Multimedia Programmer's Guide

Microsoft® WINDOWS™ SOFTWARE DEVELOPMENT KIT

Microsoft® Windows™

Version 3.1

Multimedia Programmer's Guide

For the Microsoft Windows Operating System

Microsoft Corporation

Document Number: PC30253-0492

10 9 8 7 6 5 4 3 2 1

Contents

Chapter 6 Timer and Joystick Services

Chapter 7 Multimedia File I/O Services

About This Guide

This manual contains information you can use to write applications using the multimedia services available in the Microsoft® Windows™ operating system. It describes the architecture of the multimedia elements of Windows, as well as the application programming interface (API) for creating applications that use multimedia features.

This introduction contains background information you should review, including the following:

- The basics you should already know about writing Windows applications
- The organization of this manual
- A description of the multimedia sample applications provided with the Microsoft Windows Software Development Kit (SDK)
- Typographical and notational conventions used in this manual
- A description of other manuals provided with the SDK

What You Should Know

This manual assumes you are familiar with writing applications for Microsoft Windows, version 3.1 or later, using the C programming language. For more information on programming for Windows, see the *Microsoft Windows Software Development Kit Guide to Programming*.

Some chapters in this manual assume additional knowledge on your part. The introductions to these chapters list these assumptions.

Contents of this Manual

This manual provides information that experienced Windows programmers can use to include multimedia features in their applications. The manual includes the following information:

- Chapter 1, "Introduction to Multimedia Services," introduces the features and software architecture of the multimedia services provided by Windows and outlines the development environment for writing applications using these services.
- Chapter 2, "The Media Control Interface (MCI)," describes MCI, an extensible software interface that provides high-level, device-independent capabilities for controlling media devices such as audio hardware, animation players, and audio/visual peripherals.
- Chapter 3, "Introduction to Audio," Chapter 4, "High-Level Audio Services," and Chapter 5, "Low-Level Audio Services," all describe the audio services, including waveform audio, MIDI, and compact-disc audio.
- Chapter 6, "Timer and Joystick Services," describes high-resolution timer services and joystick input services.
- Chapter 7, "Multimedia File I/O Services," describes the multimedia file I/O services.
- The Glossary defines terms used in this manual and in the *Multimedia Programmer's Reference*.

You should read Chapters 1 and 2 for a general overview of Windows multimedia services. After reading these chapters, refer to those chapters that discuss the particular multimedia features you want to include in your applications.

Sample Applications

Each chapter includes sample C code fragments illustrating important concepts discussed in the chapter. The SDK also includes several complete program examples that illustrate how to use the multimedia services. Use the SDK Setup program to install these sample programs on your system. Compile and run these examples to see how the functions work in actual programs, or use the examples as the basis for writing your own multimedia applications.

The following table lists the multimedia sample applications provided with the SDK.

Program	Function	Demonstrates
LOWPASS	Low-pass filter for waveform-audio files	Multimedia file I/O functions
MCITEST	Interactively sends command strings to MCI devices	MCI command-string interface
MIDIMON	Displays a textual listing of MIDI input messages	MIDI recording functions, low-level callback functions
REVERSE	Plays a waveform-audio file in reverse	Multimedia file I/O functions, waveform-audio functions

Document Conventions

The following document conventions are used throughout this manual:

Type Style	Used For
bold	A specific term intended to be used literally; for example, language key words, function names, and macro names. You must type these terms exactly as shown.
italic	Placeholders for information you must provide. For example, the following syntax for the **sndPlaySound** function indicates you must substitute values for the *lpszSound* and *wFlags* parameters, separated by a comma: **BOOL sndPlaySound**(*lpszSound, wFlags*) Italic is also used for terms defined in text and the Glossary.
ALL CAPITALS	Directory names, filenames, and acronyms.
`Monospace`	Code examples.
Vertical ellipsis . . .	Indicates a portion of the program is omitted in a program example.
Horizontal ellipsis ...	For a given function, there are several functions that have the same form but different prefixes; for example, **...GetNumDevs** substitutes for the **auxGetNumDevs**, **midiInGetNumDevs**, **midiOutGetNumDevs**, **waveInGetNumDevs**, and **waveOutGetNumDevs** functions, all of which have similar syntax and functionality. Also indicates a single statement is omitted in a program example.

Related Manuals

This manual explains how to write applications using the multimedia services provided by Windows. The SDK also includes the following manuals providing additional information about Windows programming:

- The *Microsoft Windows Software Development Kit Getting Started* manual provides an orientation to the SDK, explains how to install the SDK software, and highlights the changes for version 3.1.
- The *Microsoft Windows Software Development Kit Programmer's Reference* provides a comprehensive guide to all the details of the Microsoft Windows API. The reference consists of four volumes:
  - Volume 1 presents an overview of the API.
  - Volume 2 lists, in alphabetical order the functions that make up the API.
  - Volume 3 describes Windows API messages, data structures, and macros.
  - Volume 4 describes resources.
- The *Microsoft Windows Software Development Kit Programming Tools* manual explains how to use the software-development tools you'll need to build Windows applications, such as debuggers and specialized SDK editors.
- The *Microsoft Windows Software Development Kit Guide to Programming* explains how to write Windows applications and provides sample applications that you can use as templates for writing your own applications.
- The *Microsoft Windows Software Development Kit Multimedia Programmer's Reference* is a companion manual to this *Multimedia Programmer's Guide*, providing a definitive summary of the Windows multimedia API. It describes multimedia functions, messages, data structures, file formats, and MCI command strings.

Chapter 1
Introduction to Multimedia Services

Microsoft Windows provides application developers with high-level and low-level services for developing multimedia applications using the extended capabilities of a multimedia personal computer.

This chapter covers the following topics:

- An introduction to the features of the multimedia services provided by Windows
- The software architecture of the multimedia services
- The design philosophy and implementation of the multimedia services
- Basic information about developing applications using the multimedia services

Before reading this chapter, you should install Windows and the Microsoft Windows Software Development Kit on your computer. See the *Getting Started* manual for installation instructions.

About Windows Multimedia Services

Windows provides the following multimedia services:

- A Media Control Interface (MCI) for controlling media devices.
  - Extensible string-based and message-based interfaces for communicating with MCI device drivers.
  - MCI device drivers for playing and recording waveform audio, playing MIDI (Musical Instrument Digital Interface) files, and playing compact disc audio from a CD-ROM disc drive.
- Low-level API support for multimedia-related services.
  - Low-level support for playing and recording audio with waveform and MIDI audio devices.
  - Low-level support for precision timer services.
- Multimedia file I/O services providing buffered and unbuffered file I/O, and support for standard IBM/Microsoft Resource Interchange File Format (RIFF) files. The services are extensible with custom I/O procedures that can be shared among applications.
- Control Panel options that let users change display drivers, set up a screen saver, install multimedia device drivers, assign waveform sounds to system alerts, and configure the MIDI Mapper.
- A MIDI Mapper supporting standard MIDI patch services. This allows MIDI files to be authored independently of end-user MIDI synthesizer setups.

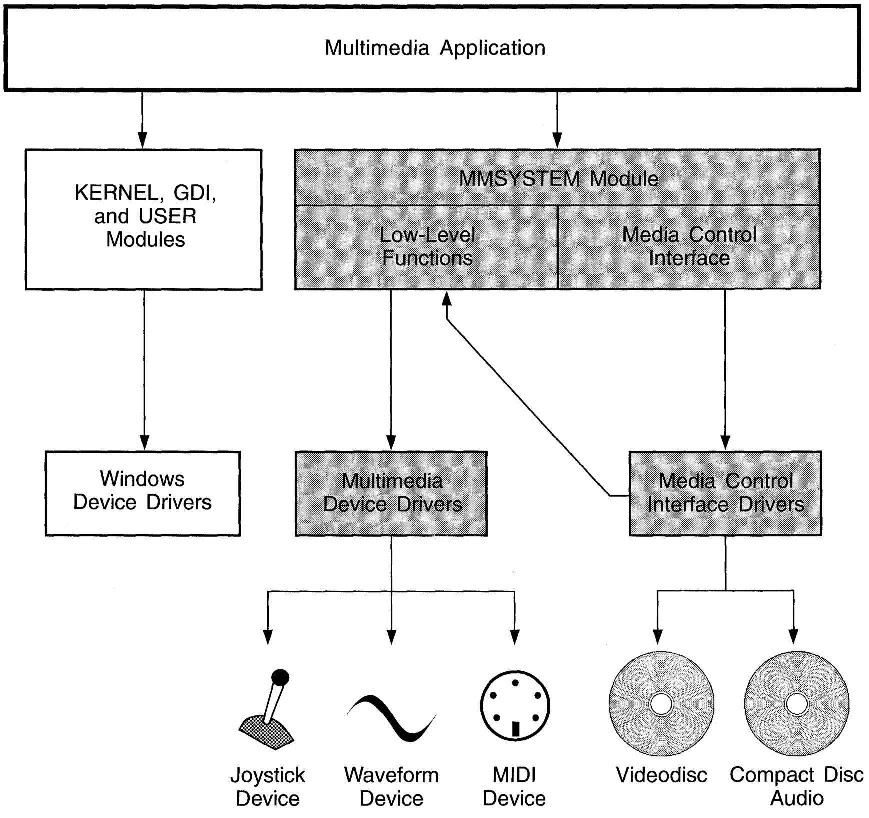
- Interrupt-callable functions for timer and MIDI devices, providing real-time response for time-critical uses. A list of the interrupt-callable functions follows:
  - **midiOutLongMsg**
  - **midiOutShortMsg**
  - **timeGetSystemTime**
  - **timeGetTime**
  - **timeKillEvent**
  - **timeSetEvent**

Architecture of Windows Multimedia Services

Although the multimedia services are provided by a number of files, the overall architecture can be viewed as consisting of just a few software modules:

- The MMSYSTEM library providing the Media Control Interface services and low-level multimedia support functions.
- Multimedia device drivers providing communication between the low-level MMSYSTEM functions and multimedia devices such as waveform, MIDI, and timer hardware.
- Drivers for the Media Control Interface providing high-level control of media devices.

The following illustration shows the relationship between the Windows modules that provide multimedia services.



The relationship between Windows and the Multimedia extensions.

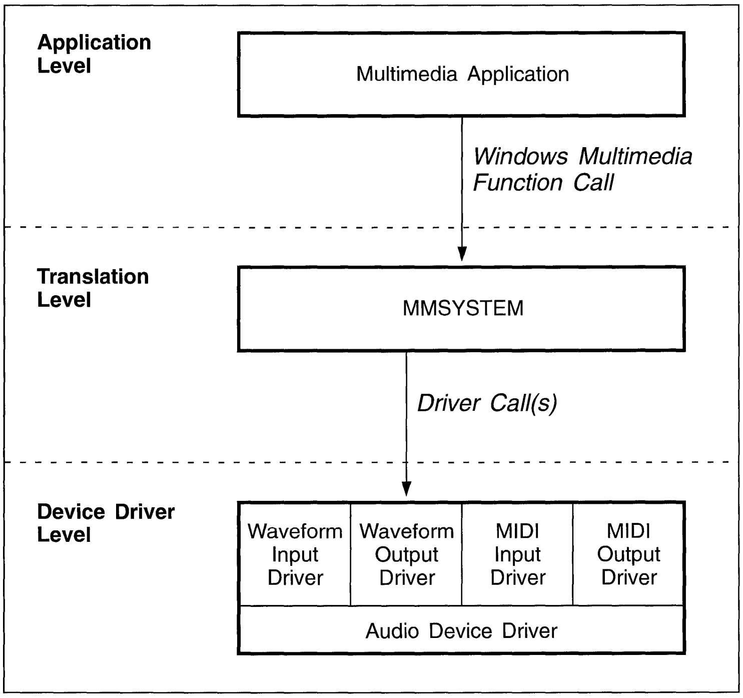
This illustration is a simplified view of the relationship between the various Windows modules. Connections between modules indicate control flow.

Windows Multimedia Design Philosophy

The architecture of the multimedia services is designed around the concepts of *extensibility* and *device-independence*. Extensibility allows the software architecture to easily accommodate advances in technology without changes to the architecture itself. Device-independence allows multimedia applications to be easily developed that will run on a wide range of hardware providing different levels of multimedia support. Three design elements of the system software provide extensibility and device-independence:

- A translation layer (MMSYSTEM) that isolates applications from device drivers and centralizes device-independent code.
- Run-time linking that allows the translation layer to link to the drivers it needs.
- A well-defined and consistent driver interface that minimizes special-case code and makes the installation and upgrade process easier.

In the following illustration, you can see how the translation layer translates a multimedia function call into a call to an audio device driver:



Relationship between an application and multimedia device drivers.

Some function calls might result in multiple driver calls, or they might be handled by MMSYSTEM without causing any driver calls.

Building a Multimedia Application

Before writing a multimedia application, you should be familiar with programming in the Windows environment.

Including Header Files

When using the multimedia services, you must include the MMSYSTEM.H header file in all source modules that call a multimedia function. This header file depends on declarations made in the WINDOWS.H header file, so you must first include WINDOWS.H. The MMSYSTEM.H header file provides function prototypes as well as definitions of multimedia data types and constants.

Linking with Multimedia Libraries

In addition to the normal Windows libraries, you must also link to the MMSYSTEM.LIB import library when linking an application that uses multimedia functions.

Debugging a Multimedia Application

The Microsoft Windows Software Development Kit (SDK) includes a debugging version of the MMSYSTEM module to aid in debugging applications that use multimedia services. This version of MMSYSTEM.LIB displays commands sent to MCI and performs error checking unavailable in the retail version.

To control the MCI debugging output, add an MCI entry to the [mmsystem] section of SYSTEM.INI. Assign the value 1 to this entry (MCI = 1) to enable MCI debugging output. The debugging output returns to the Windows function DebugOutput a string representation of each MCI command. To disable the MCI debugging output, either remove the MCI entry from the [mmsystem] section of SYSTEM.INI or assign the value 0 to the entry.

The services in the debug MMSYSTEM module supplement other Windows debugging services and tools provided with the Windows Software Development Kit. See the *Microsoft Windows Programming Tools* manual provided with the SDK for information about using these debugging tools.

Chapter 2

The Media Control Interface (MCI)

The Media Control Interface (MCI) provides Windows applications with device-independent capabilities for controlling media devices such as audio hardware, videodisc players, and animation players. This chapter presents an introduction to MCI and shows how to get started using it in Windows applications.

The chapter covers the following topics:

- The architecture of MCI
- The MCI command set
- MCI programming interfaces
- Using the command-message interface
- An introduction to the command-string interface

This chapter focuses on the C-language interface to MCI (the command-message interface). For more information on the English-language interface to MCI (the command-string interface), see the *Multimedia Programmer's Reference* which contains an alphabetical reference to MCI command strings and provides additional overview information about using the command-string interface.

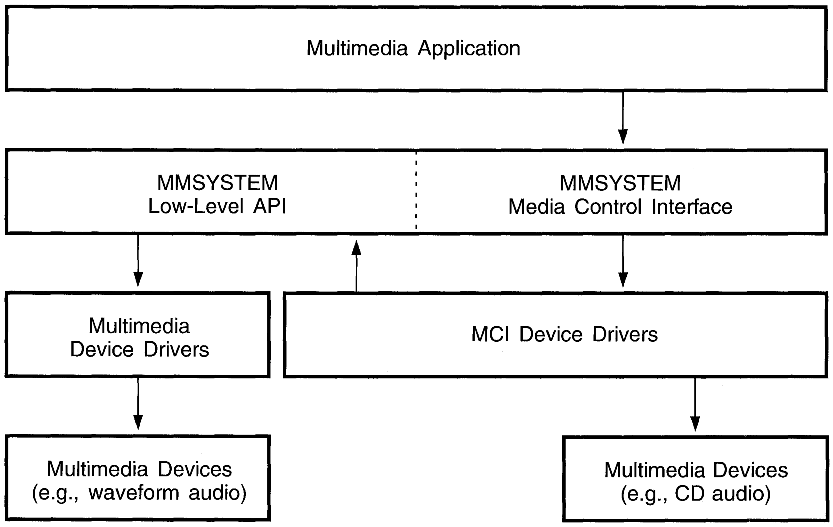
For specific details and programming examples on using MCI to control different types of audio devices, see Chapter 4, "High-Level Audio Services."

An Overview of MCI

MCI provides a device-independent, extensible interface for controlling virtually any type of media device. Because of the high level of device independence provided by MCI, you are encouraged to use MCI rather than the low-level API to access the multimedia capabilities of Windows.

The Architecture of MCI

To provide extensibility, MCI is designed around an architecture using special *MCI device drivers* to interpret and execute MCI commands. The following illustration shows the relationship between MCI and MCI device drivers:


Multimedia Application
MMSYSTEM Low-Level API
MMSYSTEM Media Control Interface
Multimedia Device Drivers
MCI Device Drivers
Multimedia Devices (e.g., waveform audio)
Multimedia Devices (e.g., CD audio)

The relationship between MCI and MCI device drivers.

MCI device drivers can control media hardware directly or through the low-level multimedia API provided by Windows. Most commonly used devices, such as waveform audio and MIDI devices, are controlled through the low-level multimedia API. Devices not supported by the low-level API are controlled directly, usually through a serial port.

It's not necessary that you understand how MCI device drivers operate to use MCI. If you want to write your own MCI device driver or just want additional information about MCI drivers, see the Microsoft Windows Device Driver Kit.

Note CD-ROM and audio CD devices are controlled through commands sent to MSCD-EX, the CD extensions for MS-DOS.

The MCI Pioneer Videodisk device driver is available from the Windows Driver Library (WDL).

The MCI Programming Interfaces

MCI provides two programming interfaces: the command-string interface and the command-message interface.

The Command-String Interface

The *command-string* interface allows you to use English-language commands to communicate with MCI devices. For example, the following command string plays a WAVE file named "TIMPANI.WAV":

```
play timpani.wav
```

The command-string interface is designed to be used with high-level programming and authoring environments such as Microsoft Visual Basic and Asymetrix ToolBook. Applications providing a text-based interface to let users control MCI devices should use the command-string interface.

The Command-Message Interface

The *command-message* interface uses a message-passing paradigm to communicate with MCI devices. For example, the following code fragment performs the same operation as the previous command-string example:

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

The command-message interface is designed to be used by applications requiring a C-language interface to control multimedia devices. Applications that directly control multimedia devices should use the command-message interface.

The MCI Command Set

The MCI command set is designed to provide a generic core set of commands to control different types of media devices. For example, MCI uses the same command to begin playback of a waveform audio file, a videodisc track, and an animation sequence. Some types of devices have unique capabilities, such as the capability of an animation player to use a frame-based time format. For such devices, MCI provides *extended commands* that are unique to a particular type of device.

The commands in the command-string interface provide a good overview of the MCI command set. Each of these commands is represented by a similar command in the command-message interface. For example, the equivalent command message for the **close** command is the MCI_CLOSE message.

The following table is an overview of some of the commonly used commands:

Command	Description
capability	Requests information about the capabilities of a device.
close	Closes a device after it has been used.
info	Requests information about a device such as a description of the hardware associated with the device.
open	Opens and initializes a device for use.
pause	Pauses playing or recording on a device.
play	Begins playing on a device.
record	Begins recording on a device.
resume	Resumes playing or recording on a paused device.
seek	Changes the current position in the media.
set	Changes control settings on the device such as the time format it is using.
status	Requests information about the status of a device such as whether it is playing or paused.
stop	Stops playing or recording on a device.

For a complete list of commands, see the *Multimedia Programmer's Reference*. The MCI command messages are in the message overview and message reference chapters—the MCI command strings are in a separate chapter.

About MCI Devices

MCI device drivers can be classified as either *simple devices* or *compound devices.* Simple devices do not require a data file for playback. Videodisc players and CD audio players are examples of simple devices. Compound devices require a data file for playback. MIDI sequencers and waveform audio players are examples of compound devices. The data file associated with a compound device is known as a *device element.* Examples of device elements are MIDI files and WAVE files.

Standard MCI Device Types

A *device type* identifies a class of MCI devices that respond to a common set of commands. The following table lists the currently defined device types:

Device Type	Description
animation	Animation device.
cdaudio	CD audio player.
dat	Digital audio tape player.
digitalvideo	Digital video in a window (not GDI based).
other	Undefined MCI device.
overlay	Overlay device (analog video in a window).
scanner	Image scanner.
sequencer	MIDI sequencer.
vcr	Videotape recorder or player.
videodisc	Videodisc player.
waveaudio	Audio device that plays digitized waveform files.

This chapter uses italic type for MCI device types.

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, there are several MCI drivers for *animation* devices that use the same command set but use different file formats. To uniquely identify an MCI driver, MCI uses *device names.*

The [mci] Section of SYSTEM.INI

Device names are identified in the [mci] section of the SYSTEM.INI file. This section identifies all MCI device drivers to Windows. The following is part of a typical [mci] section:

```
[mci]
waveaudio=mciwave.drv
sequencer=mciseq.drv
MMMovie=mcimmp.drv
cdaudio=mcicda.drv
```

The keyname (left side of the equal sign) is the device name. The value corresponding to the keyname (right side of the equal sign) identifies the filename of the MCI driver. Frequently, the device name is the same as the device type for the driver, as is the case for the *waveaudio, sequencer,* and *cdaudio* devices in the preceding example. The "MMMovie" device is an *animation* device, but it uses a unique device name.

If an MCI driver is installed using a device name that already exists in the [mci] section, Windows appends an integer to the device name of the new driver, creating a unique device name. In the preceding example, a driver installed using the *cdaudio* device name would be assigned device name "cdaudio1". A subsequent *cdaudio* device would be assigned device name "cdaudio2".

Opening MCI Devices

Before using an MCI device, you must initialize it by opening it. Opening a device loads its driver into memory (if it isn't already loaded), and establishes a *device ID* to identify the device in subsequent commands. (The device ID is not used in the command-string interface.) There are several ways to specify which device you want to open:

- With simple devices, you can open the device by specifying the device name.
- With compound devices, you can open the device by specifying the device name, the device element, or both.

Opening a Simple Device

For example, the following command string opens a CD audio device (a simple device) by specifying the device name:

```
open cdaudio
```

Opening a Compound Device

The next command string opens a waveform audio device (a compound device) by specifying the device name and a device element:

```
open bells.wav type waveaudio
```

You can also open a compound device by specifying only the device element as shown in the following example:

```
open bells.wav
```

If you specify only a device element when opening a compound device, MCI uses the file extension of the device element filename to determine which device to open. The WIN.INI file contains an [mci extensions] section that associates file extensions and corresponding MCI device types. The following is part of an [mci extensions] section:

```
[mci extensions]
wav=waveaudio
mid=sequencer
rmi=sequencer
```

Using the Command-Message Interface

Applications needing a C-language interface to MCI should use the command-message interface. This section provides background and detailed information on using this interface to MCI.

About Command Messages

MCI command messages consist of the following three elements:

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

Commands are sent with the **mciSendCommand** function which has parameters for the message, flags, and a pointer to the parameter block. See "Sending Command Messages," later in this chapter, for more information about the **mciSendCommand** function.

An Example of a Command Message

For an example of an MCI command message, consider the MCI_PLAY command. The MMSYSTEM.H header file defines the MCI_PLAY command message, the MCI_PLAY_PARMS parameter block, and related flags as follows:

```
#define MCI_PLAY 0x0806

typedef struct {
    DWORD   dwCallback;
    DWORD   dwFrom;
    DWORD   dwTo;
} MCI_PLAY_PARMS;

#define MCI_NOTIFY     0x00000001L
#define MCI_WAIT       0x00000002L
#define MCI_FROM       0x00000004L
#define MCI_TO         0x00000008L
```

The MCI_WAIT and MCI_NOTIFY flags are option flags. MCI_WAIT tells the driver to wait until the command is complete before returning control to the application. MCI_NOTIFY tells the driver to notify the application when the command is complete. For more information on using these flags, see "Using the Wait and Notify Flags," later in this chapter.

The MCI_FROM and MCI_TO flags validate the **dwFrom** and **dwTo** fields in the MCI_PLAY_PARMS parameter block. These fields specify beginning and ending positions for the play operation. Both of these flags are optional. If either flag is specified, the driver takes the value from the corresponding field in the parameter block as a beginning or ending position, otherwise the fields are ignored.

Note Data fields in a parameter block are ignored unless corresponding flags are specified to validate the fields.

Items in Query Commands

Two commands that query devices for information, MCI_GETDEVCAPS and MCI_STATUS, extend the command-message paradigm by using *item constants*. Item constants specify which item of information is being queried for.

These commands extend the command-message paradigm by adding a **dwItem** field to the parameter block and defining item constants for the field. For example, MMSYSTEM.H defines the MCI_STATUS_PARMS parameter block, flags, and item constants as follows:

```
typedef struct {
    DWORD   dwCallback;
    DWORD   dwReturn;
    DWORD   dwItem;
    DWORD   dwTrack;
} MCI_STATUS_PARMS;

/* item-constants for dwFlags parameter of MCI_STATUS command message  */
#define MCI_STATUS_ITEM                 0x00000100L

/* flags for dwItem field of the MCI_STATUS_PARMS parameter block */
#define MCI_STATUS_LENGTH               0x00000001L
#define MCI_STATUS_POSITION             0x00000002L
#define MCI_STATUS_NUMBER_OF_TRACKS     0x00000003L
#define MCI_STATUS_MODE                 0x00000004L
#define MCI_STATUS_MEDIA_PRESENT        0x00000005L
#define MCI_STATUS_TIME_FORMAT          0x00000006L
#define MCI_STATUS_READY                0x00000007L
#define MCI_STATUS_CURRENT_TRACK        0x00000008L
```

The MCI_STATUS_ITEM flag must be specified to validate the **dwItem** field which should contain one of the defined item constants to indicate exactly what status information is being requested.

