform-audio output device.

dwVolume
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.

Changing the volume on a handle changes it for an instance of the device, rather than changing the default volume for the device (and affecting all instances of the device).

Not all devices support volume changes. To determine whether the device supports volume control, use the WAVECAPS_VOLUME flag to test the **dwSupport** member of the **WAVEOUTCAPS** structure (filled by the **waveOutGetDevCaps** function). To determine whether the device supports volume control on both the left and right channels, use the WAVECAPS_LRVOLUME flag.

Most devices do not 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.

waveOutUnprepareHeader

```
MMRESULT waveOutUnprepareHeader(HWAVEOUT hwo, LPWAVEHDR pwh, UINT cbwh);
```

Cleans up the preparation performed by the **waveOutPrepareHeader** function. This function must be called after the device driver is finished with a data block. You must call this function before freeing the buffer.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
WAVERR_STILLPLAYING	The data block pointed to by the *pwh* parameter is still in the queue.

hwo
Handle of the waveform-audio output device.

pwh
Address of a **WAVEHDR** structure identifying the data block to be cleaned up.

cbwh
Size, in bytes, of the **WAVEHDR** structure.

This function complements **waveOutPrepareHeader**. You must call this function before freeing the buffer. After passing a buffer to the device driver with the **waveOutWrite** function, 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.

waveOutWrite

```
MMRESULT waveOutWrite(HWAVEOUT hwo, LPWAVEHDR pwh, UINT cbwh);
```

Sends a data block to the given waveform-audio output device.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMSYSERR_INVALHANDLE	Specified device handle is invalid.
MMSYSERR_NODRIVER	No device driver is present.
MMSYSERR_NOMEM	Unable to allocate or lock memory.
WAVERR_UNPREPARED	The data block pointed to by the *pwh* parameter hasn't been prepared.

hwo
Handle of the waveform-audio output device.

pwh
Address of a **WAVEHDR** structure containing information about the data block.

cbwh
Size, in bytes, of the **WAVEHDR** structure.

When the buffer is finished, the WHDR_DONE bit is set in the **dwFlags** member of the **WAVEHDR** structure.

The buffer must be prepared with the **waveOutPrepareHeader** function before it is passed to **waveOutWrite**. Unless the device is paused by calling the **waveOutPause** function, playback begins when the first data block is sent to the device.

MM_WIM_CLOSE

```
MM_WIM_CLOSE
wParam = (WPARAM) hInputDev  // see below
lParam = reserved            // must be zero
```

Sent to a window when a waveform-audio input device is closed. The device handle is no longer valid after this message has been sent.

- No return value.

hInputDev
Handle of the waveform-audio input device that was closed.

MM_WIM_DATA

```
MM_WIM_DATA
wParam = (WPARAM) hInputDev
lParam = (LONG) lpwvhdr
```

Sent to a window when waveform-audio 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.

- No return value.

hInputDev
Handle of the waveform-audio input device that received the data.

lpwvhdr
Address of a **WAVEHDR** structure that identifies the buffer containing the data.

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

MM_WIM_OPEN

```
MM_WIM_OPEN
wParam = (WPARAM) hInputDev  // see below
lParam = reserved            // must be zero
```

Sent to a window when a waveform-audio input device is opened.

- No return value.

hInputDev
Handle of the device that was opened.

MM_WOM_CLOSE

```
MM_WOM_CLOSE
wParam = (WPARAM) hOutputDev  // see below
lParam = reserved             // must be zero
```

Sent to a window when a waveform-audio output device is closed. The device handle is no longer valid after this message has been sent.

- No return value.

hOutputDev
Handle of the device that was closed.

MM_WOM_DONE

```
MM_WOM_DONE
wParam = (WPARAM) hOutputDev
lParam = (LONG) lpwvhdr
```

Sent to a window when the given 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 the **waveOutReset** function.

- No return value.

hOutputDev
Handle of the waveform-audio output device that played the buffer.

lpwvhdr
Address of a **WAVEHDR** structure identifying the buffer.

MM_WOM_OPEN

```
MM_WOM_OPEN
wParam = (WPARAM) hOutputDev  // see below
lParam = reserved             // must be zero
```

Sent to a window when the given waveform-audio output device is opened.

- No return value.

hOutputDev
Handle of the device that was opened.

WIM_CLOSE

```
WIM_CLOSE
dwParam1 = reserved  // must be zero
dwParam2 = reserved  // must be zero
```

Sent to the given waveform-audio input callback function when a waveform-audio input device is closed. The device handle is no longer valid after this message has been sent.

- No return value.

WIM_DATA

```
WIM_DATA
dwParam1 = (DWORD) lpwvhdr  // see below
dwParam2 = reserved         // must be zero
```

Sent to the given waveform-audio input callback function when waveform-audio data is present in the input buffer and the buffer is being returned to the application. The message can be sent when the buffer is full or after the **waveInReset** function is called.

- No return value.

lpwvhdr
Address of a **WAVEHDR** structure that identifies the buffer containing the data.

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

WIM_OPEN

```
WIM_OPEN
dwParam1 = reserved  // must be zero
dwParam2 = reserved  // must be zero
```

Sent to a waveform-audio input callback function when a waveform-audio input device is opened.

- No return value.

WOM_CLOSE

```
WOM_CLOSE
dwParam1 = reserved  // must be zero
dwParam2 = reserved  // must be zero
```

Sent to a waveform-audio output callback function when a waveform-audio output device is closed. The device handle is no longer valid after this message has been sent.

- No return value.

WOM_DONE

```
WOM_DONE
dwParam1 = (DWORD) lpwvhdr  // see below
dwParam2 = reserved         // must be zero
```

Sent to a waveform-audio output callback function when the given 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 the **waveOutReset** function.

- No return value.

lpwvhdr
Address of a **WAVEHDR** structure identifying the buffer.

WOM_OPEN

```
WOM_OPEN
dwParam1 = reserved  // must be zero
dwParam2 = reserved  // must be zero
```

Sent to a waveform-audio output callback function when a waveform-audio output device is opened.

- No return value.

AUXCAPS

```
typedef struct {
    WORD      wMid;
    WORD      wPid;
    MMVERSION vDriverVersion;
    CHAR      szPname[MAXPNAMELEN];
    WORD      wTechnology;
    WORD      wReserved1;  // padding
    DWORD     dwSupport;
} AUXCAPS;
```

Describes the capabilities of an auxiliary output device.

wMid

Manufacturer identifier for the device driver for the auxiliary audio device. The following identifier is defined:

MM_MICROSOFT

Drivers developed by Microsoft Corporation.

wPid

Product identifier for the auxiliary audio device. Currently, no product identifiers are defined for auxiliary audio devices.

vDriverVersion

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

Product name in a null-terminated string.

wTechnology

Type of the auxiliary audio output:

AUXCAPS_AUXIN

Audio output from auxiliary input jacks.

AUXCAPS_CDAUDIO

Audio output from an internal CD-ROM drive.

dwSupport

Describes optional functionality supported by the auxiliary audio device.

AUXCAPS_LRVOLUME

Supports separate left and right volume control.

AUXCAPS_VOLUME

Supports volume control.

If a device supports volume changes, the AUXCAPS_VOLUME flag will be set. If a device supports separate volume changes on the left and right channels, both AUXCAPS_VOLUME and the AUXCAPS_LRVOLUME will be set.

PCMWAVEFORMAT

```
typedef struct {
    WAVEFORMAT wf;              // see below
    WORD       wBitsPerSample;  // number of bits per sample
} PCMWAVEFORMAT;
```

Describes the data format for PCM waveform-audio data. This structure has been superseded by the **WAVEFORMATEX** structure.

wf

A **WAVEFORMAT** structure containing general information about the format of the data.

WAVEFORMAT

```
typedef struct {
    WORD  wFormatTag;       // see below
    WORD  nChannels;        // see below
    DWORD nSamplesPerSec;   // sample rate, in samples per second
    DWORD nAvgBytesPerSec;  // see below
    WORD  nBlockAlign;      // see below
} WAVEFORMAT;
```

Describes the format of waveform-audio data. Only format information common to all waveform-audio data formats is included in this structure. This structure has been superseded by the **WAVEFORMATEX** structure.

wFormatTag

Format type. The following type is defined:

WAVE_FORMAT_PCM

Waveform-audio data is PCM.

nChannels

Number of channels in the waveform-audio data. Mono data uses one channel and stereo data uses two channels.

nAvgBytesPerSec

Required average data transfer rate, in bytes per second. For example, 16-bit stereo at 44.1 kHz has an average data rate of 176,400 bytes per second (2 channels × 2 bytes per sample per channel × 44,100 samples per second).

nBlockAlign

Block alignment, in bytes. The block alignment is the minimum atomic unit of data. 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).

For formats that require additional information, this structure is included as a member in another structure along with the additional information.

WAVEFORMATEX

```
typedef struct {
    WORD  wFormatTag;
    WORD  nChannels;
    DWORD nSamplesPerSec;
    DWORD nAvgBytesPerSec;
    WORD  nBlockAlign;
    WORD  wBitsPerSample;
    WORD  cbSize;
} WAVEFORMATEX;
```

Defines the format of waveform-audio data. Only format information common to all waveform-audio data formats is included in this structure. For formats that require additional information, this structure is included as the first member in another structure, along with the additional information.

wFormatTag
Waveform-audio format type. Format tags are registered with Microsoft Corporation for many compression algorithms. A complete list of format tags can be found in the MMREG.H header file.

nChannels
Number of channels in the waveform-audio data. Monaural data uses one channel and stereo data uses two channels.

nSamplesPerSec
Sample rate, in samples per second (hertz), that each channel should be played or recorded. If **wFormatTag** is WAVE_FORMAT_PCM, then common values for **nSamplesPerSec** are 8.0 kHz, 11.025 kHz, 22.05 kHz, and 44.1 kHz. For non-PCM formats, this member must be computed according to the manufacturer's specification of the format tag.

nAvgBytesPerSec
Required average data-transfer rate, in bytes per second, for the format tag. If **wFormatTag** is WAVE_FORMAT_PCM, **nAvgBytesPerSec** should be equal to the product of **nSamplesPerSec** and **nBlockAlign**. For non-PCM formats, this member must be computed according to the manufacturer's specification of the format tag.

Playback and record software can estimate buffer sizes by using the **nAvgBytesPerSec** member.

nBlockAlign
Block alignment, in bytes. The block alignment is the minimum atomic unit of data for the **wFormatTag** format type. If **wFormatTag** is WAVE_FORMAT_PCM, **nBlockAlign** should be equal to the product of **nChannels** and **wBitsPerSample** divided by 8 (bits per byte). For non-PCM formats, this member must be computed according to the manufacturer's specification of the format tag.

Playback and record software must process a multiple of **nBlockAlign** bytes of data at a time. Data written and read from a device must always start at the beginning of a block. For example, it is illegal to start playback of PCM data in the middle of a sample (that is, on a non-block-aligned boundary).

wBitsPerSample
Bits per sample for the **wFormatTag** format type. If **wFormatTag** is WAVE_FORMAT_PCM, then **wBitsPerSample** should be equal to 8 or 16. For non-PCM formats, this member must be set according to the manufacturer's specification of the format tag. Note that some compression schemes cannot define a value for **wBitsPerSample**, so this member can be zero.

cbSize
Size, in bytes, of extra format information appended to the end of the **WAVEFORMATEX** structure. This information can be used by non-PCM formats to store extra attributes for the **wFormatTag**. If no extra information is required by the **wFormatTag**, this member must be set to zero. Note that for WAVE_FORMAT_PCM formats (and only WAVE_FORMAT_PCM formats), this member is ignored.

An example of a format that uses extra information is the Microsoft Adaptive Delta Pulse Code Modulation (MS-ADPCM) format. The **wFormatTag** for MS-ADPCM is WAVE_FORMAT_ADPCM. The **cbSize** member will typically be set to 32. The extra information stored for WAVE_FORMAT_ADPCM is coefficient pairs required for encoding and decoding the waveform-audio data.

WAVEHDR

```
typedef struct {
    LPSTR  lpData;                  // address of the waveform buffer
    DWORD  dwBufferLength;          // length, in bytes, of the buffer
    DWORD  dwBytesRecorded;         // see below
    DWORD  dwUser;                  // 32 bits of user data
    DWORD  dwFlags;                 // see below
    DWORD  dwLoops;                 // see below
    struct wavehdr_tag far * lpNext; // reserved; must be zero
    DWORD  reserved;                // reserved; must be zero
} WAVEHDR;
```

Defines the header used to identify a waveform-audio buffer.

dwBytesRecorded
When the header is used in input, this member specifies how much data is in the buffer.

dwFlags
Flags supplying information about the buffer. The following values are defined:

WHDR_BEGINLOOP
This buffer is the first buffer in a loop. This flag is used only with output buffers.

WHDR_DONE
Set by the device driver to indicate that it is finished with the buffer and is returning it to the application.

WHDR_ENDLOOP
This buffer is the last buffer in a loop. This flag is used only with output buffers.

WHDR_INQUEUE
Set by Windows to indicate that the buffer is queued for playback.

WHDR_PREPARED
Set by Windows to indicate that the buffer has been prepared with the **waveInPrepareHeader** or **waveOutPrepareHeader** function.

dwLoops
Number of times to play the loop. This member is used only with output buffers.

Use the WHDR_BEGINLOOP and WHDR_ENDLOOP flags in the **dwFlags** member 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** member in the **WAVEHDR** structure for the first block in the loop to specify the number of times to play the loop.

The **lpData**, **dwBufferLength**, and **dwFlags** members must be set before calling the **waveInPrepareHeader** or **waveOutPrepareHeader** function. (For either function, the **dwFlags** member must be set to zero.)

WAVEINCAPS

```
typedef struct {
    WORD      wMid;
    WORD      wPid;
    MMVERSION vDriverVersion;
    CHAR      szPname[MAXPNAMELEN];
    DWORD     dwFormats;
    WORD      wChannels;
    WORD      wReserved1; // padding
} WAVEINCAPS;
```

Describes the capabilities of a waveform-audio input device.

wMid
Manufacturer identifier for the device driver for the waveform-audio input device. Manufacturer identifiers are defined in Chapter 0, "Manufacturer and Product Identifiers."

wPid
Product identifier for the waveform-audio input device. Product identifiers are defined in Chapter 0, "Manufacturer and Product Identifiers."

vDriverVersion
Version number of the device driver for the waveform-audio input device. The high-order byte is the major version number, and the low-order byte is the minor version number.

szPname
Product name in a null-terminated string.

dwFormats
Standard formats that are supported. Can be a combination of the following:

WAVE_FORMAT_1M08	11.025 kHz, mono, 8-bit
WAVE_FORMAT_1M16	11.025 kHz, mono, 16-bit
WAVE_FORMAT_1S08	11.025 kHz, stereo, 8-bit
WAVE_FORMAT_1S16	11.025 kHz, stereo, 16-bit
WAVE_FORMAT_2M08	22.05 kHz, mono, 8-bit
WAVE_FORMAT_2M16	22.05 kHz, mono, 16-bit
WAVE_FORMAT_2S08	22.05 kHz, stereo, 8-bit
WAVE_FORMAT_2S16	22.05 kHz, stereo, 16-bit
WAVE_FORMAT_4M08	44.1 kHz, mono, 8-bit
WAVE_FORMAT_4M16	44.1 kHz, mono, 16-bit
WAVE_FORMAT_4S08	44.1 kHz, stereo, 8-bit
WAVE_FORMAT_4S16	44.1 kHz, stereo, 16-bit

wChannels
Number specifying whether the device supports mono (1) or stereo (2) input.

WAVEOUTCAPS

```
typedef struct {
    WORD      wMid;
    WORD      wPid;
    MMVERSION vDriverVersion;
    CHAR      szPname[MAXPNAMELEN];
    DWORD     dwFormats;
    WORD      wChannels;
    WORD      wReserved1;  // packing
    DWORD     dwSupport;
} WAVEOUTCAPS;
```

Describes the capabilities of a waveform-audio output device.

wMid
Manufacturer identifier for the device driver for the device. Manufacturer identifiers are defined in Chapter 0, "Manufacturer and Product Identifiers."

wPid
Product identifier for the device. Product identifiers are defined in Chapter 0, "Manufacturer and Product Identifiers."

vDriverVersion
Version number of the device driver for the device. The high-order byte is the major version number, and the low-order byte is the minor version number.

szPname
Product name in a null-terminated string.

dwFormats
Standard formats that are supported. Can be a combination of the following:

WAVE_FORMAT_1M08	11.025 kHz, mono, 8-bit
WAVE_FORMAT_1M16	11.025 kHz, mono, 16-bit
WAVE_FORMAT_1S08	11.025 kHz, stereo, 8-bit
WAVE_FORMAT_1S16	11.025 kHz, stereo, 16-bit
WAVE_FORMAT_2M08	22.05 kHz, mono, 8-bit
WAVE_FORMAT_2M16	22.05 kHz, mono, 16-bit
WAVE_FORMAT_2S08	22.05 kHz, stereo, 8-bit
WAVE_FORMAT_2S16	22.05 kHz, stereo, 16-bit
WAVE_FORMAT_4M08	44.1 kHz, mono, 8-bit
WAVE_FORMAT_4M16	44.1 kHz, mono, 16-bit
WAVE_FORMAT_4S08	44.1 kHz, stereo, 8-bit
WAVE_FORMAT_4S16	44.1 kHz, stereo, 16-bit

wChannels
Number specifying whether the device supports mono (1) or stereo (2) output.

dwSupport
Optional functionality supported by the device. The following values are defined:

WAVECAPS_LRVOLUME	Supports separate left and right volume control.
WAVECAPS_PITCH	Supports pitch control.
WAVECAPS_PLAYBACKRATE	Supports playback rate control.

WAVECAPS_SYNC	The driver is synchronous and will block while playing a buffer.
WAVECAPS_VOLUME	Supports volume control.
WAVECAPS_SAMPLEACCURATE	Returns sample-accurate position information.

If a device supports volume changes, the WAVECAPS_VOLUME flag will be set for the **dwSupport** member. 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 member.

Joysticks

The Win32 application programming interface supports ancillary input devices for applications that provide alternatives to using the keyboard and mouse. The joystick is one such device. The joystick provides positional information within a coordinate system that has absolute maximum and minimum values in each axis of movement.

This chapter describes the Win32 functions and messages that support joysticks, as well as other ancillary input devices that track positions within an absolute coordinate system, such as a touch screen, digitizing tablet, and light pen. Extended capabilities also provide support for rudder pedals, flight yokes, and other devices that use up to six axes of movement, a point-of-view hat, and 32 buttons.

Joystick services are loaded when the operating system is started. The joystick services can simultaneously monitor two joysticks, each with two- or three-axis movement. Each joystick can have up to four buttons. You can use the joystick functions to determine the capabilities of the joysticks and joystick driver. Also, you can process a joystick's positional and button information by querying the joystick directly or by capturing the joystick and processing messages from it. The latter method is simpler because your application does not have to manually query the joystick or track the time to generate queries at regular intervals.

Joystick Capabilities

Joysticks can support two- or three-axis movement and up to four buttons. Joysticks also support different *ranges of motion* and *polling frequencies*. The range of motion is the distance a joystick handle can move from its resting position to the position farthest from its resting position. The polling frequency is the time interval between joystick queries.

Joystick drivers can support either one or two joysticks. You can determine the number of joysticks supported by a joystick driver by using the **joyGetNumDevs** function. This function returns an unsigned integer that contains the number of supported joysticks or zero if there is no joystick support. The return value does not indicate the number of joysticks attached to the system.

You can determine if a joystick is attached to the system by using the **joyGetPos** function. This function returns JOYERR_NOERROR if the specified device is attached or JOYERR_UNPLUGGED otherwise.

Each joystick has several capabilities that are available to your application. You can retrieve the capabilities of a joystick by using the **joyGetDevCaps** function. This function fills a **JOYCAPS** structure with joystick capabilities such as the minimum and maximum values for its coordinate system, the number of buttons on the joystick, and the minimum and maximum polling frequencies.

Joystick Position

You can query a joystick for position and button information by using the **joyGetPos** function. For example, an application can query the joystick for baseline position values. The Joystick Control Panel property sheet uses this technique when calibrating the joystick.

You can also query a joystick or other device that has extended capabilities by using the **joyGetPosEx** function.

Joystick Notifications

You can capture direct joystick messages to be sent to a function by using the **joySetCapture** function. Only one application at a time can capture messages from a joystick, but you can query the joystick from another application by using the **joyGetPos** or **joyGetPosEx** function.

Note A joystick message can fail to reach the application that captured the joystick if a second application uses **joyGetPos** or **joyGetPosEx** to query the joystick near the same time that the message is sent. In this case, the second application might intercept the message.

If you want to capture messages from two joysticks attached to the system, use **joySetCapture** twice, once for each joystick. Your window receives separate and distinct messages for each device.

You can release a captured joystick by using the **joyReleaseCapture** function. If an application does not release the joystick before ending, the joystick is automatically released shortly after the capture window is destroyed.

You cannot capture an unplugged joystick. The **joySetCapture** function returns JOYERR_UNPLUGGED if the specified device is unplugged.

Time-Based Notifications

You can notify the Microsoft Windows operating system to send joystick messages to an application at regular time intervals by setting the *fChanged* parameter of **joySetCapture** to FALSE and by specifying the interval length between successive messages. Assign the *uPeriod* parameter a value between the minimum and maximum polling frequencies for the joystick. You can determine this range by using the **joyGetDevCaps** function, which fills the **wPeriodMin** and **wPeriodMax** members in the **JOYCAPS** structure. If the *uPeriod* value is outside the range of valid polling frequencies for the joystick, the joystick driver uses the minimum or maximum polling frequency, whichever is closer to the *uPeriod* value.

Note Windows sets up a timer event with each call to **joySetCapture**.

Event-Based Notifications

You can notify Windows to send joystick messages to an application whenever the position of a joystick axis changes by a value greater than the *movement threshold* of the device. 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. You can set the movement threshold by using the **joySetThreshold** function. You can retrieve the minimum polling frequency of the joystick by using the **joyGetDevCaps** function.

Joystick Notification Messages

Joystick messages notify your application that a joystick has changed position or that one of its buttons has changed states. Messages beginning with MM_JOY1 are sent to the function if your application requests input from the joystick using the identifier JOYSTICKID1, and MM_JOY2 messages are sent if your application requests input from the joystick using the identifier JOYSTICKID2.

The messages in the following table identify the status of the joystick buttons:

Message	Description
MM_JOY1BUTTONDOWN	A button of joystick JOYSTICKID1 has been pressed.
MM_JOY1BUTTONUP	A button of joystick JOYSTICKID1 has been released.
MM_JOY1MOVE	Joystick JOYSTICKID1 changed position in the x- or y-direction.
MM_JOY1ZMOVE	Joystick JOYSTICKID1 changed position in the z-direction.
MM_JOY2BUTTONDOWN	A button of joystick JOYSTICKID2 has been pressed.
MM_JOY2BUTTONUP	A button of joystick JOYSTICKID2 has been released.
MM_JOY2MOVE	Joystick JOYSTICKID2 changed position in the x- or y-direction
MM_JOY2ZMOVE	Joystick JOYSTICKID2 changed position in the z-direction.

All messages report nonexistent buttons as released.

Using Joysticks

This section contains examples demonstrating how to perform the following tasks:

- Get the driver capabilities.
- Capture joystick input.
- Process joystick messages.

The examples are taken from a simple joystick application that gets position and button-state information from the joystick services, plays waveform-audio resources, and paints bullet holes on the screen when a user presses the joystick buttons.

Getting the Driver Capabilities

The following example determines whether the joystick services are available and if a joystick is attached to one of the ports.

```
JOYINFO joyinfo;
UINT wNumDevs, wDeviceID;
BOOL bDev1Attached, bDev2Attached;

    if((wNumDevs = joyGetNumDevs()) == 0)
        return ERR_NODRIVER;
    bDev1Attached = joyGetPos(JOYSTICKID1,&joyinfo) != JOYERR_UNPLUGGED;
    bDev2Attached = wNumDevs == 2 && joyGetPos(JOYSTICKID2,&joyinfo) !=
        JOYERR_UNPLUGGED;
    if(bDev1Attached || bDev2Attached)   // decide which joystick to use
        wDeviceID = bDev1Attached ? JOYSTICKID1 : JOYSTICKID2;
    else
        return ERR_NODEVICE;
```

Capturing Joystick Input

Most of the code controlling the joystick is in the main window function. In the following portion of the message handler, the application captures input from the joystick JOYSTICKID1.

```
case WM_CREATE:
    if(joySetCapture(hWnd, JOYSTICKID1, NULL, FALSE))
    {
        MessageBeep(MB_ICONEXCLAMATION);
        MessageBox(hWnd, "Couldn't capture the joystick.", NULL,
            MB_OK | MB_ICONEXCLAMATION);
        PostMessage(hWnd,WM_CLOSE,0,0L);
    }
    break;
```

Processing Joystick Messages

The following example illustrates how an application might respond to joystick movements and changes in the button states. When the joystick changes position, the application moves the cursor and, if either button is pressed, draws a bullet hole on the desktop. When a joystick button is pressed, the application draws a hole on the desktop and plays a sound continuously until a button is released.

```
case MM_JOY1MOVE :                      // changed position
    if((UINT) wParam & (JOY_BUTTON1 | JOY_BUTTON2))
        DrawFire(hWnd);
    DrawSight(lParam);                  // calculates new cursor position
    break;
case MM_JOY1BUTTONDOWN :                // button is down
    if((UINT) wParam & JOY_BUTTON1)
    {
        PlaySound(lpButton1, SND_LOOP | SND_ASYNC | SND_MEMORY);
        DrawFire(hWnd);
    }
    else if((UINT) wParam & JOY_BUTTON2)
    {
        PlaySound(lpButton2, SND_ASYNC | SND_MEMORY |  SND_LOOP);
        DrawFire(hWnd);
    }
    break;
case MM_JOY1BUTTONUP :                  // button is up
    sndPlaySound(NULL, 0);              // stops the sound
    break;
```

Joystick Reference

This section describes the functions, structures, and messages associated with joysticks. The elements are grouped as follows:

Device Capabilities

joyGetDevCaps
joyGetNumDevs
JOYCAPS

Querying a Joystick

joyGetPos
joyGetPosEx
JOYINFO
JOYINFOEX

Capturing a Joystick

joyGetThreshold
joyReleaseCapture
joySetCapture
joySetThreshold
MM_JOY1BUTTONDOWN
MM_JOY1BUTTONUP
MM_JOY1MOVE
MM_JOY1ZMOVE
MM_JOY2BUTTONDOWN
MM_JOY2BUTTONUP
MM_JOY2MOVE
MM_JOY2ZMOVE

Joystick Functions

An application uses the joystick functions to query a joystick driver and to prepare an application to receive notification messages from a joystick driver.

joyGetDevCaps

[New - Windows 95]

```
MMRESULT joyGetDevCaps(UINT uJoyID, LPJOYCAPS pjc, UINT cbjc);
```

Queries a joystick to determine its capabilities.

- Returns JOYERR_NOERROR if successful or one of the following error values:

MMSYSERR_NODRIVER	The joystick driver is not present.
MMSYSERR_INVALPARAM	An invalid parameter was passed.

uJoyID
Identifier of the joystick (JOYSTICKID1 or JOYSTICKID2) to be queried.

pjc
Address of a **JOYCAPS** structure to contain the capabilities of the joystick.

cbjc
Size, in bytes, of the **JOYCAPS** structure.

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

joyGetNumDevs

[New - Windows 95]

```
UINT joyGetNumDevs(VOID);
```

Queries the joystick driver for the number of joysticks it supports.

- Returns the number of joysticks supported by the joystick driver or zero if no driver is present.

Use the **joyGetPos** function to determine whether a given joystick is physically attached to the system. If the specified joystick is not connected, **joyGetPos** returns a JOYERR_UNPLUGGED error value.

joyGetPos

[New - Windows 95]

```
MMRESULT joyGetPos(UINT uJoyID, LPJOYINFO pji);
```

Queries a joystick for its position and button status.

- Returns JOYERR_NOERROR if successful or one of the following error values:

MMSYSERR_NODRIVER	The joystick driver is not present.
MMSYSERR_INVALPARAM	An invalid parameter was passed.
JOYERR_UNPLUGGED	The specified joystick is not connected to the system.

uJoyID
Identifier of the joystick (JOYSTICKID1 or JOYSTICKID2) to be queried.

pji
Address of a **JOYINFO** structure that contains the position and button status of the joystick.

For devices that have four to six axes of movement, a point-of-view control, or more than four buttons, use the **joyGetPosEx** function.

joyGetPosEx

[New - Windows 95]

```
MMRESULT joyGetPosEx(UINT uJoyID, LPJOYINFOEX pji);
```

Queries a joystick for its position and button status.

- Returns JOYERR_NOERROR if successful or one of the following error values:

MMSYSERR_NODRIVER	The joystick driver is not present.
MMSYSERR_INVALPARAM	An invalid parameter was passed.
MMSYSERR_BADDEVICEID	The specified joystick identifier is invalid.
JOYERR_UNPLUGGED	The specified joystick is not connected to the system.

uJoyID
Identifier of the joystick (JOYSTICKID1 or JOYSTICKID2) to be queried.

pji
Address of a **JOYINFOEX** structure that contains extended position information and button status of the joystick.

This function provides access to extended devices such as rudder pedals, point-of-view hats, devices with a large number of buttons, and coordinate systems using up to six axes. For joystick devices that use three axes or fewer and have fewer than four buttons, use the **joyGetPos** function.

joyGetThreshold

[New - Windows 95]

```
MMRESULT joyGetThreshold(UINT uJoyID, LPUINT puThreshold);
```

Queries a joystick for its current movement threshold.

- Returns JOYERR_NOERROR if successful or one of the following error values:

MMSYSERR_NODRIVER	The joystick driver is not present.
MMSYSERR_INVALPARAM	An invalid parameter was passed.

uJoyID
Identifier of the joystick (JOYSTICKID1 or JOYSTICKID2) to be queried.

puThreshold
Address of a variable that contains the movement threshold value.

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.

joyReleaseCapture

[New - Windows 95]

```
MMRESULT joyReleaseCapture(UINT uJoyID);
```

Releases the specified captured joystick.

- Returns JOYERR_NOERROR if successful or one of the following error values:

MMSYSERR_NODRIVER	The joystick driver is not present.
JOYERR_PARMS	The specified joystick device identifier *uJoyID* is invalid.

uJoyID
Identifier of the joystick (JOYSTICKID1 or JOYSTICK2) to be released.

joySetCapture

[New - Windows 95]

```
MMRESULT joySetCapture(HWND hwnd, UINT uJoyID, UINT uPeriod,
    BOOL fChanged);
```

Captures a joystick by causing its messages to be sent to the specified window.

- Returns JOYERR_NOERROR if successful or one of the following error values:

MMSYSERR_NODRIVER	The joystick driver is not present.
JOYERR_NOCANDO	Cannot capture joystick input because a required service (such as a Windows timer) is unavailable.
JOYERR_UNPLUGGED	The specified joystick is not connected to the system.

hwnd
Handle of the window to receive the joystick messages.

uJoyID
Identifier of the joystick (JOYSTICKID1 or JOYSTICKID2) to be captured.

uPeriod
Polling frequency, in milliseconds.

fChanged
Change position flag. Specify TRUE for this parameter to send messages only when the position changes by a value greater than the joystick movement threshold. Otherwise, messages are sent at the polling frequency specified in *uPeriod*.

This function fails if the specified joystick is currently captured. Call the **joyReleaseCapture** function to release the captured joystick, or destroy the window to release the joystick automatically.

joySetThreshold

[New - Windows 95]

```
MMRESULT joySetThreshold(UINT uJoyID, UINT uThreshold);
```

Sets the movement threshold of a joystick.

- Returns JOYERR_NOERROR if successful or one of the following error values:

MMSYSERR_NODRIVER	The joystick driver is not present.
JOYERR_PARMS	The specified joystick device identifier *uJoyID* is invalid.

uJoyID
Identifier of the joystick (JOYSTICKID1 or JOYSTICKID2).

uThreshold
New movement threshold.

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.

Joystick Messages

A joystick device driver sends the following messages to notify an application that a joystick has changed position or one of its buttons has changed states.

MM_JOY1BUTTONDOWN

[New - Windows 95]

```
MM_JOY1BUTTONDOWN
fwButtons = wParam;
xPos = LOWORD(lParam);
yPos = HIWORD(lParam);
```

Notifies the window that has captured joystick JOYSTICKID1 that a button has been pressed.

fwButtons

Identifies the button that has changed state and the buttons that are pressed. It can be one of the following:

JOY_BUTTON1CHG	First joystick button has changed state.
JOY_BUTTON2CHG	Second joystick button has changed state.
JOY_BUTTON3CHG	Third joystick button has changed state.
JOY_BUTTON4CHG	Fourth joystick button has changed state.

and one or more of the following:

JOY_BUTTON1	First joystick button is pressed.
JOY_BUTTON2	Second joystick button is pressed.
JOY_BUTTON3	Third joystick button is pressed.
JOY_BUTTON4	Fourth joystick button is pressed.

xPos and *yPos*

The x- and y-coordinates of the joystick relative to the upper left corner of the client area.

MM_JOY1BUTTONUP

[New - Windows 95]

```
MM_JOY1BUTTONUP
fwButton = wParam;
xPos = LOWORD(lParam);
yPos = HIWORD(lParam);
```

Notifies the window that has captured joystick JOYSTICKID1 that a button has been released.

fwButtons

Identifies the button that has changed state and the buttons that are pressed. It can be one of the following:

JOY_BUTTON1CHG	First joystick button has changed state.
JOY_BUTTON2CHG	Second joystick button has changed state.
JOY_BUTTON3CHG	Third joystick button has changed state.
JOY_BUTTON4CHG	Fourth joystick button has changed state.

and one or more of the following:

JOY_BUTTON1	First joystick button is pressed.
JOY_BUTTON2	Second joystick button is pressed.
JOY_BUTTON3	Third joystick button is pressed.
JOY_BUTTON4	Fourth joystick button is pressed.

xPos and *yPos*

The x- and y-coordinates of the joystick relative to the upper left corner of the client area.

MM_JOY1MOVE

[New - Windows 95]

```
MM_JOY1MOVE
fwButtons = wParam;
xPos = LOWORD(lParam);
yPos = HIWORD(lParam);
```

Notifies the window that has captured joystick JOYSTICKID1 that the joystick position has changed.

fwButtons

Identifies the buttons that are pressed. It can be one or more of the following values:

JOY_BUTTON 1	First joystick button is pressed.
JOY_BUTTON 2	Second joystick button is pressed.
JOY_BUTTON 3	Third joystick button is pressed.
JOY_BUTTON 4	Fourth joystick button is pressed.

xPos and *yPos*

The x- and y-coordinates of the joystick relative to the upper left corner of the client area.

MM_JOY1ZMOVE

[New - Windows 95]

```
MM_JOY1ZMOVE
fwButtons = wParam;
zPos = LOWORD(lParam);
```

Notifies the window that has captured joystick JOYSTICKID1 that the joystick position on the z-axis has changed.

fwButtons

Identifies the buttons that are pressed. It can be one or more of the following values:

JOY_BUTTON1	First joystick button is pressed.
JOY_BUTTON2	Second joystick button is pressed.
JOY_BUTTON3	Third joystick button is pressed.
JOY_BUTTON4	Fourth joystick button is pressed.

zPos

The z-coordinate of the joystick.

MM_JOY2BUTTONDOWN

[New - Windows 95]

```
MM_JOY2BUTTONDOWN
fwButtons = wParam;
xPos = LOWORD(lParam);
yPos = HIWORD(lParam);
```

Notifies the window that has captured joystick JOYSTICKID2 that a button has been pressed.

fwButtons

Identifies the button that has changed state and the buttons that are pressed. It can be one of the following:

JOY_BUTTON1CHG	First joystick button has changed state.
JOY_BUTTON2CHG	Second joystick button has changed state.
JOY_BUTTON3CHG	Third joystick button has changed state.
JOY_BUTTON4CHG	Fourth joystick button has changed state.

and one or more of the following:

JOY_BUTTON1	First joystick button is pressed.
JOY_BUTTON2	Second joystick button is pressed.
JOY_BUTTON3	Third joystick button is pressed.
JOY_BUTTON4	Fourth joystick button is pressed.

xPos and *yPos*

The x- and y-coordinates of the joystick relative to the upper left corner of the client area.

MM_JOY2BUTTONUP

[New - Windows 95]

```
MM_JOY2BUTTONUP
fwButton = wParam;
xPos = LOWORD(lParam);
yPos = HIWORD(lParam);
```

Notifies the window that has captured joystick JOYSTICKID2 that a button has been released.

fwButtons

Identifies the button that has changed state and the buttons that are pressed. It can be one of the following:

JOY_BUTTON1CHG	First joystick button has changed state.
JOY_BUTTON2CHG	Second joystick button has changed state.
JOY_BUTTON3CHG	Third joystick button has changed state.
JOY_BUTTON4CHG	Fourth joystick button has changed state.

and one or more of the following:

JOY_BUTTON1	First joystick button is pressed.
JOY_BUTTON2	Second joystick button is pressed.
JOY_BUTTON3	Third joystick button is pressed.
JOY_BUTTON4	Fourth joystick button is pressed.

xPos and *yPos*

The x- and y-coordinates of the joystick relative to the upper left corner of the client area.

MM_JOY2MOVE

[New - Windows 95]

```
MM_JOY2MOVE
fwButtons = wParam;
xPos = LOWORD(lParam);
yPos = HIWORD(lParam);
```

Notifies the window that has captured joystick JOYSTICKID2 that the joystick position has changed.

fwButtons

Identifies the buttons that are pressed. It can be one or more of the following values:

JOY_BUTTON 1	First joystick button is pressed.
JOY_BUTTON 2	Second joystick button is pressed.
JOY_BUTTON 3	Third joystick button is pressed.
JOY_BUTTON 4	Fourth joystick button is pressed.

xPos and *yPos*

The x- and y-coordinates of the joystick relative to the upper left corner of the client area.

MM_JOY2ZMOVE

[New - Windows 95]

```
MM_JOY2ZMOVE
fwButtons = wParam;
zPos = LOWORD(lParam);
```

Notifies the window that has captured joystick JOYSTICKID2 that the joystick position on the z-axis has changed.

fwButtons

Identifies the buttons that are pressed. It can be one or more of the following values:

JOY_BUTTON1	First joystick button is pressed.
JOY_BUTTON2	Second joystick button is pressed.
JOY_BUTTON3	Third joystick button is pressed.
JOY_BUTTON4	Fourth joystick button is pressed.

zPos

The z-coordinate of the joystick.

Joystick Structures

The following structures store information about the capabilities of a joystick or its current position and button states.

JOYCAPS

[New - Windows 95]

```
typedef struct {
    WORD wMid;                  \\ manufacturer identifier
    WORD wPid;                  \\ product identifier
    CHAR szPname[MAXPNAMELEN];  \\ see below
    UINT wXmin;                 \\ min. x-coordinate
    UINT wXmax;                 \\ max. x-coordinate
    UINT wYmin;                 \\ min. y-coordinate
    UINT wYmax;                 \\ max. y-coordinate
    UINT wZmin;                 \\ min. z-coordinate
    UINT wZmax;                 \\ max. z-coordinate
    UINT wNumButtons;           \\ no. of joystick buttons
    UINT wPeriodMin;            \\ see below
    UINT wPeriodMax;            \\ see below
\\  The following members are not in previous versions of Windows.
    UINT wRmin;                 \\ see below
    UINT wRmax;                 \\ see below
    UINT wUmin;                 \\ see below
    UINT wUmax;                 \\ see below
    UINT wVmin;                 \\ see below
    UINT wVmax;                 \\ see below
    UINT wCaps;                 \\ see below
    UINT wMaxAxes;              \\ see below
    UINT wNumAxes;              \\ see below
    UINT wMaxButtons;           \\ see below
    CHAR szRegKey[MAXPNAMELEN]; \\ see below
    CHAR szOEMVxD[MAXOEMVXD];   \\ see below
} JOYCAPS;
```

Contains information about the joystick capabilities.

szPname
Null-terminated string containing the joystick product name.

wPeriodMin
Smallest polling frequency supported when captured by the **joySetCapture** function.

wPeriodMax
Largest polling frequency supported when captured by **joySetCapture**.

wRmin and **wRmax**
Minimum and maximum rudder values. The rudder is a fourth axis of movement.

wUmin and **wUmax**
Minimum and maximum u-coordinate (fifth axis) values.

wVmin and **wVmax**
Minimum and maximum v-coordinate (sixth axis) values.

wCaps
Joystick capabilities The following flags define individual capabilities that a joystick might have:

JOYCAPS_HASZ	Joystick has z-coordinate information.
JOYCAPS_HASR	Joystick has rudder (fourth axis) information.
JOYCAPS_HASU	Joystick has u-coordinate (fifth axis)

	information.
JOYCAPS_HASV	Joystick has v-coordinate (sixth axis) information.
JOYCAPS_HASPOV	Joystick has point-of-view information.
JOYCAPS_POV4DIR	Joystick point-of-view supports discrete values (centered, forward, backward, left, and right).
JOYCAPS_POVCTS	Joystick point-of-view supports continuous degree bearings.

wMaxAxes
Maximum number of axes supported by the joystick.

wNumAxes
Number of axes currently in use by the joystick.

wMaxButtons
Maximum number of buttons supported by the joystick.

szRegKey
Null-terminated string containing the registry key for the joystick.

szOEMVxD
Null-terminated string identifying the joystick driver OEM.

JOYINFO

[New - Windows 95]

```
typedef struct {
    UINT wXpos;      \\ current x-coordinate
    UINT wYpos;      \\ current y-coordinate
    UINT wZpos;      \\ current z-coordinate
    UINT wButtons;   \\ see below
} JOYINFO;
```

Contains information about the joystick position and button state.

wButtons

Current state of joystick buttons described by one or more of the following values:

JOY_BUTTON 1	First joystick button is pressed.
JOY_BUTTON 2	Second joystick button is pressed.
JOY_BUTTON 3	Third joystick button is pressed.
JOY_BUTTON 4	Fourth joystick button is pressed.

JOYINFOEX

[New - Windows 95]

```
typedef struct joyinfoex_tag {
    DWORD dwSize;         \\ size, in bytes, of this structure
    DWORD dwFlags;        \\ see below
    DWORD dwXpos;         \\ current x-coordinate
    DWORD dwYpos;         \\ current y-coordinate
    DWORD dwZpos;         \\ current z-coordinate
    DWORD dwRpos;         \\ see below
    DWORD dwUpos;         \\ current 5th axis position
    DWORD dwVpos;         \\ current 6th axis position
    DWORD dwButtons;      \\ see below
    DWORD dwButtonNumber; \\ see below
    DWORD dwPOV;          \\ see below
    DWORD dwReserved1;    \\ reserved; do not use
    DWORD dwReserved2;    \\ reserved; do not use
} JOYINFOEX;
```

Contains extended information about the joystick position, point-of-view position, and button state.

dwFlags

Flags indicating the valid information returned in this structure. Members that do not contain valid information are set to zero. The following flags are defined:

JOY_RETURNALL	Equivalent to setting all of the JOY_RETURN bits except JOY_RETURNRAWDATA.
JOY_RETURNBUTTONS	The **dwButtons** member contains valid information about the state of each joystick button.
JOY_RETURNCENTERED	Centers the joystick neutral position to the middle value of each axis of movement.
JOY_RETURNPOV	The **dwPOV** member contains valid information about the point-of-view control, expressed in discrete units.
JOY_RETURNPOVCTS	The **dwPOV** member contains valid information about the point-of-view control expressed in continuous, one-hundredth degree units.
JOY_RETURNR	The **dwRpos** member contains valid rudder pedal data. This information represents another (fourth) axis.
JOY_RETURNRAWDATA	Data stored in this structure is uncalibrated joystick readings.
JOY_RETURNU	The **dwUpos** member contains valid data for a fifth axis of the joystick, if such an axis is available, or returns zero otherwise.
JOY_RETURNV	The **dwVpos** member contains valid data for a sixth axis of the joystick, if

	such an axis is available, or returns zero otherwise.
JOY_RETURNX	The **dwXpos** member contains valid data for the x-coordinate of the joystick.
JOY_RETURNY	The **dwYpos** member contains valid data for the y-coordinate of the joystick.
JOY_RETURNZ	The **dwZpos** member contains valid data for the z-coordinate of the joystick.
JOY_USEDEADZONE	Expands the range for the neutral position of the joystick and calls this range the dead zone. The joystick driver returns a constant value for all positions in the dead zone.