Summary of MCI Command Messages

MCI defines four classifications of commands. The commands and options comprising the following two classifications are defined as the minimum command set for any MCI driver:

- *System commands* are handled directly by MCI rather than by the driver.
- *Required commands* are handled by the driver. All drivers should support the required commands and options.

Regardless of the specific driver you're using, you should be able to assume that the commands and options in the two preceding groups are available.

The commands comprising the following two classifications are not supported by all drivers:

- *Basic commands*, or optional commands, are used by some devices. If a device supports a basic command, it must support a defined set of options for the command.
- *Extended commands* are specific to a certain device types or drivers. Extended commands include new commands (like the MCI_PUT and MCI_WHERE commands for the *animation* device type) and extensions to existing commands (like the MCI_ANIM_GETDEVCAPS_CAN_STRETCH option added to the MCI_STATUS command for the *animation* device type).

If you need to use a basic or extended command or option, you should query the device driver before trying to use the command or option (that is, unless you are certain that the MCI driver you've used during development is the same one that will be available on the delivery system). The following tables summarize the specific commands in each category. For a complete description of each of these commands, see the message directory in the *Multimedia Programmer's Reference*.

System Commands

MCI directly processes the following system commands rather than passing them to MCI device drivers:

Command	**Description**
MCI_BREAK	Sets a break key for an MCI device.
MCI_SYSINFO	Returns information about MCI devices.

Required Commands

Required commands must be supported by all MCI device drivers. The following table lists the required commands:

Command	Description
MCI_CLOSE	Closes the device.
MCI_GETDEVCAPS	Obtains the capabilities of a device.
MCI_INFO	Obtains textual information from a device.
MCI_OPEN	Initializes the device.
MCI_STATUS	Obtains status information from the device.

Basic Commands

The following table summarizes the basic commands. The use of these commands by a device is optional.

Command	Description
MCI_LOAD	Loads data from a file.
MCI_PAUSE	Stops playing.
MCI_PLAY	Starts transmitting output data.
MCI_RECORD	Starts recording data.
MCI_RESUME	Resumes playing or recording on a paused device.
MCI_SAVE	Saves data to a disk file.
MCI_SEEK	Seeks forward or backward.
MCI_SET	Sets the operating state of the device.
MCI_STATUS	Obtains status information about the device. (Supplements the required MCI_STATUS command with options for devices using linear media with identifiable positions.)
MCI_STOP	Stops playing or recording.

Extended Commands

Some MCI devices have additional commands or add options to existing commands. While some extended commands only apply to a specific device driver, most of them apply to all drivers of a particular device type. For example, the *sequencer* command set extends the MCI_SET command to add time formats that are needed by MIDI sequencers.

Unless you are certain that the specific MCI driver you use during development will be available on the delivery system, you should not assume that the device supports the extended commands or options. You can use the MCI_GETDEVCAPS command to determine whether a specific feature is supported. Applications should be ready to handle an MCIERR_UNSUPPORTED_FUNCTION return value.

The following extended commands are available with certain device types:

Command	Device Types	Description
MCI_CUE	*waveaudio*	Prepares for playing or recording.
MCI_DELETE	*waveaudio*	Deletes a data segment from the media element.
MCI_ESCAPE	*videodisc*	Sends custom information to a device.
MCI_FREEZE	*overlay*	Disables video acquisition to the frame buffer.
MCI_PUT	*animation* *overlay*	Defines the source, destination, and frame windows.
MCI_REALIZE	*animation*	Tells the device to select and realize its palette into a display context of the displayed window.
MCI_SPIN	*videodisc*	Starts the disc spinning or stops the disc from spinning.
MCI_STEP	*animation* *videodisc*	Steps the play one or more frames forward or reverse.
MCI_UNFREEZE	*overlay*	Enables the frame buffer to acquire video data.
MCI_UPDATE	*animation*	Repaints the current frame into the display context.
MCI_WHERE	*animation* *overlay*	Obtains the rectangle specifying the source, destination, or frame area.
MCI_WINDOW	*animation* *overlay*	Controls display window options such as caption text.

Sending Command Messages

Windows provides the following functions to send command messages to devices and to query devices for error information:

mciSendCommand	Sends a command message to an MCI device.
mciGetErrorString	Returns the error string corresponding to an error number.

Use the **mciSendCommand** function to send command messages to MCI devices. This function has the following syntax:

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

The *wDeviceID* parameter is a UINT identifying the MCI device that is to receive the message. Use the device ID returned when the device was opened for this parameter.

The *wMessage* parameter is a UINT specifying the message. The MMSYSTEM.H header file defines MCI command messages.

The *dwParam1* parameter is a DWORD specifying flags for the command. These flags indicate options for the command and validate fields in the parameter block.

The *dwParam2* parameter is a DWORD specifying a pointer to the parameter block for the command. If the parameter block is not used, this parameter can be NULL.

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.

The MCI Device ID

MCI returns a device ID when you open a device using the MCI_OPEN command. Use this ID to identify the opened device when sending subsequent commands.

MMSYSTEM.H defines a special constant, MCI_ALL_DEVICE_ID, to indicate that a command be sent to all of the MCI devices that an application has opened. You can use MCI_ALL_DEVICE_ID with any command that does not return information. For example, the following code fragment 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
     */
    ...
```

Note While using the MCI_ALL_DEVICE_ID identifier is a convenient way to broadcast a command to all your devices, don't rely on it to synchronize devices; the timing between messages can vary.

Opening a Device

Before using a device, you must initialize it using the MCI_OPEN command message. The variations of this command make it one of the most complex of all MCI commands.

The MCI_OPEN_PARMS Parameter Block

Like all MCI command messages, MCI_OPEN has an associated parameter block. The default parameter block for MCI_OPEN is the MCI_OPEN_PARMS data structure. Certain devices such as waveform, animation, and overlay devices have extended parameter blocks to accommodate additional optional parameters. Unless you need to use these additional parameters, you can use the MCI_OPEN_PARMS parameter block with any MCI device.

The MCI_OPEN_PARMS data structure has the following fields:

```
typedef struct {
    DWORD  dwCallback;         /* callback for MCI_NOTIFY */
    UINT   wDeviceID;          /* device ID returned to user */
    UINT   wReserved0;         /* reserved */
    LPCSTR  lpstrDeviceType;   /* device type */
    LPCSTR  lpstrElementName;  /* device element */
    LPCSTR  lpstrAlias;        /* optional device alias */
} MCI_OPEN_PARMS;
```

MCI uses the **wDeviceID** field of this structure to return the device ID to the application. You should check the return value from **mciSendCommand** before using the device ID to be sure that it is valid. A non-zero return value indicates that an error occurred during the open process.

Note You can also determine the device ID with **mciGetDeviceID** by specifying the device name.

Other fields used in opening a device correspond to the following flags for the command:

Flag	Description
MCI_OPEN_ALIAS	Specifies that the **lpstrAlias** field of the data structure contains a pointer to a device alias.
MCI_OPEN_ELEMENT	Specifies that the **lpstrElementName** field of the data structure contains a pointer to the element name.
MCI_OPEN_SHAREABLE	Specifies that the device or element was opened as shareable.
MCI_OPEN_TYPE	Specifies that the **lpstrDeviceType** field of the data structure contains a pointer to the device-type identifier.
MCI_OPEN_TYPE_ID	Specifies that the **lpstrDeviceType** field of the data structure contains an integer device-type identifier.

Opening Simple Devices

To open a simple device such as a CD audio or videodisc device, use the **lpstrDeviceType** field in the MCI_OPEN_PARMS parameter block to identify which device to open. There are three ways to identify devices:

- Specify a pointer to a null-terminated string containing the device name
- Specify a device-type constant
- Specify the actual name of the device driver

Regardless of which method you use to identify the device, you must specify the MCI_OPEN_TYPE flag to validate the **lpstrDeviceType** field. For example, the following code fragment opens a CD audio device by specifying the device name:

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

/* Open a compact disc 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
     */
    ...

/* Device opened successfully, get the device ID
 */
wDeviceID = mciOpenParms.wDeviceID;
```

Using Device-Type Constants

You can also open devices by specifying a device-type constant. The following table gives the device-type constants for various devices. To be opened with a device-type constant, device drivers must be installed in the [mci] section of SYSTEM.INI using the device names given in this table.

Device Name	Device-Type Constant
animation	MCI_DEVTYPE_ANIMATION
cdaudio	MCI_DEVTYPE_CD_AUDIO
dat	MCI_DEVTYPE_DAT
digitalvideo	MCI_DEVTYPE_DIGITAL_VIDEO

Device Name	Device-Type Constant
other	MCI_DEVTYPE_OTHER
overlay	MCI_DEVTYPE_OVERLAY
scanner	MCI_DEVTYPE_SCANNER
sequencer	MCI_DEVTYPE_SEQUENCER
vcr	MCI_DEVTYPE_VIDEOTAPE
videodisc	MCI_DEVTYPE_VIDEODISC
waveaudio	MCI_DEVTYPE_WAVEFORM_AUDIO

If you open a device by specifying a device-type constant, you must specify the MCI_OPEN_TYPE_ID flag in addition to the MCI_OPEN_TYPE flag. For example, the following code fragment opens a CD audio device by specifying a device-type constant:

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

/* Open a compact disc 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
     */
    ...

/* Device opened successfully, get the device ID
 */
wDeviceID = mciOpenParms.wDeviceID;
```

When you open a device by specifying a device-type constant, MCI uses the high-order word of the **lpstrDeviceType** field to distinguish between multiple occurrences of the device type in the SYSTEM.INI file.

For example, if SYSTEM.INI lists entries for *waveaudio1* and *waveaudio2*, the following statement sets the **lpstrDeviceType** field to specify the device identified by *waveaudio2*:

```
lpstrDeviceType = MAKELONG(MCI_DEVTYPE_WAVEFORM_AUDIO, 2);
```

Likewise, to specify the device identified by *waveaudio1*, you would set the high-order word of **lpstrDeviceType** to 0 or 1 (setting the high-order word to 0 tells MCI to use an unnumbered device or the lowest-numbered device).

Using Driver Names

You can also open a device by specifying the filename of the device driver in the **lpstrDeviceType** field of the MCI_OPEN_PARMS parameter block. You don't need to specify the complete path or the file extension—MCI assumes drivers are located in the Windows system directory and have a file extension of "DRV". For example, the following code fragment opens a device by specifying the filename of the device driver:

Note Opening MCI devices by specifying the filename of the device driver may be useful during development and testing. However, this technique is not recommended for applications since it makes them device-dependent. Applications that open MCI devices by filename may not work in all systems.

Opening Compound Devices

There are three ways to open a compound device:

- Specify only the device (using the device name, a device-type constant, or the filename of the device driver).
- Specify both the device and the device element.
- Specify only the device element.

The first method, specifying only the device, is useful for determining the capabilities of a device with the MCI_GETDEVCAPS command. To actually control a media device, you must specify a device element when you open the device.

To specify a device element, fill the **lpstrElementName** field of the MCI_OPEN_PARMS parameter block with a pointer to a null-terminated string containing the filename of the device element. You must also validate the **lpstrElementName** field by specifying the MCI_OPEN_ELEMENT flag in the *dwFlags* parameter of **mciSendCommand**. To specify the device, you can use the same methods used for simple devices: specify the device by device name by device-type constant or by the filename of the driver. For example, the following code fragment opens the waveform audio device with a WAVE file named "TIMPANI.WAV":

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

/* Open a waveform device by specifying the device name and device element
 */
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
     */
    ...

/* Device opened successfully, get the device ID
 */
wDeviceID = mciOpenParms.wDeviceID;
```

You can also open a device by specifying just a device element—MCI uses the filename extension of the given device element along with the [mci extensions] section of WIN.INI to select which device to open. See "Opening MCI Devices," earlier in this chapter, for more information about opening devices by specifying only a device element.

Closing a Device

The MCI_CLOSE command message releases access to a device or device element. MCI frees a device when all tasks using a device have closed it. To help MCI manage devices, applications should explicitly close each device or device element when finished using them.

Note It is very important that applications close all opened MCI devices before exiting. You can close all devices with a single command by specifying the MCI_ALL_DEVICE_ID constant as the device ID.

Using the Wait and Notify Flags

Normally, MCI returns control immediately to the application, even if it takes several seconds to complete the action initiated by the command. For example, after a CD audio device receives a seek command, control returns to the application before the seek operation is complete. You can use the following two flags to request that MCI wait until a command is complete before returning control to the application or that MCI notify the application when the command is complete:

Flag	Description
MCI_NOTIFY	Directs the device to return control immediately and post the MM_MCINOTIFY message when the requested action is complete.
MCI_WAIT	Directs the device to wait until the requested action is complete before returning control to the application.

Using the Notify Flag

The MCI_NOTIFY flag directs MCI to post the MM_MCINOTIFY message when the device completes an action. The application specifies the handle to the destination window for the message in the low-order word of the **dwCallBack** field of the data structure sent with the command message. Every data structure associated with a command message contains this field.

Note MCI can have only one pending notification message for each device.

When MCI posts the MM_MCINOTIFY message, it sets the low-order word of *lParam* parameter of the window procedure to the ID of the device initiating the callback. The *wParam* parameter of the window procedure contains one of the following constants specifying the notification status of the callback:

Flag	Description
MCI_NOTIFY_SUCCESSFUL	Indicates the conditions required for initiating the callback are satisfied and the command completed without interruption.
MCI_NOTIFY_SUPERSEDED	Indicates MCI had a notification pending and received another notify request. MCI resets the callback conditions to correspond to the notify request of the new command.
MCI_NOTIFY_ABORTED	Indicates the application sent a new command that prevented the callback conditions set by a previous command from being satisfied. If an application interrupts a command that requested notification, MCI will not inform the window procedure of the notify command being aborted.
MCI_NOTIFY_FAILURE	Indicates a device error occurred while a device was executing the command. For example, MCI posts this message when a hardware error occurs during a play command.

Using the Wait Flag and Break Command Message

Specifying the MCI_WAIT flag with a command causes MCI to wait until the command has completed before returning control to the application. Using the wait flag with commands that take a long time to complete will essentially disable a system from receiving any user input until the command is complete.

MCI supports a break key to interrupt the wait operation and return control to the application. The break key does not interrupt the command, but by returning control to the application, allows the application to monitor user input.

By default, MCI defines the break key as CTRL+BREAK. Applications can redefine this key with the MCI_BREAK command. MCI_BREAK uses the MCI_BREAK_PARMS parameter block:

```
typedef struct {
    DWORD   dwCallback;      /* callback for MCI_NOTIFY */
    int     nVirtKey;        /* virtual key code */
    UINT    wReserved0;      /* reserved */
    HWND    hwndBreak;       /* window handle */
    UINT    wReserved1;      /* reserved */
} MCI_BREAK_PARMS;
```

The following flags validate fields in the MCI_BREAK_PARMS parameter block:

Flag	Description
MCI_BREAK_KEY	Validates the **nVirtKey** field specifying the virtual-key code to be used for the break key.
MCI_BREAK_HWND	Validates the **hwndBreak** field specifying the window that must have focus to enable break detection.
MCI_BREAK_OFF	Disables any existing break key.

Obtaining MCI System Information

The MCI_SYSINFO message obtains system information about MCI devices. MCI handles this message without relaying it to any MCI device. MCI returns the system information in the MCI_SYSINFO_PARMS data structure. The MCI_SYSINFO_PARMS data structure has the following fields:

```
typedef struct {
    DWORD   dwCallback;      /* callback for MCI_NOTIFY */
    LPSTR   lpstrReturn;     /* pointer to buffer for return information */
    DWORD   dwRetSize;       /* size of buffer */
    DWORD   dwNumber;        /* index number */
    UINT    wDeviceType;     /* device type */
    UINT    wReserved0;      /* reserved */
} MCI_SYSINFO_PARMS;
```

The MCI_SYSINFO command message has the following flags.

Flag	Description
MCI_SYSINFO_QUANTITY	Specifies that MCI will return the number of devices of a given type listed in the [mci] section of SYSTEM.INI. When used with the MCI_SYSINFO_OPEN flag, the number of open devices will be returned.
MCI_SYSINFO_NAME	Specifies that MCI will return the name of a device that satisfies the query. When used with the MCI_SYSINFO_OPEN flag, only the names of open devices will be returned.
MCI_SYSINFO_OPEN	Specifies that MCI will return the number or names of open devices.
MCI_SYSINFO_INSTALLNAME	Specifies that MCI will return the name listed in the SYSTEM.INI file used to install the device.

You use the MCI_SYSINFO_QUANTITY flag to determine the number of devices for a particular type listed in the [mci] section of the SYSTEM.INI file. For this flag, set the **wDeviceType** field of the data structure to the device type. When requesting the number of devices, set the *wDeviceID* parameter to NULL. MCI returns the number of devices as a DWORD in the **lpstrReturn** field of the data structure. If you combine the MCI_SYSINFO_QUANTITY flag with the MCI_SYSINFO_OPEN flag, MCI returns the number of devices opened by the task of that type.

Once you have the number of devices, you can enumerate the names of the devices using the MCI_SYSINFO_NAME flag. To get information using this flag, you must create a buffer for the return name. In the data structure, specify a pointer to the buffer, the size of the buffer, the index number corresponding to the device (1 is the first device), and the device type. This information is entered in the **lpstrReturn**, **dwRetSize**, **dwNumber**, and **wDeviceType** fields. MCI returns the device name in the buffer. When requesting a name of a particular device, set the *wDeviceID* parameter to NULL. To restrict the names to open devices, use the MCI_SYSINFO_OPEN flag with MCI_SYSINFO_NAME.

To obtain information on all devices in the system, you can assign MCI_ALL_DEVICE_ID to the *wDeviceID* parameter. When you use this identifier, MCI ignores the contents of the **wDeviceType** field and returns information on all MCI devices listed in the SYSTEM.INI file. When you use the MCI_SYSINFO_OPEN flag with the MCI_ALL_DEVICE_ID identifier, MCI returns information on the devices opened by the task.

You can obtain the installation name of an open device with the MCI_SYSINFO_INSTALLNAME flag. To get the installation name you must create a buffer for the return name. In the data structure, specify a pointer to the buffer and the size of the buffer in the **lpstrReturn** and **dwRetSize** fields. MCI places the device that corresponds to the *wDeviceID* parameter name in the buffer.

About the Command-String Interface

This section presents an introduction to the MCI command-string interface. MCI has the following functions to work with command strings:

mciSendString
Sends a command string to an MCI device driver. This function also has parameters for callback functions and return strings.

mciGetErrorString
Returns the error string corresponding to an error number.

Sending Command Strings Using mciSendString

Use the **mciSendString** function to send command strings to a device. It has the following syntax:

DWORD mciSendString(*lpstrCommand, lpstrRtnString, wRtnLength, hCallback*)

The far pointer *lpstrCommand* points to a null-terminated string containing the MCI string command. The string command has the following form:

command device_name arguments

The second parameter, *lpstrRtnString*, points to an application-supplied buffer for a return string. The third parameter, *wRtnLength*, specifies the size of this buffer. If an MCI command returns a value or a string, MCI copies it into *lpstrRtnString* as a null-terminated string (integers are converted to strings). If the length of the return string exceeds the size of the buffer, MCI returns an error. You can assign NULL to *lpstrRtnString* if you don't want the return string, or for commands that don't supply return information.

The last parameter, *hCallback*, is a handle to the window that receives the MM_MCINOTIFY message. MCI ignores this parameter unless the command contains the **notify** flag. This parameter must contain a valid window handle if the notify flag is used.

The following statement passes the "play to 500" command string to a waveform audio device driver:

```
dwErrorCode = mciSendString ("play waveaudio to 500", NULL, 0, 0L);
```

The **mciSendString** function returns zero if successful. If the function fails, it returns an error code. If there is an error, you can get a description of it by passing the error code to **mciGetErrorString**.

Additional Information about Command Strings

For more information on using the MCI command-string interface, see the MCI command strings chapter in the *Multimedia Programmer's Reference*.

MCITEST Sample Application

The MCITEST application contains examples of using **mciSendString** and **mciGetErrorString**. You can use this application to experiment with MCI devices and command strings.

Chapter 3
Introduction to Audio

Windows provides two levels of audio services. Before you can use either level, you must understand what each level offers and how it fits into the audio architecture.

This chapter covers the following topics:

- The different types and levels of audio services
- An overview of the audio architecture
- Audio file formats used by Windows
- Books and specifications providing additional information about audio

This chapter assumes you have a basic knowledge of digital audio and MIDI. For more information on these subjects, see "Further Reading," at the end of this chapter, for a list of related books and specifications.

About Audio Services

Windows provides several different types and levels of audio services. You should become familiar with these types and levels so you can choose the services appropriate for your application.

Types of Audio Services

Different types of audio services require different formats for the audio information and different technologies to reproduce audio. Windows offers the following types of audio services:

- *Waveform audio services* provide playback and recording support for digital audio hardware. Waveform audio is useful for reproducing non-musical audio material such as sound effects and voice-over narration. Waveform audio has moderate storage space and data transfer rate requirements—as low as 11K per second of audio.
- *MIDI audio services* provide support for MIDI file and MIDI event playback through internal or external synthesizers and MIDI event recording. MIDI audio is useful with music-related applications such as music composition and MIDI sequencer programs. Because it requires less storage space and a lower data transfer rate than waveform audio, you might want to use MIDI audio services to provide introductory and background music in your applications.
- *Compact disc audio services* provide support for playback of Red Book audio information on compact discs with the CD-ROM drive on multimedia computers. Compact disc audio offers the highest quality reproduction of musical material, but has the highest storage space requirements—176K per second of audio. You cannot read from the CD-ROM drive while playing compact disc audio.

Note CD-ROM and audio CD devices are controlled through commands sent to MSCD-EX, the CD extensions for MS-DOS.

Levels of Audio Services

The different levels of audio services let you choose services appropriate to the requirements of your application and to your programming ability. Windows offers two levels of audio services:

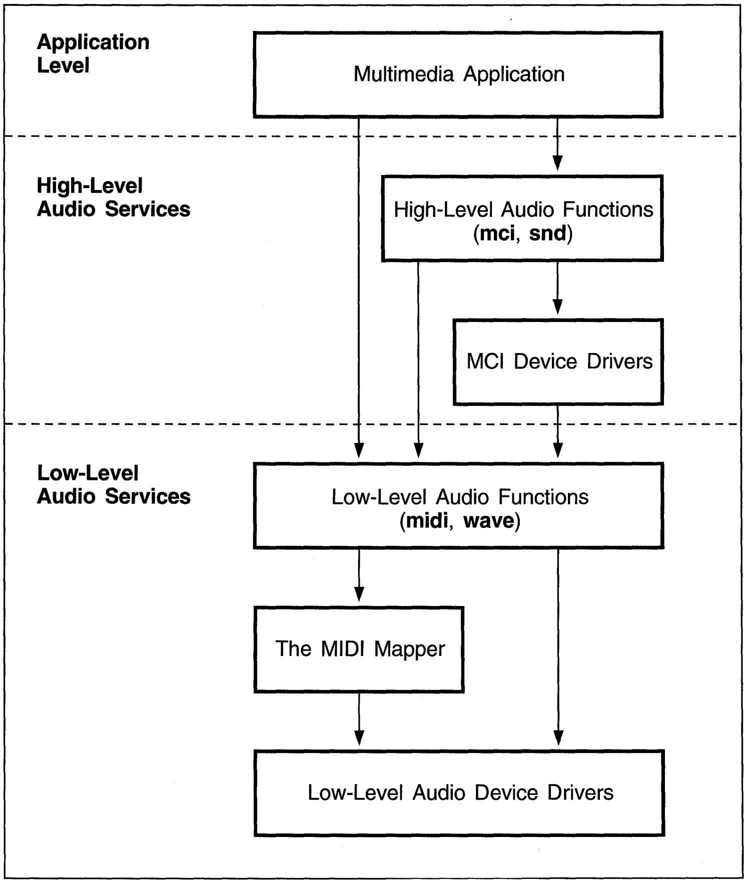
- *High-level audio services* allow you to play and record audio files with as little as one function call. Compared to low-level audio services, high-level services are easier to use and require less programming. High-level services also meet the audio requirements of most applications.
- *Low-level audio services* allow you to communicate directly with audio device drivers to manage playback and recording. Low-level audio services require more programming than high-level services, but give you more control over audio playback and recording.

Choosing the Appropriate Services

The easiest way of playing and recording audio is to use high-level audio services. If the high-level audio services don't meet the audio needs of your application, you can use low-level audio services to write your own routines to manage audio playback and recording.

Windows Audio Architecture

Windows provides audio services through high-level audio functions, Media Control Interface (MCI) device drivers, low-level audio functions, the MIDI Mapper, and low-level audio device drivers. The following illustration shows the relationship between an application and the elements of Windows that provide audio support:


Application Level
Multimedia Application
High-Level Audio Services
High-Level Audio Functions (**mci**, **snd**)
MCI Device Drivers
Low-Level Audio Services
Low-Level Audio Functions (**midi**, **wave**)
The MIDI Mapper
Low-Level Audio Device Drivers

The relationship between an application and the Multimedia functions.

High-Level Audio Functions

MCI functions, along with MCI device drivers, provide high-level audio support for MIDI, waveform, and compact disc audio devices. MCI drivers can play disk-resident (hard disk and CD-ROM) audio files in the background while an application runs in the foreground. In addition to MCI, Windows provides high-level audio support with two special-purpose functions that play waveform sounds: **MessageBeep** and **sndPlaySound**. You can also use these functions to play waveform sounds associated with system alerts and sounds identified by entries in the WIN.INI file.

High-level audio functions should meet the audio requirements of most applications. For information on using MCI and the **MessageBeep** and **sndPlaySound** functions, see Chapter 4, "High-Level Audio Services."

Low-Level Audio Functions

Low-level audio functions provide a device-independent interface to audio hardware in multimedia computers. These functions supply low-level audio services by letting applications communicate directly with the audio device drivers. MCI drivers and the **MessageBeep** and **sndPlaySound** functions use these low-level functions to provide high-level audio services.

Low-level audio functions are provided for applications, such as audio tools programs, that have special-purpose audio requirements. For example, a MIDI sequencer application would use low-level audio functions to record and play MIDI data. For information on using low-level audio services, see Chapter 5, "Low-Level Audio Services."

The MIDI Mapper

The MIDI Mapper provides *standard patch services* for device-independent playback of MIDI files. Standard patch services ensure different MIDI snythesizers use the same instrument sounds to reproduce the music in a MIDI file.

The MIDI Mapper translates and redirects messages sent to it by low-level MIDI output functions. Because the high-level MIDI services use low-level MIDI output functions, the MIDI Mapper can be used with high-level MIDI services, as well as low-level MIDI services. For information on the architecture of the MIDI Mapper and how to use the Mapper with high-level audio services, see Chapter 4, "High-Level Audio Services." For information on using the Mapper with low-level audio services, see Chapter 5, "Low-Level Audio Services."

MIDI files must follow certain authoring guidelines to use the standard patch services provided by the MIDI Mapper. For details on these authoring guidelines, see Chapter 4, "High-Level Audio Services."

Audio File Formats

Windows supports a tagged file structure called the Resource Interchange File Format (RIFF). There are two RIFF file formats currently defined for audio files:

RIFF File Format	File Extension	Description
RMID	.RMI	MIDI audio file
WAVE	.WAV	Waveform audio file

The WAVE file format supports a number of different digital audio data formats. All multimedia computers support PCM (pulse code modulated) data formats of 8-bit mono at sample rates of 11.025 kHz and 22.05 kHz.

In addition to RMID files, the MCI MIDI sequencer plays standard MIDI files in the format defined by the International MIDI Association in the "Standard MIDI Files 1.0" specification. See "Further Reading," at the end of this chapter, for information on how to obtain this specification. RMID files are standard MIDI files with a RIFF header.

For more information on RIFF files and the RMID and WAVE file formats, see the file formats chapter in the *Multimedia Programmer's Reference.*

Using RIFF Files

The multimedia file I/O services include functions for working with RIFF files. For information on using these functions, see Chapter 7, "Multimedia File I/O Services."

Further Reading

For more information on digital audio, digital-signal processing, and computer music, see the following books:

Boom, Michael. *Music Through MIDI.* Redmond: Microsoft Press, 1987.

Chamberlin, Hal. *Musical Applications of Microprocessors.* Hasbrouk Heights: Hayden Book Company, Inc., 1985.

Pohlmann, Ken. *Principles of Digital Audio.* Indianapolis: Howard W. Sams & Co., 1985.

Roads, Curtis, and John Strawn. *Foundations of Computer Music.* Cambridge: The MIT Press, 1985.

Strawn, John F. *Digital Audio Engineering, An Anthology.* Los Altos: William Kaufmann, Inc., 1985.

Strawn, John F. *Digital Audio Signal Processing, An Anthology.* Los Altos: William Kaufmann, Inc., 1985.

For information on MIDI and standard MIDI files, see the following MIDI specifications:

MIDI 1.0 Detailed Specification

Standard MIDI Files 1.0

The specifications listed above are available from the International MIDI Association at the following address:

International MIDI Association
5316 West 57th Street
Los Angeles, CA 90056

Chapter 4
High-Level Audio Services

This chapter explains how to use the high-level audio services of Windows to add sound to applications. For an overview of the audio services, see Chapter 3, "Introduction to Audio."

This chapter covers the following topics:

- Playing waveform sounds
- Using the Media Control Interface (MCI) to play and record audio
- Using the MIDI Mapper for device-independent MIDI file playback
- Authoring MIDI files

This chapter assumes you have a basic knowledge of digital audio and MIDI. If you need additional information on these subjects, see "Further Reading," at the end of Chapter 3, "Introduction to Audio."

Function Prefixes

High-level audio function names begin with the following prefixes:

Prefix	Description
snd	System alert sound functions
mci	Media Control Interface functions

Playing Waveform Sounds

One basic use of sound is associating a sound with a user action or with a warning or alert message. These sounds tend to be short in duration and are often played repeatedly. Windows provides two functions to play waveform sounds:

MessageBeep
Plays the sound that corresponds to a given system-alert level.

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

These functions provide the following methods of playing waveform sounds:

- Playing WAVE files stored on a hard disk or CD-ROM
- Playing in-memory WAVE resources
- Playing WAVE files specified by entries in the WIN.INI file
- Playing WAVE files associated with system-alert levels

Restrictions in Playing Waveform Sounds

There are some restrictions to using **sndPlaySound** and **MessageBeep** to play waveform sounds:

- The entire sound must fit in available physical memory.
- The sound must be in a data format playable by one of the installed waveform audio device drivers.

Use **sndPlaySound** and **MessageBeep** to play WAVE files that are relatively small in size—up to about 100K. For larger sound files, use the Media Control Interface (MCI) services. For information on using MCI, see "Playing and Recording Audio Using MCI," later in this chapter.

Using the sndPlaySound Function

The **sndPlaySound** function has the following syntax:

BOOL sndPlaySound(*lpszSoundName, wFlags*)

The parameter *lpszSoundName* is an LPCSTR and points to a null-terminated string containing the name of the sound to be played. This name can be a keyname in the [sounds] section of the WIN.INI file or it can be the filename of a WAVE file. Optionally, *lpszSoundName* can be a far pointer to an in-memory image of a WAVE file, which can be Clipboard data in CF_WAVE format or a resource that has been loaded into memory. See "Playing WAVE Resources," later in this section, for details on using in-memory WAVE file images.

The *wFlags* parameter specifies optional flags that affect how the sound is played. If the SND_SYNC flag is specified, the sound is played synchronously—**sndPlaySound** doesn't return until playback is complete. If the SND_ASYNC flag is specified, the sound is played asynchronously—**sndPlaySound** returns as soon as the sound begins playing. If neither of these flags is specified, the sound is played synchronously.

Playing WAVE Files

As an example, the following statement plays the C:\SOUNDS\BELLS.WAV file:

```
sndPlaySound("C:\\SOUNDS\\BELLS.WAV", SND_SYNC);
```

If the specified file does not exist, or will not fit into the available physical memory, **sndPlaySound** plays the default system sound specified by the SystemDefault entry in the [sounds] section of the WIN.INI file. If there is no SystemDefault entry, **sndPlaySound** fails without producing any sound. If you don't want the default system sound to play, specify the SND_NODEFAULT flag, as shown in the following example:

```
sndPlaySound("C:\\SOUNDS\\BELLS.WAV", SND_SYNC | SND_NODEFAULT);
```

Looping Sounds

If you specify the SND_LOOP and SND_ASYNC flags for the *wFlags* parameter, the sound will continue to play repeatedly; for example:

```
sndPlaySound("C:\\SOUNDS\\BELLS.WAV", 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 **sndPlaySound** is called to play another sound. To stop playing a sound (looped or asynchronous) without playing another sound, use the following statement:

```
sndPlaySound(NULL, 0);
```

Playing WAVE Resources

You can also build WAVE files into an application as resources and use **sndPlaySound** to play these sounds by specifying the SND_MEMORY flag. The SND_MEMORY flag indicates the *lpszSoundName* parameter is a pointer to an in-memory image of the WAVE file. To include a WAVE file as a resource in an application, add the following entry to the application's resource script (.RC) file:

```
soundName WAVE c:\sounds\bells.wav
```

The name "soundName" is a placeholder for a name that you create to refer to this WAVE resource. WAVE resources are loaded and accessed just like other user-defined Windows resources. The function in the following example plays a specified WAVE resource.

```
/* Plays a specified WAVE resource */
BOOL PlayResource(LPSTR lpName)
{
    HANDLE hResInfo, hRes;
    LPSTR lpRes;
    BOOL bRtn;

    /* Find the WAVE resource */
    hResInfo = FindResource(hInst, lpName, "WAVE");
    if (!hResInfo) return FALSE;
    /* Load the WAVE resource */
    hRes = LoadResource(hInst, hResInfo);
    if (!hRes) return FALSE;

    /* Lock the WAVE resource and play it */
    lpRes = LockResource(hRes);
    if (lpRes) {
        bRtn = sndPlaySound(lpRes, SND_MEMORY | SND_SYNC | SND_NODEFAULT);
        UnlockResource(hRes);
    }
    else
        bRtn = 0;

    /* Free the WAVE resource and return success or failure */
    FreeResource(hRes);
    return bRtn;
}
```

To play a WAVE resource using this function, pass the function a far pointer to a string containing the name of the resource, as shown in the following example:

```
PlayResource("soundName");
```

Playing Sounds Specified in WIN.INI

The **sndPlaySound** function will also play waveform sounds referred to by a keyname in the [sounds] section of WIN.INI. This allows users to assign their own sounds to system alerts and warnings, or to user actions, such as a mouse button click. For example, the [sounds] section of WIN.INI might look like this:

```
[sounds]
SystemDefault=C:\SOUNDS\BUMMER.WAV, Says BUMMER
SystemAsterisk=C:\SOUNDS\WHALES.WAV, Whale sounds
SystemExclamation=C:\SOUNDS\LASER.WAV, Laser gun sounds
SystemHand=C:\SOUNDS\OHOH.WAV, Says OH-OH
SystemQuestion=C:\SOUNDS\JIBERISH.WAV, Person mumbling
SystemStart=C:\SOUNDS\CHORD.WAV, Plays a chord
MouseClick=C:\SOUNDS\CLICK.WAV, Recording of a clicking sound
```

To play a sound identified by a WIN.INI entry, call **sndPlaySound** with the *lpszSoundName* parameter pointing to a string containing the name of the entry that identifies the sound. For example, to play the sound associated with the "MouseClick" entry in the [sounds] section of WIN.INI, and wait for the sound to complete before returning, use the following statement:

```
sndPlaySound("MouseClick", SND_SYNC);
```

If the specified WIN.INI entry or the waveform file it identifies does not exist, or if the sound will not fit into the available physical memory, **sndPlaySound** plays the default system sound specified by the SystemDefault entry. If there is no SystemDefault entry, **sndPlaySound** fails without producing any sound. If you don't want the default system sound to play, specify the SND_NODEFAULT flag when you call **sndPlaySound**, as in the following example:

```
sndPlaySound("MouseClick", SND_SYNC | SND_NODEFAULT);
```

Note The **sndPlaySound** function always searches the [sounds] section of WIN.INI for a keyname matching *lpszSoundName* before attempting to load a file with this name.

The [sounds] section of WIN.INI includes an optional string following the waveform filename for descriptive text about the sound in the file. This descriptive text is displayed in the Sound Option from the Control Panel.

Playing System Alert Sounds

If you are familiar with programming for the Windows environment, you should recognize the **MessageBeep** function. Windows 3.1 replaces the Windows 3.0 **MessageBeep** function with a function that uses the waveform audio hardware to produce a variety of user-selectable sounds. The new **MessageBeep** function has the following syntax:

void MessageBeep(*wAlert*)

The syntax of **MessageBeep** remains the same, except its previously unused parameter is now used. The *wAlert* parameter is a UINT and specifies the alert level. Valid flags for *wAlert* are the same as those passed to **MessageBox**: MB_ICONASTERISK, MB_ICONEXCLAMATION, MB_ICONHAND, and MB_ICONQUESTION.

When you call the **MessageBeep** function, it searches the [sounds] section of the WIN.INI file for the WAVE file that corresponds to the specified alert level. The following table identifies the WIN.INI entries for system-alert sounds:

WIN.INI Entry	Description
SystemDefault	Identifies the sound produced when **MessageBeep** is called with *wAlert* set to an invalid alert level or when the requested alert sound can't be found. This is called the default system sound.
SystemAsterisk	Identifies the sound produced when **MessageBeep** is called with *wAlert* set to MB_ICONASTERISK.
SystemExclamation	Identifies the sound produced when **MessageBeep** is called with *wAlert* set to MB_ICONEXCLAMATION.
SystemHand	Identifies the sound produced when **MessageBeep** is called with *wAlert* set to MB_ICONHAND.
SystemQuestion	Identifies the sound produced when **MessageBeep** is called with *wAlert* set to MB_ICONQUESTION.

If **MessageBeep** is called with *wAlert* set to zero, it plays the default system sound. If *wAlert* is set to –1, it uses the computer speaker to produce a standard beep sound. If **MessageBeep** can't play the requested sound, it plays the default system sound. If it can't play the default system sound, then it produces a beep sound using the computer speaker.

The Sound option of the Control Panel allows users to set these WIN.INI entries to customize their environment for system-alert sounds. It also allows users to disable these alert sounds.

Playing and Recording Audio Using MCI

The Media Control Interface (MCI) provides a high-level interface for controlling both internal and external media devices. MCI provides support for playing waveform, MIDI, and compact disc audio and for recording waveform audio. MCI is the easiest way for multimedia applications to play and record audio.

MCI uses device drivers to interpret and execute high-level MCI commands. MCI device drivers can stream digital audio and MIDI data directly from a storage device to the appropriate device driver, allowing applications to play files too large to fit in available physical memory. This data streaming takes place in the background while an application is running. The application is responsible only for setting up MCI and telling it to start playing or recording.

Before reading this section, you should become acquainted with MCI by reading Chapter 2, "The Media Control Interface (MCI)." MCI provides two types of interfaces: a command-message interface and a command-string interface. This section explains how to use the command-message interface for waveform, MIDI, and compact disc audio devices. See the MCI chapter for information about the command-string interface.

MCI Audio Data Types

The MMSYSTEM.H header file defines data types and function prototypes for MCI. You must include this header file in any source module that uses MCI functions.

The following list contains data types for parameter blocks used with extended MCI commands related to audio. For a complete list of all MCI data types, see the chapter on data types in the *Multimedia Programmer's Reference*.

MCI_SEQ_SET_PARMS
A data structure for specifying a parameter block for the MCI_SET command for MIDI sequencer devices.

MCI_WAVE_OPEN_PARMS
A data structure for specifying a parameter block for the MCI_OPEN command for waveform audio devices.

MCI_WAVE_SET_PARMS
A data structure for specifying a parameter block for the MCI_SET command for waveform audio devices.

MCI Audio Commands

MCI provides a standard set of commands applying to all types of media devices. Some of these commands can be extended to accommodate unique features of a particular type of device. For a complete reference to MCI command messages, see the message-directory chapter in the *Multimedia Programmer's Reference*.

The following table lists common audio playback and recording tasks along with the corresponding MCI command message to perform each task:

Audio Task	MCI Command
Open and close audio devices	
Open an audio device	MCI_OPEN
Close an audio device	MCI_CLOSE
Control playback and recording	
Play all or part of audio selection, resume playback from pause	MCI_PLAY
Stop playback	MCI_STOP
Pause playback	MCI_PAUSE
Change current location	MCI_SEEK
Cue a device so playback or recording begins with minimum delay	MCI_CUE
Begin recording on a waveform audio device	MCI_RECORD
Save a recorded waveform audio file	MCI_SAVE
Query and set audio devices	
Query device information such as the product name and the name of the device element currently associated with the device (returns information in string format)	MCI_INFO
Query device capabilities such as the device name, number of inputs and outputs (if the device can record)	MCI_GETDEVCAPS

Audio Task	MCI Command
Query and set audio devices (*continued*)	
Query device status such as current playback position, media length, media format, time format, record level, CD audio track, MIDI sequencer tempo	MCI_STATUS
Set device parameters such as time format, waveform data format, MIDI sequencer tempo	MCI_SET

Opening MCI Audio Devices

MCI requires audio devices be opened before they can be accessed. Use the MCI_OPEN command along with the MCI_OPEN_PARMS parameter block to open an audio device and obtain an MCI device ID. Subsequent commands use this device ID to identify the device to receive the command. MMSYSTEM.H defines the MCI_OPEN_PARMS parameter block as follows:

```
typedef struct {
    DWORD   dwCallback;         /* callback for MCI_NOTIFY flag */
    UINT    wDeviceID;          /* device ID returned to user */
    UINT    wReserved0;         /* reserved */
    LPCSTR  lpstrDeviceType;    /* device name */
    LPCSTR  lpstrElementName;   /* device element */
    LPCSTR  lpstrAlias;         /* optional device alias (reserved) */
} MCI_OPEN_PARMS;
```

The MCI device ID is returned in the **wDeviceID** field. You should always check the return value of **mciSendCommand** after sending an MCI_OPEN command before using this device ID. A non-zero return value indicates that there was an error in opening the device and the returned device ID will not be valid.

Extended Parameter Block for Waveform Devices

For waveform audio devices, you can use an extended parameter block, MCI_WAVE_OPEN_PARMS. This structure has a **dwBufferSeconds** field to specify the number of seconds of buffering used by the MCI waveform device driver. If you use this field, you must specify the MCI_WAVE_OPEN_BUFFER flag for the *dwParam1* parameter of **mciSendCommand** to validate the field. MMSYSTEM.H defines the MCI_WAVE_OPEN_PARMS parameter block as follows:

```
typedef struct {
    DWORD   dwCallback;          /* callback for MCI_NOTIFY flag */
    UINT    wDeviceID;           /* device ID returned to user */
    UINT    wReserved0;          /* reserved */
    LPCSTR  lpstrDeviceType;     /* device name */
    LPCSTR  lpstrElementName;    /* device element */
    LPCSTR  lpstrAlias;          /* optional device alias (reserved) */
    DWORD   dwBufferSeconds;     /* buffer size in seconds */
} MCI_WAVE_OPEN_PARMS;
```

Unless you want to specify the number of seconds of buffering for the driver to use, you can use the MCI_WAVE_OPEN_PARMS parameter block when you open waveform audio devices.

Specifying Device Names

When you open a simple device, such as a compact disc audio device, you must specify the device name in the **lpstrDeviceType** field. When you open a compound device, such as a waveform or MIDI sequencer device, the device name is optional.

Use the **lpstrDeviceType** field of the MCI_OPEN_PARMS structure to specify the device name. MCI lets you use a string or a constant for this field. The following table shows the strings and constants for MCI audio devices:

Device	String	Constant
Compact disc	"cdaudio"	MCI_DEVTYPE_CD_AUDIO
Waveform	"waveaudio"	MCI_DEVTYPE_WAVEFORM_AUDIO
MIDI sequencer	"sequencer"	MCI_DEVTYPE_SEQUENCER

Using a string is the default convention for specifying device names. If you use a constant to specify the device name, you must specify the MCI_OPEN_TYPE_ID flag in addition to the MCI_OPEN_TYPE flag.

Opening Waveform and MIDI Sequencer Devices

Waveform and MIDI sequencer devices are compound devices. Compound devices require an associated device element—a WAVE or MIDI file. There are three ways to open compound devices:

- Specify only the device name.
- Specify only the device element and let MCI select the device from the file extension of the device element.
- Specify both the device name and the device element.

Use the first approach, specifying only the device name, when opening a device to query its capabilities with the MCI_GETDEVCAPS command and when you plan to use the device to play more than one device element.

Example of Opening a MIDI Sequencer Audio Device

If you don't specify a device name when you open a compound device, MCI will choose an appropriate device type by looking at the file extension of the device element and at entries in the [mci extensions] section of WIN.INI. The following code fragment uses this technique to open a MIDI sequencer device to play a MIDI file named CHOPIN.RMI:

```
UINT wDeviceID;
MCI_OPEN_PARMS mciOpenParms;
.
.
.
/* Open the device by specifying only the device element
 */
mciOpenParms.lpstrElementName = "CHOPIN.RMI";
if (mciSendCommand(0,                                  // device ID
                   MCI_OPEN,                           // command
                   MCI_OPEN_ELEMENT,                   // flags
                   (DWORD)(LPVOID) &mciOpenParms))     // parameter block
    /* Error, unable to open device
     */
     ...
else
    /* Device opened successfully, get the device ID
     */
    wDeviceID = mciOpenParms.wDeviceID;
```

Example of Opening a Waveform Audio Device

Instead of letting MCI choose the device, you can specify the device name when you open the device. The following code fragment uses this technique to open a waveform audio device to play the C:\SOUNDS\BELLS.WAV file. This example uses a string to specify the device name. For an example using a constant to specify the device name, see "Opening Compact Disc Devices," later in this chapter.

```
UINT wDeviceID;
MCI_OPEN_PARMS mciOpenParms;
.
.
.
/* Open the device by specifying both the device element and the device name
 */
mciOpenParms.lpstrDeviceType = "waveaudio";
mciOpenParms.lpstrElementName = "C:\\SOUNDS\\BELLS.WAV";
if (mciSendCommand(0,                                       // device ID
                   MCI_OPEN,                                // command
                   MCI_OPEN_ELEMENT | MCI_OPEN_TYPE,        // flags
                   (DWORD)(LPVOID) &mciOpenParms))          // parameter block
    /* Error, unable to open device
     */
    ...
else
    /* Device opened successfully, get the device ID
     */
    wDeviceID = mciOpenParms.wDeviceID;
```

Using the MIDI Mapper with the MCI Sequencer

The MIDI Mapper is the default output device for the MCI MIDI sequencer. It provides standard patch services for device-independent playback of MIDI files. Applications that use the MCI sequencer to play MIDI files should use the MIDI Mapper. For details on the MIDI Mapper, see "The MIDI Mapper," later in this chapter. For information on authoring device-independent MIDI files, see "Authoring MIDI Files," also later in this chapter.

When you open the MCI MIDI sequencer, MCI attempts to select the MIDI Mapper as the output device. If the Mapper is unavailable because it's already in use, MCI selects another MIDI output device.

Note The MIDI Mapper currently supports only one client at a time. This might change in future versions of Windows.

Example of Verifying the Output Device

After opening the sequencer, you should check to see if the MIDI Mapper was available and selected as the output device. The following code fragment uses the MCI_STATUS command to verify that the MIDI Mapper is the output device for the MCI sequencer:

```
UINT wDeviceID;
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 */
    ...
```

Example Selecting MIDI Mapper as the Output Device

MCI also provides a command to explicitly select the MIDI Mapper as the output device for the sequencer. The following code fragment uses the MCI_SET command to select the MIDI Mapper as the output device for the MCI sequencer:

```
UINT wDeviceID;
DWORD dwReturn;
MCI_SEQ_SET_PARMS mciSeqSetParms;
.
.
.
/* Set the MIDI Mapper as the output port for the open device
 */
mciSeqSetParms.dwPort = MIDI_MAPPER;
if (dwReturn = mciSendCommand(wDeviceID, MCI_SET, MCI_SEQ_SET_PORT,
                              (DWORD)(LPVOID) &mciSeqSetParms))
{
    /* Error, unable to set Mapper as output port */
    ...
}
```

Before querying the sequencer or setting an output port, you must successfully open the sequencer. Both of the previous examples assume that the *wDeviceID* parameter contains a valid device ID for the sequencer.

Opening Compact Disc Devices

Because a compact disc audio device is a simple device, you need only specify the device name when opening it by using either a string or a constant ID. The following code fragment opens a compact disc device using a constant ID to specify the device name:

```
UINT wDeviceID;
MCI_OPEN_PARMS mciOpenParms;
.
.
.
/* Open the device by specifying a device ID constant
 */
mciOpenParms.lpstrDeviceType = (LPCSTR) MCI_DEVTYPE_CD_AUDIO;
if (mciSendCommand(0,                                    // device ID
                   MCI_OPEN,                             // command
                   MCI_OPEN_TYPE | MCI_OPEN_TYPE_ID,     // flags
                   (DWORD)(LPVOID) &mciOpenParms))       // parameter block
    /* Error, unable to open device
     */
    ...
else
    /* Device opened successfully, get the device ID
     */
    wDeviceID = mciOpenParms.wDeviceID;
```

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. You can also interpret the error code yourself—MMSYSTEM.H defines constants for MCI error return codes.

Note To interpret an **mciSendCommand** error return value yourself, mask the high-order word—the low-order word contains the error code. However, if you pass the error return to **mciGetErrorString**, you must pass the entire DWORD value.

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

```
/* 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);
}
```

Starting Playback

Once you successfully open an MCI audio device, you can use the MCI_PLAY command along with the MCI_PLAY_PARMS parameter block to begin playback. MMSYSTEM.H defines the MCI_PLAY_PARMS parameter block as follows:

```
typedef struct {
    DWORD   dwCallback;         /* callback for MCI_NOTIFY flag */
    DWORD   dwFrom;             /* playback begin position */
    DWORD   dwTo;               /* playback end position */
} MCI_PLAY_PARMS;
```

Playback begins at the current position in the device element. When a device element is opened, the current position is set to the beginning of the media. After a device element is played, the current position is at the end of the media. You can use the MCI_SEEK command to change the current position, as explained in "Changing the Current Position," later in this chapter.

You can also set beginning and ending positions for playback by specifying the MCI_FROM and MCI_TO flags with the MCI_PLAY command. If you specify one of these flags, you must fill in the corresponding **dwFrom** or **dwTo** field in the MCI_PLAY_PARMS structure with the desired beginning or ending time. If you are using a time format other than the default time format (milliseconds), you must set the time format before specifying a beginning or ending time with MCI_PLAY.

Example of Playing a WAVE File

The following function opens a waveform device and plays the WAVE file specified by the *lpszWAVEFileName* parameter:

```
/* Plays a given WAVE file using MCI_OPEN, 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 name and device element.
     * 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);
    }

    /* 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);
}
```

Example of Playing a MIDI File

The following function 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 given MIDI file 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, 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 name and device element.
     * 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);
    }

    /* Device opened successfully, get the device ID.
     */
    wDeviceID = mciOpenParms.wDeviceID;

    /* See 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 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);
}
```