The following flags provide data to calibrate a joystick and are intended for custom calibration applications.

JOY_CAL_READ3	Read the x-, y-, and z-coordinates and store the raw values in **dwXpos**, **dwYpos**, and **dwZpos**.
JOY_CAL_READ4	Read the rudder information and the x-, y-, and z-coordinates and store the raw values in **dwXpos**, **dwYpos**, **dwZpos**, and **dwRpos**.
JOY_CAL_READ5	Read the rudder information and the x-, y-, z-, and u-coordinates and store the raw values in **dwXpos**, **dwYpos**, **dwZpos**, **dwRpos**, and **dwUpos**.
JOY_CAL_READ6	Read the raw v-axis data if a joystick mini driver is present that will provide the data. Returns zero otherwise.
JOY_CAL_READALWAYS	Read the joystick port even if the driver does not detect a device.
JOY_CAL_READRONLY	Read the rudder information if a joystick mini-driver is present that will provide the data and store the raw value in **dwRpos**. Return zero otherwise.
JOY_CAL_READXONLY	Read the x-coordinate and store the raw (uncalibrated) value in **dwXpos**.
JOY_CAL_READXYONLY	Reads the x- and y-coordinates and place the raw values in **dwXpos** and **dwYpos**.
JOY_CAL_READYONLY	Reads the y-coordinate and store the raw value in **dwYpos**.
JOY_CAL_READZONLY	Read the z-coordinate and store the raw value in **dwZpos**.

JOY_CAL_READUONLY	Read the u-coordinate if a joystick mini-driver is present that will provide the data and store the raw value in **dwUpos**. Return zero otherwise.
JOY_CAL_READVONLY	Read the v-coordinate if a joystick mini-driver is present that will provide the data and store the raw value in **dwVpos**. Return zero otherwise.

dwRpos
Current position of the rudder or fourth joystick axis.

dwButtons
Current state of the 32 joystick buttons. The value of this member can be set to any combination of JOY_BUTTON*n* flags, where *n* is a value in the range of 1 through 32 corresponding to the button that is pressed.

dwButtonNumber
Current button number that is pressed.

dwPOV
Current position of the point-of-view control. Values for this member are in the range 0 through 35,900. These values represent the angle, in degrees, of each view multiplied by 100.

The value of the **dwSize** member is also used to identify the version number for the structure when it's passed to the **joyGetPosEx** function.

Most devices with a point-of-view control have only five positions. When the JOY_RETURNPOV flag is set, these positions are reported by using the following constants:

Point-of-view flags	Description
JOY_POVBACKWARD	Point-of-view hat is pressed backward. The value 18,000 represents an orientation of 180.00 degrees (to the rear).
JOY_POVCENTERED	Point-of-view hat is in the neutral position. The value -1 means the point-of-view hat has no angle to report.
JOY_POVFORWARD	Point-of-view hat is pressed forward. The value 0 represents an orientation of 0.00 degrees (straight ahead).
JOY_POVLEFT	Point-of-view hat is being pressed to the left. The value 27,000 represents an orientation of 270.00 degrees (90.00 degrees to the left).
JOY_POVRIGHT	Point-of-view hat is pressed to the right. The value 9,000 represents an orientation of 90.00 degrees (to the right).

The default Windows 95 joystick driver currently supports these five discrete directions. If an application can accept only the defined point-of-view values, it must use the JOY_RETURNPOV flag. If an application can accept other degree readings, it should use the JOY_RETURNPOVCTS flag to obtain continuous data if it is available. The JOY_RETURNPOVCTS flag also supports the JOY_POV

constants used with the JOY_RETURNPOV flag.

File Input and Output

Most multimedia applications require file input and output (I/O) – that is, the ability to create, read, and write disk files. 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.

Most applications need only the basic file I/O services and the RIFF file I/O services. Applications sensitive to file I/O performance, such as applications that stream data from compact disc - read-only memory (CD-ROM) in real time, can optimize performance by using services to directly access the file I/O buffer. Applications that access custom storage systems, such as file archives and databases, can provide their own I/O procedure that reads and writes elements of the storage system.

The multimedia file I/O services provide more functionality than the standard Microsoft Windows operating system services, including support for buffered I/O, RIFF files, memory files, and custom storage systems. In addition, the multimedia file I/O services are optimized for applications sensitive to performance.

File Input and Output Services

This section describes procedures for using the following multimedia file I/O services:

- Basic
- Buffered
- RIFF
- Custom

Basic Services

Using the basic I/O services is similar to using the run-time file I/O services of the C development system. Files must be opened before they can be read or written. After reading or writing, the file must be closed. You can also change the current read or write location within an open file.

Before you begin any I/O operations to a file, you must open the file by using the **mmioOpen** function. This function returns a file handle of type **HMMIO**. You can use this file handle to identify the open file when calling other file I/O functions.

Note An **HMMIO** file handle is not a standard file handle. Do not use **HMMIO** file handles with MS-DOS, standard Windows, or C run-time file I/O functions.

When you use **mmioOpen** to open a file, you use a flag to specify whether you are opening it for reading, writing, or both. You can also specify flags that enable you to create or delete a file. Use the **mmioClose** function to close a file when you are finished reading or writing to it.

You can read and write files by using the **mmioRead** and **mmioWrite** functions respectively. The next read or write operation occurs at the current file position or file pointer in a file. The current file position is advanced each time a file is read or written.

You can also change the current file position by using the **mmioSeek** function. You should ensure that you seek to a valid location in a file; if you seek to an invalid location, such as past the end of the file, **mmioSeek** might not return an error, but subsequent I/O operations might fail.

There are flags you can use with the **mmioOpen** function for operations beyond basic file I/O. By specifying an **MMIOINFO** structure, for example, you can open memory files, specify a custom I/O procedure, or supply a buffer for buffered I/O.

Buffered Services

Most of the overhead in file I/O occurs when accessing the media device. If you are reading or writing many small blocks of information, the device can spend a lot of time seeking to the physical location on the media for each read or write operation. In this case, you can achieve better performance by using buffered file I/O services. With buffered I/O, the file I/O manager maintains an intermediate buffer larger than the blocks of information you are reading or writing. It accesses the device only when the buffer must be filled from or written to the disk.

Before you set up and use buffered file I/O, you must decide whether you want the file I/O manager or the application to allocate the buffer. It is simpler to let the file I/O manager allocate the buffer; however, you can let the application allocate the buffer if you want to directly access the buffer or open a memory file. For more information about using memory files, see "Performing Memory File Input and Output" later in this chapter. For an example of directly accessing an I/O buffer, see "Accessing a File I/O Buffer" later in this chapter.

A buffer allocated by the file I/O manager is called an internal I/O buffer. To open a file for buffered I/O using an internal buffer, specify the MMIO_ALLOCBUF flag with the **mmioOpen** function when you open the file. After a file is opened for buffered I/O, the buffer is essentially transparent to the application; you can read, write, and seek the same way as with unbuffered I/O.

The default size of the internal I/O buffer is 8K. If this size is not adequate, you can use the **mmioSetBuffer** function to change it. You can also use this function to enable buffering on a file opened for unbuffered I/O, or to supply your own buffer for use as a memory file.

You can force the contents of an I/O buffer to be written to disk by using the **mmioFlush** function. When you close a file by using the **mmioClose** function, however, you don't have to call **mmioFlush** to flush an I/O buffer; the **mmioClose** function automatically flushes it. If you run out of disk space, **mmioFlush** might fail, even if the preceding calls of the **mmioWrite** function were successful. Similarly, **mmioClose** might fail when it is flushing its I/O buffer.

Applications that are performance-sensitive – such as those that stream data in real time from a CD-ROM – can optimize file I/O performance by directly accessing the I/O buffer. You should be careful if you choose to do this because you bypass some of the safeguards and error checking provided by the file I/O manager.

The multimedia file I/O manager uses the **MMIOINFO** structure to maintain state information about an open file. You use three members in this structure to read and write the I/O buffer: **pchNext**, **pchEndRead**, and **pchEndWrite**. The **pchNext** member points to the next location in the buffer to read or write. You must increment this member as you read and write the buffer. The **pchEndRead** member identifies the last valid character you can read from the buffer. Likewise, this member identifies the last location in the buffer you can write. More precisely, both **pchEndRead** and **pchEndWrite** point to the memory location that follows the last valid data in the buffer. Use the **mmioGetInfo** and **mmioSetInfo** functions to retrieve and set state information about the file I/O buffer.

When you reach the end of the I/O buffer, you must advance the buffer to fill it from the disk (if you are reading) or flush it to the disk (if you are writing). Use the **mmioAdvance** function to advance an I/O buffer. To fill an I/O buffer from disk, use **mmioAdvance** with the MMIO_READ flag. If there is not enough data remaining in the file to fill the buffer, the **pchEndRead** member of the **MMIOINFO** structure points to the location following the last valid byte in the buffer. To flush a buffer to disk, set the MMIO_DIRTY flag in the **dwFlags** member of the **MMIOINFO** structure, and then call **mmioAdvance** with the MMIO_WRITE flag.

Resource Interchange File Format Services

The preferred format for multimedia files is RIFF. The RIFF file I/O functions work with the basic buffered and unbuffered file I/O services. You can open, read, and write RIFF files in the same way as other file types. For detailed information about RIFF, see Chapter 6, "AVIFile Functions and Macros ."

RIFF files use four-character codes to identify file elements. These codes are 32-bit quantities representing a sequence of one to four ASCII alphanumeric characters, padded on the right with space characters. The data type for four-character codes is **FOURCC**. Use the **mmioFOURCC** macro to convert four characters into a four-character code. To convert a null-terminated string into a four-character code, use the **mmioStringToFOURCC** function.

The basic building block of a RIFF file is a *chunk*. A chunk is a logical unit of multimedia data, such as a single frame in a video clip. Each chunk contains the following fields:

- A four-character code specifying the chunk identifier
- A doubleword value specifying the size of the data member in the chunk
- A data field

A chunk contained in another chunk is a *subchunk*. The only chunks allowed to contain subchunks are those with a chunk identifier of "RIFF" or "LIST". A chunk that contains another chunk is called a *parent chunk*. The first chunk in a RIFF file must be a "RIFF" chunk. All other chunks in the file are subchunks of the "RIFF" chunk.

"RIFF" chunks include an additional field in the first four bytes of the data field. This additional field provides the *form type* of the field. The form type is a four-character code identifying the format of the data stored in the file. For example, Microsoft waveform-audio files have a form type of "WAVE".

"LIST" chunks also include an additional field in the first four bytes of the data field. This additional field contains the *list type* of the field. The list type is a four-character code identifying the contents of the list. For example, a "LIST" chunk with a list type of "INFO" can contain "ICOP" and "ICRD" chunks providing copyright and creation date information.

Multimedia file I/O services include two functions you can use to navigate among chunks in a RIFF file: **mmioAscend** and **mmioDescend**. You can use these functions as high-level seek functions. When you descend into a chunk, the file position is set to the data field of the chunk (8 bytes from the beginning of the chunk). For "RIFF" and "LIST" chunks, the file position is set to the location following the form type or list type (12 bytes from the beginning of the chunk). When you ascend out of a chunk, the file position is set to the location following the end of the chunk. To create a new chunk, use the **mmioCreateChunk** function to write a chunk header at the current position in an open file. The **mmioAscend**, **mmioDescend**, and **mmioCreateChunk** functions use the **MMCKINFO** structure to specify and retrieve information about "RIFF" chunks.

Custom Services

Multimedia file I/O services use I/O procedures to handle the physical input and output associated with reading and writing to different types of storage systems, such as file-archival systems and database-storage systems. Predefined I/O procedures exist for the standard file systems and for memory files, but you can supply a custom I/O procedure for accessing a unique storage system by using the **mmioInstallIOProc** function.

To open a file by using a custom I/O procedure, use the **mmioOpen** function. Include a plus sign (+) or the CFSEPCHAR constant in the filename to separate the name of the physical file from the name of the element of the file you want to open. The following example opens a file element named "element" from a file named FILENAME.ARC:

```
mmioOpen("filename.arc+element", NULL, MMIO_READ);
```

When the file I/O manager encounters a plus sign in a filename, it examines the filename extension to determine which I/O procedure to associate with the file. In the previous example, the file I/O manager would attempt to use the I/O procedure associated with the .ARC filename extension; this I/O procedure would have been installed by using **mmioInstallIOProc**. If no I/O procedure is installed, **mmioOpen** returns an error.

I/O procedures must respond to the MMIOM_CLOSE, MMIOM_OPEN, MMIOM_READ, MMIOM_WRITE, MMIOM_SEEK, MMIOM_RENAME, and MMIOM_WRITEFLUSH messages. You can also create custom messages and send them to your I/O procedure by using the **mmioSendMessage** function. If you define your own messages, make sure they are defined at or above the value defined by the MMIOM_USER constant.

In addition to processing messages, an I/O procedure must maintain the **lDiskOffset** member of the **MMIOINFO** structure (pointed to by the *lpmmioinfo* parameter of the **mmioOpen** function). The **lDiskOffset** member must always contain the file offset to the location that the next MMIOM_READ or MMIOM_WRITE message will access. The offset is specified in bytes and is relative to the beginning of the file. The I/O procedure can use the **adwInfo** member to maintain any required state information. The I/O procedure should not modify any other members in the **MMIOINFO** structure.

Using File Input and Output

This section contains examples demonstrating how to perform the following tasks:

- Open a file.
- Create and delete a file.
- Seek to a new position in a file.
- Change the I/O buffer size.
- Access a file I/O buffer.
- Generate four-character codes.
- Create a "RIFF" chunk.
- Search for a "RIFF" chunk.
- Search for a subchunk.
- Perform file I/O on RIFF files.
- Perform memory file input and output.
- Install custom I/O procedures.
- Share I/O procedures with other applications.

Using mioOpen to Open a File

To open a file for basic I/O operations, set the *lpmmioinfo* parameter of the **mmioOpen** function to NULL. The following example opens a file named "C:\SAMPLES\SAMPLE1.TXT" for reading, and checks the return value for error.

```
HMMIO hFile;
.
.
.
if ((hFile = mmioOpen("C:\\SAMPLES\\SAMPLE1.TXT", NULL,
    MMIO_READ)) != NULL)
    // File opened successfully.
else
    // File cannot be opened.
```

Use the *dwFlags* parameter of **mmioOpen** to specify flags for opening a file.

Creating and Deleting a File

To create a file, set the *dwOpenFlags* parameter of the **mmioOpen** function to MMIO_CREATE. The following example creates a file and opens it for reading and writing.

```
HMMIO hFile;
.
.
.
hFile = mmioOpen("NEWFILE.TXT", NULL, MMIO_CREATE | MMIO_READWRITE);
if (hFile != NULL)
    // File created successfully.
else
    // File cannot be created.
```

If the file you are creating already exists, it will be truncated to zero length.

To delete a file, set the *dwOpenFlags* parameter of the **mmioOpen** function to MMIO_DELETE. After you delete a file, it cannot be recovered by any standard means. If your application is deleting a file at the request of a user, query the user before deleting the file to make sure the user wants to delete it.

Seeking to a New Position in a File

The following example seeks to the beginning of an open file.

```
mmioSeek(hFile, 0L, SEEK_SET);
```

The following example seeks to the end of an open file.

```
mmioSeek(hFile, 0L, SEEK_END);
```

The following example seeks to a position 10 bytes from the end of an open file.

```
mmioSeek(hFile, -10L, SEEK_END);
```

Changing the I/O Buffer Size

The following example opens a file named "SAMPLE.TXT" for unbuffered I/O, and then enables buffered I/O with an internal 16K buffer.

```
HMMIO hFile;
.
.
.
if ((hFile = mmioOpen("SAMPLE.TXT", NULL, MMIO_READ)) != NULL)
{
    // File opened successfully; request an I/O buffer.
    if (mmioSetBuffer(hFile, NULL, 16384L, 0))
        // Buffer cannot be allocated.
    else
        // Buffer allocated successfully.
}
else
    // File cannot be opened.
```

Accessing a File I/O Buffer

The following example accesses an I/O buffer directly to read data from a waveform-audio file.

```
HMMIO    hmmio;
MMIOINFO mmioinfo;
DWORD    dwDataSize;
DWORD    dwCount;
HPSTR    hptr;
.
.
.
// Get information about the file I/O buffer.
if (mmioGetInfo(hmmio, &mmioinfo, 0)) {
    Error("Failed to get I/O buffer info.");
    .
    .
    .
    mmioClose(hmmio, 0);
    return;
}

// Read the entire file by directly reading the file I/O buffer.
// When the end of the I/O buffer is reached, advance the buffer.
for (dwCount = dwDataSize, hptr = lpData; dwCount  0; dwCount--)
{
    // Check to see if the I/O buffer must be advanced.
    if (mmioinfo.pchNext == mmioinfo.pchEndRead)
    {
        if(mmioAdvance(hmmio, &mmioinfo, MMIO_READ))
        {
            Error("Failed to advance buffer.");
            .
            .
            .
            mmioClose(hmmio, 0);
            return;
        }
    }

    // Get a character from the buffer.
    *hptr++ = *mmioinfo.pchNext++;
}

// End direct buffer access and close the file.
mmioSetInfo(hmmio, &mmioinfo, 0);
mmioClose(hmmio, 0);
```

When you finish accessing a file I/O buffer, call the **mmioSetInfo** function, passing an address of the **MMIOINFO** structure filled by the **mmioGetInfo** function. If you wrote to the buffer, set the MMIO_DIRTY flag in the **dwFlags** member of the **MMIOINFO** structure before calling **mmioSetInfo**. Otherwise, the buffer will not be flushed to disk.

Generating Four-Character Codes

You can use the **mmioFOURCC** macro or the **mmioStringToFOURCC** function to generate four-character codes. The following example uses **mmioFOURCC** to generate a four-character code for "WAVE".

```
FOURCC fourccID;
.
.
.
fourccID = mmioFOURCC('W', 'A', 'V', 'E');
```

The following example uses **mmioStringToFOURCC** to generate a four-character code for "WAVE".

```
FOURCC fourccID;
.
.
.
fourccID = mmioStringToFOURCC("WAVE", 0);
```

The second parameter in **mmioStringToFOURCC** specifies flags for converting the string to a four-character code. If you specify the MMIO_TOUPPER flag, **mmioStringToFOURCC** converts all alphabetic characters in the string to uppercase. This is useful when you need to specify a four-character code to identify a custom I/O procedure because four-character codes identifying file-extension names must be all uppercase.

Creating a "RIFF" Chunk

The following example creates a chunk with a chunk identifier of "RIFF" and a form type of "RDIB".

```
HMMIO    hmmio;
MMCKINFO mmckinfo;
.
.
.
mmckinfo.fccType = mmioFOURCC('R', 'D', 'I', 'B');
mmioCreateChunk(hmmio, &mmckinfo, MMIO_CREATERIFF);
```

If you're creating a "RIFF" or "LIST" chunk, you must specify the form type or list type in the **fccType** member of the **MMCKINFO** structure. In the previous example, the form type is "RDIB".

If you know the size of the data field in a new chunk, you can set the **cksize** member of the **MMCKINFO** structure when you create the chunk. This value will be written to the data size field in the new chunk. If this value is not correct when you call **mmioAscend** to mark the end of the chunk, it will be automatically rewritten to reflect the correct size of the data field.

After you create a chunk by using the **mmioCreateChunk** function, the file position is set to the data field of the chunk (8 bytes from the beginning of the chunk). If the chunk is a "RIFF" or "LIST" chunk, the file position is set to the location following the form type or list type (12 bytes from the beginning of the chunk).

Searching for a "RIFF" Chunk

The following example searches for a "RIFF" chunk with a form type of "WAVE" to verify that the file that has just been opened is a waveform-audio file.

```
HMMIO    hmmio;
MMCKINFO mmckinfoParent;
MMCKINFO mmckinfoSubchunk;
.
.
.
// Locate a "RIFF" chunk with a "WAVE" form type to make
// sure the file is a waveform-audio file.
mmckinfoParent.fccType = mmioFOURCC('W', 'A', 'V', 'E');
if (mmioDescend(hmmio, (LPMMCKINFO) &mmckinfoParent, NULL,
    MMIO_FINDRIFF))
    // The file is not a waveform-audio file.
else
    // The file is a waveform-audio file
```

Searching for a Subchunk

The following example searches for the "FMT" chunk in the "RIFF" chunk of the previous example.

```
// Find the format chunk (form type "FMT"); it should be
// a subchunk of the "RIFF" parent chunk.
mmckinfoSubchunk.ckid = mmioFOURCC('f', 'm', 't', ' ');
if (mmioDescend(hmmio, &mmckinfoSubchunk, &mmckinfoParent,
    MMIO_FINDCHUNK))
    // Error, cannot find the "FMT" chunk.
else
    // "FMT" chunk found.
```

To search for a subchunk (that is, any chunk other than a "RIFF" or "LIST" chunk), identify its parent chunk in the *lpckParent* parameter of the **mmioDescend** function.

If you do not specify a parent chunk, the current file position should be at the beginning of a chunk before you call the **mmioDescend** function. If you do specify a parent chunk, the current file position can be anywhere in that chunk.