Example of Playing a Compact Disc Track

The following function opens a compact disc device, plays the track specified by the *wTrack* parameter, and closes the device after playback is complete:

```
/* Plays a given compact disc 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, it 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 compact disc 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);
    }

    /* Device opened successfully, get the device ID.
     */
    wDeviceID = mciOpenParms.wDeviceID;

    /* Set the time format to track/minute/second/frame.
     */
    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 track-relative position with a compact disc device, you must use the track/minute/second/frame time format. See "Setting the Time Format," later in this chapter, for details on setting time formats.

Changing the Current Position

To change the current position in a device element, use the MCI_SEEK command along with the MCI_TO flag and the MCI_SEEK_PARMS parameter block. MMSYSTEM.H defines the MCI_SEEK_PARMS parameter block as follows:

```
typedef struct {
    DWORD   dwCallback;         /* callback for MCI_NOTIFY flag */
    DWORD   dwTo;               /* seek position */
} MCI_SEEK_PARMS;
```

If you use the **dwTo** field to specify a seek position with MCI_SEEK, you should query the time format and set it if necessary.

In addition to specifying a position with the **dwTo** field, you can specify the MCI_SEEK_TO_START or MCI_SEEK_TO_END flags for the *dwParam1* parameter of **mciSendCommand** to seek to the starting and ending positions of the device element. If you use one of these flags, don't specify the MCI_TO flag.

Setting the Time Format

Use the MCI_SET command along with the MCI_SET_PARMS parameter block to set the time format for an open device. Set the **dwTimeFormat** field in the parameter block to one of the constants identified in the following table:

Time Format	Constant
Waveform Devices	
Milliseconds	MCI_FORMAT_MILLISECONDS
Samples	MCI_FORMAT_SAMPLES
Bytes (in PCM format files)	MCI_FORMAT_BYTES
Compact Disc Devices	
Milliseconds	MCI_FORMAT_MILLISECONDS
Minute/Second/Frame	MCI_FORMAT_MSF
Track/Minute/Second/Frame	MCI_FORMAT_TMSF
MIDI Sequencer Devices	
Milliseconds	MCI_FORMAT_MILLISECONDS
MIDI Song Pointer	MCI_SEQ_FORMAT_SONGPTR
SMPTE, 24 Frame	MCI_FORMAT_SMPTE_24
SMPTE, 25 Frame	MCI_FORMAT_SMPTE_25
SMPTE, 30 Frame	MCI_FORMAT_SMPTE_30
SMPTE, 30 Frame Drop	MCI_FORMAT_SMPTE_30DROP

As an example, the following code fragment sets the time format to milliseconds on the device specified by *wDeviceID*:

```
UINT wDeviceID;
MCI_SET_PARMS mciSetParms;
.
.
.
/* Set time format to milliseconds
 */
mciSetParms.dwTimeFormat = MCI_FORMAT_MILLISECONDS;
if (mciSendCommand(wDeviceID, MCI_SET, MCI_SET_TIME_FORMAT,
                   (DWORD)(LPVOID) &mciSetParms))
```

```
    /* Error, unable to set time format
     */
    ...
else
    /* Time format set successfully
     */
    ...
```

Using the Minute/Second/Frame Time Format

For the minute/second/frame time format specified with the MCI_FORMAT_MSF constant, the time is relative to the beginning of the media. The time is packed into a DWORD, as shown in the following illustration:

HIWORD		LOWORD	
HIBYTE (Unused)	**LOBYTE** Frames	**HIBYTE** Seconds	**LOBYTE** Minutes

DWORD packing for the minute/second/frame time format.

The MMSYSTEM.H header file defines the following macros to get and set elements of a minute/second/frame packed DWORD:

MCI_MSF_MINUTE
Gets minute value in a minute/second/frame DWORD.

MCI_MSF_SECOND
Gets second value in a minute/second/frame DWORD.

MCI_MSF_FRAME
Gets frame value in a minute/second/frame DWORD.

MCI_MAKE_MSF
Sets minute, second, and frame values in a minute/second/frame DWORD.

Note The first few seconds of audio compact discs contain table of contents data. To play from the beginning of a disc using the minute/second/frame time format, use the MCI_STATUS command to get the position of the first track and play from that position.

Using the Track/Minute/Second/Frame Time Format

For the track/minute/second/frame time format specified with the MCI_FORMAT_TMSF constant, the time is relative to the beginning of the specified track. The time is packed into a DWORD, as shown in the following illustration:

HIWORD		LOWORD	
HIBYTE Frames	**LOBYTE** Seconds	**HIBYTE** Minutes	**LOBYTE** Track

DWORD packing for the track/minute/second/frame time format.

The MMSYSTEM.H header file defines the following macros to get and set elements of a track/minute/second/frame packed DWORD:

MCI_TMSF_TRACK
Gets track value in a track/minute/second/frame DWORD.

MCI_TMSF_MINUTE
Gets minute value in a track/minute/second/frame DWORD.

MCI_TMSF_SECOND
Gets second value in a track/minute/second/frame DWORD.

MCI_TMSF_FRAME
Gets frame value in a track/minute/second/frame DWORD.

MCI_MAKE_TMSF
Sets track, minute, second, and frame values in a track/minute/second/frame DWORD.

Using the SMPTE Time Formats

SMPTE (Society of Motion Picture and Television Engineers) time formats are based on standard time formats developed for the motion picture and television industries. For SMPTE time formats, the time is packed into a DWORD, as shown in the following illustration:

HIWORD		LOWORD	
HIBYTE Frames	**LOBYTE** Seconds	**HIBYTE** Minutes	**LOBYTE** Hours

DWORD packing for SMPTE time format.

Closing MCI Audio Devices

After you finish using an MCI device, you must use the MCI_CLOSE command to close the device. If you are playing a compound device, such as a waveform or MIDI device, and want to play a different device element using the same device, close the device and reopen it using the new device element.

Note If you're playing multiple device elements with the same device, you'll get better performance if you open the device by explicitly specifying its name and then open, play, and close the individual device elements separately. Don't close the device until you are through playing all of the device elements. Otherwise, the driver will be reloaded each time you open the device.

Getting Information About Devices and Media

Use the MCI_STATUS command along with the MCI_STATUS_PARMS parameter block to get information about the status of an open device and its associated device element. MMSYSTEM.H defines the MCI_STATUS_PARMS parameter block as follows:

```
typedef struct {
    DWORD   dwCallback;     /* callback for MCI_NOTIFY flag */
    DWORD   dwReturn;       /* status information is returned here */
    DWORD   dwItem;         /* identifies status item */
    DWORD   dwTrack;        /* track number */
} MCI_STATUS_PARMS;
```

Before using the MCI_STATUS command, you must identify the status item to query for by putting a constant in the **dwItem** field of the parameter block. The following list shows different status items you can query for and the corresponding constant for each item for different types of audio devices:

Status Item	Constant
All Audio Devices	
Length of the media	MCI_STATUS_LENGTH
Current position	MCI_STATUS_POSITION
Current mode	MCI_STATUS_MODE
Time format	MCI_STATUS_TIME_FORMAT
Ready state	MCI_STATUS_READY
Waveform Devices	
Block alignment	MCI_WAVE_STATUS_BLOCKALIGN
Format tag	MCI_WAVE_STATUS_FORMATTAG
Number of channels	MCI_WAVE_STATUS_CHANNELS
Sample rate	MCI_WAVE_STATUS_SAMPLESPERSEC
Average bytes per second	MCI_WAVE_STATUS_AVGBYTESPERSEC
Bits per sample (in PCM format files)	MCI_WAVE_STATUS_BITSPERSAMPLE
Record level	MCI_WAVE_STATUS_LEVEL
Compact Disc Devices	
Number of tracks	MCI_STATUS_NUMBER_OF_TRACKS
Media present	MCI_STATUS_MEDIA_PRESENT
Current track	MCI_STATUS_CURRENT_TRACK
MIDI Sequencer Devices	
Tempo	MCI_SEQ_STATUS_TEMPO
Port	MCI_SEQ_STATUS_PORT
SMPTE offset	MCI_SEQ_STATUS_OFFSET
Division type of file	MCI_SEQ_STATUS_DIVTYPE

Getting Track-Relative Information for Compact Disc Devices

For compact disc devices, you can get the starting position and length of a track by specifying the MCI_TRACK flag and setting the **dwTrack** field of MCI_STATUS_PARMS to the desired track number. To get the starting position of a track, set the **dwItem** field to MCI_STATUS_POSITION. To get the length of a track, set **dwItem** to MCI_STATUS_LENGTH. For example, the following function gets the total number of tracks on the disc and the starting position of each track. It then uses the **MessageBox** function to report the starting positions of the tracks.

```
/* Uses the MCI_STATUS command to get and display the starting times
 * for the tracks on a compact disc. Returns 0L on success; 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";      // big enough for 20 tracks
    MCI_OPEN_PARMS mciOpenParms;
    MCI_SET_PARMS mciSetParms;
    MCI_STATUS_PARMS mciStatusParms;

    /* Open the compact disc 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);
    }

    /* Device opened successfully, get the device ID.
     */
    wDeviceID = mciOpenParms.wDeviceID;
```

```
/* Set the time format to minute/second/frame 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 we can display (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 position and
     * build a string containing starting positions.
     */
    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 Waveform Audio Devices

MCI supports recording with waveform audio devices. 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 device and device element as you would normally. To record into a new file, specify a zero-length filename for the device element when you open the device.

When MCI creates a new file for recording, the waveform 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 MCI_SET to change the format.

To begin recording, use the MCI_RECORD command along with the MCI_RECORD_PARMS parameter block. MMSYSTEM.H defines the MCI_RECORD_PARMS parameter block as follows:

```
typedef struct {
    DWORD   dwCallback;         /* callback for MCI_NOTIFY flag */
    DWORD   dwFrom;             /* record begin position */
    DWORD   dwTo;               /* record end position */
} MCI_RECORD_PARMS;
```

If you record to an existing file, you can use the MCI_TO and MCI_FROM flags to specify beginning and ending points for recording. For example, if you record to an existing file 20 seconds long, and you begin recording at 5 seconds and end recording at 10 seconds, you will have a recording 25 seconds long, as shown in the following illustration:

5 Seconds
Original
5 Seconds
New
15 Seconds
Original

Recording waveform audio to an existing file.

If you don't specify an ending position, recording continues until you send an MCI_STOP command, or until the driver runs out of free disk space. If you record to a new file, you can omit the MCI_FROM flag or set it to 0 to start recording at the beginning of a new file. You can specify an end position to terminate recording when recording to a new file.

If you record with overwrite mode to an existing file, you can use the MCI_TO and MCI_FROM flags to specify beginning and ending points of the waveform section that is overwritten. For example, if you record to an existing file 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 has been replaced.

Saving a Recorded File

When recording is complete, use the MCI_SAVE command along with the MCI_SAVE_PARMS parameter block to save the recording before closing the device. MMSYSTEM.H defines the MCI_SAVE_PARMS parameter block as follows:

```
typedef struct {
    DWORD   dwCallback;     /* callback for MCI_NOTIFY flag */
    LPCSTR  lpfilename;     /* filename for saved file */
} MCI_SAVE_PARMS;
```

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 waveform input device, use the MCI_STATUS command. Specify the MCI_STATUS_ITEM flag and
set **dwItem** in MCI_STATUS_PARMS to MCI_WAVE_STATUS_LEVEL.
The average input signal level is returned in the **dwReturn** field of the MCI_STATUS_PARMS parameter block. The left-channel value is in the high-order word and the right- or mono-channel value is in the low-order word, as shown in the following illustration:

HIWORD	LOWORD
Left-Channel Level	Right- or Mono-Channel Level

DWORD packing for waveform input levels.

The input level is represented as an unsigned value. For 8-bit samples, this value ranges from 0 through 127 (0x7F). For 16-bit samples, it ranges from 0 through 32,767 (0x7FFF).

Example of Recording with a Waveform Audio Device

The following function opens a waveform audio device with a new file, records for the specified time, plays the recording, and prompts the user to see if the recording should be saved as a file:

```
/* Uses the MCI_OPEN, MCI_RECORD, MCI_SAVE commands to record and
 * save a waveform audio file. Returns 0L on success; 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 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);
    }

    /* 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 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);
}
```

Using the MCI MIDI Sequencer

The MCI MIDI sequencer plays standard MIDI files and RIFF MIDI files (RMID files). Standard MIDI files conform to the "Standard MIDI Files 1.0" specification. See Chapter 3, "Introduction to Audio," for information about this specification. Because RMID files are standard MIDI files with a RIFF header, information on standard MIDI files also applies to RMID files.

There are currently three variations of standard MIDI files. The MCI sequencer plays only two of these: Format 0 and Format 1 MIDI files.

Note If the MCI MIDI sequencer encounters a MIDI file not marked as being authored for Windows, it will display a dialog box informing the user. MIDI files authored for Windows should be marked with the **MARKMIDI** utility program. For additional information about the **MARKMIDI** utility, see the last section of this chapter, "Using the MARKMIDI Utility."

Querying for Sequence Division Types

The *division type* of a MIDI sequence refers to the technique used to represent the time between MIDI events in the sequence. Use the MCI_STATUS command and set the **dwItem** field of the MCI_STATUS_PARMS parameter block to MCI_SEQ_STATUS_DIVTYPE to determine the division type of a sequence.

If the MCI_STATUS command is successful, the **dwReturn** field of the MCI_STATUS_PARMS parameter block 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 can't change the division type of a sequence using the MCI sequencer.

Querying and Setting the Tempo

Use the MCI_STATUS command and set the **dwItem** field of the MCI_STATUS_PARMS parameter block to MCI_SEQ_STATUS_TEMPO to get the tempo of a sequence. If the MCI_STATUS command is successful, the **dwReturn** field of the MCI_STATUS_PARMS parameter block contains the current tempo.

To change tempo, use the MCI_SET command with the MCI_SEQ_SET_PARMS parameter block. Specify the MCI_SEQ_SET_TEMPO flag and set the **dwTempo** field of the parameter block to the desired tempo. MMSYSTEM.H defines the MCI_SEQ_SET_PARMS parameter block as follows:

```
typedef struct {
    DWORD   dwCallback;     /* callback for MCI_NOTIFY flag */
    DWORD   dwTimeFormat;   /* time format */
    DWORD   dwAudio;        /* audio channel (not used by sequencer) */
    DWORD   dwTempo;        /* tempo */
    DWORD   dwPort;         /* output port */
    DWORD   dwSlave;        /* slave sync type */
    DWORD   dwMaster;       /* master sync type */
    DWORD   dwOffset;       /* SMPTE offset */
} MCI_SEQ_SET_PARMS;
```

Tempo Representation and Division Type

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. See the previous section, "Querying for Sequence Division Types," for information on determining the division type of a sequence.

Changing Sequencer Synchronization

To change the synchronization mode of a sequencer device, use the MCI_SET command along with the MCI_SEQ_SET_MASTER and MCI_SEQ_SET_SLAVE flags. Two fields in the MCI_SEQ_SET_PARMS parameter block, **dwMaster** and **dwSlave**, are used to specify the master and slave synchronization modes.

The master synchronization mode controls synchronization information sent by the sequencer to an output port. The slave synchronization mode controls where the sequencer gets its timing information to play a MIDI file. The following table shows the different modes for master and slave synchronization and the corresponding constant for the **dwMaster** and **dwSlave** fields:

Synchronization Mode	Constant
Master	
MIDI Sync—Send timing information to output port using MIDI timing clock messages	MCI_SEQ_MIDI
SMPTE Sync—Send timing information to output port using MIDI quarter frame messages	MCI_SEQ_SMPTE
No Sync—Send no timing information	MCI_SEQ_NONE
Slave	
File Sync—Get timing information from MIDI file	MCI_SEQ_FILE
MIDI Sync—Get timing information from input port using MIDI timing clock messages	MCI_SEQ_MIDI
SMPTE Sync—Get timing information from input port using MIDI quarter frame messages	MCI_SEQ_SMPTE
No Sync—Get timing information from MCI commands only and ignore timing information such as tempo changes that are in the MIDI file	MCI_SEQ_NONE

Note Currently, for master synchronization, the MCI MIDI sequencer supports only the "no synchronization" mode (MCI_SEQ_NONE). For slave synchronization, it only supports the file synchronization mode (MCI_SEQ_FILE) and the "no synchronization" mode (MCI_SEQ_NONE).

The MIDI Mapper

The MIDI Mapper's standard patch services provide device-independent MIDI file playback for applications. You don't need to understand exactly how the MIDI Mapper works to use these services. The MIDI Mapper can be used with the MCI MIDI sequencer or with low-level MIDI output services.

To learn more about the MIDI Mapper or about authoring standard MIDI files for Windows, read this section to become familiar with how the MIDI Mapper works. For specific information on how to use the MIDI Mapper, see "Using the MIDI Mapper with the MCI Sequencer," earlier in this chapter, or see Chapter 5, "Low-Level Audio Services."

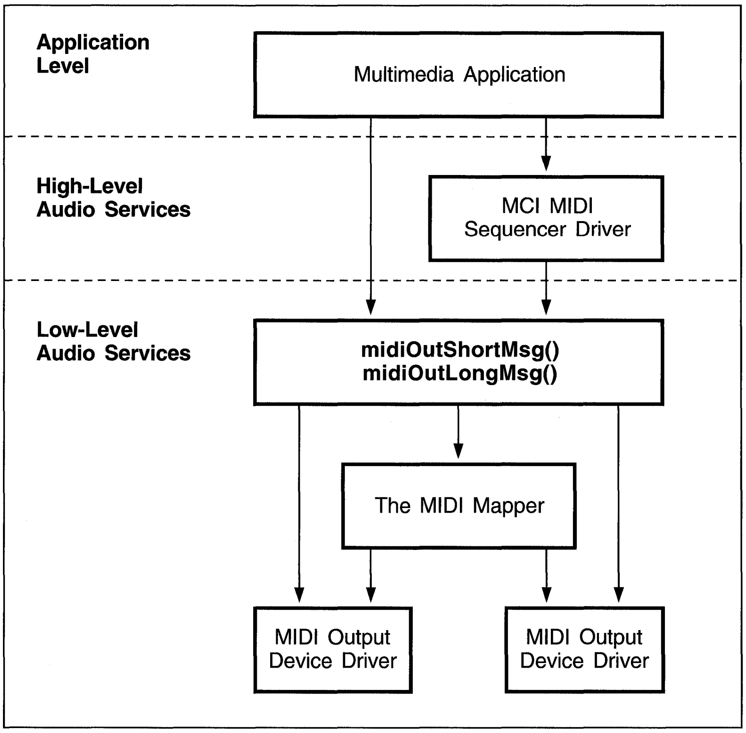
MIDI Notational Conventions

Unless stated otherwise, all references to MIDI channel numbers use the logical channel numbers 1 through 16. These logical channel numbers correspond to the physical channel numbers 0 through 15 that are actually part of the MIDI message. All references to MIDI program-change and key values use the physical values 0 through 127. All numbers are decimal unless preceded by a "0x" prefix, in which case they are hexadecimal.

In the discussion of the MIDI Mapper, the term *source* refers to the input side of the Mapper. The term *destination* refers to the output side of the Mapper. For example, a source channel is the MIDI channel of a message sent to the Mapper, a destination channel is the MIDI channel of a message sent from the Mapper to an output device.

The MIDI Mapper and Windows

The MIDI Mapper is part of the system software. The following illustration shows how the MIDI Mapper relates to other elements of the audio services:



The relationship of the MIDI Mapper to the Multimedia extensions.

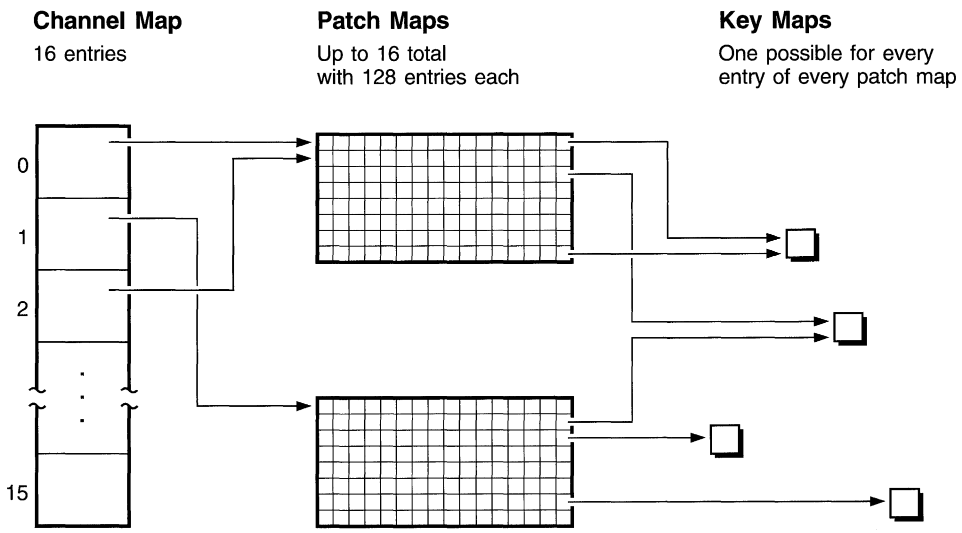
From the viewpoint of an application, the MIDI Mapper looks like another MIDI output device. The MIDI Mapper receives messages sent to it by the low-level MIDI output functions **midiOutShortMsg** and **midiOutLongMsg**. The Mapper modifies these messages and redirects them to a MIDI output device according to the current MIDI setup map. The current MIDI setup map is selected by the user using the MIDI Control Panel option. Only the user can select the current setup map; applications cannot change the current setup map.

The MIDI Mapper Architecture

The MIDI Mapper uses a MIDI setup map to determine how to translate and redirect messages it receives. A MIDI setup map consists of the following types of maps:

- Channel map
- Patch map
- Key map

The following illustration shows how channel, patch, and key maps comprise a MIDI setup map:



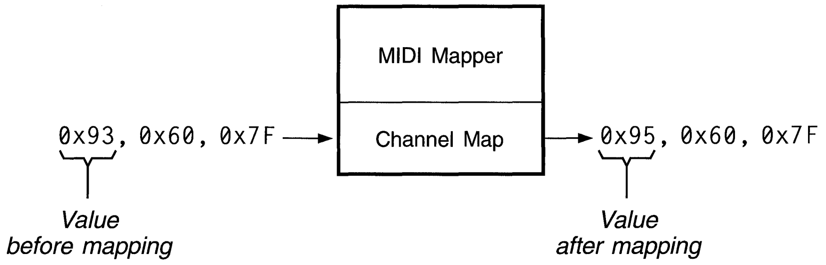
The MIDI Mapper architecture.

The Channel Map

The channel map affects all MIDI channel messages. MIDI channel messages include note-on, note-off, polyphonic-key-aftertouch, control-change, program-change, channel-aftertouch, and pitch-bend-change messages. The MIDI Mapper uses a single channel map with an entry for each of the 16 MIDI channels. Each channel map entry specifies the following:

- A destination channel for the MIDI message
- A destination output device for the MIDI message
- An optional patch map specifying other possible modifications for the MIDI message

The destination channel is set to one of the 16 MIDI channels. MIDI messages are modified to reflect each new channel assignment. For example, if the destination channel entry for MIDI channel 4 is set to 6, all MIDI messages sent to channel 4 will be mapped to channel 6, as shown in the following illustration:



Channel mapping.

In this example, the MIDI status byte 0x93 is mapped to 0x95. The low nibble of a MIDI status byte specifies the channel number. Source channels are set to either active or inactive. Messages sent to inactive source channels are ignored, allowing the channel to be muted or turned off.

The destination output device is set to one of the available MIDI output devices. A MIDI output device can be an internal synthesizer or a physical MIDI output port attached to an external MIDI synthesizer.

MIDI system messages are MIDI messages from 0xF0 to 0xFF. There is no channel associated with MIDI system messages, so they can't be mapped. MIDI system messages are sent to all MIDI output devices listed in a channel map.

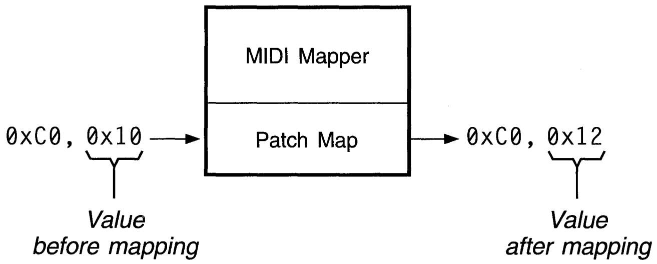
Patch Maps

Each channel map entry can have an associated patch map. Patch maps affect MIDI program-change and volume-controller messages. Program-change messages tell a synthesizer to change the instrument sound for a specified channel. Volume-controller messages set the volume for a channel.

A patch map has a translation table with an entry for each of the 128 program-change values. Each patch map specifies the following:

- A destination program-change value
- A volume scalar
- An optional key map

When program-change messages are received by the MIDI Mapper, the destination program-change value is substituted for the program-change value in the message. For example, if the destination program-change value for program-change 16 is 18, the Mapper modifies the MIDI program-change message, as shown in the following illustration:

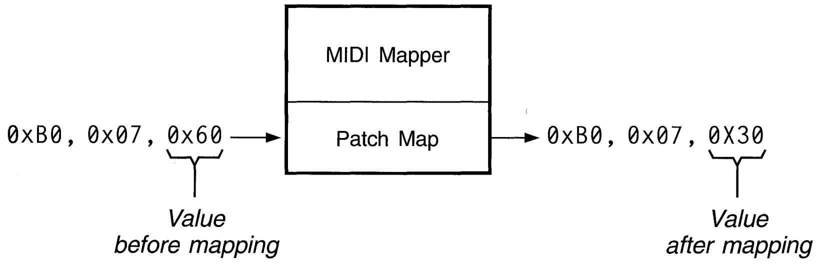


Patch mapping.

The Volume Scalar

The purpose of the volume scalar is to allow adjustments between the relative output levels of different patches on a synthesizer. For example, if the bass patch on a synthesizer is too loud compared to its piano patch, you can change the setup map to scale the bass volume down or the piano volume up.

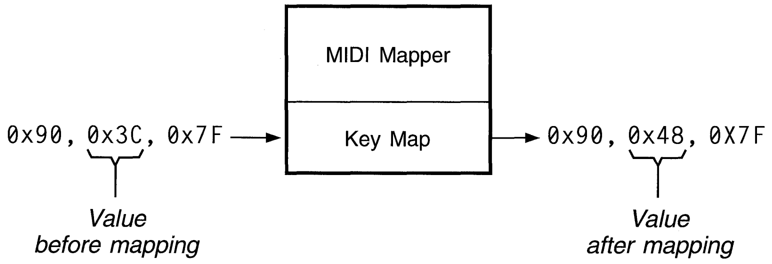
The volume scalar specifies a percentage value for changing all MIDI main-volume controller messages that follow an associated program-change message. For example, if the volume scalar value is 50%, then the Mapper modifies MIDI main-volume controller messages, as shown in the following illustration:



Volume scaling.

Key Maps

Each entry in the patch map translation table can have an associated key map. Key maps affect note-on, note-off, and polyphonic-key-aftertouch messages. A key map has a translation table with an entry for each of the 128 MIDI key values. For example, if the entry for key value 60 is 72, then the Mapper modifies MIDI note-on messages, as shown in the following illustration:



Key mapping.

Key maps are useful with synthesizers having key-based percussion instruments where a particular percussion sound is assigned to each key. Key maps are usually assigned to the first patch in the patch maps on the percussion channels (10 and 16).

Summary of Maps and MIDI Messages

The following table contains a list of status bytes for MIDI messages and shows which types of maps affect each message:

MIDI Status	Description	Map Types
0x80–0x8F	Note-off	Channel maps, key maps
0x90–0x9F	Note-on	Channel maps, key maps
0xA0–0xAF	Polyphonic-key-aftertouch	Channel maps, key maps
0xB0–0xBF	Control-change	Channel maps, patch maps[1]
0xC0–0xCF	Program-change	Channel maps, patch maps
0xD0–0xDF	Channel-aftertouch	Channel maps
0xE0–0xEF	Pitch-bend-change	Channel maps
0xF0–0xFF	System	Not mapped[2]

1. Patch maps affect only controller 7 (main volume).
2. System messages are sent to all devices listed in a channel map.

Authoring MIDI Files

The "MIDI 1.0 Detailed Specification" does not define any standard patch assignments for synthesizers. Therefore, when you create a MIDI file, it won't be reproduced correctly unless it is played back on the same MIDI synthesizer setup used to create it. For example, if you create a piano concerto for your Yamaha DX7 and try to play it back on a Roland LAPC-1, it might be played with a flute instead of a piano.

To enable MIDI files to be a viable format for representing music in multimedia computing, Windows provides MIDI authoring guidelines. These guidelines include a list of standard patch assignments and standard key assignments for percussion instruments. Using the MIDI Mapper, MIDI files authored to these guidelines can be played on any multimedia computer with internal or external MIDI synthesizers.

About Base-Level and Extended Synthesizers

Although it is difficult to clearly quantify distinctions between synthesizers, it is important to have some guidelines so you can create MIDI files that will play on all multimedia computers. The terms used to distinguish synthesizers for the purpose of authoring MIDI files are *base-level synthesizer* and *extended synthesizer.*

The distinctions between base-level and extended synthesizers are made solely on the number of instruments and notes the synthesizer can play, not on the quality or the cost of the synthesizer. The following table shows the minimum capabilities of base-level and extended synthesizers:

Synthesizer	**Melodic Instruments**		**Percussive Instruments**	
	Number	**Polyphony**	**Number**	**Polyphony**
Base-Level	3 instruments	6 notes	3 instruments	3 notes
Extended	9 instruments	16 notes	8 instruments	16 notes

Polyphony is the number of notes the synthesizer can play simultaneously. The polyphony expressed above applies to each group of instruments—melodic and percussive. For example, a base-level synthesizer is capable of playing six notes distributed among three melodic instruments and three notes distributed among three percussive instruments. The melodic instruments are each on different MIDI channels, and the percussive instruments are key-based—all on a single MIDI channel.

All multimedia computers provide at least a base-level synthesizer. Users can enhance their computer by adding internal or external synthesizers, which can be either base-level or extended synthesizers. When a user adds a synthesizer, the user must configure the MIDI Mapper to use the new device, or the instrument sounds will not be correct when playing MIDI files. The MIDI Control Panel option allows a user to configure the MIDI Mapper as needed.

Authoring Guidelines for MIDI Files

Follow these guidelines to author device-independent MIDI files for Windows:

- Author for both base-level and extended synthesizer setups.
- Use MIDI channels 13 through 16 for base-level synthesizer data (reserve channel 16 for key-based percussion instruments).
- Use MIDI channels 1 through 10 for extended synthesizer data (reserve channel 10 for key-based percussion instruments).
- Prioritize MIDI data by putting crucial data in the lower-numbered channels.
- Limit the polyphony of non-percussive channels to a total of 6 notes for base-level data and 16 notes for extended data.
- Limit the polyphony of percussive channels to a total of 3 notes for base-level data and 16 notes for extended data.
- Use the standard MIDI patch assignments and key assignments.
- Always send a program-change message to a channel to select a patch before sending other messages to that channel. For the two percussion channels (10 and 16), select program number 0.
- Always follow a MIDI program-change message with a MIDI main-volume-controller message (controller number 7) to set the relative volume of the patch.
- Use a value of 80 (0x50) for the main volume controller for normal listening levels. For quieter or louder levels, you can use lower or higher values.
- Use only the following MIDI messages: note-on with velocity, note-off, program change, pitch bend, main volume (controller 7), and damper pedal (controller 64). Internal synthesizers are required to respond to these messages and most MIDI musical instruments will respond to them as well.
- Use the **MARKMIDI** utility to mark MIDI files authored for Windows.

The following illustration summarizes the use of the 16 MIDI channels in a standard MIDI file authored for Windows:

Channel	Description	Polyphony
1	Extended Melodic Tracks	
2		
3		
4		
5		16 Notes
6		
7		
8		
9	↓	
10	Extended Percussion Track	16 Notes
11	Unused Tracks	
12		
13	Base-Level Melodic Tracks	6 Notes
14		
15	↓	
16	Base-Level Percussion Track	3 Notes

Prioritizing MIDI Data

Synthesizers don't always fall cleanly into the base-level and extended designations defined earlier. It's up to the end-user to determine how to use synthesizers capable of more than the base-level requirements, but not fully meeting the extended requirements. For this reason, it's important to prioritize the melodic data by putting the most critical data in lower-numbered channels. For example, a user may have a synthesizer capable of playing six melodic instruments with 12-note polyphony. The user can use this device as an extended synthesizer by setting up the MIDI Mapper to play only the first six melodic channels and ignore any information on channels seven, eight, and nine.

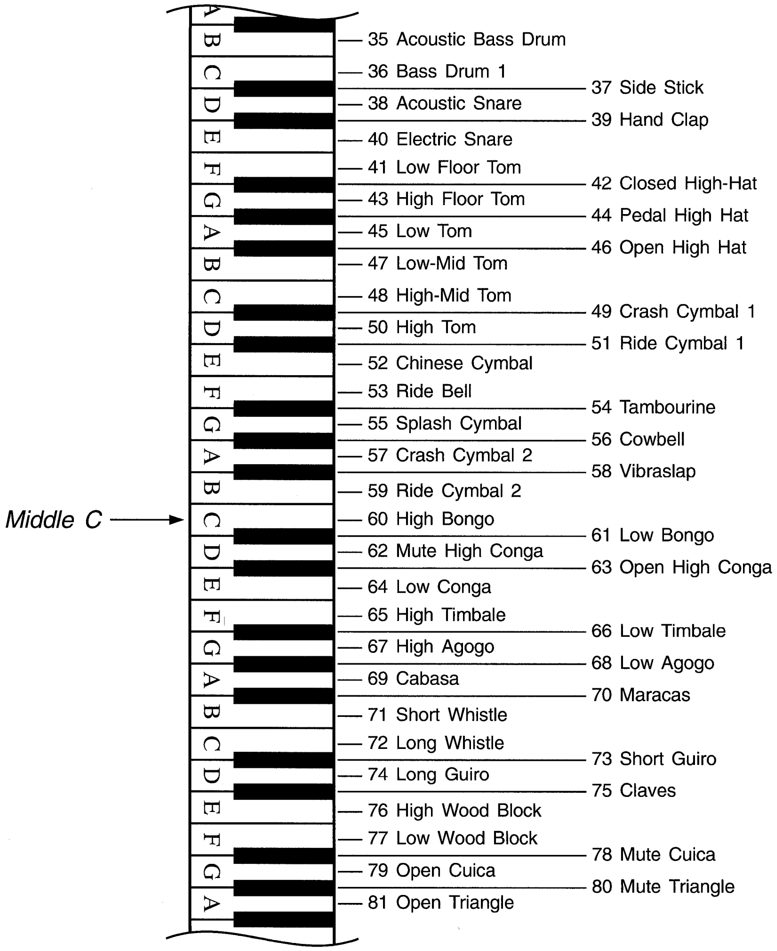
Standard MIDI Patch Assignments

The standard MIDI patch assignments for authoring MIDI files for use with Windows are based on the MIDI Manufacturers Association (MMA) General MIDI Mode specification. The following illustration shows the standard MIDI patch assignments.

Piano		**Chromatic Percussion**		**Organ**		**Guitar**	
0	Acoustic Grand Piano	8	Celesta	16	Hammond Organ	24	Acoustic Guitar (nylon)
1	Bright Acoustic Piano	9	Glockenspiel	17	Percussive Organ	25	Acoustic Guitar (steel)
2	Electric Grand Piano	10	Music box	18	Rock Organ	26	Electric Guitar (jazz)
3	Honky-tonk Piano	11	Vibraphone	19	Church Organ	27	Electric Guitar (clean)
4	Rhodes Piano	12	Marimba	20	Reed Organ	28	Electric Guitar (muted)
5	Chorused Piano	13	Xylophone	21	Accordion	29	Overdriven Guitar
6	Harpsichord	14	Tubular Bells	22	Harmonica	30	Distortion Guitar
7	Clavinet	15	Dulcimer	23	Tango Accordion	31	Guitar Harmonics
Bass		**Strings**		**Ensemble**		**Brass**	
32	Acoustic Bass	40	Violin	48	String Ensemble 1	56	Trumpet
33	Electric Bass (finger)	41	Viola	49	String Ensemble 2	57	Trombone
34	Electric Bass (pick)	42	Cello	50	SynthStrings 1	58	Tuba
35	Fretless Bass	43	Contrabass	51	SynthStrings 2	59	Muted Trumpet
36	Slap Bass 1	44	Tremolo Strings	52	Choir Aahs	60	French Horn
37	Slap Bass 2	45	Pizzicato Strings	53	Voice Oohs	61	Brass Section
38	Synth Bass 1	46	Orchestral Harp	54	Synth Voice	62	Synth Brass 1
39	Synth Bass 2	47	Timpani	55	Orchestra Hit	63	Synth Brass 2
Reed		**Pipe**		**Synth Lead**		**Synth Pad**	
64	Soprano Sax	72	Piccolo	80	Lead 1 (square)	88	Pad 1 (new age)
65	Alto Sax	73	Flute	81	Lead 2 (sawtooth)	89	Pad 2 (warm)
66	Tenor Sax	74	Recorder	82	Lead 3 (caliope lead)	90	Pad 3 (polysynth)
67	Baritone Sax	75	Pan Flute	83	Lead 4 (chiff lead)	91	Pad 4 (choir)
68	Oboe	76	Bottle Blow	84	Lead 5 (charang)	92	Pad 5 (bowed)
69	English Horn	77	Shakuhachi	85	Lead 6 (voice)	93	Pad 6 (metallic)
70	Bassoon	78	Whistle	86	Lead 7 (fifths)	94	Pad 7 (halo)
71	Clarinet	79	Ocarina	87	Lead 8 (brass + lead)	95	Pad 8 (sweep)
Synth Effects		**Ethnic**		**Percussive**		**Sound Effects**	
96	FX 1 (rain)	104	Sitar	112	Tinkle Bell	120	Guitar Fret Noise
97	FX 2 (soundtrack)	105	Banjo	113	Agogo	121	Breath Noise
98	FX 3 (crystal)	106	Shamisen	114	Steel Drums	122	Seashore
99	FX 4 (atmosphere)	107	Koto	115	Woodblock	123	Bird Tweet
100	FX 5 (brightness)	108	Kalimba	116	Taiko Drum	124	Telephone Ring
101	FX 6 (goblins)	109	Bagpipe	117	Melodic Tom	125	Helicopter
102	FX 7 (echoes)	110	Fiddle	118	Synth Drum	126	Applause
103	FX 8 (sci-fi)	111	Shanai	119	Reverse Cymbal	127	Gunshot

Standard MIDI Key Assignments

The standard MIDI key assignments for percussion instruments are based on the MIDI Manufacturers Association (MMA) General MIDI Mode specification. The following illustration shows the standard key assignments for MIDI files authored for Windows:



Standard MIDI key assignments for key-based percussion instruments.

Using the MARKMIDI Utility

The **MARKMIDI** utility marks MIDI files as following the authoring guidelines for Windows MIDI files. The utility does not verify that the file is authored correctly, it only marks the file by adding a sequencer-specific meta-event to the first track chunk. **MARKMIDI** is an MS-DOS application with the following command syntax:

MARKMIDI *src-filename dest-filename*

If the MCI MIDI sequencer is used to play a MIDI file not marked with **MARKMIDI**, it will display a dialog informing the user that the file is not authored for Windows.

Note Do not use **MARKMIDI** to mark MIDI files unless the files were authored according to the guidelines presented in this chapter. If your application uses MIDI files not authored to these guidelines, the application must instruct users to install or create the proper maps using the MIDI Mapper Control Panel applet. For example, if your application uses MIDI files following the General MIDI channel recommendations, your users should create or install a General MIDI map for their synthesizers.

Chapter 5

Low-Level Audio Services

This chapter explains how to use the low-level audio services of Windows to manage playback and recording of waveform and MIDI audio. For an overview of the audio services, see Chapter 3, "Introduction to Audio."

This chapter covers the following topics:

- Using low-level audio services
- Playing waveform audio
- Recording waveform audio
- Playing MIDI audio
- Recording MIDI audio
- Using auxiliary audio devices
- Using audio Clipboard formats

You should have a basic knowledge of digital audio and MIDI before reading this chapter. If you need additional information on these subjects, see "Further Reading" at the end of Chapter 3, "Introduction to Audio."

Function Prefixes

Low-level audio function names begin with the following prefixes:

Prefix	Description
aux	Auxiliary audio functions
midi	MIDI audio functions
wave	Waveform audio functions

Using Low-Level Audio Services

Low-level audio services control different types of audio devices, including waveform, MIDI, and auxiliary audio devices. Many of the concepts of using low-level services apply to more than one type of device. This section presents general information on using low-level audio services. It covers the following topics:

- Querying audio devices
- Opening and closing device drivers
- Allocating and preparing audio data blocks
- Managing audio data blocks
- Using the MMTIME data structure
- Handling errors

Subsequent sections in this chapter discuss using specific types of audio devices.

Querying Audio Devices

Before playing or recording audio, you must determine the capabilities of the audio hardware present in the system. Audio capability can vary from one multimedia computer to the next; applications should not make assumptions about the audio hardware present in a given system.

Getting the Number of Devices

Windows provides the following functions to determine how many devices of a certain type are available in a given system:

auxGetNumDevs	Retrieves the number of auxiliary audio devices present in the 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.
waveInGetNumDevs	Retrieves the number of waveform input devices present in the system.
waveOutGetNumDevs	Retrieves the number of waveform output devices present in the system.

Audio devices are identified by a device identifier (device ID). The device ID is determined implicitly from the number of devices present in a given system. Device IDs range from 0 to 1 less than the number of devices present. For example, if there are 2 MIDI output devices in a system, valid device IDs are 0 and 1.

Getting the Capabilities of a Device

Once you determine how many devices of a certain type 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:

auxGetDevCaps
Retrieves the capabilities of a given auxiliary audio device.

midiInGetDevCaps
Retrieves the capabilities of a given MIDI input device.

midiOutGetDevCaps
Retrieves the capabilities of a given MIDI output device.

waveInGetDevCaps
Retrieves the capabilities of a given waveform input device.

waveOutGetDevCaps
Retrieves the capabilities of a given waveform output device.

Device-Capability Data Structures

Each of these functions takes a far pointer to a data structure the function fills with information on the capabilities of a specified device. The following are the data structures that correspond to each of the device-inquiry functions:

Function	Data Structure
auxGetDevCaps	AUXCAPS
midiInGetDevCaps	MIDIINCAPS
midiOutGetDevCaps	MIDIOUTCAPS
waveInGetDevCaps	WAVEINCAPS
waveOutGetDevCaps	WAVEOUTCAPS

All of the device capabilities data structures have the following fields:

Field	Description
wMid	Specifies a manufacturer ID for the author of the device driver.
wPid	Specifies a product ID for the device.
szPname	Specifies an array of characters containing the name of the device in a null-terminated string.
vDriverVersion	Specifies the version number of the device driver.

Microsoft will assign manufacturer IDs and product IDs specified by the **wMid** and **wPid** fields. The MMSYSTEM.H file contains constants for currently defined IDs.

The **szPname** field points to a null-terminated string containing the product name. You should use the product name to identify devices to users.

The **vDriverVersion** field specifies a version number for the device driver. The high-order of this field is the major version number; the low-order byte is the minor version number.

Opening and Closing Device Drivers

After getting the capabilities of an audio device, you must open the device before you can use it. Audio devices aren't guaranteed to be shareable, so a particular device might not be available when you request it. If this happens, you should notify the user and allow the user to try to open the device again. When you open an audio device, you should close the device as soon as you finish using it.

Windows provides the following functions to open and close different types of audio devices:

midiInOpen	Opens a specified MIDI input device for recording.
midiInClose	Closes a specified MIDI input device.
midiOutOpen	Opens a MIDI output device for playback.
midiOutClose	Closes a specified MIDI output device.
waveInOpen	Opens a waveform input device for recording.
waveInClose	Closes a specified waveform input device.
waveOutOpen	Opens a waveform output device for playback.
waveOutClose	Closes a specified waveform output device.

These functions are discussed in detail later in this chapter.

About Device Handles

Each function that opens an audio device takes as parameters a device ID, a pointer to a memory location, and some parameters 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 distinction between audio-device IDs and audio-device handles is subtle, but very important. Don't confuse the two in your application. The differences between device IDs and device handles are as follows:

- Device IDs are determined implicitly from the number of devices present in a system, which is obtained by using the **...GetNumDevs** functions.
- Device handles are returned when device drivers are opened by using the **...Open** functions.
- The only functions that take device IDs as parameters are the **...GetDevCaps**, **...Open**, and **...Volume** functions. All other functions take device handles.

There are no functions for opening and closing auxiliary audio devices. Auxiliary audio devices don't need to be opened and closed like MIDI and waveform devices because there is no continuous data transfer associated with them. All auxiliary audio functions take device IDs to identify devices.

Allocating and Preparing Audio Data Blocks

Some low-level audio functions require applications to allocate data blocks to pass to the device drivers for playback or recording purposes. Each of these functions uses a data structure (or header) to describe its data block. The following table identifies these functions and their associated header structures (the MMSYSTEM.H file defines the data structures for these headers):

Function	Header	Purpose
waveOutWrite	WAVEHDR	Waveform playback
waveInAddBuffer	WAVEHDR	Waveform recording
midiOutLongMsg	MIDIHDR	MIDI system-exclusive playback
midiInAddBuffer	MIDIHDR	MIDI system-exclusive recording

Before you use one of the functions listed above to pass a data block to a device driver, you must allocate memory for the data block according to the guidelines discussed in the following sections of this chapter.

Allocating Memory for Audio Data Blocks

Before preparing a data block, you must allocate memory for the data block and the header structure that describes the data block.

To allocate memory, use **GlobalAlloc** with the GMEM_MOVEABLE and GMEM_SHARE flags to get a handle to the memory block. Then, pass this handle to **GlobalLock** to get a pointer to the memory block.

To free a data block, use **GlobalUnlock** and **GlobalFree**.

Preparing Audio Data Blocks

Before you pass an audio data block to a device driver, you must prepare the data block by passing it to a **...PrepareHeader** function. 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 an **...UnprepareHeader** function before any allocated memory can be freed.

Windows provides the following functions for preparing and cleaning up audio data blocks:

midiInPrepareHeader
Prepares a MIDI input data block.

midiInUnprepareHeader
Cleans up the preparation on a MIDI input data block.

midiOutPrepareHeader
Prepares a MIDI output data block.

midiOutUnprepareHeader
Cleans up the preparation on a MIDI output data block.

waveInPrepareHeader
Prepares a waveform input data block.

waveInUnprepareHeader
Cleans up the preparation on a waveform input data block.

waveOutPrepareHeader
Prepares a waveform output data block.

waveOutUnprepareHeader
Cleans up the preparation on a waveform output data block.

Managing Audio Data Blocks

Unless the audio 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. This is true for waveform input and output, and for MIDI system-exclusive input messages. Regular MIDI channel messages don't require data blocks for input or output.

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 the application can free the memory associated with the data block and header structure. There are three ways to determine when a device driver is finished with a data block:

- By specifying a window to receive a message sent by the driver when it 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 polling a bit in the **dwFlags** field of the WAVEHDR or MIDIHDR data structure sent with each data block.

If an application doesn't 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.

Note To get time stamped MIDI input data, you must use a callback function.

Using a Window to Process Driver Messages

The easiest type of callback to use to process driver messages is a window callback. 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 **...Open** function. Driver messages will be sent to the window-procedure function for the window identified by the handle in *dwCallback*.

Messages sent to the window function are specific to the audio device type used. For details on these messages, see the sections later in this chapter on using window messages for each specific audio device type.

Using a Callback Function to Process Driver Messages

You can also write your own low-level callback function to process messages sent by the device driver. To use a low-level callback function, specify the CALLBACK_FUNCTION flag in the *dwFlags* parameter and the address of the callback in the *dwCallback* parameter of the **...Open** 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 the sections on using low-level callbacks for each specific audio device type.

Writing Low-Level Callback Functions

Callback functions for the low-level audio services are accessed at interrupt time, and therefore must be carefully written to adhere to the following set of rules:

- The callback function must reside in a dynamic-link library (DLL) and be exported in the module-definition file for the DLL.
- The code and data segments for the callback functions must be specified as FIXED in the module-definition file for the DLL.
- Any data the callback function accesses must be handled in one of the following ways:
  - Declared in the FIXED data segment of the callback DLL.
  - Allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked using **GlobalLock** and **GlobalPageLock**.
  - Allocated with **LocalAlloc** from a FIXED local heap.
- The callback cannot make any Windows calls except **PostMessage**, **timeGetTime**, **timeGetSystemTime**, **timeSetEvent**, **timeKillEvent**, **midiOutShortMsg**, **midiOutLongMsg**, and **OutputDebugStr**.

Note Since low-level callback functions must reside in a DLL, you don't need to use **MakeProcInstance** to get a procedure instance address for the callback.

Passing Instance Data to Callbacks

To pass instance data from an application to a low-level callback residing in a DLL, 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** field of the WAVEHDR and MIDIHDR data structures that identify 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 data structure containing the additional information. Be sure to follow the memory-allocation guidelines listed in "Using a Callback Function to Process Driver Messages," earlier in this chapter.

Managing Data Blocks by Polling

In addition to using a callback, you can poll the **dwFlags** field of a WAVEHDR or MIDIHDR structure to determine when an audio device is finished with a data block. There are times when it's better to poll **dwFlags** rather than waiting for a window to receive messages from the drivers. For example, after you call **waveOutReset** to release pending data blocks, you can immediately poll to be sure that the data blocks are indeed done before proceeding to call **waveOutUnprepareHeader** and free the memory for the data block.

MMSYSTEM.H defines two flags for testing the **dwFlags** field: WHDR_DONE for a WAVEHDR structure, and MHDR_DONE for a MIDIHDR structure. For example, to test a MIDIHDR structure, use the following technique:

```
if(lpMidiHdr->dwFlags & MHDR_DONE)
    /* Driver is finished with the data block */
else
    /* Driver is not finished with the data block */
```

Using the MMTIME Structure

Windows uses a structure called MMTIME to represent time. Low-level audio functions that use MMTIME include **waveInGetPosition** and **waveOutGetPosition**. The **timeGetSystemTime** function also uses MMTIME to represent system time.

The MMTIME structure is defined in the MMSYSTEM.H header file as follows:

```
typedef struct mmtime_tag {
    UINT  wType;                        // Contents of the union
    union {
        DWORD ms;                       // Milliseconds (wType = TIME_MS)
        DWORD sample;                   // Samples (wType = TIME_SAMPLES)
        DWORD cb;                       // Byte count (wType = TIME_BYTES)

        struct {                        // SMPTE (wType = TIME_SMPTE)
            BYTE hour;                  // Hours
            BYTE min;                   // Minutes
            BYTE sec;                   // Seconds
            BYTE frame;                 // Frames
            BYTE fps;                   // Frames per second
            BYTE dummy;                 // Pad byte
        } smpte;

        struct {                        // MIDI (wType = TIME_MIDI)
            DWORD songptrpos;           // Song pointer position
        } midi;
    } u;
} MMTIME;
```

Setting the Time Format

MMTIME can represent time in one or more different formats including milliseconds, samples, SMPTE, and MIDI song-pointer formats. The **wType** field specifies the format used to represent time. Before calling a function that uses the MMTIME structure, you must set the **wType** field to indicate your requested time format. Be sure to check **wType** after the call to see if the requested time format is supported. 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** field is changed to indicate the selected time format. MMSYSTEM.H defines the following flags for the **wType** field of the MMTIME structure:

Flag	Description
TIME_MS	Milliseconds
TIME_SAMPLES	Number of waveform audio samples
TIME_BYTES	Number of waveform audio bytes
TIME_SMPTE	SMPTE time
TIME_MIDI	MIDI song-position pointer

For details on using MMTIME with the **waveOutGetPosition** function, see "Getting the Current Playback Position," later in this chapter.