If the search for a subchunk fails, the current file position is undefined. You can use the **mmioSeek** function and the **dwDataOffset** member of the **MMCKINFO** structure describing the parent chunk to seek back to the beginning of the parent chunk, as in the following example:

```
mmioSeek(hmmio, mmckinfoParent.dwDataOffset + 4, SEEK_SET);
```

Because **dwDataOffset** specifies the offset to the beginning of the data portion of the chunk, you must seek 4 bytes past **dwDataOffset** to set the file position after the form type.

Performing File Input and Output on RIFF Files

The following example shows how to open a RIFF file for buffered I/O, as well as how to descend, ascend, and read "RIFF" chunks.

```
// ReversePlay--Plays a waveform-audio file backward.
void ReversePlay()
{
    char        szFileName[128];    // filename of file to open
    HMMIO       hmmio;              // file handle for open file
    MMCKINFO    mmckinfoParent;     // parent chunk information
    MMCKINFO    mmckinfoSubchunk;   // subchunk information structure
    DWORD       dwFmtSize;          // size of "FMT" chunk
    DWORD       dwDataSize;         // size of "DATA" chunk
    WAVEFORMAT  *pFormat;           // address of "FMT" chunk
    HPSTR       lpData;             // address of "DATA" chunk

    // Get the filename from the edit control.
    .
    .
    .
    // Open the file for reading with buffered I/O
    // by using the default internal buffer
    if(!(hmmio = mmioOpen(szFileName, NULL,
        MMIO_READ | MMIO_ALLOCBUF)))
    {
        Error("Failed to open file.");
        return;
    }

    // Locate a "RIFF" chunk with a "WAVE" form type to make
    // sure the file is a waveform-audio file.
    mmckinfoParent.fccType = mmioFOURCC('W', 'A', 'V', 'E');
    if (mmioDescend(hmmio, (LPMMCKINFO) &mmckinfoParent, NULL,
        MMIO_FINDRIFF))
    {
        Error("This is not a waveform-audio file.");
        mmioClose(hmmio, 0);
        return;
    }
    // Find the "FMT" chunk (form type "FMT"); it must be
    // a subchunk of the "RIFF" chunk.
    mmckinfoSubchunk.ckid = mmioFOURCC('f', 'm', 't', ' ');
    if (mmioDescend(hmmio, &mmckinfoSubchunk, &mmckinfoParent,
        MMIO_FINDCHUNK))
    {
        Error("Waveform-audio file has no "FMT" chunk.");
        mmioClose(hmmio, 0);
        return;
    }

    // Get the size of the "FMT" chunk. Allocate
    // and lock memory for it.
    dwFmtSize = mmckinfoSubchunk.cksize;
    .
```

```
.
.
// Read the "FMT" chunk.
if (mmioRead(hmmio, (HPSTR) pFormat, dwFmtSize) != dwFmtSize){
    Error("Failed to read format chunk.");
    .
    .
    .
    mmioClose(hmmio, 0);
    return;
}

// Ascend out of the "FMT" subchunk.
mmioAscend(hmmio, &mmckinfoSubchunk 0);

// Find the data subchunk. The current file position should be at
// the beginning of the data chunk; however, you should not make
// this assumption. Use mmioDescend to locate the data chunk.
mmckinfoSubchunk.ckid = mmioFOURCC('d', 'a', 't', 'a');
if (mmioDescend(hmmio, &mmckinfoSubchunk, &mmckinfoParent,
    MMIO_FINDCHUNK))
{
    Error("Waveform-audio file has no data chunk.");
    .
    .
    .
    mmioClose(hmmio, 0);
    return;
}

// Get the size of the data subchunk.
dwDataSize = mmckinfoSubchunk.cksize;
if (dwDataSize == 0L){
    Error("The data chunk contains no data.");
    .
    .
    .
    mmioClose(hmmio, 0);
    return;
}

// Open a waveform-audio output device.
.
.
.

// Allocate and lock memory for the waveform-audio data.
.
.
.

// Read the waveform-audio data subchunk.
if(mmioRead(hmmio, (HPSTR) lpData, dwDataSize) != dwDataSize){
    Error("Failed to read data chunk.");
    .
```

```
        .
        .
        mmioClose(hmmio, 0);
        return;
    }

    // Close the file.
    mmioClose(hmmio, 0);

    // Reverse the sound and play it.
    .
    .
    .
}
```

Performing Memory File Input and Output

The multimedia file I/O services let you to treat a block of memory as a file. This can be useful if you already have a file image in memory. Memory files let you reduce the number of special-case conditions in your code because, for I/O purposes, you can treat memory files as if they were disk-based files. You can also use memory files with the clipboard.

As with I/O buffers, memory files can use memory allocated by the application or by the file I/O manager. In addition, memory files can be either expandable or nonexpandable. When the file I/O manager reaches the end of an expandable memory file, it expands the memory file by a predefined increment.

To open a memory file, use the **mmioOpen** function with the *szFilename* parameter set to NULL and the MMIO_READWRITE flag set in the *dwOpenFlags* parameter. Set the *lpmmioinfo* parameter to point to an **MMIOINFO** structure that has been set up as follows:

1. Set the **pIOProc** member to NULL.
2. Set the **fccIOProc** member to FOURCC_MEM.
3. Set the **pchBuffer** member to point to the memory block. To request that the file I/O manager allocate the memory block, set **pchBuffer** to NULL.
4. Set the **cchBuffer** member to the initial size of the memory block.
5. Set the **adwInfo** member to the minimum expansion size of the memory block. For a nonexpandable memory file, set **adwInfo** to NULL.
6. Set all other members to zero.

There are no restrictions on allocating memory for use as a nonexpandable memory file.

Installing Custom I/O Procedures

To install an I/O procedure associated with the .ARC filename extension, use the **mmioInstallIOProc** function as follows:

```
mmioInstallIOProc (mmioFOURCC('A', 'R', 'C', ' '),
    (LPMMIOPROC)lpmmioproc, MMIO_INSTALLPROC);
```

When you install an I/O procedure using **mmioInstallIOProc**, the procedure remains installed until you remove it. The I/O procedure is used for any file you open as long as the file has the appropriate filename extension.

You can also temporarily install an I/O procedure by using the **mmioOpen** function. In this case, the I/O procedure is used only with a file opened by using **mmioOpen** and is removed when the file is closed by using the **mmioClose** function. To specify an I/O procedure when you open a file by using **mmioOpen**, use the *lpmmioinfo* parameter to reference an **MMIOINFO** structure as follows:

1. Set the **fccIOProc** member to NULL.
2. Set the **pIOProc** member to the procedure-instance address of the I/O procedure.
3. Set all other members to zero (unless you are opening a memory file, or directly reading or writing to the file I/O buffer).

Be sure to remove any I/O procedures you have installed before you exit your application.

Sharing an I/O Procedure with Other Applications

If you want to share an I/O procedure with other applications, you need to write a dynamic-link library (DLL) for your application. You can share the I/O procedure by doing one of the following:

- Include the code for the I/O procedure in the DLL.
- Create a function in the DLL that calls the **mmioInstallIOProc** function to install the I/O procedure.
- Export this function in the module-definitions file of the DLL.

To use the shared I/O procedure, an application must first call the function in the DLL to install the I/O procedure.

File Input and Output Reference

This section describes the functions, macros, messages, and structures associated with multimedia file input and output. These elements are grouped as follows.

Basic I/O

mmioClose
mmioOpen
mmioRead
mmioRename
mmioSeek
mmioWrite

Buffered I/O

mmioAdvance
mmioFlush
mmioGetInfo
MMIOINFO
mmioSetBuffer
mmioSetInfo

RIFF I/O

mmioAscend
MMCKINFO
mmioCreateChunk
mmioDescend
mmioFOURCC
mmioStringToFOURCC

Custom I/O Procedures

IOProc
mmioInstallIOProc
MMIOM_CLOSE
MMIOM_OPEN
MMIOM_READ
MMIOM_RENAME
MMIOM_SEEK
MMIOM_WRITE
MMIOM_WRITEFLUSH
mmioSendMessage

IOProc

```
LRESULT FAR PASCAL IOProc(LPSTR lpmmioinfo, UINT wMsg, LPARAM lParam1,
    LPARAM lParam2);
```

Accesses a unique storage system, such as a database or file archive. You can install or remove this callback function by using the **mmioInstallIOProc** function.

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

- Returns a value that corresponds to the message specified by *wMsg*. If the I/O procedure does not recognize a message, it should return zero.

lpmmioinfo
Address of an **MMIOINFO** structure containing information about the open file. The I/O procedure must maintain the **lDiskOffset** member in this structure to indicate the file offset to the next read or write location. The I/O procedure can use the **adwInfo** member to store state information. The I/O procedure should not modify any other members of the **MMIOINFO** structure.

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

lParam1 and *lParam2*
Parameters for the message.

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

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

The **mmioInstallIOProc** function does 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 or MEM). The most recently installed procedure takes precedence, and the most recently installed procedure is the first one to be removed.

When searching for a specified I/O procedure, local procedures are searched first, then global procedures.

mmioAdvance

```
MMRESULT mmioAdvance(HMMIO hmmio, LPMMIOINFO lpmmioinfo, UINT wFlags);
```

Advances the I/O buffer of a file set up for direct I/O buffer access with the **mmioGetInfo** function.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMIOERR_CANNOTEXPAND	The specified memory file cannot be expanded, probably because the **adwInfo** member of the **MMIOINFO** structure was set to zero in the initial call to the **mmioOpen** function.
MMIOERR_CANNOTREAD	An error occurred while refilling the buffer.
MMIOERR_CANNOTWRITE	The contents of the buffer could not be written to disk.
MMIOERR_OUTOFMEMORY	There was not enough memory to expand a memory file for further writing.
MMIOERR_UNBUFFERED	The specified file is not opened for buffered I/O.

hmmio
File handle of a file opened by using the **mmioOpen** function.

lpmmioinfo
Address of the **MMIOINFO** structure obtained by using the **mmioGetInfo** function. This structure is used to set the current file information, and then it is updated after the buffer is advanced. This parameter is optional.

wFlags
Flags for the operation. It can be one of the following:

MMIO_READ
Buffer is filled from the file.

MMIO_WRITE
Buffer is written to the file.

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** member of the **MMIOINFO** structure, the buffer is written to disk. The **pchNext**, **pchEndRead**, and **pchEndWrite** members of the **MMIOINFO** structure are updated to reflect the new state of the I/O buffer.

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

If the specified file is open only 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** member of the **MMIOINFO** structure before calling **mmioAdvance**. Otherwise, the buffer will not be written to disk.

If the end of file is reached, **mmioAdvance** still returns successfully even though no more data can be read. To check for the end of the file, check if the **pchNext** and **pchEndRead** members of the **MMIOINFO** structure are equal after calling **mmioAdvance**.

mmioAscend

```
MMRESULT mmioAscend(HMMIO hmmio, LPMMCKINFO lpck, UINT wFlags);
```

Ascends out of a chunk in a RIFF file descended into with the **mmioDescend** function or created with the **mmioCreateChunk** function.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMIOERR_CANNOTSEEK	There was an error while seeking to the end of the chunk.
MMIOERR_CANNOTWRITE	The contents of the buffer could not be written to disk.

hmmio
File handle of an open RIFF file.

lpck
Address of an application-defined **MMCKINFO** structure previously filled by the **mmioDescend** or **mmioCreateChunk** function.

wFlags
Reserved; must be zero.

If the chunk was descended into by using **mmioDescend**, **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 by using **mmioCreateChunk**, or if the MMIO_DIRTY flag is set in the **dwFlags** member of the **MMCKINFO** structure referenced by *lpck*, 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** member of the **MMCKINFO** structure when **mmioCreateChunk** was called, **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).

mmioClose

```
MMRESULT mmioClose(HMMIO hmmio, UINT wFlags);
```

Closes a file that was opened by using the **mmioOpen** function.

- Returns zero if successful or an error otherwise. The error value can originate from the **mmioFlush** function or from the I/O procedure. Possible error values include the following:

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

hmmio
File handle of the file to close.

wFlags
Flags for the close operation. The following value is defined:

MMIO_FHOPEN
If the file was opened by passing a file handle whose type is not **HMMIO**, using this flag tells the **mmioClose** function to close the multimedia file handle, but not the standard file handle.

mmioCreateChunk

```
MMRESULT mmioCreateChunk(HMMIO hmmio, LPMMCKINFO lpck, UINT wFlags);
```

Creates a chunk in a RIFF file that was opened by using the **mmioOpen** function. 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.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMIOERR_CANNOTSEEK	Unable to determine offset of the data portion of the chunk.
MMIOERR_CANNOTWRITE	Unable to write the chunk header.

hmmio
File handle of an open RIFF file.

lpck
Address an application-defined **MMCKINFO** structure containing information about the chunk to be created.

wFlags
Flags identifying what type of chunk to create. The following values are defined:

MMIO_CREATELIST
"LIST" chunk.

MMIO_CREATERIFF
"RIFF" chunk.

This function cannot insert a chunk into the middle of a file. If an application attempts to create a chunk somewhere other than at the end of a file, **mmioCreateChunk** overwrites existing information in the file.

The **MMCKINFO** structure pointed to by the *lpck* parameter should be set up as follows:

- The **ckid** member specifies the chunk identifier. If *wFlags* includes MMIO_CREATERIFF or MMIO_CREATELIST, this member will be filled by **mmioCreateChunk**.
- The **cksize** member 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 the **mmioAscend** function is called to mark the end of the chunk, **mmioAscend** corrects the chunk size.
- The **fccType** member 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 member does not need to be filled in.
- The **dwDataOffset** member does not need to be filled in. The **mmioCreateChunk** function fills this member with the file offset of the data portion of the chunk.
- The **dwFlags** member does not need to be filled in. The **mmioCreateChunk** function sets the MMIO_DIRTY flag in **dwFlags**.

mmioDescend

```
MMRESULT mmioDescend(HMMIO hmmio, LPMMCKINFO lpck,
    LPMMCKINFO lpckParent, UINT wFlags);
```

Descends into a chunk of a RIFF file that was opened by using the **mmioOpen** function. It can also search for a given chunk.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMIOERR_CHUNKNOTFOUND	The end of the file (or the end of the parent chunk, if given) was reached before the desired chunk was found.

hmmio
File handle of an open RIFF file.

lpck
Address an application-defined **MMCKINFO** structure.

lpckParent
Address of an optional application-defined **MMCKINFO** structure identifying the parent of the chunk being searched for. If this parameter is not NULL, **mmioDescend** assumes the **MMCKINFO** structure it refers to was filled when **mmioDescend** was called to descend into the parent chunk, and **mmioDescend** searches for a chunk within the parent chunk. Set this parameter to NULL if no parent chunk is being specified.

wFlags
Search flags. If no flags are specified, **mmioDescend** descends into the chunk beginning at the current file position. The following values are defined:

MMIO_FINDCHUNK
Searches for a chunk with the specified chunk identifier.

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

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

A "RIFF" chunk consists of a four-byte chunk identifier (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 identifier 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 you use **mmioDescend** to search for a chunk, make sure the file position is at the beginning of a chunk before calling the function. 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).

The **mmioDescend** function fills the **MMCKINFO** structure pointed to by the *lpck* parameter with the following information:

- The **ckid** member is the chunk. If the MMIO_FINDCHUNK, MMIO_FINDRIFF, or MMIO_FINDLIST flag is specified for *wFlags*, the **MMCKINFO** structure is also used to pass parameters to

mmioDescend. In this case, the **ckid** member specifies the four-character code of the chunk identifier, form type, or list type to search for.

- The **cksize** member is the size, in bytes, of the data portion of the chunk. The 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** member is the form type if **ckid** is "RIFF", or the list type if **ckid** is "LIST". Otherwise, it is NULL.
- The **dwDataOffset** member 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, this member is the offset of the form type or list type.
- The **dwFlags** member contains other information about the chunk. Currently, this information is not used and is set to zero.

mmioFlush

```
MMRESULT mmioFlush(HMMIO hmmio, UINT fuFlush);
```

Writes the I/O buffer of a file to disk if the buffer has been written to.

- Returns zero if successful or an error otherwise. The following value is defined:

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

hmmio

File handle of a file opened by using the **mmioOpen** function.

fuFlush

Flag determining how the flush is carried out. It can be zero or the following:

MMIO_EMPTYBUF Empties the buffer after writing it to the disk.

Closing a file with the **mmioClose** function automatically flushes its buffer.

If there is insufficient disk space to write the buffer, **mmioFlush** fails, even if the preceding calls of the **mmioWrite** function were successful.

mmioFOURCC

```
FOURCC mmioFOURCC(CHAR ch0, CHAR ch1, CHAR ch2, CHAR ch3)
```

Converts four characters into a four-character code.

- Returns the four-character code created from the given characters.

ch0, *ch1*, *ch2*, and *ch3*
First, second, third, and fourth characters of the four-character code.

This macro does not check whether the four-character code it returns is valid.

The **mmioFOURCC** macro is defined as follows:

```
#define mmioFOURCC(ch0, ch1, ch2, ch3)  MAKEFOURCC(ch0, ch1, ch2, ch3);
```

The **MAKEFOURCC** macro, in turn, is defined as follows:

```
#define MAKEFOURCC(ch0, ch1, ch2, ch3)  \
    ((DWORD)(BYTE)(ch0) | ((DWORD)(BYTE)(ch1) << 8) |  \
    ((DWORD)(BYTE)(ch2) << 16) | ((DWORD)(BYTE)(ch3) << 24 ));
```

mmioGetInfo

```
MMRESULT mmioGetInfo(HMMIO hmmio, LPMMIOINFO lpmmioinfo, UINT wFlags);
```

Retrieves information about a file opened by using the **mmioOpen** function. This information allows the application to directly access the I/O buffer, if the file is opened for buffered I/O.

- Returns zero if successful or an error otherwise.

hmmio
File handle of the file.

lpmmioinfo
Address an **MMIOINFO** structure that **mmioGetInfo** fills with information about the file.

wFlags
Reserved; must be zero.

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

- The **pchNext** member 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** member points to 1 byte past the last valid byte in the buffer that can be read.
- The **pchEndWrite** member points to 1 byte past the last location in the buffer that can be written.

After you read or write to the buffer and modify **pchNext**, do not call any multimedia file I/O functions except **mmioAdvance** until you call the **mmioSetInfo** function. Call **mmioSetInfo** when you are finished directly accessing the buffer.

When you reach the end of the buffer specified by the **pchEndRead** or **pchEndWrite** member, call **mmioAdvance** to fill the buffer from the disk or write the buffer to the disk. The **mmioAdvance** function updates the **pchNext**, **pchEndRead**, and **pchEndWrite** members 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 **dwFlags** member of the **MMIOINFO** structure for the file. Otherwise, the buffer will not be written to disk.

Do not decrement **pchNext** or modify any members in the **MMIOINFO** structure other than **pchNext** and **dwFlags**. Do not set any flags in **dwFlags** except MMIO_DIRTY.

mmioInstallIOProc

```
LPMMIOPROC mmioInstallIOProc(FOURCC fccIOProc, LPMMIOPROC pIOProc,
    DWORD dwFlags);
```

Installs or removes a custom I/O procedure. This function also locates an installed I/O procedure, using its corresponding four-character code.

- Returns the address of the I/O procedure installed, removed, or located. Returns NULL if there is an error.

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

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

dwFlags
Flag indicating whether the I/O procedure is being installed, removed, or located. The following values are defined:

MMIO_FINDPROC
Searches for the specified I/O procedure.

MMIO_GLOBALPROC
This flag is a modifier to the MMIO_INSTALLPROC flag and indicates the I/O procedure should be installed for global use. This flag is ignored if MMIO_FINDPROC or MMIO_REMOVEPROC is specified.

MMIO_INSTALLPROC
Installs the specified I/O procedure.

MMIO_REMOVEPROC
Removes the specified I/O procedure.

mmioOpen

```
HMMIO mmioOpen(LPSTR szFilename, LPMMIOINFO lpmmioinfo,
    DWORD dwOpenFlags);
```

Opens a file for unbuffered or buffered I/O. The file can be a standard file, a memory file, or an element of a custom storage system. The handle returned by **mmioOpen** is not a standard file handle; do not use it with any file I/O functions other than multimedia file I/O functions.

- Returns a handle of the opened file. If the file cannot be opened, the return value is NULL. If *lpmmioinfo* is not NULL, the **wErrorRet** member of the **MMIOINFO** structure will contain one of the following error values:

MMIOERR_ACCESSDENIED	The file is protected and cannot be opened.
MMIOERR_INVALIDFILE	Another failure condition occurred. This is the default error for an open-file failure.
MMIOERR_NETWORKERROR	The network is not responding to the request to open a remote file.
MMIOERR_PATHNOTFOUND	The directory specification is incorrect.
MMIOERR_SHARINGVIOLATION	The file is being used by another application and is unavailable.
MMIOERR_TOOMANYOPENFILES	The number of files simultaneously open is at a maximum level. The system has run out of available file handles.

szFilename

Address of a string containing the filename of the file to open. If no I/O procedure is specified to open the file, the filename determines how the file is opened, as follows:

- If the filename does not contain a plus sign (+), it is assumed to be the name of a standard file (that is, a file whose type is not **HMMIO**).
- If the filename is of the form EXAMPLE.EXT+ABC, the extension EXT is assumed to identify an installed I/O procedure which is called to perform I/O on the file. For more information, see **mmioInstallIOProc**.
- If the filename is NULL and no I/O procedure is given, the **adwInfo** member of the **MMIOINFO** structure is assumed to be the standard (non-**HMMIO**) file handle of a currently open file.

The filename should not be longer than 128 bytes, including the terminating NULL character.

When opening a memory file, set *szFilename* to NULL.

lpmmioinfo

Address of 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 this parameter is not NULL, all unused members of the **MMIOINFO** structure it references must be set to zero, including the reserved members.

dwOpenFlags

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 file-sharing flags. The following values are defined:

MMIO_ALLOCBUF

Opens a file for buffered I/O. To allocate a buffer larger or smaller than the default buffer size (8K, defined as MMIO_DEFAULTBUFFER), set the **cchBuffer** member of the **MMIOINFO** structure to the desired buffer size. If **cchBuffer** is zero, the default buffer size is used. If you are providing your own I/O buffer, this 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. If the file has been opened with any of the other sharing modes, **mmioOpen** fails.