Getting the System Time

Use the **timeGetSystemTime** or **timeGetTime** functions to get the system time. System time is defined as the time (in milliseconds) since Windows was started. For more information on **timeGetSystemTime** and **timeGetTime**, see Chapter 6, "Timer and Joystick Services."

Handling Errors with Audio Functions

Low-level audio functions return a non-zero error code. The multimedia extensions provide a set of functions that convert these error codes into a textual description of the error. The application must still look at the error value itself to determine how to proceed, but textual descriptions of errors can be used in dialog boxes describing errors to users.

The following functions can be used to get textual descriptions of low-level audio errors:

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

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

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

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

The only low-level audio functions that don't return error codes are the **...GetNumDevs** functions. These functions return a value of 0 if no devices are present in a system, or if any errors are encountered by the function.

Playing Waveform Audio

If your application plays waveform audio, you should use the Media Control Interface (MCI) to control waveform output devices. If the MCI waveform playback services don't meet the needs of your application, you can manage waveform playback by using the low-level waveform services.

Waveform Output Data Types

The MMSYSTEM.H header file defines data types and function prototypes for all of the audio functions. You must include this header file in any source module that uses these functions. MMSYSTEM.H defines the following data types for the waveform output functions.

HWAVEOUT
A handle to an open waveform output device.

WAVEOUTCAPS
A data structure used to inquire about the capabilities of a particular waveform output device.

WAVEFORMAT
A data structure that specifies the data formats supported by a particular waveform output device. This data structure is also used for waveform input devices.

WAVEHDR
A data structure that is a header for a block of waveform output data. This data structure is also used for waveform input devices.

Querying Waveform Output Devices

Before playing a waveform, you should call the **waveOutGetDevCaps** function to determine the waveform output capabilities of the playback device, as described earlier in this chapter. This function takes a pointer to a WAVEOUTCAPS structure, which it fills with information about the capabilities of a given device. This information includes the manufacturer and product IDs, a product name for the device, and the version number of the device driver.

In addition, the WAVEOUTCAPS structure provides information on the standard waveform formats and features supported by the device driver. The MMSYSTEM.H header file defines WAVEOUTCAPS as follows:

```
typedef struct waveoutcaps_tag {
  UINT  wMid;                      /* manufacturer ID */
  UINT  wPid;                      /* product ID */
  VERSION vDriverVersion;          /* driver version */
  char  szPname[MAXPNAMELEN];      /* product name */
  DWORD  dwFormats;                /* supported standard formats */
  UINT  wChannels;                 /* number of channels */
  DWORD  dwSupport;                /* supported features */
} WAVEOUTCAPS;
```

Determining Standard Format Support

The **dwFormats** field of the WAVEOUTCAPS structure specifies the standard waveform formats supported by a device. The MMSYSTEM.H header file defines the following standard waveform format identifiers for the **dwFormats** field:

Format Identifier	Waveform Format
WAVE_FORMAT_1M08	8-bit mono at 11.025 kHz
WAVE_FORMAT_1S08	8-bit stereo at 11.025 kHz
WAVE_FORMAT_1M16	16-bit mono at 11.025 kHz
WAVE_FORMAT_1S16	16-bit stereo at 11.025 kHz
WAVE_FORMAT_2M08	8-bit mono at 22.05 kHz
WAVE_FORMAT_2S08	8-bit stereo at 22.05 kHz
WAVE_FORMAT_2M16	16-bit mono at 22.05 kHz
WAVE_FORMAT_2S16	16-bit stereo at 22.05 kHz
WAVE_FORMAT_4M08	8-bit mono at 44.1 kHz
WAVE_FORMAT_4S08	8-bit stereo at 44.1 kHz
WAVE_FORMAT_4M16	16-bit mono at 44.1 kHz
WAVE_FORMAT_4S16	16-bit stereo at 44.1 kHz

The **dwFormats** field is a logical OR of the flags listed above. For example, to determine if a device supports a waveform format of 16-bit stereo at 44.1 kHz, use this technique:

```
if(waveOutCaps.dwFormats & WAVE_FORMAT_4S16)
    /* Format is supported */
else
    /* Format is not supported */
```

This information on standard-format support also applies to the WAVEINCAPS structure used with waveform input devices. For information on the WAVEINCAPS structure, see "Querying Waveform Input Devices," later in this chapter.

To determine if a specific format is supported by a device (as opposed to all standard formats supported by a device), use the **waveOutOpen** function with the WAVE_FORMAT_QUERY flag as shown in the next section.

Determining Non-Standard Format Support

Waveform devices can support non-standard formats not listed in the preceding table. To see if a particular format (standard or non-standard) is supported by a device, you can call **waveOutOpen** with the WAVE_FORMAT_QUERY flag. The WAVE_FORMAT_QUERY flag tells **waveOutOpen** to check if the requested format is supported. The wave device is not actually opened. The requested format is specified by the structure pointed to by the *lpFormat* parameter passed to **waveOutOpen**. For information about setting up this structure, see "Specifying Waveform Data Formats," later in this chapter. The following code fragment uses this technique to determine if a given waveform device supports a given format:

```
/* Determines if the given waveform output device supports a given wave-
 * form format. Returns 0 if the format is supported, WAVEERR_BADFORMAT
 * if the format is not supported, and one of the MMSYSERR_ error codes if
 * there are other errors encountered in opening the given waveform device.
 */
UINT IsFormatSupported(LPPCMWAVEFORMAT lpPCMWaveFormat, UINT wDeviceID)
{
    return (waveOutOpen(
            NULL,                             // ptr can be NULL for query
            wDeviceID,                        // the device ID
            (LPWAVEFORMAT)lpPCMWaveFormat,    // defines requested format
            NULL,                             // no callback
            NULL,                             // no instance data
            WAVE_FORMAT_QUERY));              // query only, don't open device
}
```

This technique to determine non-standard format support also applies to waveform input devices. The only difference is that the **waveInOpen** function is used in place of **waveOutOpen** to query for format support.

Note To determine if a particular waveform-data format is supported by any of the waveform devices in a system, use the technique illustrated in the previous example, but specify the WAVE_MAPPER constant for the *wDeviceID* parameter. See "Selecting a Waveform Output Device," later in this chapter, for more information on using the WAVE_MAPPER constant.

Determining Capabilities of Waveform Output Devices

Waveform output devices vary in the capabilities they support. The **dwSupport** field of the WAVEOUTCAPS structure indicates whether a given device supports capabilities such as volume and pitch changes. MMSYSTEM.H defines the following flags for the **dwSupport** field:

Flag	Description
WAVECAPS_PITCH	Pitch-change support
WAVECAPS_PLAYBACKRATE	Playback-rate-change support
WAVECAPS_VOLUME	Volume-control support
WAVECAPS_LRVOLUME	Individual volume-control support for both left and right channels

The **dwSupport** field is a logical OR of the flags listed in the preceding table. For example, to determine if a device supports volume changes, use this technique:

```
if(waveOutCaps.dwSupport & WAVECAPS_VOLUME)
    /* Volume changes are supported */
else
    /* Volume changes are not supported */
```

For more information on playback volume levels, see "Changing Waveform Playback Volume," later in this chapter. For more information on pitch and playback rates, see "Changing Pitch and Playback Rate," also later in this chapter.

Opening Waveform Output Devices

Use **waveOutOpen** to open a waveform output device for playback. This function opens the device associated with the specified device ID and returns a handle to the open device by writing the handle to a specified memory location. The syntax of the **waveOutOpen** function is as follows:

UINT waveOutOpen(*lphWaveOut, wDeviceID, lpFormat, dwCallback, dwInstance, dwFlags*)

The *lphWaveOut* parameter is an LPHWAVEOUT and specifies a far pointer to a memory location the function fills with a handle to the open waveform output device. Use this handle to identify the waveform device when calling other waveform-output functions.

The *wDeviceID* parameter is a UINT and identifies the waveform output device to open. See "Querying Audio Devices," earlier in this chapter, for details about device IDs. If you specify the WAVE_MAPPER constant, the function finds a waveform output device capable of playing the given format and attempts to open it.

The *lpFormat* parameter is an LPWAVEFORMAT and specifies a far pointer to a WAVEFORMAT data structure. This data structure contains information on the format of the waveform data that will be sent to the waveform device. The following section explains how to use this data structure. You can free the WAVEFORMAT data structure immediately after passing it to the **waveOutOpen** function.

The *dwCallback* parameter is a DWORD and specifies either a window handle or the address of a low-level callback function. The callback can be used to monitor the progress of the playback of waveform data so an application can determine when to send additional data blocks or when to free data blocks that have been sent. You must specify the appropriate flag in the *dwFlags* parameter to indicate which type of callback you want. If no callback is desired, this parameter is NULL.

The *dwInstance* parameter is a DWORD and specifies instance data sent to the callback function each time it is called.

The *dwFlags* parameter is a DWORD and specifies one or more flags for opening a waveform device. Use the WAVE_FORMAT_QUERY flag to specify that you don't want to actually open a device, but just query whether the device supports the specified format. For information on using WAVE_FORMAT_QUERY, see "Determining Non-Standard Format Support," earlier in this chapter. If you specify a window or low-level callback in the *dwCallback* parameter, you must specify either the CALLBACK_WINDOW or the CALLBACK_FUNCTION flag to indicate the type of callback you are using.

Selecting a Waveform Output Device

Some multimedia computers have multiple waveform output devices. Unless you know you want to open a specific waveform output device in a system, you should use the WAVE_MAPPER constant for the device ID when you open a device. The **waveOutOpen** function chooses the device in the system best capable of playing the given data format.

Specifying Waveform Data Formats

When you call **waveOutOpen** to open a device driver for playback or to query if the driver supports a particular data format, use the *lpFormat* parameter to specify a pointer to a structure containing the requested waveform data format.

The WAVEFORMAT Structure

The WAVEFORMAT structure specifies format information common to all types of waveform data formats. Currently, the only format supported is PCM, but in the future, other types such as ADPCM might be supported. The MMSYSTEM.H file defines the WAVEFORMAT structure as follows:

```
typedef struct waveformat_tag {
    UINT    wFormatTag;        /* format type */
    UINT    nChannels;         /* number of channels */
    DWORD   nSamplesPerSec;    /* number of samples per second */
    DWORD   nAvgBytesPerSec;   /* average data rate */
    UINT    nBlockAlign;       /* block alignment */
} WAVEFORMAT;
```

The **wFormatTag** field specifies the format type for the data. Currently, the only flag defined for this field is WAVE_FORMAT_PCM for PCM waveform data.

The **nChannels** field specifies the number of discrete channels in the format. Use a value of 1 for mono data and 2 for stereo data.

The **nSamplesPerSec** field specifies the sample rate.

The **nAvgBytesPerSec** field specifies the average data rate in bytes per second. For example, 16-bit stereo at 44.1 kHz has an average data rate of 176400 bytes per second (2 channels × 2 bytes per sample per channel × 44100 samples per second).

The **nBlockAlign** field specifies the minimum atomic unit of data that can be passed to a driver. 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).

The PCMWAVEFORMAT Structure

In addition to the general information in the WAVEFORMAT structure, specific information is needed to describe a PCM waveform data format completely. For PCM waveform data, the PCMWAVEFORMAT structure includes a WAVEFORMAT structure along with an additional field containing PCM-specific information as follows.

```
typedef struct pcmwaveformat_tag {
    WAVEFORMAT  wf;                 /* general format information */
    UINT        wBitsPerSample;     /* number of bits per sample */
} PCMWAVEFORMAT;
```

The **wf** field specifies general format information. The **wBitsPerSample** field specifies the number of bits per sample for PCM data.

Using the PCMWAVEFORMAT Structure

For PCM audio data, use the PCMWAVEFORMAT structure to specify the data format. The following code fragment shows how to set up a PCMWAVEFORMAT structure for 11.025 kHz 8-bit mono and for 44.1 kHz 16-bit stereo. After setting up PCMWAVEFORMAT, the example calls the **IsFormatSupported** function to verify that the PCM waveform output device supports the format. The source for **IsFormatSupported** is given in an example in "Determining Non-Standard Format Support," earlier in this chapter.

```
UINT wReturn;
PCMWAVEFORMAT pcmWaveFormat;

/* Set up PCMWAVEFORMAT for 11 kHz 8-bit mono
 */
pcmWaveFormat.wf.wFormatTag = WAVE_FORMAT_PCM;
pcmWaveFormat.wf.nChannels = 1;
pcmWaveFormat.wf.nSamplesPerSec = 11025L;
pcmWaveFormat.wf.nAvgBytesPerSec = 11025L;
pcmWaveFormat.wf.nBlockAlign = 1;
pcmWaveFormat.wBitsPerSample = 8;

/* See if format is supported by any device in system
 */
wReturn = IsFormatSupported(&pcmWaveFormat, WAVE_MAPPER);

/* Report results
 */
if (wReturn == 0)
     MessageBox(hMainWnd, "11 kHz 8-bit mono is supported.",
       "", MB_ICONINFORMATION);
else if (wReturn == WAVERR_BADFORMAT)
     MessageBox(hMainWnd, "11 kHz 8-bit mono is NOT supported.",
       "", MB_ICONINFORMATION);
else
     MessageBox(hMainWnd, "Error opening waveform device.",
       "Error", MB_ICONEXCLAMATION);


/* Set up PCMWAVEFORMAT for 44.1 kHz 16-bit stereo
 */
pcmWaveFormat.wf.wFormatTag = WAVE_FORMAT_PCM;
pcmWaveFormat.wf.nChannels = 2;
pcmWaveFormat.wf.nSamplesPerSec = 44100L;
pcmWaveFormat.wf.nAvgBytesPerSec = 176400L;
pcmWaveFormat.wf.nBlockAlign = 4;
pcmWaveFormat.wBitsPerSample = 32;
```

```
/* See if format is supported by any device in the system
 */
wReturn = IsFormatSupported(&pcmWaveFormat, WAVE_MAPPER);

/* Report results
 */
If (wReturn == 0)
    MessageBox(hMainWnd, "44.1 kHz 16-bit stereo is supported.",
      "", MB_ICONINFORMATION);
else if (wReturn == WAVERR_BADFORMAT)
    MessageBox(hMainWnd, "44.1 kHz 16-bit stereo is NOT supported.",
      "", MB_ICONINFORMATION);
else
    MessageBox(hMainWnd, "Error opening waveform device.",
      "Error", MB_ICONEXCLAMATION);
```

Getting Format Information from a WAVE File

The easiest way to get waveform-format information from a WAVE file is by using the multimedia file I/O services. To do this, use **mmioDescend** to locate the “fmt ” chunk containing the format information and then **mmioRead** to read the format chunk directly into the proper format structure (chunks are the basic building blocks of RIFF files). The following code fragment illustrates this technique. For more information about multimedia file I/O, see Chapter 7, “Multimedia File I/O Services.”

```
void ReversePlay()
{
    HMMIO        hmmio;
    MMCKINFO     mmckinfoParent;
    MMCKINFO     mmckinfoSubchunk;
    DWORD        dwFmtSize;
    char         szFileName[ MAX_FILENAME_SIZE ];
    HANDLE       hFormat;
    WAVEFORMAT   *pFormat;
    .
    .
    .

    /* Open the given file for reading using buffered I/O.
     */
    ...

    /* Locate a "RIFF" chunk with a "WAVE" form type
     * to make sure it's a WAVE file.
     */
    ...
```

```
/* Now, 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))
{
    MessageBox(hwndApp, "WAVE file is corrupted.",
               NULL, MB_OK | MB_ICONEXCLAMATION);
    mmioClose(hmmio, 0);
    return;
}

/* Get the size of the format chunk, allocate and lock memory for it.
 */
dwFmtSize = mmckinfoSubchunk.cksize;
hFormat = LocalAlloc(LMEM_MOVEABLE, LOWORD(dwFmtSize));
if (!hFormat)
{
    MessageBox(hwndApp, "Out of memory.",
               NULL, MB_OK | MB_ICONEXCLAMATION);
    mmioClose(hmmio, 0);
    return;
}
pFormat = (WAVEFORMAT *) LocalLock(hFormat);
if (!pFormat)
{
    MessageBox(hwndApp, "Failed to lock memory for format chunk.",
               NULL, MB_OK | MB_ICONEXCLAMATION);
    LocalFree(hFormat);
    mmioClose(hmmio, 0);
    return;
}
```

```
        /* Read the format chunk.
         */
        if (mmioRead(hmmio, pFormat, dwFmtSize) != dwFmtSize)
        {
            MessageBox(hwndApp, "Failed to read format chunk.",
                       NULL, MB_OK | MB_ICONEXCLAMATION);
            LocalUnlock(hFormat);
            LocalFree(hFormat);
            mmioClose(hmmio, 0);
            return;
        }

        /* Make sure it's a PCM file.
         */
        if (pFormat->wFormatTag != WAVE_FORMAT_PCM)
        {
            LocalUnlock(hFormat);
            LocalFree(hFormat);
            mmioClose(hmmio, 0);
            MessageBox(hwndApp, "The file is not a PCM file.",
                       NULL, MB_OK | MB_ICONEXCLAMATION);
            return;
        }

        /* Make sure the system has a waveform output
         * device capable of playing this format.
         */
        if (waveOutOpen(&hWaveOut, WAVE_MAPPER, (LPWAVEFORMAT)pFormat, NULL,
                        0L, WAVE_FORMAT_QUERY))
        {
            LocalUnlock(hFormat);
            LocalFree(hFormat);
            mmioClose(hmmio, 0);
            MessageBox(hwndApp, "The waveform device can't play this format.",
                       NULL, MB_OK | MB_ICONEXCLAMATION);
            return;
        }
        .
        .
        .
    }
```

Writing Waveform Data

After successfully opening a waveform output device driver, you can begin waveform playback. Windows provides the following function for sending data blocks to waveform output devices:

waveOutWrite
Writes a data block to a waveform output device.

Use the WAVEHDR data structure to specify the waveform 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 assorted flags. The MMSYSTEM.H file defines the WAVEHDR data structure as follows:

```
typedef struct wavehdr_tag {
    LPSTR                   lpData;            /* pointer to data block */
    DWORD                   dwBufferLength;    /* length of data block */
    DWORD                   dwBytesRecorded;   /* number of bytes recorded */
    DWORD                   dwUser;            /* user instance data */
    DWORD                   dwFlags;           /* assorted flags */
    DWORD                   dwLoops;           /* loop control counter */
    struct wavehdr_tag far  *lpNext;           /* private to driver */
    DWORD                   reserved;          /* private to driver */
} WAVEHDR;
```

After you send a data block to an output device 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 details on different techniques for monitoring data block completion, see “Managing Audio Data Blocks,” earlier in this chapter.

Example of Writing Waveform Data

The following code fragment illustrates the steps required to allocate and set up a WAVEHDR data structure, and write a block of data to a waveform output device.

```
/* Global variables--Must be visible to window-procedure function so it
 * can unlock and free the data block after it has been played.
 */
HANDLE      hData       = NULL;         // handle to waveform data memory
HPSTR       lpData      = NULL;         // pointer to waveform data memory

void ReversePlay()
{
    HWAVEOUT    hWaveOut;
    HWAVEHDR    hWaveHdr;
    LPWAVEHDR   lpWaveHdr;
    HMMIO       hmmio;
    MMCKINFO    mmckinfoParent;
    MMCKINFO    mmckinfoSubchunk;
    UINT        wResult;
    HANDLE      hFormat;
    WAVEFORMAT  *pFormat;
    DWORD       dwDataSize;
        .
    .
    .

    /* Open a waveform device for output using window callback.
     */
    if (waveOutOpen((LPHWAVEOUT)&hWaveOut, WAVE_MAPPER,
                    (LPWAVEFORMAT)pFormat,
                    (LONG)hwndApp, 0L, CALLBACK_WINDOW))
    {
        MessageBox(hwndApp, "Failed to open waveform output device.",
                   NULL, MB_OK | MB_ICONEXCLAMATION);
        LocalUnlock(hFormat);
        LocalFree(hFormat);
        mmioClose(hmmio, 0);
        return;
    }
```

```
/* Allocate and lock memory for the waveform data. The memory for
 * waveform data must be globally allocated with GMEM_MOVEABLE and
 * GMEM_SHARE flags.
 */
hData = GlobalAlloc(GMEM_MOVEABLE | GMEM_SHARE, dwDataSize );
if (!hData)
{
    MessageBox(hwndApp, "Out of memory.",
               NULL, MB_OK | MB_ICONEXCLAMATION);
    mmioClose(hmmio, 0);
    return;
}
lpData = GlobalLock(hData);
if (!lpData)
{
    MessageBox(hwndApp, "Failed to lock memory for data chunk.",
               NULL, MB_OK | MB_ICONEXCLAMATION);
    GlobalFree(hData);
    mmioClose(hmmio, 0);
    return;
}

/* Read the waveform data subchunk.
 */
if(mmioRead(hmmio, (HPSTR) lpData, dwDataSize) != dwDataSize)
{
    MessageBox(hwndApp, "Failed to read data chunk.",
               NULL, MB_OK | MB_ICONEXCLAMATION);
    GlobalUnlock(hData);
    GlobalFree(hData);
    mmioClose(hmmio, 0);
    return;
}

/* Allocate and lock memory for the header. This memory must also be
 * globally allocated with GMEM_MOVEABLE and GMEM_SHARE flags.
 */
hWaveHdr = GlobalAlloc(GMEM_MOVEABLE | GMEM_SHARE,
                    (DWORD) sizeof(WAVEHDR));
if (!hWaveHdr)
{
    GlobalUnlock(hData);
    GlobalFree(hData);
    MessageBox(hwndApp, "Not enough memory for header.",
               NULL, MB_OK | MB_ICONEXCLAMATION);
    return;
}
```

```
        lpWaveHdr = (LPWAVEHDR) GlobalLock(hWaveHdr);
        if (!lpWaveHdr)
        {
            GlobalUnlock(hData);
            GlobalFree(hData);
            MessageBox(hwndApp, "Failed to lock memory for header.",
                       NULL, MB_OK | MB_ICONEXCLAMATION);
            return;
        }

        /* After allocation, the header must be set up and prepared for use.
         */
        lpWaveHdr->lpData = lpData;
        lpWaveHdr->dwBufferLength = dwDataSize;
        lpWaveHdr->dwFlags = 0L;
        lpWaveHdr->dwLoops = 0L;
        waveOutPrepareHeader(hWaveOut, lpWaveHdr, sizeof(WAVEHDR));

        /* Then the data block can be sent to the output device. The
         * waveOutWrite function returns immediately and waveform data
         * is sent to the output device in the background.
         */
        wResult = waveOutWrite(hWaveOut, lpWaveHdr, sizeof(WAVEHDR));
        if (wResult != 0)
        {
            waveOutUnprepareHeader(hWaveOut, lpWaveHdr, sizeof(WAVEHDR));
            GlobalUnlock( hData);
            GlobalFree(hData);
            MessageBox(hwndApp, "Failed to write block to device",
                       NULL, MB_OK | MB_ICONEXCLAMATION);
            return;
        }
        .
        .
        .
    }
```

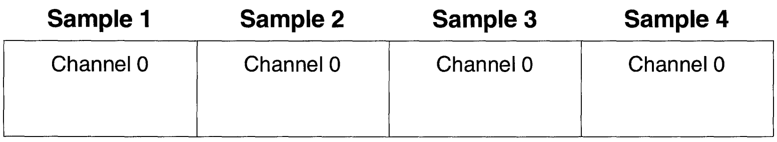
PCM Waveform Data Format

The **lpData** field in the WAVEHDR structure points to the waveform 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 data.

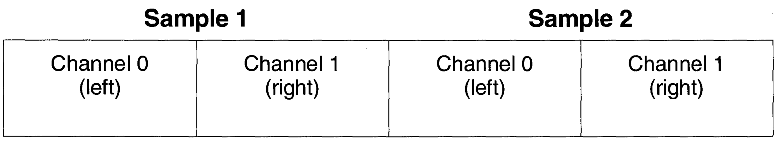
Data Format	Maximum Value	Minimum Value	Midpoint Value
8-bit PCM	255 (0xFF)	0	128 (0x80)
16-bit PCM	32767 (0x7FFF)	–32768 (0x8000)	0

PCM Data Packing

The order of the data bytes varies between 8-bit and 16-bit, and mono and stereo formats. The following illustrations show data packing for the first four words of different PCM waveform data formats:



Data packing for 8-bit mono PCM.



Data packing for 8-bit stereo PCM.

Sample 1		Sample 2	
Channel 0 Low-Order Byte	Channel 0 High-Order Byte	Channel 0 Low-Order Byte	Channel 0 High-Order Byte

Data packing for 16-bit mono PCM.

Sample 1			
Channel 0 (left) Low-Order Byte	Channel 0 (left) High-Order Byte	Channel 1 (right) Low-Order Byte	Channel 1 (right) High-Order Byte

Data packing for 16-bit stereo PCM.

Using Window Messages to Manage Waveform Playback

The following messages can be sent to a window-procedure function for managing waveform playback:

Message	Description
MM_WOM_CLOSE	Sent when the device is closed using **waveOutClose**.
MM_WOM_DONE	Sent when the device driver is finished with a data block sent using **waveOutWrite**.
MM_WOM_OPEN	Sent when the device is opened using **waveOutOpen**.