MMIO_CREATE
Creates a new file. If the file already exists, it is truncated to zero length. For memory files, this flag 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 is TRUE (cast to **HMMIO**) if the file was deleted successfully or FALSE otherwise. Do not call the **mmioClose** function for a file that has been deleted. If this flag is specified, all other flags that open files are ignored.

MMIO_DENYNONE
Opens the file without denying other processes read or write access to the file. If the file has been opened in compatibility mode by any other process, **mmioOpen** fails.

MMIO_DENYREAD
Opens the file and denies other processes read access to the file. If the file has been opened in compatibility mode or for read access by any other process, **mmioOpen** fails.

MMIO_DENYWRITE
Opens the file and denies other processes write access to the file. If the file has been opened in compatibility mode or for write access by any other process, **mmioOpen** fails.

MMIO_EXCLUSIVE
Opens the file and denies other processes read and write access to the file. If the file has been opened in any other mode for read or write access, even by the current process, **mmioOpen** fails.

MMIO_EXIST
Determines whether the specified file exists and creates a fully qualified filename from the path specified in *szFilename*. The filename is placed back into *szFilename*. The return value is TRUE (cast to **HMMIO**) if the qualification was successful and the file exists or FALSE otherwise. The file is not opened, and the function does not return a valid multimedia file I/O file handle, so do not attempt to close the file.

MMIO_GETTEMP
Creates a temporary filename, optionally using the parameters passed in *szFilename.* For example, you can specify "C:F" to create a temporary file residing on drive C, starting with letter "F". The resulting filename is placed in the buffer pointed to by *szFilename*. The return value is MMSYSERR_NOERROR (cast to **HMMIO**) if the temporary filename was created successfully or MMIOERR_FILENOTFOUND otherwise. The file is not opened, and the function does not return a valid multimedia file I/O file handle, so do not attempt to close the file. This flag overrides all other flags.

MMIO_PARSE
Creates a fully qualified filename from the path specified in *szFilename*. The filename is placed back into *szFilename*. The return value is TRUE (cast to **HMMIO**) if the qualification was successful or FALSE otherwise. The file is not opened, and the function does not return a valid multimedia file I/O file handle, so do not attempt to close the file. If this flag is specified, all flags that open files are ignored.

MMIO_READ
Opens the file for reading only. This is the default if MMIO_WRITE and MMIO_READWRITE are not specified.

MMIO_READWRITE
Opens the file for reading and writing.

MMIO_WRITE
Opens the file for writing only.

If *lpmmioinfo* references an **MMIOINFO** structure, set up the members of that structure as described below. All unused members must be set to zero, including reserved members.

- To request that a file be opened with an installed I/O procedure, set **fccIOProc** to the four-character code of the I/O procedure, and set **pIOProc** to NULL.
- To request that a file be opened with an uninstalled I/O procedure, set **IOProc** 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 **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 **pchBuffer** to NULL, **fccIOProc** to FOURCC_MEM, **cchBuffer** to the initial size of the buffer, and **adwInfo** to the incremental expansion size of the buffer. This memory file will automatically be expanded in increments of the number of bytes specified in **adwInfo** 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 an application-supplied buffer, set **pchBuffer** to point to the memory buffer, **fccIOProc** to FOURCC_MEM, **cchBuffer** to the size of the buffer, and **adwInfo** to the incremental expansion size of the buffer. The expansion size in **adwInfo** should be nonzero only if **pchBuffer** is a pointer obtained by calling the **GlobalAlloc** and **GlobalLock** functions; in this case, the **GlobalReAlloc** function will be called to expand the buffer. In other words, if **pchBuffer** points to a local or global array, a block of memory in the local heap, or a block of memory allocated by the **GlobalDosAlloc** function, **adwInfo** 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 is considered readable.
- To use a currently open standard file handle (that is, a file handle that does not have the **HMMIO** type) with multimedia file I/O services, set **fccIOProc** to FOURCC_DOS, **pchBuffer** to NULL, and **adwInfo** to the standard file handle. Offsets within the file will be relative to the beginning of the file and are not related to the position in the standard file at the time **mmioOpen** is called; the initial multimedia file I/O offset will be the same as the offset in the standard file when **mmioOpen** is called. To close the multimedia file I/O file handle without closing the standard file handle, pass the MMIO_FHOPEN flag to **mmioClose**.

You must call **mmioClose** to close a file opened by using **mmioOpen**. Open files are not automatically closed when an application exits.

mmioRead

```
LONG mmioRead(HMMIO hmmio, HPSTR pch, LONG cch);
```

Reads a specified number of bytes from a file opened by using the **mmioOpen** function.

- Returns 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 0. If there is an error reading from the file, the return value is -1.

hmmio
File handle of the file to be read.

pch
Address of a buffer to contain the data read from the file.

cch
Number of bytes to read from the file.

mmioRename

```
MMRESULT mmioRename(LPCSTR szFilename, LPCSTR szNewFilename,
    const LPMMIOINFO lpmmioinfo, DWORD dwRenameFlags);
```

Renames the specified file.

- Returns zero if the file was renamed. Otherwise, returns an error code returned from **mmioRename** or from the I/O procedure.

szFilename
Address of a string containing the filename of the file to rename.

szNewFileName
Address of a string containing the new filename.

lpmmioinfo
Address of an **MMIOINFO** structure containing extra parameters used by **mmioRename**. If this parameter is not NULL, all unused members of the **MMIOINFO** structure it references must be set to zero, including the reserved members.

dwRenameFlags
Flags for the rename operation. This parameter should be set to zero.

mmioSeek

```
LONG mmioSeek(HMMIO hmmio, LONG lOffset, int iOrigin);
```

Changes the current file position in a file opened by using the **mmioOpen** function.

- Returns the new file position, in bytes, relative to the beginning of the file. If there is an error, the return value is - 1.

hmmio
File handle of the file to seek in.

lOffset
Offset to change the file position.

iOrigin
Flags indicating how the offset specified by *lOffset* is interpreted. The following values are defined:

SEEK_CUR
Seeks to *lOffset* bytes from the current file position.

SEEK_END
Seeks to *lOffset* bytes from the end of the file.

SEEK_SET
Seeks to *lOffset* bytes from the beginning of the file.

Seeking to an invalid location in the file, such as past the end of the file, might not cause **mmioSeek** to return an error, but it might 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

```
LRESULT mmioSendMessage(HMMIO hmmio, UINT wMsg, LPARAM lParam1,
    LPARAM lParam2);
```

Sends a message to the I/O procedure associated with the specified file.

- Returns a value that corresponds to the message. If the I/O procedure does not recognize the message, the return value should be zero.

hmmio
File handle for a file opened by using the **mmioOpen** function.

wMsg
Message to send to the I/O procedure.

lParam1 and *lParam2*
Parameters for the message.

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.

mmioSetBuffer

```
MMRESULT mmioSetBuffer(HMMIO hmmio, LPSTR pchBuffer, LONG cchBuffer,
    UINT wFlags);
```

Enables or disables buffered I/O, or changes the buffer or buffer size for a file opened by using the **mmioOpen** function.

- Returns zero if successful or an error otherwise. If an error occurs, the file handle remains valid. The following values are defined:

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.

hmmio
File handle of the file.

pchBuffer
Address of an application-defined buffer to use for buffered I/O. If this parameter is NULL, **mmioSetBuffer** allocates an internal buffer for buffered I/O.

cchBuffer
Size, in characters, of the application-defined buffer, or the size of the buffer for **mmioSetBuffer** to allocate.

wFlags
Reserved; must be zero.

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 can be changed after resizing.

mmioSetInfo

```
MMRESULT mmioSetInfo(HMMIO hmmio, LPMMIOINFO lpmmioinfo, UINT wFlags);
```

Updates the information retrieved by the **mmioGetInfo** function about a file opened by using the **mmioOpen** function. Use this function to terminate direct buffer access of a file opened for buffered I/O.

- Returns zero if successful or an error otherwise.

hmmio
File handle of the file.

lpmmioinfo
Address of an **MMIOINFO** structure filled with information by the **mmioGetInfo** function.

wFlags
Reserved; must be zero.

If you have written to the file I/O buffer, set the MMIO_DIRTY flag in the **dwFlags** member of the **MMIOINFO** structure before calling **mmioSetInfo** to terminate direct buffer access. Otherwise, the buffer will not get flushed to disk.

mmioStringToFOURCC

```
FOURCC mmioStringToFOURCC(LPCSTR sz, UINT wFlags);
```

Converts a null-terminated string to a four-character code.

- Returns the four-character code created from the given string.

sz
Address of a null-terminated string to convert to a four-character code.

wFlags
Flags for the conversion. The following value is defined:

MMIO_TOUPPER
Converts all characters to uppercase.

This function copies the string to a four-character code and pads it with space characters or truncates it if necessary. It does not check whether the code it returns is valid.

mmioWrite

```
LONG mmioWrite(HMMIO hmmio, char _huge* pch, LONG cch);
```

Writes a specified number of bytes to a file opened by using the **mmioOpen** function.

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

hmmio
File handle of the file.

pch
Address of the buffer to be written to the file.

cch
Number of bytes to write to the file.

The current file position is incremented by the number of bytes written.

MMCKINFO

```
typedef struct {
    FOURCC ckid;           // chunk identifier
    DWORD  cksize;         // see below
    FOURCC fccType;        // see below
    DWORD  dwDataOffset;   // see below
    DWORD  dwFlags;        // see below
} MMCKINFO;
```

Contains information about a chunk in a RIFF file.

cksize

Size, in bytes, of the data member of the chunk. The size of the data member does not include the 4-byte chunk identifier, the 4-byte chunk size, or the optional pad byte at the end of the data member.

fccType

Form type for "RIFF" chunks or the list type for "LIST" chunks.

dwDataOffset

File offset of the beginning of the chunk's data member, relative to the beginning of the file.

dwFlags

Flags specifying additional information about the chunk. It can be zero or the following flag:

MMIO_DIRTY

The length of the chunk might have changed and should be updated by the **mmioAscend** function. This flag is set when a chunk is created by using the **mmioCreateChunk** function.

MMIOINFO

```
typedef struct {
    DWORD       dwFlags;      // see below
    FOURCC      fccIOProc;    // see below
    LPMMIOPROC  pIOProc;      // address of file's I/O procedure
    UINT        wErrorRet;    // see below
    HTASK       hTask;        // see below
    LONG        cchBuffer;    // see below
    HPSTR       pchBuffer;    // see below
    HPSTR       pchNext;      // see below
    HPSTR       pchEndRead;   // see below
    HPSTR       pchEndWrite;  // see below
    LONG        lBufOffset;   // reserved
    LONG        lDiskOffset;  // see below
    DWORD       adwInfo[4];   // see below
    DWORD       dwReserved1;  // reserved
    DWORD       dwReserved2;  // reserved
    HMMIO       hmmio;        // see below
} MMIOINFO;
```

Contains the current state of a file opened by using the **mmioOpen** function.

dwFlags

Flags specifying how a file was opened. The following values are defined:

MMIO_ALLOCBUF
File's I/O buffer was allocated by the **mmioOpen** or **mmioSetBuffer** function.

MMIO_CREATE
The **mmioOpen** function was directed to create the file (or truncate it to zero length if it already existed).

MMIO_DIRTY
The I/O buffer has been modified.

MMIO_EXIST
Checks for the existence of the file.

MMIO_GETTEMP
A temporary name was retrieved by the **mmioOpen** function.

MMIO_PARSE
The new file's path is returned.

The following values may be set when the file is opened in share mode (identified by using the MMIO_SHAREMODE bit mask):

MMIO_COMPAT
File was opened with compatibility mode, allowing any process on a given machine to open the file any number of times.

MMIO_DENYNONE
Other processes are not denied read or write access to the file.

MMIO_DENYREAD
Other processes are denied read access to the file.

MMIO_DENYWRITE
Other processes are denied write access to the file.

MMIO_EXCLUSIVE
Other processes are denied read and write access to the file.

The following values may be set when the file is opened in read/write mode (identified by using the

MMIO_RWMODE bit mask):

MMIO_READ
File was opened only for reading.

MMIO_READWRITE
File was opened for reading and writing.

MMIO_WRITE
File was opened only for writing.

fccIOProc
Four-character code identifying the file's I/O procedure. If the I/O procedure is not an installed I/O procedure, this member is NULL.

wErrorRet
Extended error value from the **mmioOpen** function if it returns NULL. This member is not used to return extended error information from any other functions.

hTask
Handle of a local I/O procedure. Media Control Interface (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
Size, in bytes, of the file's I/O buffer. If the file does not have an I/O buffer, this member is zero.

pchBuffer
Address of the file's I/O buffer. If the file is unbuffered, this member is NULL.

pchNext
Address of the next location in the I/O buffer to be read or written. If no more bytes can be read without calling the **mmioAdvance** or **mmioRead** function, this member points to the **pchEndRead** member. If no more bytes can be written without calling the **mmioAdvance** or **mmioWrite** function, this member points to the **pchEndWrite** member.

pchEndRead
Address of the location that is 1 byte past the last location in the buffer that can be read.

pchEndWrite
Address of the location that is 1 byte past the last location in the buffer that can be written.

lDiskOffset
Current file position, which is an offset, in bytes, from the beginning of the file. I/O procedures are responsible for maintaining this member.

adwInfo
State information maintained by the I/O procedure. I/O procedures can also use these members to transfer information from the application to the I/O procedure when the application opens a file.

hmmio
Handle of the open file, as returned by the **mmioOpen** function. I/O procedures can use this handle when calling other multimedia file I/O functions.

MMIOM_CLOSE

```
MMIOM_CLOSE
lParam1 = (LPARAM) lCloseFlags
lParam2 = reserved
```

Sent to an I/O procedure by the **mmioClose** function to request that a file be closed.

- Returns zero if the file is successfully closed or an error otherwise.

lCloseFlags
Flags contained in the *wFlags* parameter of **mmioClose**.

MMIOM_OPEN

```
MMIOM_OPEN
lParam1 = (LPARAM) lpszFileName
lParam2 = reserved
```

Sent to an I/O procedure by the **mmioOpen** function to request that a file be opened or deleted.

- Returns MMSYSERR_NOERROR if successful or an error otherwise. Possible error values include the following:

MMIOM_CANNOTOPEN	The file could not be opened.
MMIOM_OUTOFMEMORY	Not enough memory to perform the operation.

lpszFileName
Null-terminated string containing the name of the file to open.

The **dwFlags** member of the **MMIOINFO** structure contains flags passed to the **mmioOpen** function.

The **lDiskOffset** member of the **MMIOINFO** structure is initialized to zero. If this value is incorrect, the I/O procedure must correct it.

If the application passed an **MMIOINFO** structure to **mmioOpen**, the return value is returned in the **wErrorRet** member.

MMIOM_READ

```
MMIOM_READ
lParam1 = (LPARAM) lBuffer // see below
lParam2 = (LPARAM) cbRead  // number of bytes to read from file
```

Sent to an I/O procedure by the **mmioRead** function to request that a specified number of bytes be read from an open file.

- Returns the number of bytes actually read from the file. If no more bytes can be read, the return value is 0. If there is an error, the return value is - 1.

lBuffer
Address of the buffer to be filled with data read from the file.

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

MMIOM_RENAME

```
MMIOM_RENAME
lParam1 = (LPARAM) lpszOldFilename
lParam2 = (LPARAM) lpszNewFilename
```

Sent to an I/O procedure by the **mmioRename** function to request that the specified file be renamed.

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

lpszOldFilename
Address of a string containing the filename of the file to rename.

lpszNewFilename
Address of a string containing the new filename.

MMIOM_SEEK

```
MMIOM_SEEK
lParam1 = (LPARAM) lNewFilePos
lParam2 = (LPARAM) lChangeFlag
```

Sent to an I/O procedure by the **mmioSeek** function to request that the current file position be moved.

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

lNewFilePos
New file position. The meaning of this value is dependent on the flag specified in *lChangeFlag*.

lChangeFlag
Flag specifying how the file position is changed. The following values are defined:

SEEK_CUR
Move the file position to be *lNewFilePos* bytes from the current position. *NewFilePos* can be positive or negative.

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

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

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

MMIOM_WRITE

```
MMIOM_WRITE
lParam1 = (LPARAM) lpBuffer // see below
lParam2 = (LPARAM) cbWrite  // number of bytes to write to file
```

Sent to an I/O procedure by the **mmioWrite** function to request that data be written to an open file.

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

lpBuffer
Address of a buffer containing the data to write to the file.

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

MMIOM_WRITEFLUSH

```
MMIOM_WRITEFLUSH
lParam1 = (LPARAM) lpBuffer // see below
lParam2 = (LPARAM) cbWrite  // number of bytes to write to file
```

Sent to an I/O procedure by the **mmioWrite** function to request that data be written to an open file and that any internal buffers used by the I/O procedure be flushed to disk.

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

lpBuffer
Address of a buffer containing the data to write to the file.

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

This message is equivalent to the MMIOM_WRITE message except that it 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.

Multimedia Timers

Multimedia timer services allow applications to schedule timer events with the greatest resolution (or accuracy) possible for the hardware platform. These multimedia timer services allow you to schedule timer events at a higher resolution than through other timer services.

The multimedia timer services are useful for applications that demand high-resolution timing. For example, a MIDI sequencer requires a high-resolution timer because it must maintain the pace of MIDI events within a resolution of 1 millisecond.

Applications that do not use high-resolution timing should use the **SetTimer** function instead of multimedia timer services. The timer services provided by **SetTimer** post WM_TIMER messages to a message queue; the multimedia timer services call a callback function.

The multimedia timer services allow an application to schedule periodic timer events – that is, the application can request and receive timer messages at application-specified intervals.

Obtaining the System Time

Typically, before an application begins using the multimedia timer services, it retrieves the current *system time*. The system time is the time, in milliseconds, since the Microsoft Windows operating system was started. You can use the **timeGetTime** or **timeGetSystemTime** function to retrieve the system time. These two functions are very similar: **timeGetTime** returns the system time, and **timeGetSystemTime** fills an **MMTIME** structure with the system time.

Timer Resolution

You can use the **timeGetDevCaps** function to determine the minimum and maximum timer resolutions supported by the timer services. This function fills the **wPeriodMin** and **wPeriodMax** members of the **TIMECAPS** structure with the minimum and maximum resolutions. This range can vary across computers and Windows platforms.

After you determine the minimum and maximum available timer resolutions, you must establish the minimum resolution you want your application to use. Use the **timeBeginPeriod** and **timeEndPeriod** functions to set and clear this resolution. You must match each call to **timeBeginPeriod** with a call to **timeEndPeriod**, specifying the same minimum resolution in both calls. An application can make multiple **timeBeginPeriod** calls, as long as each call is matched with a call to **timeEndPeriod**.

In both functions, the *uPeriod* parameter indicates the minimum timer resolution, in milliseconds. You can specify any timer resolution value within the range supported by the timer.

Timer Event Operations

After you have established your application's timer resolution, you can start timer events by using the **timeSetEvent** function. This function returns a timer identifier that can be used to stop or identify timer events. One of the function's parameters is the address of a **TimeProc** callback function that is called when the timer event takes place.

There are two types of timer events: *single* and *periodic*. A single timer event occurs once, after a specified number of milliseconds. A periodic timer event occurs every time a specified number of milliseconds elapses. The interval between periodic events is called an *event delay*. Periodic timer events with an event delay of 10 milliseconds or less consume a significant portion of CPU resources.

The relationship between the resolution of a timer event and the length of the event delay is important in timer events. For example, if you specify a resolution of 5 and an event delay of 100, the timer services notify the callback function after an interval ranging from 95 to 105 milliseconds.

You can cancel an active timer event at any time by using the **timeKillEvent** function. Be sure to cancel any outstanding timers before freeing the memory containing the callback function.

Using Multimedia Timers

This section contains examples demonstrating how to perform the following tasks:

- Obtain and set the timer resolution.
- Start a single timer event.
- Write a timer callback function.
- Cancel a timer event.

Obtaining and Setting Timer Resolution

The following example calls the **timeGetDevCaps** function to determine the minimum and maximum timer resolutions supported by the timer services. Before it sets up any timer events, the example establishes the minimum timer resolution by using the **timeBeginPeriod** function.

```
#define TARGET_RESOLUTION 1         // 1-millisecond target resolution

TIMECAPS tc;
UINT     wTimerRes;

if (timeGetDevCaps(&tc, sizeof(TIMECAPS)) != TIMERR_NOERROR) {
    // Error; application can't continue.
}

wTimerRes = min(max(tc.wPeriodMin, TARGET_RESOLUTION), tc.wPeriodMax);
timeBeginPeriod(wTimerRes);
```

Starting a Single Timer Event

To start a single timer event, an application must specify the amount of time before the callback occurs, the resolution, the address of the callback function, and the user data to supply with the callback function. An application can use a function like the following to start a single timer event.

```
UINT SetTimerCallback(NPSEQ npSeq,  // sequencer data
    UINT msInterval)                // event interval
{
    npSeq->wTimerID = timeSetEvent(
        msInterval,                     // delay
        wTimerRes,                      // resolution (global variable)
        OneShotCallback,                // callback function
        (DWORD)npSeq,                   // user data
        TIME_ONESHOT );                 // single timer event
    if(! npSeq->wTimerID)
        return ERR_TIMER;
    else
        return ERR_NOERROR;
}
```

Writing a Timer Callback Function

The following callback function invalidates the identifier for the single timer event and calls a timer routine to handle the application-specific tasks.

```
void FAR PASCAL OneShotTimer(UINT wTimerID, UINT msg,
    DWORD dwUser, DWORD dw1, DWORD dw2)
{
    NPSEQ npSeq;              // pointer to sequencer data
    npSeq = (NPSEQ)dwUser;
    npSeq->wTimerID = 0;      // invalidate timer ID (no longer in use)
    TimerRoutine(npSeq);      // handle tasks
}
```

Canceling a Timer Event

The application must cancel any outstanding timers before it frees the memory that contains the callback function. To cancel a timer event, it might call the following function.

```
void DestroyTimer(NPSEQ npSeq)
{
    if(npSeq->wTimerID) {                // is timer event pending?
        timeKillEvent(npSeq->wTimerID);  // cancel the event
        npSeq->wTimerID = 0;
    }
}
```

Timer Reference

This section describes the functions and structures associated with multimedia timer services. These elements are grouped as follows.

Retrieving the System Time

MMTIME
timeGetSystemTime
timeGetTime

Retrieving Timer Information

TIMECAPS
timeGetDevCaps

Time Events

timeKillEvent
TimeProc
timeSetEvent

Time Periods

timeBeginPeriod
timeEndPeriod

timeBeginPeriod

```
MMRESULT timeBeginPeriod(UINT uPeriod);
```

Sets the minimum timer resolution for an application or device driver.

- Returns TIMERR_NOERROR if successful or TIMERR_NOCANDO if the resolution specified in *uPeriod* is out of range.

uPeriod
Minimum timer resolution, in milliseconds, for the application or device driver.

Call this function immediately before using timer services, and call the **timeEndPeriod** function immediately after you are finished using the timer services.

You must match each call to **timeBeginPeriod** with a call to **timeEndPeriod**, specifying the same minimum resolution in both calls. An application can make multiple **timeBeginPeriod** calls as long as each call is matched with a call to **timeEndPeriod**.

timeEndPeriod

```
MMRESULT timeEndPeriod(UINT uPeriod);
```

Clears a previously set minimum timer resolution.

- Returns TIMERR_NOERROR if successful or TIMERR_NOCANDO if the resolution specified in *uPeriod* is out of range.

uPeriod
Minimum timer resolution specified in the previous call to the **timeBeginPeriod** function.

Call this function immediately after you are finished using timer services.

You must match each call to **timeBeginPeriod** with a call to **timeEndPeriod**, specifying the same minimum resolution in both calls. An application can make multiple **timeBeginPeriod** calls as long as each call is matched with a call to **timeEndPeriod**.

timeGetDevCaps

```
MMRESULT timeGetDevCaps(LPTIMECAPS ptc, UINT cbtc);
```

Queries the timer device to determine its resolution.

- Returns TIMERR_NOERROR if successful or TIMERR_STRUCT if it fails to return the timer device capabilities.

ptc
Address of a **TIMECAPS** structure. This structure is filled with information about the resolution of the timer device.

cbtc
Size, in bytes, of the **TIMECAPS** structure.

timeGetSystemTime

```
MMRESULT timeGetSystemTime(LPMMTIME pmmt, UINT cbmmt);
```

Retrieves the system time, in milliseconds. The system time is the time elapsed since Windows was started. This function works very much like the **timeGetTime** function. See **timeGetTime** for details of these functions' operation.

- Returns TIMERR_NOERROR. The system time is returned in the **ms** member of the **MMTIME** structure.

pmmt
Address of an **MMTIME** structure.

cbmmt
Size, in bytes, of the **MMTIME** structure.

timeGetTime

```
DWORD timeGetTime(VOID);
```

Retrieves the system time, in milliseconds. The system time is the time elapsed since Windows was started.

- Returns the system time, in milliseconds.

The only difference between this function and the **timeGetSystemTime** function is that **timeGetSystemTime** uses the **MMTIME** structure to return the system time. The **timeGetTime** function has less overhead than **timeGetSystemTime**.

Note that the value returned by the **timeGetTime** function is a DWORD value. The return value wraps around to 0 every 2^32 milliseconds, which is about 49.71 days.This can cause problems in code that directly uses the **timeGetTime** return value in computations, particularly where the value is used to control code execution. You should always use the difference between two **timeGetTime** return values in computations.

Windows NT: The default precision of the **timeGetTime** function can be five milliseconds or more, depending on the machine. You can use the **timeBeginPeriod** and **timeEndPeriod** functions to increase the precision of **timeGetTime**. If you do so, the minimum difference between successive values returned by **timeGetTime** can be as large as the minimum period value set using **timeBeginPeriod** and **timeEndPeriod**. Use the **QueryPerformanceCounter** and **QueryPerformanceFrequency** functions to measure short time intervals at a high resolution,

Windows 95: The default precision of the **timeGetTime** function is 1 millisecond. In other words, the **timeGetTime** function can return successive values that differ by just 1 millisecond. This is true no matter what calls have been made to the **timeBeginPeriod** and **timeEndPeriod** functions.

timeKillEvent

```
MMRESULT timeKillEvent(UINT uTimerID);
```

Cancels a specified timer event.

- Returns TIMERR_NOERROR if successful or MMSYSERR_INVALPARAM if the specified timer event does not exist.

uTimerID
Identifier of the timer event to cancel. This identifier was returned by the **timeSetEvent** function when the timer event was set up.

TimeProc

```
void CALLBACK TimeProc(UINT uID, UINT uMsg, DWORD dwUser, DWORD dw1,
    DWORD dw2);
```

Callback function that is called once upon the expiration of a single event or periodically upon the expiration of periodic events.

TimeProc is a placeholder for the application-defined function name.

uID
Identifier of the timer event. This identifier was returned by the **timeSetEvent** function when the timer event was set up.

uMsg
Reserved; do not use.

dwUser
User instance data supplied to the *dwUser* parameter of **timeSetEvent**.

dw1 and *dw2*
Reserved; do not use.

Applications should not call any system-defined functions from inside a callback function, except for **PostMessage**, **timeGetSystemTime**, **timeGetTime**, **timeSetEvent**, **timeKillEvent**, **midiOutShortMsg**, **midiOutLongMsg**, and **OutputDebugString**.

timeSetEvent

```
MMRESULT timeSetEvent(UINT uDelay, UINT uResolution,
    LPTIMECALLBACK lpTimeProc, DWORD dwUser, UINT fuEvent);
```

Starts a specified timer event. After the event is activated, it calls the specified callback function.

- Returns an identifier for the timer event if successful or an error otherwise. This function returns NULL if it fails and the timer event was not created. (This identifier is also passed to the callback function.)

uDelay
Event delay, in milliseconds. If this value is not in the range of the minimum and maximum event delays supported by the timer, the function returns an error.

uResolution
Resolution of the timer event, in milliseconds. The resolution increases with smaller values; a resolution of 0 indicates periodic events should occur with the greatest possible accuracy. To reduce system overhead, however, you should use the maximum value appropriate for your application.

lpTimeProc
Address of a callback function that is called once upon expiration of a single event or periodically upon expiration of periodic events.

dwUser
User-supplied callback data.

fuEvent
Timer event type. The following values are defined:

TIME_ONESHOT
Event occurs once, after *uDelay* milliseconds.

TIME_PERIODIC
Event occurs every *uDelay* milliseconds.

Each call to **timeSetEvent** for periodic timer events must be matched with a call to the **timeKillEvent** function.

MMTIME

```
typedef struct mmtime_tag {
    UINT wType;
    union {              // start u union
        DWORD ms;        // see below
        DWORD sample;    // see below
        DWORD cb;        // see below
        DWORD ticks;     // see below
        struct {
            BYTE hour;   // hours
            BYTE min;    // minutes
            BYTE sec;    // seconds
            BYTE frame;  // frames
            BYTE fps;    // frames/sec. (24, 25, 29 (30 drop), or 30)
            BYTE dummy;  // dummy byte for alignment
            BYTE pad[2]  // more padding
        } smpte;
        struct {
            DWORD songptrpos;  // song pointer position
        } midi;
    } u;
} MMTIME;
```

Contains timing information for different types of multimedia data.

wType
Time format. It can be one of the following values:

TIME_BYTES	Current byte offset from beginning of the file.
TIME_MIDI	MIDI time.
TIME_MS	Time in milliseconds.
TIME_SAMPLES	Number of waveform-audio samples.
TIME_SMPTE	SMPTE (Society of Motion Picture and Television Engineers) time.
TIME_TICKS	Ticks within a MIDI stream.

ms
Number of milliseconds. Used when **wType** is TIME_MS.

sample
Number of samples. Used when **wType** is TIME_SAMPLES.

cb
Byte count. Used when **wType** is TIME_BYTES.

ticks
Ticks in MIDI stream. Used when **wType** is TIME_TICKS.

smpte
SMPTE time structure. Used when **wType** is TIME_SMPTE.

midi
MIDI time structure. Used when **wType** is TIME_MIDI.

TIMECAPS

```
typedef struct {
    UINT wPeriodMin;  \\ minimum supported resolution
    UINT wPeriodMax;  \\ maximum supported resolution
} TIMECAPS;
```

Contains information about the resolution of the timer.

Installable Drivers

An installable driver is a Microsoft Windows dynamic-link library (DLL) that provides a standard interface through which Windows-based applications and DLLs communicate with and manage the driver. Installable drivers are used most commonly for as multimedia device drivers, but installable drivers can also be used for other purposes and are particularly useful in situations that require a standard interface and control of multiple instances.

This chapter describes the general format of installable drivers and defines the installable driver functions that applications and DLLs use to open and manage installable drivers.

Installable Driver Format

Every installable driver exports a **DriverProc** function. This common entry-point function receives *driver messages* from the system that direct the driver to carry out actions or provide information. The system sends driver messages to the **DriverProc** function when an application or DLL opens or closes the driver or requests that a message be sent to the driver. The **DriverProc** function either processes the message or passes the message to the default message handler, the **DefDriverProc** function. In either case, **DriverProc** must return a value indicating whether the requested action was successful.

Driver Messages

Each driver message consists of a message identifier and two 32-bit parameters. The message identifier is a unique value that the **DriverProc** function checks to determine which action to carry out. The meaning of the message parameters depends on the message. The parameters may represent values or addresses. In many cases, the parameters are not used and are set to zero.

Driver messages can be standard or custom. Window sends standard driver messages, such as DRV_OPEN, DRV_CLOSE, and DRV_CONFIGURE, to an installable driver in response to a request to open, close, or configure the driver. The standard messages direct the installable driver to load or unload its resources, enable or disable its operation, open or close a driver instance, and display a configuration dialog box. Some standard messages, such as DRV_POWER and DRV_EXITSESSION, notify the driver of system-wide events that affect the operation of the driver or any related hardware.

Applications and DLLs send custom driver messages to direct an installable driver to carry out driver-specific actions. Installable drivers that support custom messages must include appropriate processing in the **DriverProc** function. To prevent conflict between custom and standard driver messages, custom message identifiers must have values ranging from DRV_RESERVED to DRV_USER. Custom messages passed to the **DefDriverProc** function are ignored.

Driver Instances

Windows allows for multiples instances of an installable driver. The system creates an instance of the driver each time the driver is opened and destroys the instance when the driver is closed. Driver instances are especially useful for installable drivers that support multiple devices or that are opened by multiple applications or by the same application multiple times.

To help the driver keep track of the instances, the system sends a driver instance handle with each driver message after the instance has been created. Because this handle uniquely identifies the instance, installable drivers often associate the handle with memory and other resources that they have specifically allocated for the instance.

When the first instance is opened, the system sends the DRV_LOAD, DRV_ENABLE, and DRV_OPEN messages to the driver in that order. The DRV_LOAD and DRV_ENABLE messages notify the driver that it is now in memory and is enabled for operation. The DRV_OPEN message identifies the instance handle and may include configuration information for the instance. On each subsequent opening of an instance of the same driver, the system sends only a DRV_OPEN message.

When processing a DRV_LOAD message, a driver typically reads configuration settings from the registry, configures the driver and any associated hardware, and allocates memory for use by all instances of the driver. If a driver cannot complete the configuration or allocate memory, it returns zero to direct the system to immediately remove the driver from memory and prevent any subsequent messages from being sent. When processing the DRV_ENABLE message, the driver prepares the hardware to receive and process input and output (I/O) requests. The preparation may include installing interrupt handlers.

When processing the DRV_OPEN message, the driver allocates memory or resources required by the given instance of the driver and then returns a nonzero value. The system uses this nonzero value as the *driver identifier* in subsequent driver messages for the instance. The driver can use this identifier for any purpose. For example, some drivers use a memory handle for the identifier to gain quick access to memory containing information about the given instance.

Many installable drivers process the second parameter of the DRV_OPEN message, giving the system and applications the means to send additional information to the driver when opening an instance. The parameter can be a single value or an address of a structure containing a set of values. When processing DRV_OPEN, the driver checks the parameter to determine whether it is a value and uses the given values, if any, to complete the creation of the instance.

The system sends a DRV_CLOSE message each time an instance is closed. The instance handle sent with the message identifies which instance to close. When the last remaining instance is closed, the system sends the DRV_CLOSE, DRV_DISABLE, and DRV_FREE messages in that order. The DRV_CLOSE message directs the driver to close the instance, and the DRV_DISABLE and DRV_FREE messages notify the driver that it is now disabled and will be immediately freed from memory.

When processing the DRV_CLOSE message, the driver typically frees any memory or resources allocated for the instance. When processing the DRV_DISABLE message, the driver places any hardware in an inactive state, which may include the removal of interrupt handlers. When processing the DRV_FREE message, the driver frees any memory or resources that are still allocated.

Installable drivers are not required to support multiple instances. A driver can prevent any instance from being created by returning zero for the DRV_OPEN message.

Configuration

An installable driver can let users choose configuration settings for the driver and associated hardware by displaying a configuration dialog box when processing the DRV_CONFIGURE message. The driver is responsible for creating and managing the dialog box, processing any user input from the dialog box, and changing the configuration of the driver or hardware as requested by the user. The driver must provide a separate dialog box procedure to process window messages for the dialog box and a dialog box template to define the appearance and content of the dialog box.

Before receiving the DRV_CONFIGURE message, a driver receives the DRV_QUERYCONFIGURE message. The driver must return a nonzero value to the query to ensure receipt of the subsequent DRV_CONFIGURE message.

When initializing the configuration dialog box, the driver typically retrieves configuration information from the registry. To help locate this information, the DRV_CONFIGURE message usually includes the address of a **DRVCONFIGINFO** structure that contains the names of the registry key and value associated with the driver. If the user requests changes to the configuration, the driver should update the configuration information in the registry.

Installation

An installable driver can carry out driver-specific installation tasks when processing the DRV_INSTALL and DRV_REMOVE messages. An installation application, such as a Control Panel application, sends the messages to the driver when installing or removing the driver, respectively.

When processing the DRV_INSTALL message, the driver typically verifies that the required hardware is present and then displays the configuration dialog box to let the user choose the initial configuration settings for the driver and associated hardware. The message includes the address of a **DRVCONFIGINFO** structure that contains the names of the registry key and value associated with the driver; the driver checks the registry value for default configuration information. Finally, the driver also creates any additional registry keys and values needed for successful operation.

When processing the DRV_REMOVE message, the driver removes any registry keys and values it may have created.

Callback Functions

Installable drivers can notify the application, window, or task that opened the given instance about events by using the **DriverCallback** function. This function gives the driver the means to return information to an application or DLL while continuing to process a request.

If a driver supports callback functions, the application or DLL that opens the instance must supply a value this is either the address of a callback function, a window handle, or a task handle. This value and a flag identifying the type of the value are typically passed in a structure pointed to by the second parameter of the DRV_OPEN message.

Installable Drive Module-Definition File

The module-definition (.DEF) file of an installable driver names the driver, exports the **DriverProc** function, and defines a driver description. The following example shows a typical module-definition file for an installable driver:

```
LIBRARY OSCI
DESCRIPTION 'FREQ,AMPL:Oscilloscope frequency and amplitude drivers.'
EXPORTS
    DriverProc
```

Some installation applications may open the driver and retrieve the description line to use when installing the driver. To remain compatible with these installation applications, the description line should have this form:

DESCRIPTION *alias*[,*alias*]...**:***text*

The *alias* specifies a unique name for the driver that applications can use to open the driver. The alias also serves as the value name associated with the driver in the registry. Multiple aliases are separated by commas. The *text* describes the purpose of the driver.

Installable Driver Functions and Messages

You can open an installable driver from an application by using the **OpenDriver** function. This function creates an instance of the driver, loading the driver into memory if no other instance exists, and returns the handle of the new instance. When opening an installable driver, you must supply either the full path of the driver or the names of the registry key and value associated with the driver.

Once a driver is open, you can direct it to carry out tasks by using the **SendDriverMessage** function to send driver messages to the driver. For example, you can direct the driver to display its configuration dialog box by sending the DRV_CONFIGURE message. Before sending this message, you must determine whether the driver has a configuration dialog box by sending the DRV_QUERYCONFIGURE message and checking for a nonzero return value. Many drivers provide a set of custom messages that you can send to direct the operations of the driver.

If you need special access to an installable driver, such as access to its resources, you can retrieve the module handle of the driver by using the **GetDriverModuleHandle** function.

When you no longer need the installable driver, you can close it by using the **CloseDriver** function.

You can use the installable driver functions and messages to open and manage any installable driver. However, the recommended course of action for opening and managing multimedia devices is to first use standard services (such as **waveOutOpen**, **waveOutMessage**, and **waveOutClose** for waveform output devices), if they exist. If standard services do not exist for a multimedia driver, then open and manage the driver using the installable driver functions and messages.

Note The **SendDriverMessage** and **GetDriverModuleHandle** functions are the preferred functions to use to send messages to a driver and to obtain a handle to a module instance. The older **DrvSendMessage** and **DrvGetModuleHandle** functions, however, have been included to maintain compatibility with previous versions of the Windows operating system.

Using Installable Drivers

You use installable drivers to give applications or DLLs a standard way to access a device or a set of useful routines. The following sections show how to create an installable driver by using a **DriverProc** function and how to open an installable driver and direct it to carry out useful tasks.

Creating a DriverProc Function

You create a **DriverProc** function in much the same way as you create a window procedure. The function consists of a **switch** statement, and each case processes a given driver message, returning a value indicating success or failure. The **DriverProc** function has the following form:

```
LONG DriverProc(DWORD dwDriverId, HDRVR hdrvr, UINT msg,
    LONG lParam1, LONG lParam2)
{
    DWORD dwRes = 0L;

    switch (msg) {
    case DRV_LOAD:
    // Sent when the driver is loaded. This is always
    // the first message received by a driver.
        dwRes = 1L;  // returns 0L to fail
        break;

    case DRV_FREE:
    // Sent when the driver is about to be discarded.
    // This is the last message a driver receives
    // before it is freed.
        dwRes = 1L;  // return value ignored
        break;

    case DRV_OPEN:
    // Sent when the driver is opened.
        dwRes = 1L;  // returns 0L to fail
        break;       // value subsequently used
                     // for dwDriverId.

    case DRV_CLOSE:
    // Sent when the driver is closed. Drivers are
    // unloaded when the open count reaches zero.
        dwRes = 1L;  // returns 0L to fail
        break;

    case DRV_ENABLE:
    // Sent when the driver is loaded or reloaded and
    // when Windows is enabled. Install interrupt
    // handlers and initialize hardware. Expect the
    // driver to be in memory only between the enable
    // and disable messages.
        dwRes = 1L;  // return value ignored
        break;

    case DRV_DISABLE:
    // Sent before the driver is freed or when Windows
    // is disabled. Remove interrupt handlers and place
    // hardware in an inactive state.
        dwRes = 1L;  // return value ignored
        break;

    case DRV_INSTALL:
    // Sent when the driver is installed.
        dwRes = DRVCNF_OK;  // Can also return
        break;              // DRVCNF_CANCEL
                            // and DRV_RESTART

    case DRV_REMOVE:
    // Sent when the driver is removed.
```

```
        dwRes = 1L;  // return value ignored
        break;

    case DRV_QUERYCONFIGURE:
    // Sent to determine if the driver can be
    // configured.
        dwRes = 0L;  // Zero indicates configuration
        break;       // NOT supported

    case DRV_CONFIGURE:
    // Sent to display the configuration
    // dialog box for the driver.
        dwRes = DRVCNF_OK;  // Can also return
        break;              // DRVCNF_CANCEL
                            // and DRVCNF_RESTART

    default:
    // Process any other messages.
        return DefDriverProc (dwDriverId, hdrvr,
            msg, lParam1, lParam2);

    }
    return dwRes;
}
```

Configuring an Installable Driver

To direct an installable driver to carry out useful tasks, you must open the driver by using the **OpenDriver** function and send it messages by using the **SendDriverMessage** function. The following example shows how to direct the driver to display its configuration dialog box.

```
LONG MyConfigureDriver()
{
    HDRVR hdrvr;
    DRVCONFIGINFO dci;
    LONG lRes;

    // Open the driver (no additional parameters needed this time).
    if ((hdrvr = OpenDriver(L"\\samples\\sample.drv", 0, 0)) == 0) {
        // Can't open the driver
        return DRVCNF_CANCEL;
    }

    // Make sure driver has a configuration dialog box.
    if (SendDriverMessage(hdrvr, DRV_QUERYCONFIGURE, 0, 0) != 0) {
        // Set the DRVCONFIGINFO structure and send the message
        dci.dwDCISize = sizeof (dci);
        dci.lpszDCISectionName = (LPWSTR)0;
        dci.lpszDCIAliasName = (LPWSTR)0;
        lRes = SendDriverMessage(hdrvr, DRV_CONFIGURE, 0, (LONG)&dci);
     }

    // Close the driver (no additional parameters needed this time).
    CloseDriver(hdrvr, 0, 0);

    return lRes;
}
```

Installable Driver Reference

Thie functions and messages associated with installable drivers are grouped as follows.

Loading and Unloading Drivers

OpenDriver
SendDriverMessage
GetDriverModuleHandle

CloseDriver

DRV_LOAD
DRV_ENABLE
DRV_OPEN
DRV_CLOSE
DRV_DISABLE
DRV_FREE

Configuring a Driver

DRV_CONFIGURE
DRV_QUERYCONFIGURE
DRVCONFIGINFO

Installing a Driver

DRV_INSTALL
DRV_REMOVE

Driver Functions

DefDriverProc
DriverProc
DriverCallback

Installable Driver Functions

The functions in this sectioncan be used in an application to open, close, and communicate with an installable driver.

CloseDriver

```
LRESULT CloseDriver(HDRVR hdrvr, LONG lParam1, LONG lParam2);
```

Closes an installable driver.

- Returns nonzero if successful or zero otherwise.

hdrvr
Handle of an installable driver instance. The handle must have been previously created by using the **OpenDriver** function.

lParam1 and *lParam2*
32-bit driver-specific data.

The function passes the *lParam1* and *lParam2* parameters to the **DriverProc** function of the installable driver.

DrvGetModuleHandle

```
WINMMAPI HMODULE WINAPI DrvGetModuleHandle(HDRVR hDriver);
```

Retrieves the instance handle of the module that contains the installable driver.

- Returns an instance handle of the driver module if successful or NULL otherwise.

hDriver

Handle of the installable driver instance. The handle must have been previously created by using the **OpenDriver** function.

This function is provided for compatibility with previous versions of Windows.

DrvSendMessage

```
WINMMAPI LRESULT WINAPI DrvSendMessage(HDRVR hdrvr, UINT uMsg,
    LPARAM lParam1, LPARAM lParam2);
```

Sends the specified message to the insntallable driver.

- Returns nonzero if successful or zero otherwise.

hdrvr
Handle of the installable driver instance. The handle must have been previously created by using the **OpenDriver** function.

uMsg
Driver message value. It can be a custom message value or one of these standard message values:

DRV_QUERYCONFIGURE	Queries an installable driver about whether it supports the DRV_CONFIGURE message and can display a configuration dialog box.
DRV_CONFIGURE	Notifies an installable driver that it should display a configuration dialog box. (This message should be sent only if the driver returns a nonzero value when the DRV_QUERYCONFIGURE message is processed.)
DRV_INSTALL	Notifies an installable driver that it has been successfully installed.
DRV_REMOVE	Notifies an installable driver that it is about to be removed from the system.

lParam1 and *lParam2*
Message parameters containing 32-bit information specific to the message value.

This function is provided for compatibility with previous versions of Windows.

GetDriverModuleHandle

```
HMODULE GetDriverModuleHandle(HDRVR hdrvr);
```

Retrieves the instance handle of the module that contains the installable driver.

- Returns an instance handle of the driver module if successful or NULL otherwise.

hdrvr
Handle of the installable driver instance. The handle must have been previously created by using the **OpenDriver** function.

This function is specific to Windows 95.

OpenDriver

```
HDRVR OpenDriver(LPCWSTR lpDriverName, LPCWSTR lpSectionName,
    LONG lParam);
```

Opens an instance of an installable driver and initializes the instance using either the driver's default settings or a driver-specific value.

- Returns the handle of the installable driver instance if successful or NULL otherwise.

lpDriverName
Address of a null-terminated, wide-character string that specifies the filename of an installable driver or the name of a registry value associated with the installable driver. (This value must have been previously set when the driver was installed.)

lpSectionName
Address of a null-terminated, wide-character string that specifies the name of the registry key containing the registry value given by the *lpDriverName* parameter. If *lpSectionName* is NULL, the registry key is assumed to be **Drivers32**.

lParam
32-bit driver-specific value. This value is passed as the *lParam2* parameter to the **DriverProc** function of the installable driver.

SendDriverMessage

```
LRESULT SendDriverMessage(HDRVR hdrvr, UINT msg, LONG lParam1,
    LONG lParam2);
```

Sends the specified message to the installable driver.

- Returns nonzero if successful or zero otherwise.

hdrvr
Handle of the installable driver instance. The handle must been previously created by using the **OpenDriver** function.

msg
Driver message value. It can be a custom message value or one of these standard message values:

DRV_QUERYCONFIGURE	Queries an installable driver about whether it supports the DRV_CONFIGURE message and can display a configuration dialog box.
DRV_CONFIGURE	Notifies an installable driver that it should display a configuration dialog box. (This message should only be sent if the driver returns a nonzero value when the DRV_QUERYCONFIGURE message is processed.)
DRV_INSTALL	Notifies an installable driver that it has been successfully installed.
DRV_REMOVE	Notifies an installable driver that it is about to be removed from the system.

lParam1 and *lParam2*
32-bit message-dependent information.

This function is specific to Windows 95.

Driver Functions

The functions is this sectiondescribe the entry point, default processing, and callback functions to use in an installable driver.

DefDriverProc

```
LONG DefDriverProc(DWORD dwDriverId, HDRVR hdrvr, UINT msg,
    LONG lParam1, LONG lParam2);
```

Provides default processing for any messages not processed by an installable driver. This function is intended to be used only within the **DriverProc** function of an installable driver.

- Returns nonzero if successful or zero otherwise.

dwDriverId
Identifier of the installable driver.

hdrvr
Handle of the installable driver instance.

msg
Driver message value.

lParam1 and *lParam2*
32-bit message-dependent information.

DriverProc

```
LONG DriverProc(DWORD dwDriverId, HDRVR hdrvr, UINT msg,
    LONG lParam1, LONG lParam2);
```

Processes driver messages for the installable driver. **DriverProc** is a driver-supplied function.

- Returns nonzero if successful or zero otherwise.

dwDriverId
Identifier of the installable driver.

hdrvr
Handle of the installable driver instance. Each instance of the installable driver has a unique handle.

msg
Driver message value. It can be a custom value or one of these standard values:

DRV_CLOSE	Notifies the driver that it should decrement its usage count and unload the driver if the count is zero.
DRV_CONFIGURE	Notifies the driver that it should display a configuration dialog box. This message is sent only if the driver returns a nonzero value when processing the DRV_QUERYCONFIGURE message.
DRV_DISABLE	Notifies the driver that its allocated memory is about to be freed.
DRV_ENABLE	Notifies the driver that it has been loaded or reloaded or that Windows has been enabled.
DRV_FREE	Notifies the driver that it will be discarded.
DRV_INSTALL	Notifies the driver that it has been successfully installed.
DRV_LOAD	Notifies the driver that it has been successfully loaded.
DRV_OPEN	Notifies the driver that it is about to be opened.
DRV_POWER	Notifies the driver that the device's power source is about to be turned on or off.
DRV_QUERYCONFIGURE	Directs the driver to specify whether it supports the DRV_CONFIGURE message.
DRV_REMOVE	Notifies the driver that it is about to be removed from the system.

lParam1 and *lParam2*
32-bit message-specific value.

When *msg* is DRV_OPEN, *lParam1* is the string following the driver filename from the SYSTEM.INI file and *lParam2* is the value given as the *lParam* parameter in a call to the **OpenDriver** function.

When *msg* is DRV_CLOSE, *lParam1* and *lParam2* are the same values as the *lParam1* and *lParam2* parameters in a call to the **CloseDriver** function.

DriverCallback

```
BOOLEAN DriverCallback(DWORD dwCallBack, DWORD dwFlags, HDRVR hdrvr,
    DWORD msg, DWORD dwUser, DWORD dwParam1, DWORD dwParam2);
```

Calls a callback function, sends a message to a window, or unblocks a thread. The action depends on the value of the notification flag. This function is intended to be used only within the **DriverProc** function of an installable driver.

- Returns TRUE if successful or FALSE if a parameter is invalid or the task's message queue is full.

dwCallBack
Address of the callback function, a window handle, or a task handle, depending on the flag specified in the *dwFlags* parameter.

dwFlags
Notification flags. It can be one of these values:

DCB_NOSWITCH	The system is prevented from switching stacks. This value is only used if enough stack space for the callback function is known to exist.
DCB_FUNCTION	The *dwCallback* parameter is the address of an application-defined callback function. The system sends the callback message to the callback function.
DCB_WINDOW	The *dwCallback* parameter is the handle of an application-defined window. The system sends subsequent notifications to the window.
DCB_TASK	The *dwCallback* parameter is the handle of an application or task. The system sends subsequent notifications to the application or task.

hdrvr
Handle of the installable driver instance.

msg
Message value.

dwUser
32-bit user-instance data supplied by the application when the device was opened.

dwParam1 and *dwParam2*
32-bit message-dependent parameter.

The client specifies how to notify it when the device is opened. The DCB_FUNCTION and DCB_WINDOW flags are equivalent to the high-order word of the corresponding flags CALLBACK_FUNCTION and CALLBACK_WINDOW specified in the *lParam2* parameter of the DRV_OPEN message when the device was opened.

If notification is accomplished with a callback function, *hdrvr*, *msg*, *dwUser*, *dwParam1*, and *dwParam2* are passed to the callback function. If notification is accomplished by means of a window, only *msg*, *hdrvr*, and *dwParam1* are passed to the window.

DRV_CLOSE

Directs the driver to close the given instance. If no other instances are open, the driver should prepare for subsequent release from memory.

- Returns nonzero if successful or zero otherwise.

dwDriverId
Identifier of the installable driver. This is the same value previously returned by the driver from the DRV_OPEN message.

hdrvr
Handle of the installable driver instance.

lParam1
32-bit value specified as the *lParam1* parameter in a call to the **DriverClose** function.

lParam2
32-bit value specified as the *lParam2* parameter in a call to the **DriverClose** function.

DRV_CONFIGURE

Directs the installable driver to display its configuration dialog box and let the user specify new settings for the given installable driver instance.

- Returns one of these values:

DRVCNF_OK	The configuration is successful; no further action is required.
DRVCNF_CANCEL	The user canceled the dialog box; no further action is required.
DRVCNF_RESTART	The configuration is successful, but the changes do not take effect until the system is restarted.

dwDriverId
Identifier of the installable driver. This is the same value previously returned by the driver from the DRV_OPEN message.

hdrvr
Handle of the installable driver instance.

lParam1
Handle of the parent window. This window is used as the parent window for the configuration dialog box.

lParam2
Address of a **DRVCONFIGINFO** structure or NULL. If the structure is given, it contains the names of the registry key and value associated with the driver.

Some installable drivers append configuration information to the value assigned to the registry value associated with the driver.

The DRV_CANCEL, DRV_OK, and DRV_RESTART return values are obsolete; they have been replaced by DRVCNF_CANCEL, DRVCNF_OK, and DRVCNF_RESTART, respectively.

DRV_DISABLE

Disables the driver. The driver should place the corresponding device, if any, in an inactive state and terminate any callback functions or threads.

- No return value.

hdrvr
Handle of the installable driver instance.

The *dwDriverId*, *lParam1*, and *lParam2* parameters are not used.

After disabling the driver, the system typically sends the driver a DRV_FREE message before removing the driver from memory.

DRV_ENABLE

Enables the driver. The driver should initialize any variables and locate devices with the input and output (I/O) interface.

- No return value.

hdrvr
Handle of the installable driver instance.

The *dwDriverId*, *lParam1*, and *lParam2* parameters are not used.

Drivers are considered enabled from the time they receive this message until they are disabled by using the DRV_DISABLE message.

DRV_EXITSESSION

Notifies the driver that Windows is preparing to exit. The driver should prepare for termination.

- No return value.

dwDriverId
Identifier of the installable driver. This is the same value previously returned by the driver from the DRV_OPEN message.

hdrvr
Handle of the installable driver instance.

The *lParam1* and *lParam2* parameters are not used.

DRV_FREE

Notifies the driver that it is being removed from memory. The driver should free any memory and other system resources that it has allocated.

- No return value.

hdrvr
Handle of the installable driver instance.

The *dwDriverId*, *lParam1*, and *lParam2* parameters are not used.

The DRV_FREE message is always the last message that a device driver receives.

DRV_INSTALL

Notifies the driver that is it being installed. The driver should create and initialize any needed registry keys and values and verify that the supporting drivers and hardware are installed and properly configured.

- Returns one of these values:

DRVCNF_OK	The installation is successful; no further action is required.
DRVCNF_CANCEL	The installation failed..
DRVCNF_RESTART	The installation is successful, but it does not take effect until the system is restarted.

dwDriverId
Identifier of the installable driver. This is the same value previously returned by the driver from the DRV_OPEN message.

hdrvr
Handle of the installable driver instance.

lParam2
Address of a **DRVCONFIGINFO** structure or NULL. If a structure is given, it contains the names of the registry key and value associated with the driver.

The *lParam1* parameter is not used.

Some installable drivers append configuration information to the value assigned to the registry value associated with the driver.

DRV_LOAD

Notifies the driver that it has been loaded. The driver should make sure that any hardware and supporting drivers it needs to function properly are present.

- Returns nonzero if successful or zero otherwise.

The *hdrvr* parameter is always zero. The *dwDriverId*, *lParam1*, and *lParam2* parameters are not used.

The DRV_LOAD message is always the first message that a device driver receives.

DRV_OPEN

Directs the driver to open an new instance.

- Returns a nonzero value if successful or zero otherwise.

dwDriverId
Identifier of the installable driver.

hdrvr
Handle of the installable driver instance.

lParam1
Address of a null-terminated, wide-character string that specifies configuration information used to open the instance. If no configuration information is available, either this string is empty or the parameter is NULL.

lParam2
32-bit driver-specific data.

If the driver returns a nonzero value, the system uses that value as the driver identifier (the *dwDriverId* parameter) in messages it subsequently sends to the driver instance. The driver can return any type of value as the identifier. For example, some drivers return memory addresses that point to instance-specific information. Using this method of specifying identifiers for a driver instance gives the drivers ready access to the information while they are processing messages.

DRV_POWER

Notifies the driver that power to the device is being turned on or off.

- Returns nonzero if successful or zero otherwise.

dwDriverId
Identifier of the installable driver. This is the same value previously returned by the driver from the DRV_OPEN message.

hdrvr
Handle of the installable driver instance.

The *lParam1* and *lParam2* parameters are not used.

DRV_QUERYCONFIGURE

Directs the driver to specify whether it supports custom configuration.

- Returns a nonzero value if the driver can display a configuration dialog box or zero otherwise.

dwDriverId
Identifier of the installable driver. This is the same value previously returned by the driver from the DRV_OPEN message.

hdrvr
Handle of the installable driver instance.

The *lParam1* and *lParam2* parameters are not used.

DRV_REMOVE

Notifies the driver that it is about to be removed from the system. When a driver receives this message, it should remove any sections it created in the registry.

- No return value.

dwDriverId
: Identifier of the installable driver. This is the same value previously returned by the driver from the DRV_OPEN message.

hdrvr
: Handle of the installable driver instance.

The *lParam1* and *lParam2* parameters are not used.

DRVCONFIGINFO

```
typedef struct tagDRVCONFIGINFO {
    DWORD dwDCISize;             // see below
    LPCWSTR lpszDCISectionName;  // see below
    LPCWSTR lpszDCIAliasName;    // see below
} DRVCONFIGINFO;
```

Contains the registry key and value names associated with the installable driver.

dwDCISize
Size of the structure, in bytes.

lpszDCISectionName
Address of a null-terminated, wide-character string specifying the name of the registry key associated with the driver.

lpszDCIAliasName
Address of a null-terminated, wide-character string specifying the name of the registry value associated with the driver.

Multimedia PC Specifications

The Multimedia PC (MPC) Marketing Council has developed two specifications to encourage the adoption of multimedia capabilities. The Level 1 specification, developed in 1990, provides a baseline definition of multimedia computing in functionality, hardware components, and software components. The Level 2 specification, issued in 1993, builds on the first specification and focuses on enhanced multimedia capabilities since the first specification was issued.

This appendix summarizes the specifications. For more information or complete specifications, contact the MPC Marketing Council at the following address:

Multimedia PC Marketing Council
1730 M Street NW, Suite 707
Washington, DC 20036

Level 1 Specification

The Level 1 specification was developed to encourage the adoption of basic multimedia capabilities at a minimum performance level. The most common multimedia components available to the marketplace when the Level 1 specification was issued included several compact disc - read-only memory (CD-ROM) drives that provided data at sustained transfer rates varying from 90 to 150 kilobytes per second, 8-bit sound cards, and 16-color and 256-color (SVGA) video adapters.

Level 1 System Resources

The minimum configuration for a PC to satisfy the Level 1 specification includes the following items:

- A 386SX microprocessor with 2 megabytes (MB) of random-access memory (RAM)
- A 3.5-inch high-density floppy disk drive
- A hard disk drive with at least 30 MB of disk space
- A color monitor with a display resolution of 640 by 480 pixels with 16 colors
- System software that offers binary compatibility with the Microsoft Windows operating system version 3.0

Level 1 Optical Storage

The minimum performance optical storage device is a CD-ROM drive that meets the following criteria:

- A sustained data transfer rate of 150 kilobytes per second
- A CPU bandwidth usage of 40 percent or less when maintaining a sustained data transfer rate of 150 kilobytes per second
- An average seek time of 1 second or less

Level 1 Audio Requirements

The audio subsystem of a PC satisfying the Level 1 specification includes the following items:

- An 8-bit digital-to-analog converter (DAC) capable of processing waveform-audio files recorded at 22.05 and 11.025 kHz sampling rates
- An 8-bit analog-to-digital converter (ADC) capable of recording waveform-audio files at the sample rate of 11.025 kHz through an external source, such as a microphone
- Internal synthesizer capabilities with four or nine multivoice, multitimbral capacity, and two simultaneous percussive notes

Level 2 Specification

The Level 2 specification was developed to encourage the adoption of enhanced multimedia capabilities. This specification builds on the requirements set in the Level 1 specification and is a superset of it. The Level 2 specification defines the minimum system functionality for enhanced multimedia capabilities. It is not a recommendation for a system configuration.

Level 2 System Resources

The minimum configuration for a PC to satisfy the Level 2 specification includes the following items:

- A 25 Mhz 486SX microprocessor with 4 MB of RAM
- A 3.5-inch high-density floppy disk drive
- A hard disk drive with at least 160 MB of disk space
- A 101-key keyboard with a standard DIN connector or one that provides identical functionality by using key combinations
- A two-button mouse with a serial or bus connector
- A MIDI (Musical Instrument Digital Interface) port that includes MIDI Out, MIDI In, and MIDI Thru, and that has interrupt support for input and FIFO transfer
- An IBM-style analog or digital joystick (game) port
- A color monitor with a display resolution of 640 by 480 pixels with 65,536 colors
- System software that offers binary compatibility with Windows 3.0 or Windows 3.1

In addition, the recommended performance goal for the video adapter (VGA+) is 1.2 million pixels per second using 40 percent or less of the CPU bandwidth. The device-independent bitmaps (DIBs) used to measure this performance goal have color depths of 1, 4, and 8 bits, and can use run-length encoding or be unencoded. A second method of measuring the video performance is to deliver 256-color, 320 by 240 pixel digital-video images at 15 frames per second.

Level 2 Optical Storage

The minimum performance optical storage device is a double-speed CD-ROM drive that meets the following criteria:

- A sustained data transfer rate of 300 kilobytes per second
- A CPU bandwidth usage of 40 percent or less when maintaining a sustained data transfer rate of 150 kilobytes per second, or a CPU bandwidth of 60 percent or less when maintaining a sustained data transfer rate of 300 kilobytes per second
- An average seek time of 400 milliseconds or less
- A 10,000 hour mean-time-between-failures rating
- CD-ROM XA ready (mode 1 capable, mode 2 form 1 capable, mode form 2 capable)
- Multisession capable
- MSCDEX-2.2 driver or equivalent that implements the extended audio functions

The recommended CPU bandwidth should be reached by using a read-block size of at least 16K and a lead time of no more than the time needed to load one read-block of data into the CD-ROM buffer.

Level 2 Audio Requirements

The audio subsystem of a PC satisfying the Level 2 specification includes the following items:

- A CD-ROM driver with CD-DA (Red Book audio) outputs and volume control
- A 16-bit DAC with the following characteristics:
  - Linear PCM (Pulse Code Modulation) sampling
  - DMA or FIFO buffered transfer capability with interrupt on buffer empty
  - Mandatory sample rates of 44.1, 22.05, and 11.025 kHz
  - Stereo channels
  - CPU bandwidth usage of 10 percent or less when outputting audio of 22.05 or 11.025 kHz sample rate, or a CPU bandwidth of 15 percent or less when outputting audio of 44.1 kHz sample rate
- A 16-bit ADC with the following characteristics:
  - Linear PCM sampling
  - DMA or FIFO buffered transfer capability with interrupt on buffer empty
  - Mandatory sample rates of 44.1, 22.05, and 11.025 kHz
  - Microphone input
- Internal synthesizer capabilities with multivoice, multitimbral, six simultaneous melody notes plus two simultaneous percussive notes
- Internal mixing with the following capabilities:
  - Can combine three audio sources and present the output as a stereo, line-level audio signal at the back panel
  - Mixing sources are CD Red Book audio, synthesizer, and DAC
  - Each mixing source has 3-bit volume control with a logarithmic taper

Manufacturer and Product Identifiers

This appendix documents the manufacturer and product identifiers defined for multimedia applications. These identifiers are used when an application issues a query about the installed devices on a computer.

Manufacturer Identifiers

Company name	Identifier
Advanced Gravis Computer Technology, Ltd.	MM_GRAVIS
Antex Electronics Corporation	MM_ANTEX
APPS Software	MM_APPS
Artisoft, Inc.	MM_ARTISOFT
AST Research, Inc.	MM_AST
ATI Technologies, Inc.	MM_ATI
Audio, Inc.	MM_AUDIOFILE
Audio Processing Technology	MM_APT
Audio Processing Technology	MM_AUDIOPT
Auravision Corporation	MM_AURAVISION
Aztech Labs, Inc.	MM_AZTECH
Canopus, Co., Ltd.	MM_CANOPUS
Compusic	MM_COMPUSIC
Computer Aided Technology, Inc.	MM_CAT
Computer Friends, Inc.	MM_COMPUTER_FRIENDS
Control Resources Corporation	MM_CONTROLRES
Creative Labs, Inc.	MM_CREATIVE
Dialogic Corporation	MM_DIALOGIC
Dolby Laboratories, Inc.	MM_DOLBY
DSP Group, Inc.	MM_DSP_GROUP
DSP Solutions, Inc.	MM_DSP_SOLUTIONS
Echo Speech Corporation	MM_ECHO
ESS Technology, Inc.	MM_ESS
Everex Systems, Inc.	MM_EVEREX
EXAN, Ltd.	MM_EXAN
Fujitsu, Ltd.	MM_FUJITSU
I/O Magic Corporation	MM_IOMAGIC
ICL Personal Systems	MM_ICL_PS
Ing. C. Olivetti & C., S.p.A.	MM_OLIVETTI
Integrated Circuit Systems, Inc.	MM_ICS
Intel Corporation	MM_INTEL
InterActive, Inc.	MM_INTERACTIVE
International Business Machines	MM_IBM
Iterated Systems, Inc.	MM_ITERATEDSYS
Logitech, Inc.	MM_LOGITECH
Lyrrus, Inc.	MM_LYRRUS
Matsushita Electric Corporation of America	MM_MATSUSHITA
Media Vision, Inc.	MM_MEDIAVISION
Metheus Corporation	MM_METHEUS

microEngineering Labs	MM_MELABS
Microsoft Corporation	MM_MICROSOFT
MOSCOM Corporation	MM_MOSCOM
Motorola, Inc.	MM_MOTOROLA
Natural MicroSystems Corporation	MM_NMS
NCR Corporation	MM_NCR
NEC Corporation	MM_NEC
New Media Corporation	MM_NEWMEDIA
OKI	MM_OKI
OPTi, Inc.	MM_OPTI
Roland Corporation	MM_ROLAND
SCALACS	MM_SCALACS
Seiko Epson Corporation, Inc.	MM_EPSON
Sierra Semiconductor Corporation	MM_SIERRA
Silicon Software, Inc.	MM_SILICONSOFT
Sonic Foundry	MM_SONICFOUNDRY
Speech Compression	MM_SPEECHCOMP
Supermac Technology, Inc.	MM_SUPERMAC
Tandy Corporation	MM_TANDY
Toshihiko Okuhura, Korg, Inc.	MM_KORG
Truevision, Inc.	MM_TRUEVISION
Turtle Beach Systems	MM_TURTLE_BEACH
Video Associates Labs, Inc.	MM_VAL
VideoLogic, Inc.	MM_VIDEOLOGIC
Visual Information Technologies, Inc.	MM_VITEC
VocalTec, Inc.	MM_VOCALTEC
Voyetra Technologies	MM_VOYETRA
Wang Laboratories	MM_WANGLABS
Willow Pond Corporation	MM_WILLOWPOND
Winnov, LP	MM_WINNOV
Xebec Multimedia Solutions Limited	MM_XEBEC
Yamaha Corporation of America	MM_YAMAHA

Microsoft Corporation Product Identifiers

Product name	Identifier
Adlib-compatible synthesizer	MM_ADLIB
G.711 codec	MM_MSFT_ACM_G711
GSM 610 codec	MM_MSFT_ACM_GSM610
IMA ADPCM codec	MM_MSFT_ACM_IMAADPCM
Joystick adapter	MM_PC_JOYSTICK
MIDI mapper	MM_MIDI_MAPPER
MPU 401-compatible MIDI input port	MM_MPU401_MIDIIN
MPU 401-compatible MIDI output port	MM_MPU401_MIDIOUT
MS ADPCM codec	MM_MSFT_ACM_MSADPCM
MS audio board stereo FM synthesizer	MM_MSFT_WSS_FMSYNTH_STEREO
MS audio board aux port	MM_MSFT_WSS_AUX
MS audio board mixer driver	MM_MSFT_WSS_MIXER
MS audio board waveform input	MM_MSFT_WSS_WAVEIN
MS audio board waveform output	MM_MSFT_WSS_WAVEOUT
MS audio compression manager	MM_MSFT_MSACM
MS filter	MM_MSFT_ACM_MSFILTER
MS OEM audio aux	MM_MSFT_WSS_OEM_AUX

port	
MS OEM audio board mixer driver	MM_MSFT_WSS_OEM_MIXER
MS OEM audio board stereo FM synthesizer	MM_MSFT_WSS_OEM_FMSYNTH_STEREO
MS OEM audio board waveform input	MM_MSFT_WSS_OEM_WAVEIN
MS OEM audio board waveform output	MM_MSFT_WSS_OEM_WAVEOUT
MS vanilla driver aux (CD)	MM_MSFT_GENERIC_AUX_CD
MS vanilla driver aux (line in)	MM_MSFT_GENERIC_AUX_LINE
MS vanilla driver aux (mic)	MM_MSFT_GENERIC_AUX_MIC
MS vanilla driver MIDI external out	MM_MSFT_GENERIC_MIDIOUT
MS vanilla driver MIDI in	MM_MSFT_GENERIC_MIDIIN
MS vanilla driver MIDI synthesizer	MM_MSFT_GENERIC_MIDISYNTH
MS vanilla driver waveform input	MM_MSFT_GENERIC_WAVEIN
MS vanilla driver wavefrom output	MM_MSFT_GENERIC_WAVEOUT
PC speaker waveform output	MM_PCSPEAKER_WAVEOUT
PCM converter	MM_MSFT_ACM_PCM
Sound Blaster internal synthesizer	MM_SNDBLST_SYNTH

Sound Blaster MIDI input port	MM_SNDBLST_MIDIIN
Sound Blaster MIDI output port	MM_SNDBLST_MIDIOUT
Sound Blaster waveform input	MM_SNDBLST_WAVEIN
Sound Blaster waveform output	MM_SNDBLST_WAVEOUT
Wave mapper	MM_WAVE_MAPPER

Microsoft Windows Sound System Drivers

Driver	Identifier
Sound Blaster 16 waveform output	MM_MSFT_SB16_WAVEOUT
Sound Blaster 16 aux (CD)	MM_WSS_SB16_AUX_CD
Sound Blaster 16 aux (CD)	MM_MSFT_SB16_AUX_CD
Sound Blaster 16 aux (line in)	MM_WSS_SB16_AUX_LINE
Sound Blaster 16 aux (line in)	MM_MSFT_SB16_AUX_LINE
Sound Blaster 16 FM synthesizer	MM_WSS_SB16_SYNTH
Sound Blaster 16 FM synthesizer	MM_MSFT_SB16_SYNTH
Sound Blaster 16 MIDI out	MM_WSS_SB16_MIDIOUT
Sound Blaster 16 MIDI out	MM_MSFT_SB16_MIDIOUT
Sound Blaster 16 MIDI in	MM_WSS_SB16_MIDIIN
Sound Blaster 16 MIDI in	MM_MSFT_SB16_MIDIIN
Sound Blaster 16 mixer device	MM_WSS_SB16_MIXER
Sound Blaster 16 mixer device	MM_MSFT_SB16_MIXER
Sound Blaster 16 waveform input	MM_WSS_SB16_WAVEIN
Sound Blaster 16 waveform input	MM_MSFT_SB16_WAVEIN
Sound Blaster 16 waveform output	MM_WSS_SB16_WAVEOUT
Sound Blaster Pro aux (CD)	MM_WSS_SBPRO_AUX_CD
Sound Blaster Pro aux (CD)	MM_MSFT_SBPRO_AUX_CD
Sound Blaster Pro aux (line in)	MM_WSS_SBPRO_AUX_LINE
Sound Blaster Pro aux (line in)	MM_MSFT_SBPRO_AUX_LINE
Sound Blaster Pro FM synthesizer	MM_WSS_SBPRO_SYNTH
Sound Blaster Pro FM synthesizer	MM_MSFT_SBPRO_SYNTH
Sound Blaster Pro MIDI in	MM_WSS_SBPRO_MIDIIN
Sound Blaster Pro MIDI in	MM_MSFT_SBPRO_MIDIIN
Sound Blaster Pro MIDI out	MM_WSS_SBPRO_MIDIOUT
Sound Blaster Pro MIDI out	MM_MSFT_SBPRO_MIDIOUT
Sound Blaster Pro mixer	MM_WSS_SBPRO_MIXER
Sound Blaster Pro mixer	MM_MSFT_SBPRO_MIXER
Sound Blaster Pro waveform input	MM_WSS_SBPRO_WAVEIN

Sound Blaster Pro waveform input	MM_MSFT_SBPRO_WAVEIN
Sound Blaster Pro waveform output	MM_WSS_SBPRO_WAVEOUT
Sound Blaster Pro waveform output	MM_MSFT_SBPRO_WAVEOUT
WSS NT aux	MM_MSFT_WSS_NT_AUX
WSS NT FM synthesizer	MM_MSFT_WSS_NT_FMSYNTH_STEREO
WSS NT mixer	MM_MSFT_WSS_NT_MIXER
WSS NT wave in	MM_MSFT_WSS_NT_WAVEIN
WSS NT wave out	MM_MSFT_WSS_NT_WAVEOUT

Product Identifiers

Company	Product identifiers
Artisoft, Inc.	MM_ARTISOFT_SBWAVEIN MM_ARTISOFT_SBWAVEOUT
Audio Processing Technology	MM_APT_ACE100CD
Aztech Labs, Inc.	MM_AZTECH_AUX_CD MM_AZTECH_AUX_LINE MM_AZTECH_AUX_MIC MM_AZTECH_DSP16_FMSYNTH MM_AZTECH_DSP16_WAVEIN MM_AZTECH_DSP16_WAVEOUT MM_AZTECH_DSP16_WAVESYNTH MM_AZTECH_FMSYNTH MM_AZTECH_MIDIIN MM_AZTECH_MIDIOUT MM_AZTECH_PRO16_FMSYNTH MM_AZTECH_PRO16_WAVEIN MM_AZTECH_PRO16_WAVEOUT MM_AZTECH_WAVEIN MM_AZTECH_WAVEOUT
Computer Aided Technology, Inc.	MM_CAT_WAVEOUT
Creative Labs, Inc.	MM_CREATIVE_AUX_CD MM_CREATIVE_AUX_LINE MM_CREATIVE_AUX_MASTER MM_CREATIVE_AUX_MIC MM_CREATIVE_AUX_MIDI MM_CREATIVE_AUX_PCSPK MM_CREATIVE_AUX_WAVE MM_CREATIVE_FMSYNTH_MONO MM_CREATIVE_FMSYNTH_STEREO MM_CREATIVE_MIDIIN MM_CREATIVE_MIDIOUT MM_CREATIVE_SB15_WAVEIN MM_CREATIVE_SB15_WAVEOUT MM_CREATIVE_SB16_MIXER MM_CREATIVE_SB20_WAVEIN MM_CREATIVE_SB20_WAVEOUT MM_CREATIVE_SBP16_WAVEIN MM_CREATIVE_SBP16_WAVEOUT MM_CREATIVE_SBPRO_MIXER MM_CREATIVE_SBPRO_WAVEIN MM_CREATIVE_SBPRO_WAVEOUT
DSP Group, Inc.	MM_DSP_GROUP_TRUESPEECH
DSP Solutions, Inc.	MM_DSP_SOLUTIONS_AUX MM_DSP_SOLUTIONS_SYNTH MM_DSP_SOLUTIONS_WAVEIN MM_DSP_SOLUTIONS_WAVEOUT
Echo Speech Corporation	MM_ECHO_AUX MM_ECHO_MIDIIN

	MM_ECHO_MIDIOUT
	MM_ECHO_SYNTH
	MM_ECHO_WAVEIN
	MM_ECHO_WAVEOUT
ESS Technology, Inc.	MM_ESS_AMAUX
	MM_ESS_AMMIDIIN
	MM_ESS_AMMIDIOUT
	MM_ESS_AMSYNTH
	MM_ESS_AMWAVEIN
	MM_ESS_AMWAVEOUT
Everex Systems, Inc.	MM_EVEREX_CARRIER
I/O Magic Corporation	MM_IOMAGIC_TEMPO_AUXOUT
	MM_IOMAGIC_TEMPO_MIDIOUT
	MM_IOMAGIC_TEMPO_MXDOUT
	MM_IOMAGIC_TEMPO_SYNTH
	MM_IOMAGIC_TEMPO_WAVEIN
	MM_IOMAGIC_TEMPO_WAVEOUT
Ing. C. Olivetti & C., S.p.A.	MM_OLIVETTI_ACM_ADPCM
	MM_OLIVETTI_ACM_CELP
	MM_OLIVETTI_ACM_GSM
	MM_OLIVETTI_ACM_OPR
	MM_OLIVETTI_ACM_SBC
	MM_OLIVETTI_AUX
	MM_OLIVETTI_JOYSTICK
	MM_OLIVETTI_MIDIIN
	MM_OLIVETTI_MIDIOUT
	MM_OLIVETTI_MIXER
	MM_OLIVETTI_SYNTH
	MM_OLIVETTI_WAVEIN
	MM_OLIVETTI_WAVEOUT
Integrated Circuit Systems, Inc.	MM_ICS_WAVEDECK_AUX
	MM_ICS_WAVEDECK_MIXER
	MM_ICS_WAVEDECK_SYNTH
	MM_ICS_WAVEDECK_WAVEIN
	MM_ICS_WAVEDECK_WAVEOUT
InterActive, Inc.	MM_INTERACTIVE_WAVEIN
	MM_INTERACTIVE_WAVEOUT
International Business Machines	MM_IBM_PCMCIA_AUX
	MM_IBM_PCMCIA_MIDIIN
	MM_IBM_PCMCIA_MIDIOUT
	MM_IBM_PCMCIA_SYNTH
	MM_IBM_PCMCIA_WAVEIN
	MM_IBM_PCMCIA_WAVEOUT
	MM_MMOTION_WAVEAUX
	MM_MMOTION_WAVEIN
	MM_MMOTION_WAVEOUT
Iterated Systems, Inc.	MM_ITERATEDSYS_FUFCODEC
Lyrrus, Inc.	MM_LYRRUS_BRIDGE_GUITAR
Matsushita Electric Corporation of America	MM_MATSUSHITA_AUX
	MM_MATSUSHITA_FMSYNTH_STEREO
	MM_MATSUSHITA_MIXER
	MM_MATSUSHITA_WAVEIN

	MM_MATSUSHITA_WAVEOUT
Media Vision, Inc.	MM_MEDIAVISION_CDPC
	MM_CDPC_AUX
	MM_CDPC_MIDIIN
	MM_CDPC_MIDIOUT
	MM_CDPC_MIXER
	MM_CDPC_SYNTH
	MM_CDPC_WAVEIN
	MM_CDPC_WAVEOUT
	MM_OPUS401_MIDIIN
	MM_OPUS401_MIDIOUT
	MM_MEDIAVISION_OPUS1208
	MM_OPUS1208_AUX
	MM_OPUS1208_MIXER
	MM_OPUS1208_SYNTH
	MM_OPUS1208_WAVEIN
	MM_OPUS1208_WAVEOUT
	MM_MEDIAVISION_OPUS1216
	MM_OPUS1216_AUX
	MM_OPUS1216_MIDIIN
	MM_OPUS1216_MIDIOUT
	MM_OPUS1216_MIXER
	MM_OPUS1216_SYNTH
	MM_OPUS1216_WAVEIN
	MM_OPUS1216_WAVEOUT
	MM_MEDIAVISION_PROAUDIO
	MM_PROAUD_AUX
	MM_PROAUD_MIDIIN
	MM_PROAUD_MIDIOUT
	MM_PROAUD_MIXER
	MM_MEDIAVISION_PROAUDIO_16
	MM_PROAUD_16_AUX
	MM_PROAUD_16_MIDIIN
	MM_PROAUD_16_MIDIOUT
	MM_PROAUD_16_MIXER
	MM_PROAUD_16_SYNTH
	MM_PROAUD_16_WAVEIN
	MM_PROAUD_16_WAVEOUT
	MM_MEDIAVISION_PROAUDIO_PLUS
	MM_PROAUD_PLUS_AUX
	MM_PROAUD_PLUS_MIDIIN
	MM_PROAUD_PLUS_MIDIOUT
	MM_PROAUD_PLUS_MIXER
	MM_PROAUD_PLUS_SYNTH
	MM_PROAUD_PLUS_WAVEIN
	MM_PROAUD_PLUS_WAVEOUT
	MM_PROAUD_SYNTH
	MM_PROAUD_WAVEIN
	MM_PROAUD_WAVEOUT
	MM_MEDIAVISION_PROSTUDIO_16
	MM_STUDIO_16_AUX
	MM_STUDIO_16_MIDIIN
	MM_STUDIO_16_MIDIOUT

	MM_STUDIO_16_MIXER MM_STUDIO_16_SYNTH MM_STUDIO_16_WAVEIN MM_STUDIO_16_WAVEOUT
	MM_MEDIAVISION_THUNDER MM_THUNDER_AUX MM_THUNDER_SYNTH MM_THUNDER_WAVEIN MM_THUNDER_WAVEOUT
	MM_MEDIAVISION_TPORT MM_TPORT_SYNTH MM_TPORT_WAVEIN MM_TPORT_WAVEOUT
Metheus Corporation	MM_METHEUS_ZIPPER
microEngineering Labs	MM_MELABS_MIDI2GO
MOSCOM Corporation	MM_MOSCOM_VPC2400
NCR Corporation	MM_NCR_BA_AUX MM_NCR_BA_MIXER MM_NCR_BA_SYNTH MM_NCR_BA_WAVEIN MM_NCR_BA_WAVEOUT
New Media Corporation	MM_NEWMEDIA_WAVJAMMER
OPTi, Inc.	MM_OPTI_M16_AUX MM_OPTI_M16_FMSYNTH_STEREO MM_OPTI_M16_MIDIIN MM_OPTI_M16_MIDIOUT MM_OPTI_M16_MIXER MM_OPTI_M16_WAVEIN MM_OPTI_M16_WAVEOUT MM_OPTI_M32_AUX MM_OPTI_M32_MIDIIN MM_OPTI_M32_MIDIOUT MM_OPTI_M32_MIXER MM_OPTI_M32_SYNTH_STEREO MM_OPTI_M32_WAVEIN MM_OPTI_M32_WAVEOUT MM_OPTI_P16_AUX MM_OPTI_P16_FMSYNTH_STEREO MM_OPTI_P16_MIDIIN MM_OPTI_P16_MIDIOUT MM_OPTI_P16_MIXER MM_OPTI_P16_WAVEIN MM_OPTI_P16_WAVEOUT
Roland Corporation	MM_ROLAND_MPU401_MIDIIN MM_ROLAND_MPU401_MIDIOUT MM_ROLAND_SC7_MIDIIN MM_ROLAND_SC7_MIDIOUT MM_ROLAND_SERIAL_MIDIIN MM_ROLAND_SERIAL_MIDIOUT MM_ROLAND_SMPU_MIDIINA MM_ROLAND_SMPU_MIDIINB MM_ROLAND_SMPU_MIDIOUTA

	MM_ROLAND_SMPU_MIDIOUTB
Sierra Semiconductor Corporation	MM_SIERRA_ARIA_AUX MM_SIERRA_ARIA_AUX2 MM_SIERRA_ARIA_MIDIIN MM_SIERRA_ARIA_MIDIOUT MM_SIERRA_ARIA_SYNTH MM_SIERRA_ARIA_WAVEIN MM_SIERRA_ARIA_WAVEOUT
Silicon Software, Inc.	MM_SILICONSOFT_SC1_WAVEIN MM_SILICONSOFT_SC1_WAVEOUT MM_SILICONSOFT_SC2_WAVEIN MM_SILICONSOFT_SC2_WAVEOUT MM_SILICONSOFT_SOUNDJR2_WAVEOUT MM_SILICONSOFT_SOUNDJR2PR_WAVEIN MM_SILICONSOFT_SOUNDJR2PR_WAVEOUT MM_SILICONSOFT_SOUNDJR3_WAVEOUT
Tandy Corporation	MM_TANDY_PSSJWAVEIN MM_TANDY_PSSJWAVEOUT MM_TANDY_SENS_MMAMIDIIN MM_TANDY_SENS_MMAMIDIOUT MM_TANDY_SENS_MMAWAVEIN MM_TANDY_SENS_MMAWAVEOUT MM_TANDY_SENS_VISWAVEOUT MM_TANDY_VISBIOSSYNTH MM_TANDY_VISWAVEIN MM_TANDY_VISWAVEOUT
Toshihiko Okuhura, Korg, Inc.	MM_KORG_PCIF_MIDIIN MM_KORG_PCIF_MIDIOUT
Truevision, Inc.	MM_TRUEVISION_WAVEIN1 MM_TRUEVISION_WAVEOUT1
VideoLogic, Inc.	MM_VIDEOLOGIC_MSWAVEIN MM_VIDEOLOGIC_MSWAVEOUT
Visual Information Technologies, Inc.	MM_VITEC_VMAKER MM_VITEC_VMPRO
VocalTec, Inc.	MM_VOCALTEC_WAVEIN MM_VOCALTEC_WAVEOUT
Wang Laboratories	MM_WANGLABS_WAVEIN1 MM_WANGLABS_WAVEOUT1
Winnov, LP	MM_WINNOV_CAVIAR_CHAMPAGNE MM_WINNOV_CAVIAR_VIDC MM_WINNOV_CAVIAR_WAVEIN MM_WINNOV_CAVIAR_WAVEOUT MM_WINNOV_CAVIAR_YUV8
Yamaha Corporation of America	MM_YAMAHA_GSS_AUX MM_YAMAHA_GSS_MIDIIN MM_YAMAHA_GSS_MIDIOUT MM_YAMAHA_GSS_SYNTH MM_YAMAHA_GSS_WAVEIN MM_YAMAHA_GSS_WAVEOUT

Legal Notice

Olivetti is a registered trademark of Ing. C. Olivetti.

PostScript is a registered trademark of Adobe Systems, Inc.

R4000 is a trademark of MIPS Computer Systems, Inc.

Roland is a registered trademark of Roland Corporation.

SCSI is a registered trademark of Security Control Systems, Inc.

Epson is a registered trademark of Seiko Epson Corporation, Inc.

Silicon Graphics is a registered trademark and OpenGL is a trademark of Silicon Graphics, Inc.

Stacker is a registered trademark of STAC Electronics.

Tandy is a registered trademark of Tandy Corporation.

Unicode is a registered trademark of Unicode, Incorporated.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

VAX is a trademark of Digital Equipment Corporation

Yamaha is a registered trademark of Yamaha Corporation of America.

Paintbrush is a trademark of Wordstar Atlanta Technology Center.

Microsoft Win32 Developer's Reference

You have requested information from the **Microsoft Win32 Developer's Reference**. One or more of these help files is not available on your system.