A *wParam* and *lParam* parameter is associated with each of these messages. The *wParam* parameter always specifies a handle to the open waveform device. For the MM_WOM_DONE message, *lParam* specifies a far pointer to a WAVEHDR structure identifying the completed data block. The *lParam* parameter is unused for the MM_WOM_CLOSE and MM_WOM_OPEN messages.

The most useful message is 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 don't need to process the MM_WOM_OPEN and MM_WOM_CLOSE messages.

Example of Processing MM_WOM_DONE

The following code fragment shows how to process the MM_WOM_DONE message. This fragment 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 function.
 */
LRESULT FAR PASCAL WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM
lParam)
{
    switch (msg)
    {
        .
        .
        .

        case MM_WOM_DONE:
            /* A waveform data block has been played and can now be freed.
             */
            waveOutUnprepareHeader((HWAVEOUT) wParam,
                                    (LPWAVEHDR) lParam, sizeof(WAVEHDR) );
            GlobalUnlock(hData);
            GlobalFree(hData);
            waveOutClose((HWAVEOUT) wParam);

            break;
    }

    return DefWindowProc(hWnd,msg,wParam,lParam);
}
```

Using a Low-Level Callback to Manage Waveform Playback

The syntax of the low-level callback function for waveform output devices is as follows:

void FAR PASCAL waveOutCallback(*hWaveOut, wMsg, dwInstance, dwParam1, dwParam2*)

The following messages can be sent to the *wMsg* parameter of waveform output callback functions:

Message	Description
WOM_CLOSE	Sent when the device is closed using **waveOutClose**.
WOM_OPEN	Sent when the device is opened using **waveOutOpen**.
WOM_DONE	Sent when the device driver is finished with a data block sent using **waveOutWrite**.

These messages are similar to the messages sent to window-procedure functions, however, the parameters are different. A handle to the open waveform device is passed as a parameter to the callback, along with the DWORD of instance data that was passed using **waveOutOpen**.

Message-Dependent Parameters

The callback has two message-dependent parameters: *dwParam1* and *dwParam2*. For the WOM_DONE message, *dwParam1* specifies a far pointer to a WAVEHDR structure identifying the completed data block and *dwParam2* is not used. For the WOM_OPEN and WOM_CLOSE messages, neither of the parameters are used.

After the driver is finished with a data block, you can clean up and free the data block, as described in “Allocating and Preparing Audio Data Blocks,” earlier in this chapter. Because of the restrictions of low-level audio callbacks, you can’t do this within the callback—you must do this work outside of the callback.

Getting the Current Playback Position

While a waveform is playing, you can monitor the current playback position within the waveform. Windows provides the following function for this purpose:

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

This function takes three parameters: a handle to a waveform output device, a far pointer to an MMTIME structure, and a UINT specifying the size of the MMTIME structure.

For waveform devices, the preferred time format to represent the current position is in samples. Thus, the current position of a waveform device is specified as the number of samples for one channel from the beginning of the waveform.

To query the current position of a waveform device, set the **wType** field of the MMTIME structure to the constant TIME_SAMPLES and pass this structure to **waveOutGetPosition**.

Stopping, Pausing, and Restarting Playback

While a waveform is playing, you can stop or pause playback. Once playback has been paused, you can restart it. Windows provides the following functions for controlling waveform playback:

waveOutPause
Pauses playback on a waveform output device.

waveOutReset
Stops playback on a waveform output device and marks all pending data blocks as done.

waveOutRestart
Resumes playback on a paused waveform output device.

Use **waveOutPause** to pause a waveform device that is currently playing. To resume playback on a paused device, use **waveOutRestart**. These functions take a single parameter: the waveform output device handle returned by **waveOutOpen**. Pausing a waveform device might not be instantaneous—the driver can finish playing the current block before pausing playback.

Generally, as soon as the first waveform data block is sent using **waveOutWrite**, the waveform device begins playing. If you don't want the waveform to start playing immediately, call **waveOutPause** before calling **waveOutWrite**. Then, when you want to begin playing the waveform, call **waveOutRestart**.

To stop a waveform from playing, use **waveOutReset**. This function differs from **waveOutPause** in that it also marks all pending data blocks as being done. You can't restart a device that has been stopped with **waveOutReset** by using **waveOutRestart**—you must use **waveOutWrite** to send the first data block to resume playback on the device.

Closing Waveform Output Devices

After waveform playback is complete, call **waveOutClose** to close the waveform device. If **waveOutClose** is called while a waveform is playing, the close operation will fail and the function returns an error code indicating that the device was not closed. If you don't want to wait for playback to end before closing the device, call **waveOutReset** before closing. This terminates playback and allows the device to be closed. Be sure to clean up the preparation on all data blocks before closing the waveform output device.

Changing Waveform Playback Volume

Windows provides the following functions to query and set the volume level of waveform output devices:

waveOutGetVolume	Gets the volume level of the specified waveform device.
waveOutSetVolume	Sets the volume level of the specified waveform device.

Not all waveform devices support volume changes. Some devices support individual volume control on both the left and right channels. See "Determining Capabilities of Waveform Output Devices," earlier in this chapter, for information on how to determine the volume-control capabilities of waveform devices.

Note 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 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 DWORD; 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, as shown in the following illustration:

HIWORD	LOWORD
Right-Channel Volume	Left- or Mono-Channel Volume

DWORD packing for waveform volume levels.

For devices that don't 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.

When querying with **waveOutGetVolume**, the volume is returned in a DWORD location specified by a far pointer parameter.

Changing Pitch and Playback Rate

Some waveform output devices can vary the pitch and the playback rate of waveform data. Not all waveform devices support pitch and playback-rate changes. See "Determining Capabilities of Waveform Output Devices," earlier in this chapter, for information on how to determine if a particular waveform device supports pitch and playback rate changes.

The differences between changing pitch and playback rate are:

- 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 pitch and playback rates:

waveOutGetPitch
Gets the pitch for the specified waveform output device.

waveOutGetPlaybackRate
Gets the playback rate for the specified waveform output device.

waveOutSetPitch
Sets the pitch for the specified waveform output device.

waveOutSetPlaybackRate
Sets the playback rate for the specified waveform output device.

Specifying Pitch and Playback Rate

The pitch and playback rates are changed by a factor specified with a fixed-point number packed into a DWORD. The upper 16 bits specify the integer part; the lower 16 bits specify the fractional part of the number. 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-recording services don't meet the needs of your application, you can handle waveform recording using the low-level waveform services.

Waveform Input Data Types

The MMSYSTEM.H file defines data types and function prototypes for all of the audio functions. You must include this header file in any source module that uses these functions. MMSYSTEM.H defines the following data types for waveform-input functions:

HWAVEIN
A handle to an open waveform input device.

WAVEINCAPS
A data structure used to inquire about the capabilities of a particular waveform input device.

WAVEFORMAT
A data structure that specifies the data formats supported by a particular waveform input device. This data structure is also used for waveform output devices.

WAVEHDR
A data structure that is a header for a block of waveform input data. This data structure is also used for waveform output devices.

Querying Waveform Input Devices

Before recording a waveform, you should call the **waveInGetDevCaps** function to determine the waveform input capabilities of the system. This function takes a pointer to a WAVEINCAPS structure, which it fills with information on the capabilities of a given device. This information includes the manufacturer and product IDs, a product name for the device, and the version number of the device driver. In addition, the WAVEINCAPS structure provides information on the standard waveform formats that the device supports. MMSYSTEM.H defines WAVEINCAPS as follows.

```
typedef struct waveincaps_tag {
  UINT  wMid;                       /* manufacturer ID */
  UINT  wPid;                       /* product ID */
  VERSION vDriverVersion;           /* driver version */
  char  szPname[MAXPNAMELEN];       /* product name */
  DWORD  dwFormats;                 /* supported standard formats */
  UINT  wChannels;                  /* number of channels */
} WAVEINCAPS;
```

Opening Waveform Input Devices

Use **waveInOpen** to open a waveform input device for recording. This function opens the device associated with the specified device ID and returns a handle to the open device by writing the handle to a specified memory location. The syntax of the **waveInOpen** function is as follows:

UINT waveInOpen(*lphWaveIn, wDeviceID, lpFormat, dwCallback, dwInstance, dwFlags*)

The *lphWaveIn* parameter is an LPHWAVEIN and specifies a far pointer to a memory location the function fills with a handle to the open waveform input device. Use this handle to identify the device when calling other waveform input functions.

The *wDeviceID* parameter is a UINT and identifies the waveform input device to be opened. See "Querying Audio Devices," earlier in this chapter, for details about device IDs. If you specify the WAVE_MAPPER constant, the function will find a waveform input device capable of recording in the given format and attempt to open it.

The *lpFormat* parameter is an LPWAVEFORMAT and specifies a far pointer to a WAVEFORMAT data structure. This data structure contains information on the format of the recorded waveform data that will be sent back to the application. For details on using this data structure, see "Specifying Waveform Data Formats," earlier in this chapter. You can free the WAVEFORMAT data structure immediately after passing it to **waveInOpen**.

The *dwCallback* parameter is a DWORD and specifies either a window handle or the address of a low-level callback function. The callback can be used to monitor the progress of waveform recording so an application can determine when data blocks have been filled with waveform data and when to send additional data blocks for recording. You must specify the appropriate flag in the *dwFlags* parameter to indicate which type of callback you want. If no callback is needed, this parameter is NULL.

The *dwInstance* parameter is a DWORD and specifies 32 bits of instance data sent to the callback function each time it is called.

The *dwFlags* parameter is a DWORD and specifies one or more flags for opening a waveform device. Use the WAVE_FORMAT_QUERY flag to specify that you don't want to actually open a device, but just query whether the device supports a given format. For information on using WAVE_FORMAT_QUERY, see "Determining Non-Standard Format Support," earlier in this chapter. If you are specifying a window or low-level callback in the *dwCallback* parameter, you must specify either the CALLBACK_WINDOW or the CALLBACK_FUNCTION flag to indicate the type of callback used.

Selecting a Waveform Input Device

Some multimedia computers will have multiple waveform input devices. Unless you know you want to open a specific waveform input device in a system, you should use the WAVE_MAPPER constant for the device ID when you open a device. The **waveOutOpen** function will choose the device in the system best able to record in the given data format.

Managing Waveform Recording

Once you open a waveform input device, you can begin recording waveform data. Waveform data is recorded into application-supplied buffers specified by a WAVEHDR data structure. This is the same data structure used for waveform playback described in "Writing Waveform Data," earlier in this chapter. Memory for the WAVEHDR structure and its accompanying data buffer must be allocated and prepared, as shown in "Allocating and Preparing Audio Data Blocks," earlier in this chapter.

Windows provides the following functions to manage waveform recording:

waveInAddBuffer
Sends a buffer to the device driver so it can be filled with recorded waveform data.

waveInReset
Stops waveform recording and marks all pending buffers as done.

waveInStart
Starts waveform recording.

waveInStop
Stops waveform recording.

Use the **waveInAddBuffer** function to send data buffers to the device driver. As the buffers are filled with recorded waveform data, the application is notified with either a window message or with a callback message, depending on the flag specified when the device was opened.

Use the **waveInStart** function to begin recording. Before beginning recording, you should send at least one buffer to the driver, or incoming data might be lost. To stop waveform recording, use **waveInStop**.

Before closing the device using **waveOutClose**, call **waveOutReset** to mark any pending data blocks as being done.

Using Window Messages to Manage Waveform Recording

The following messages can be sent to a window procedure function for managing waveform recording:

Message	Description
MM_WIM_CLOSE	Sent when the device is closed using **waveInClose**.
MM_WIM_DATA	Sent when the device driver is finished with a data buffer sent using **waveInAddBuffer**.
MM_WIM_OPEN	Sent when the device is opened using **waveInOpen**.

There is a *wParam* and *lParam* parameter associated with each of these messages. The *wParam* parameter always specifies a handle to the open waveform device. The *lParam* parameter is unused for the MM_WIM_CLOSE and MM_WIM_OPEN messages.

For the MM_WIM_DATA message, *lParam* specifies a far pointer to a WAVEHDR structure that identifies the data buffer. This data buffer might not be completely filled with waveform data—recording can stop before the buffer is filled. Use the **dwBytesRecorded** field of the WAVEHDR structure to determine the amount of valid data present in the buffer.

The most useful message is MM_WIM_DATA. Unless you need to allocate memory or initialize variables, you probably don't need to use the MM_WIM_OPEN and MM_WIM_CLOSE messages. When the device driver is finished with a data block, you can clean up and free the data block as described in "Allocating and Preparing Audio Data Blocks," earlier in this chapter.

Using a Low-Level Callback to Manage Waveform Recording

This syntax of the low-level callback function for waveform input devices is as follows:

void FAR PASCAL waveInCallback(*hWaveIn, wMsg, dwInstance, dwParam1, dwParam2*)

The following messages can be sent to the *wMsg* parameter of waveform input callback functions:

Message	**Description**
WIM_CLOSE	Sent when the device is closed using **waveInClose**.
WIM_OPEN	Sent when the device is opened using **waveInOpen**.
WIM_DONE	Sent when the device driver is finished with a data block sent using **waveInAddBuffer**.

These messages resemble messages sent to window-procedure functions, but their parameters are different. A handle to an open waveform device is passed as a parameter to the callback function, along with the DWORD of instance data that was passed using **waveInOpen**.

Message-Dependent Parameters The callback has two message-dependent parameters: *dwParam1* and *dwParam2*. For the WIM_CLOSE and WIM_OPEN messages, these parameters are not used. For the WIM_DONE message, *dwParam1* specifies a far pointer to a WAVEHDR structure identifying the completed data block and *dwParam2* is not used.

After the driver is finished with a data block, you can clean up and free the data block. Because of the restrictions of low-level audio callback functions, you can't do this within the callback. You must set some semaphores and do this outside of the callback. See "Using a Callback Function to Process Driver Messages," earlier in this chapter, for details on the restrictions on using callback functions.

Playing MIDI Audio

To play MIDI files, you should use the MCI sequencer. If the MCI sequencer services don't meet the needs of your application, you can manage MIDI playback using the low-level MIDI services.

MIDI Output Data Types

The MMSYSTEM.H header file defines data types and function prototypes for all low-level audio functions. You must include this header file in any source module that uses these functions. MMSYSTEM.H defines the following data types for low-level MIDI output functions:

HMIDIOUT
A handle to a MIDI output device.

MIDIHDR
A data structure that is a header for a block of MIDI system-exclusive data. This data structure is used for input as well as output.

MIDIOUTCAPS
A data structure used to inquire about the capabilities of a particular MIDI output device.

Querying MIDI Output Devices

Before playing MIDI audio, you should call the **midiOutGetDevCaps** function to determine the capabilities of the MIDI output hardware present in the system. This function takes a pointer to a MIDIOUTCAPS structure that it fills with information on the capabilities of a given device. This information includes the manufacturer and product IDs, a product name for the device, and the version number of the device driver. In addition, the MIDIOUTCAPS structure provides information on the device technology, the number of voices and notes supported, the MIDI channels that the device responds to, and features supported by the driver.

The MMSYSTEM.H file defines the MIDIOUTCAPS structure as follows:

```
typedef struct midioutcaps_tag {
  UINT  wMid;                    /* manufacturer ID */
  UINT  wPid;                    /* product ID */
  VERSION vDriverVersion;        /* driver version */
  char  szPname[MAXPNAMELEN];    /* product name */
  UINT  wTechnology;             /* device technology */
  UINT  wVoices;                 /* total simultaneous instruments */
  UINT  wNotes;                  /* total simultaneous notes */
  UINT  wChannelMask;            /* channels device responds to */
  DWORD  dwSupport;              /* features supported */
} MIDIOUTCAPS;
```

Determining the Technology of the Device

MIDI output devices can be either internal synthesizers or external MIDI output ports. The **wTechnology** field specifies the technology of the device. MMSYSTEM.H defines the following flags to identify device technology:

Flag	Description
MOD_MIDIPORT	The device is an external MIDI output port.
MOD_SQSYNTH	The device is an internal square-wave synthesizer.
MOD_FMSYNTH	The device is an internal FM synthesizer.
MOD_SYNTH	The device is an internal synthesizer (generic).
MOD_MAPPER	The device is the MIDI Mapper.

Determining Capabilities of Internal Synthesizers

If the device is an internal synthesizer, additional device information is available in the **wVoices**, **wNotes**, and **wChannelMask** fields. If the device is an external output port, these fields are unused.

The **wVoices** field specifies the number of voices the device supports. Each voice can have a different sound or timbre. Voices are differentiated by MIDI channel. For example, a four-voice synthesizer uses four MIDI channels. The **wNotes** field specifies the *polyphony* of the device—the maximum number of notes that can be played simultaneously. The **wChannelMask** field 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.

The **dwSupport** field of the MIDIOUTCAPS structure indicates if the device driver supports volume changes and patch caching. MMSYSTEM.H defines the following flags for the **dwSupport** field:

Flag	Description
MIDICAPS_VOLUME	Indicates the driver supports volume control.
MIDICAPS_LRVOLUME	Indicates the driver supports individual volume control for the left and right channels.
MIDICAPS_CACHE	Indicates the driver supports patch caching.

Volume changes are only supported by internal synthesizer devices. External MIDI output ports don't support volume changes. For information on changing volume, see "Changing Internal MIDI Synthesizer Volume," later in this chapter.

Opening MIDI Output Devices

Use the **midiOutOpen** function to open a MIDI output device for playback. This function opens the device associated with the specified device ID and returns a handle to the open device by writing the handle to a specified memory location. The syntax of **midiOutOpen** is as follows:

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

The *lphMidiOut* parameter is an LPHMIDIOUT and specifies a far pointer to a memory location the function fills with a handle to the open MIDI output device.

Use this handle to identify the MIDI device when calling other MIDI output functions.

The *wDeviceID* parameter is a UINT that identifies the MIDI output device to be opened. See "Querying Audio Devices," earlier in this chapter, for details on device IDs.

The *dwCallback* parameter is a DWORD that specifies either a window handle or the address of a low-level callback function. The callback can be used to monitor the progress of the playback of MIDI system-exclusive data so the application can determine when to send additional data blocks, or when to free data blocks that have been sent. You must specify the appropriate flag in the *dwFlags* parameter to indicate which type of callback you want. If no callback is needed, this parameter is NULL.

The *dwInstance* parameter is a DWORD that specifies 32 bits of instance data sent to the callback function each time it is called.

The *dwFlags* parameter is a DWORD and specifies one or more flags for opening the MIDI device. If you are specifying a window or low-level callback in the *dwCallback* parameter, you must specify either the CALLBACK_WINDOW or the CALLBACK_FUNCTION flag to indicate the type of callback used.

Sending MIDI Messages

Once you open a MIDI output device, you can begin sending it MIDI messages using the following functions:

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

midiOutShortMsg
Sends a MIDI message to a specified MIDI output device.

midiOutReset
Turns off all notes on all channels for a specified MIDI output device. Any pending system-exclusive buffers are marked as done and returned to the application.

Use **midiOutShortMsg** to send any MIDI message (except for system-exclusive messages). This function takes an HMIDIOUT parameter specifying the MIDI output device to send the message to, and a DWORD for the MIDI message. The message is packed into the DWORD, as shown in the following illustration:

HIWORD		LOWORD	
HIBYTE (not used)	**LOBYTE** MIDI Data 2 (optional)	**HIBYTE** MIDI Data 1 (optional)	**LOBYTE** MIDI Status

DWORD packing for the midiOutShortMsg function.

The two MIDI data bytes are optional, depending on the MIDI status byte. The following code fragment uses **midiOutShortMsg** to send a given MIDI event to a given MIDI output device:

```
/* Sends a given MIDI event to the given output device
 */
UINT sendMIDIEvent(hMidiOut, bStatus, bData1, bData2)
HMIDIOUT hMidiOut;  // handle to the output device
BYTE bStatus;       // MIDI status byte
BYTE bData1;        // first MIDI data byte
BYTE bData2;        // second MIDI data byte
{
    union {
        DWORD dwData;
        BYTE bData[4];
    } u;

    /* Construct the MIDI message */
    u.bData[0] = bStatus;
    u.bData[1] = bData1;
    u.bData[2] = bData2;
    u.bData[3] = 0;

    /* Send the message */
    return midiOutShortMsg(hMidiOut, u.dwData);
}
```

Note MIDI output drivers are not required to verify data before sending it to an output port. It is up to applications to ensure only valid data is sent using **midiOutShortMsg**.

Sending Buffered Messages

MIDI system-exclusive messages are the only MIDI messages that will not fit into a single DWORD. System exclusive messages can be any length. Windows provides the **midiOutLongMsg** function for sending one or more messages, including system exclusive messages, to MIDI output devices.

Use the MIDIHDR data structure to specify MIDI system-exclusive data blocks. This structure contains a pointer to a locked data block, the data-block length, and some assorted flags. The MMSYSTEM.H file defines the MIDIHDR data structure as follows:

```
typedef struct midihdr_tag {
  LPSTR            lpData;              /* pointer to data block */
  DWORD            dwBufferLength;      /* length of data block */
  DWORD            dwBytesRecorded;     /* number of bytes recorded */
  DWORD            dwUser;              /* user instance data */
  DWORD            dwFlags;             /* assorted flags */
  struct wavehdr_tag far *lpNext;       /* private to the driver */
  DWORD            reserved;            /* private to the driver */
} MIDIHDR;
```

Memory for the MIDIHDR data structure and the data block pointed to by **lpData** must be allocated and prepared, as shown in "Allocating and Preparing Audio Data Blocks," earlier in this chapter.

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 you know when to send additional blocks. For information on different techniques for monitoring data-block completion, see "Managing Audio 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 Window Messages to Manage Buffered Playback

The following messages can be sent to a window-procedure function for managing MIDI system-exclusive playback:

Message	Description
MM_MOM_CLOSE	Sent when the device is closed using **midiOutClose**.
MM_MOM_DONE	Sent when the device driver is finished with a data block sent using **midiOutLongMsg**.
MM_MOM_OPEN	Sent when the device is opened using **midiOutOpen**.

A *wParam* and *lParam* parameter is associated with each of these messages. The *wParam* parameter always specifies a handle to the open MIDI device. For the MM_MOM_DONE message, *lParam* specifies a far pointer to a MIDIHDR structure identifying the completed data block. The *lParam* parameter is unused for the MM_MOM_CLOSE and MM_MOM_OPEN messages.

The most useful message is the MM_MOM_DONE message. Unless you need to allocate memory or initialize variables, you probably don't need to process the MM_MOM_OPEN and MM_MOM_CLOSE messages. When playback of a data block is completed, you can clean up and free the data block as described in "Allocating and Preparing Audio Data Blocks," earlier in this chapter.

Using a Callback to Manage Buffered Playback

This syntax of the low-level callback function for MIDI output devices is as follows:

void FAR PASCAL midiOutCallback(*hMidiOut, wMsg, dwInstance, dwParam1, dwParam2*)

The following messages can be sent to the *wMsg* parameter of MIDI output callback functions:

Message	Description
MOM_CLOSE	Sent when the device is closed using **midiOutClose**.
MOM_OPEN	Sent when the device is opened using **midiOutOpen**.
MOM_DONE	Sent when the device driver is finished with a data block sent using **midiOutLongMsg**.

These messages are similar to those sent to window-procedure functions, but the parameters are different. A handle to the open MIDI device is passed as a parameter to the callback, along with the DWORD of instance data passed using **midiOutOpen**.

Message-Dependent Parameters

The callback has two message-dependent parameters: *dwParam1* and *dwParam2*. For the MOM_OPEN and MOM_CLOSE messages, these parameters are not used. For the MOM_DONE message, *dwParam1* specifies a far pointer to a MIDIHDR structure identifying the completed data block and *dwParam2* is not used.

After the driver is finished with a data block, you can clean up and free the data block. Because of the restrictions of low-level audio callbacks, you can't do this within the callback. See "Using a Callback Function to Process Driver Messages," earlier in this chapter, for details on the restrictions when using callback functions.

Sending MIDI Messages Using Running-Status

The MIDI 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. You can send MIDI messages using running status with the **midiOutShortMsg** function by packing the message into a DWORD, as shown in the following illustration:

HIWORD		LOWORD	
HIBYTE (not used)	LOBYTE (not used)	HIBYTE MIDI Data 2 (optional)	LOBYTE MIDI Data 1

DWORD packing for midiOutShortMsg when using running status.

Resetting MIDI Output

The **midiOutReset** function will turn off all notes and mark any pending system-exclusive buffers as done and return them to the application. It may be useful in an application using 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 may 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 query and set the volume level of internal MIDI synthesizer devices:

midiOutGetVolume	Gets 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 the right channels. See "Determining Capabilities of Internal Synthesizers," earlier in this chapter, for information on how to determine if a particular device supports volume changes.

Note 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 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 DWORD; 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, as shown in the following illustration.

HIWORD	LOWORD
Right-Channel Volume	Left- or Mono-Channel Volume

DWORD packing for internal MIDI synthesizer volume levels.

For devices that don't 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.

When querying using **midiOutGetVolume**, the volume is returned in a DWORD location specified by a far pointer parameter.

Preloading Patches with Internal MIDI Synthesizers

Some internal MIDI synthesizer devices can't 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.

The PATCHARRAY Data Type

The **midiOutCachePatches** function takes a pointer to a PATCHARRAY to indicate the patches to be cached. The MMSYSTEM.H file defines the PATCHARRAY data type as follows:

```
typedef WORD PATCHARRAY[MIDIPATCHSIZE];
```

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, set element 0 of the array to 0x0101:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1

The KEYARRAY Data Type

The **midiOutCacheDrumPatches** function takes a pointer to a KEYARRAY to indicate the key-based percussion patches to be cached. The MMSYSTEM.H file defines the KEYARRAY data type as follows:

```
typedef WORD KEYARRAY[MIDIPATCHSIZE];
```

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, set element 60 of the array to 0x8200:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

Using the MIDI Mapper with Low-Level MIDI Functions

The MIDI Mapper provides standard patch services for device-independent playback of MIDI files. Applications that use MIDI files for audio should use the MIDI Mapper. For information on the MIDI Mapper, see “The MIDI Mapper” in Chapter 4, “High-Level Audio Services.” For information on authoring device-independent MIDI files, see “Authoring MIDI Files,” also in Chapter 4.

To use the MIDI mapper, open it using the **midiOutOpen** function with the *wDeviceID* parameter set to the constant MIDIMAPPER. Then you can send it MIDI messages using **midiOutShortMsg** or **midiOutLongMsg**.

Recording MIDI Audio

To record MIDI audio data, you must use low-level MIDI input functions. MCI does not currently provide a device handler for recording MIDI audio.

MIDI Input Data Types

The MMSYSTEM.H file defines data types and function prototypes for all of the low-level audio functions. MMSYSTEM.H defines the following data types for low-level MIDI input functions:

HMIDIIN
A handle to a MIDI input device.

MIDIHDR
A data structure that is a header for a block of MIDI system-exclusive data. This data structure is used for input as well as output.

MIDIINCAPS
A data structure used to inquire about the capabilities of a MIDI input device.

Querying MIDI Input Devices

Before recording MIDI audio, you should call the **midiInGetDevCaps** function to determine the capabilities of the MIDI input hardware present in the system. This function takes a pointer to a MIDIINCAPS structure, which it fills with information about the capabilities of a given device. This information includes the manufacturer and product IDs, a product name for the device, and the version number of the device driver. MMSYSTEM.H defines the MIDIINCAPS structure as follows:

```
typedef struct midiincaps_tag {
  UINT  wMid;                 /* manufacturer ID */
  UINT  wPid;                 /* product ID */
  VERSION vDriverVersion;     /* driver version */
  char  szPname[MAXPNAMELEN];  /* product name */
} MIDIINCAPS;
```

Opening MIDI Input Devices

Use the **midiInOpen** function to open a MIDI input device for recording. This function opens the device associated with the specified device ID and returns a handle to the open device by writing the handle to a specified memory location. The syntax of **midiInOpen** is as follows:

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

The *lphMidiIn* parameter is an LPHMIDIIN and specifies a far pointer to a memory location the function fills with a handle to the open MIDI input device. Use this handle to identify the MIDI device when calling other MIDI input functions.

The *wDeviceID* parameter is a UINT that identifies the MIDI input device to be opened. See "Querying Audio Devices," earlier in this chapter, for details on device IDs.

The *dwCallback* parameter is a DWORD and specifies either a window handle or the address of a low-level callback function. You must specify the appropriate flag in the *dwFlags* parameter to indicate which type of callback you want.

The *dwInstance* parameter is a DWORD that specifies 32 bits of instance data sent to the callback function each time it is called.

The *dwFlags* parameter is a DWORD and specifies one or more flags for opening the MIDI device. You must specify either the CALLBACK_WINDOW or the CALLBACK_FUNCTION flag to indicate the type of callback you are using.

Managing MIDI Recording

Once you open a waveform input 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 it can be filled with recorded 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.

Use the **midiInAddBuffer** function to send data buffers to the device driver for recording system-exclusive messages. As the buffers are filled with recorded data, the application is notified by one of the techniques discussed in "Managing Audio Data Blocks," earlier in this chapter.

Use the **midiInStart** function to begin recording. To record system-exclusive messages, send at least one buffer to the driver before starting recording. To stop recording, use **midiInStop**. Before closing the device using **midiInClose**, call **midiInReset** to mark any pending data blocks as being done.

You must use either a window-procedure function or a low-level callback function to receive MIDI data. If you want time-stamped data, you must use a low-level callback function.

To record system-exclusive messages, you must supply the device driver with data buffers. These buffers are specified by a MIDIHDR data structure. This is the same data structure used for MIDI buffered playback described in "Sending Buffered Messages," earlier in this chapter. Memory for the MIDIHDR structure and its accompanying data buffer must be allocated and prepared as shown in "Allocating and Preparing Audio Data Blocks," earlier in this chapter.

Using Window Messages to Manage MIDI Recording

The following messages can be sent to a window-procedure function for managing MIDI recording:

Message	Description
MM_MIM_CLOSE	Sent when the device is closed using **midiInClose**.
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 data buffer sent using **midiInAddBuffer** is filled with system-exclusive data.
MM_MIM_LONGERROR	Sent when an invalid MIDI system-exclusive message is received.
MM_MIM_OPEN	Sent when the device is opened using **midiInOpen**.

A *wParam* and *lParam* parameter is associated with each of these messages. The *wParam* parameter always specifies a handle to the open MIDI device. The *lParam* parameter is unused for the MM_MIM_CLOSE and MM_MIM_OPEN messages.

Receiving Regular MIDI Data

For the MM_MIM_DATA message, *lParam* specifies the received MIDI data. This data is packed into a DWORD, as shown in the following illustration:

HIWORD		LOWORD	
HIBYTE (Unused)	**LOBYTE** MIDI Data 2 (optional)	**HIBYTE** MIDI Data 1 (optional)	**LOBYTE** MIDI Status

DWORD packing for recorded MIDI data.

Receiving System-Exclusive MIDI Data

For the MM_MIM_LONGDATA message, *lParam* specifies a far pointer to a MIDIHDR structure that identifies the data buffer for system-exclusive messages. The data buffer might not be completely filled—you usually don't know the size of the system-exclusive messages before recording them and must allocate a buffer large enough for the largest expected message. Use the **dwBytesRecorded** field of the MIDIHDR structure to determine the amount of valid data present in the buffer.

Using a Low-Level Callback to Manage MIDI Recording

This syntax of the low-level callback function for MIDI input devices is as follows:

void FAR PASCAL midiInCallback(*hMidiIn, wMsg, dwInstance, dwParam1, dwParam2*)

The following messages can be sent to the *wMsg* parameter of MIDI input callback functions:

Message	Description
MIM_CLOSE	Sent when the device is closed using **midiInClose**.
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_LONGERROR	Sent when an invalid MIDI system-exclusive message is received.
MIM_LONGDATA	Sent when either a complete MIDI system-exclusive message is received, or when a data buffer is filled with system-exclusive data.
MIM_OPEN	Sent when the device is opened using midiInOpen.

These messages are similar to those sent to window-procedure functions, but the parameters are different. A handle to the open MIDI device is passed as a parameter to the callback, along with the DWORD of instance data that was passed using **midiInOpen**.

The callback has two message-dependent parameters: *dwParam1* and *dwParam2*. For the MIM_OPEN and MIM_CLOSE messages, these parameters are unused.

For the MIM_DATA message, *dwParam1* specifies the received MIDI data and *dwParam2* specifies a time stamp for the data. The data is packed into a DWORD, as shown in the previous section on using window messages.

For the MIM_LONGDATA message, *dwParam1* specifies a far pointer to a MIDIHDR structure that identifies the data buffer for system-exclusive messages. As with the MIM_DATA message, *dwParam2* specifies a time stamp for the data. The data buffer might not be completely filled. Use the **dwBytesRecorded** field of the MIDIHDR structure to determine the amount of valid data present in the buffer.

After the device driver is finished with a data block, you can clean up and free the data block. Because of the restrictions of low-level audio callbacks, you can't do this within the callback. You must set some semaphores and do this outside of the callback.

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 the message was received. MIDI time stamps are defined as the time the first byte of the message was received and are specified in milliseconds. The **midiInStart** function resets the time stamps for a device to zero.

As stated earlier, to receive time stamps with MIDI input, you must use a low-level callback function. The *dwParam2* parameter of the callback function specifies the time stamp for data associated with the MIM_DATA and MIM_LONGDATA messages.

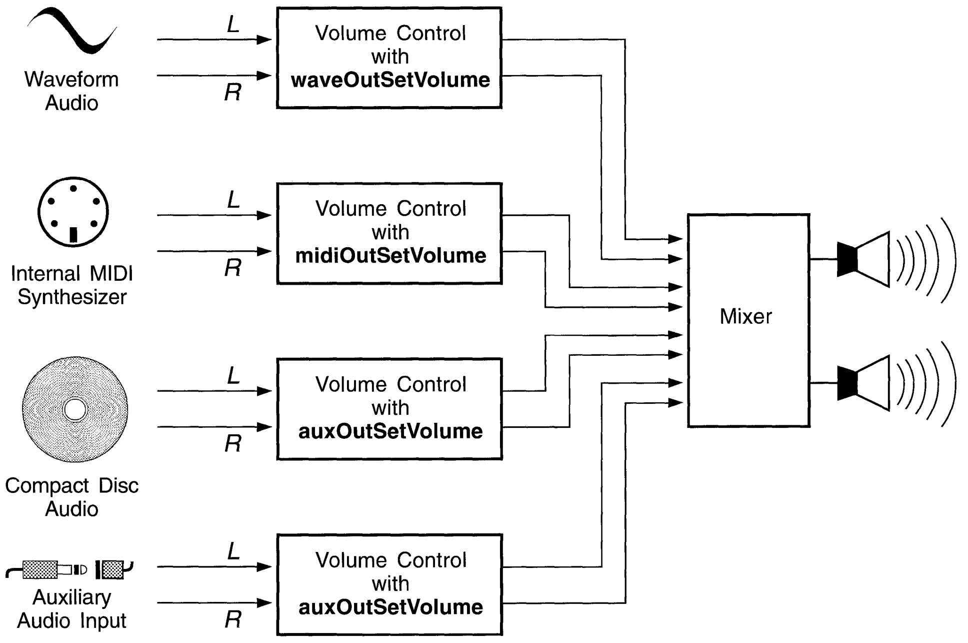
Receiving Running-Status Messages

The MIDI 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 to complete messages—you always receive complete MIDI messages from a MIDI input device driver.

Auxiliary Audio Devices

Auxiliary audio devices are audio devices whose output is mixed with the MIDI and waveform output devices in a multimedia computer. An example of an auxiliary audio device is the compact disc audio output from a CD-ROM drive.

Control for auxiliary audio devices is provided by a software-controlled audio mixer. The mixer can reside on the motherboard of a multimedia computer, or it can be on an add-in sound card. The following illustration shows the conceptual audio-signal routing in a multimedia computer:


Waveform Audio
L
R
Volume Control with waveOutSetVolume
Internal MIDI Synthesizer
L
R
Volume Control with midiOutSetVolume
Compact Disc Audio
L
R
Volume Control with auxOutSetVolume
Auxiliary Audio Input
L
R
Volume Control with auxOutSetVolume
Mixer

Audio signal routing for a multimedia computer.

In this multimedia computer, there are two auxiliary audio devices: the CD-ROM and the external audio input. The external audio input is an input jack that allows the user to connect other audio devices.

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 on a particular auxiliary audio device, use the **auxGetDevCaps** function. This function takes a pointer to an AUXCAPS structure, which it fills with information on the capabilities of a given device. This information includes the manufacturer and product IDs, a product name for the device, and the device-driver version number. For information on these fields, see "Getting the Capabilities of a Device," earlier in this chapter. The AUXCAPS structure also contains information on the device type and the functionality the device supports. The MMSYSTEM.H file defines the AUXCAPS structure as follows:

```
typedef struct auxcaps_tag {
    UINT    wMid;                       /* manufacturer ID */
    UINT    wPid;                       /* product ID */
    VERSION vDriverVersion;             /* driver version */
    char    szPname[MAXPNAMELEN];       /* product name */
    UINT    wTechnology;                /* device type */
    DWORD   dwSupport;                  /* functionality supported */
} AUXCAPS;
```

Determining Auxiliary Audio Device Types

The MMSYSTEM.H file defines the following flags for the **wTechnology** field of the AUXCAPS structure to determine the device type of an auxiliary audio device:

Flag	Description
AUXCAPS_CDAUDIO	The device is an internal CD-ROM drive.
AUXCAPS_AUXIN	The device is an auxiliary audio input jack.

Determining Capabilities of Auxiliary Audio Devices

The **dwSupport** field of the AUXCAPS structure indicates whether the device driver supports volume changes. The MMSYSTEM.H file defines the following flags for the **dwSupport** field:

Flag	Description
AUXCAPS_VOLUME	Volume-control support.
AUXCAPS_LRVOLUME	Individual volume-control support for both the left and right channels.

Changing Auxiliary Audio-Device Volume

Windows provides the following functions to query and set the volume for auxiliary audio devices:

auxGetVolume
Gets the volume level of the specified auxiliary audio device.

auxSetVolume
Sets the volume level of the specified auxiliary audio device.

Not all auxiliary audio devices support volume changes. Some devices can support individual volume changes on both the left and the right channels. See "Determining Capabilities of Auxiliary Audio Devices," earlier in this chapter, for information on how to determine if a particular device supports volume changes.

Note 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 query the volume level before changing it and restore the volume level to its previous level as soon as possible.

The volume level is specified as in the waveform and MIDI volume-control functions: in a DWORD with the upper 16 bits specifying the relative volume of the right channel and the lower 16 bits specifying the relative volume of the left channel. For devices that don't support individual volume control 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. 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.

When querying with **auxGetVolume**, the volume is returned in a DWORD location specified by a far pointer parameter.

Audio Clipboard Formats

In addition to the standard and nonstandard waveform formats previously presented in this chapter, there are two Clipboard formats that can be used 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 wave files.

Chapter 6
Timer and Joystick Services

This chapter shows you how to add joystick input capabilities to your application and how to control event timing. Timer interrupt services provide improved timer resolution with up to one-millisecond accuracy. The joystick services can receive control signals from up to two joysticks.

This chapter includes the following main topics:

- Timer interrupt services
- Joystick services

Functions and data structures associated with the joystick and timer-interrupt services are defined in MMSYSTEM.H and MMSYSTEM.LIB.

Function Prefixes

The names of functions discussed in this chapter begin with the following prefixes:

Prefix	**Description**
time	Timer-interrupt functions
joy	Joystick functions

Timer Services

The multimedia timer services provided with Windows 3.1 let applications schedule timed periodic or one-time interrupt events at a higher resolution than was available through the Windows 3.0 timer services.

Unlike the timer services provided by **SetTimer**, the multimedia timer services are interrupt-based; rather than post WM_TIMER messages to a message queue, they call a callback function at interrupt time. Because the callback code is accessed at interrupt time, it must adhere to strict programming guidelines. Also, high-resolution, periodic interrupt events require significant processor time. This can drastically affect the performance of your application and any other application running at the same time.

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 one-millisecond accuracy rate. For less-demanding synchronization tasks, use the Windows **SetTimer** function.

Timer Data Types

The MMSYSTEM.H file defines new data types and function prototypes for timer functions. You must include this header file in any source module that uses timer services. MMSYSTEM.H defines the following new data types:

MMTIME
A data structure for representing time in one of several formats.

TIMECAPS
A data structure for querying timer capabilities.

Using Timer Services

Timer services let an application request and receive timer messages at application-specified intervals. Real-time multimedia applications can use the following functions to control the pace of data and synchronized presentations:

timeBeginPeriod	Establishes the minimum timer resolution an application will use.
timeEndPeriod	Clears a minimum timer resolution previously set using **timeBeginPeriod**.
timeGetDevCaps	Returns information about the capabilities of the timer services.
timeGetTime	Returns the system time in milliseconds.
timeGetSystemTime	Fills an MMTIME structure with the system time in milliseconds.
timeSetEvent	Creates a timer event that executes a specific action at a specific time or at periodic intervals.
timeKillEvent	Cancels a timer event previously created using **timeSetEvent**.

Getting the System Time

An application can get the current system time using the **timeGetTime** or **timeGetSystemTime** functions. The system time is the count of milliseconds since Windows was started. The **timeGetTime** function returns the system time, and the **timeGetSystemTime** function fills an MMTIME structure with the system time.

The **timeGetTime** function has the following syntax:

DWORD timeGetTime()

The **timeGetSystemTime** function has the following syntax:

UINT timeGetSystemTime(*lpMMTime, wSize*)

The *lpMMTime* parameter is a far pointer to an MMTIME structure. The *wSize* parameter specifies the size of the MMTIME structure.

The **timeGetSystemTime** function returns TIMERR_NOERROR if successful.

Determining Maximum and Minimum Event Periods

You can use the **timeGetDevCaps** function to determine the minimum and maximum timer-event periods provided by the timer services. These values vary across computers and can vary depending on the current Windows mode. The **timeGetDevCaps** function has the following syntax:

UINT timeGetDevCaps(*lpCaps, wSize*)

The *lpCaps* parameter is a far pointer to a TIMECAPS structure. The second parameter, *wSize*, specifies the size of the TIMECAPS structure. The TIMECAPS structure has the following format:

```
typedef struct timecaps_tag {
    UINT    wPeriodMin;
    UINT    wPeriodMax;
 } TIMECAPS;
```

The two fields in this structure specify, in milliseconds, the minimum and maximum period (and resolution) supported.

Establishing Minimum Timer Resolution

Before starting timer events, your application must establish the minimum timer resolution that it intends to use. It must clear this value after finishing with the timer-event services.

Use the **timeBeginPeriod** and **timeEndPeriod** functions to set and clear the minimum timer-event resolution for your application. You must match a call to **timeBeginPeriod** with a corresponding 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**.

The **timeBeginPeriod** function has the following syntax:

void timeBeginPeriod(*wMinRes*)

The **timeEndPeriod** function has the following syntax:

void timeEndPeriod(*wMinRes*)

In both functions, the *wMinRes* parameter indicates the minimum timer resolution in milliseconds. You can specify any resolution value within the range of resolution values supported by the timer. The **wPeriodMin** and **wPeriodMax** fields of the TIMECAPS structure (filled by the **timeGetDevCaps** function) specify the minimum and maximum resolution supported by the timer services.

Starting Timer Events

To initialize and start timer events, use the **timeSetEvent** function. This function returns a timer ID that can be used to stop or identify timer events. The **timeSetEvent** function has the following syntax:

UINT timeSetEvent(*wDelay, wResolution, lpFunction, dwUser, wFlags*)

The *wDelay* parameter specifies the period, in milliseconds, for timer events. If this value is less than the minimum timer period or greater than the maximum period, **timeSetEvent** fails.

The *wResolution* value establishes the accuracy of the timer event. The accuracy of the timer event can increase with smaller *wResolution* values. For example, on a one-time event with a *wResolution* value of 5 and a *wDelay* value of 100, the timer services notify your callback function after an interval ranging from 95 to 105 milliseconds. The application must have called **timeBeginPeriod** to specify a minimum resolution of 5 milliseconds.

Larger *wResolution* values provide flexibility to reduce the number of timer interrupts, which can seriously affect system performance. To reduce system overhead, use the maximum *wResolution* value appropriate for your application. To ensure that periodic events occur at specified intervals, use a resolution of zero.

Pass the name of the callback function in the *lpFunction* parameter, and pass any instance data in the *dwUser* parameter. The callback function must reside in a DLL, so you don't need to call **MakeProcInstance** to get the procedure-instance address of the callback function.

The *wFlags* parameter takes one of the following flags:

Flag	Description
TIME_ONESHOT	Event should occur once, after *wPeriod* elapses
TIME_PERIODIC	Event should occur repeatedly, waiting *wPeriod* between each event

The **timeSetEvent** function returns a timer ID if successful or NULL if unsuccessful. Interrupt timers are a scarce resource, and periodic timers with resolution less than 100 milliseconds consume a significant portion of CPU time. For periodic timers, you must pair calls to **timeSetEvent** with calls to **timeKillEvent**. For more information, see the following section, "Canceling a Timer Event."

The *lpFunction* parameter contains the procedure-instance address of the function to be called when the timer event takes place.

Since the callback function is accessed at interrupt time, it must adhere to strict programming guidelines. Timer-callback functions follow the same programming guidelines as callback functions for the low-level audio services. See "Using a Callback Function to Process Driver Messages" in Chapter 5, "Low-Level Audio Services," for information on writing an interrupt callback function.

The timer-event callback function must have the following syntax:

```
void FAR PASCAL TimerCallback(
    UINT idTimer,                       // Timer ID
    UINT msg,                           // Not used
    DWORD dwUser,                       // User-instance data
    DWORD dw1,                          // Not used
    DWORD dw2 )                         // Not used
```

The *idTimer* parameter receives the timer ID, and the *dwUser* parameter receives the user-instance data passed to the **timeSetEvent** function. The *msg*, *dw1*, and *dw2* parameters are not used.

Canceling a Timer Event

You can cancel an active timer event at any time. Be sure to cancel any outstanding timers before freeing the DLL containing the callback function. To cancel a timer event, use the **timeKillEvent** function, which has the following syntax:

UINT timeKillEvent(*wTimerID*)

Pass the timer ID returned by **timeSetEvent** to the *wTimerID* parameter.

Using Timer Callbacks

This section describes how an application might use the timer services. First, the application calls the **timeGetDevCaps** function to determine the minimum and maximum resolution supported by the timer services. Before setting up any timer events, the application uses **timeBeginPeriod** to establish the minimum timer resolution it will use, as shown in the following code fragment:

```
#define TARGET_RESOLUTION 1         // Try for 1-millisecond accuracy

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);
```

To start the timer event, the application specifies the amount of time before the callback occurs, the required resolution, the address of the callback function, and user data to supply with the callback. The application might use a function like the following to start a one-time 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 );                      // Event type (one-time)

    if(! npSeq->wTimerID)
        return ERR_TIMER;
    else
        return ERR_NOERROR;
}
```

The following callback function resides in a fixed code segment in a DLL. It is limited to calling those functions that are interrupt-callable. The TimerIntRoutine procedure it calls also resides in a fixed code segment.

```
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, since no longer in use

    TimerIntRoutine(npSeq);   // Handle interrupt-time tasks
}
```

Before freeing the DLL that contains the callback function, the application cancels any outstanding timers. To cancel one timer event, it might call the following function:

```
void DestroyTimer(NPSEQ npSeq)
{
    if(npSeq->wTimerID)                     // If timer event is pending
    {
        timeKillEvent(npSeq->wTimerID);     // Cancel the event
        npSeq->wTimerID = 0;
    }
}
```

Finally, to cancel the minimum timer resolution it established, the application calls **timeEndPeriod** as follows:

```
timeEndPeriod(wTimerRes);
```

Joystick Services

The joystick is an input device that provides position information. It is an additional supported input device, not a replacement for the mouse. All absolute-position devices, including touch screens, digitizing tablets, and light pens, can use the joystick services to provide position and button information to applications.

The joystick services are loaded when Windows is started. The joystick services can monitor two joysticks, each with two- or three-axis movement, and up to four buttons. Applications access the joystick services through the set of functions described in this section.

Note The driver for the IBM Game Adapter supports two 2-axis joysticks or one 3-axis joystick.

Joystick Data Types

The MMSYSTEM.H file defines new data types and function prototypes for joystick functions. MMSYSTEM.H defines the following new data types:

JOYCAPS
A data structure that contains joystick capability information.

JOYINFO
A data structure that contains joystick position and button information.

Using Joystick Services

Joystick services include functions to query each joystick for its capabilities, to poll each joystick for position and button information, and to receive messages in response to joystick events. Your application can use the following joystick functions to accept input from one or two joysticks:

joyGetNumDevs
Returns the number of joysticks supported by the joystick services.

joyGetDevCaps
Returns joystick capabilities.

joyGetPos
Returns joystick position and button information.

joySetCapture
Causes joystick input to be sent to a specified window at regular intervals or when the joystick state changes.

joyReleaseCapture
Releases the joystick captured using **joySetCapture**.

joyGetThreshold
Returns the movement threshold of a joystick.

joySetThreshold
Sets the movement threshold of a joystick.

Determining Joystick Capabilities

The various joysticks in use today can support two or three axes and a variety of button configurations. Also, joysticks support different ranges of motion and polling frequencies. Joystick drivers can support either one or two joysticks. Two functions allow you to determine the capabilities of the joystick services and joystick devices installed on a system.

Getting the Driver Capabilities

You can use the **joyGetNumDevs** function to determine the number of joystick devices supported by the joystick services. This function has the following syntax:

UINT joyGetNumDevs()

This function returns the number of supported joysticks, or zero if there is no joystick support. The value returned is not necessarily the number of joysticks attached to the system. To determine whether a joystick is attached, call the **joyGetPos** function for the device. The **joyGetPos** function, discussed in "Polling the Joystick," later in this chapter, returns JOYERR_UNPLUGGED if the specified device is disconnected.

The following code fragment determines whether the joystick services are available and then determines 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;
```

Getting the Joystick Capabilities

You can use the **joyGetDevCaps** function to obtain the specific capabilities of each joystick attached to a given system. The **joyGetDevCaps** function has the following syntax:

UINT joyGetDevCaps(*wJoyID, lpJoyCaps, wSize*)

The *wJoyID* parameter identifies the joystick as either JOYSTICKID1 or JOYSTICKID2. The *lpJoyCaps* parameter points to a JOYCAPS structure to be filled by the function. The *wSize* parameter specifies the size of the JOYCAPS structure.

JOYCAPS Structure The JOYCAPS structure specifies the range of each axis on the joystick, the number of buttons, and the maximum and minimum polling frequency. This structure has the following fields:

Field	Description
wMid	Manufacturer identification
wPid	Product identification
szPname[MAXPNAMELEN]	Product name in a null-terminated string
wXmin, wXmax	Minimum and maximum *x*-position values
wYmin, wYmax	Minimum and maximum *y*-position values
wZmin, wZmax	Minimum and maximum *z*-position values
wNumButtons	Number of buttons
wPeriodMin	Minimum period between messages
wPeriodMax	Maximum period between messages

Methods for Checking Joystick Status

An application can receive information from the joystick in one of two ways:

- By processing joystick messages from a captured joystick
- By polling the joystick directly

The message-processing method can be simpler to use; your application is sent messages that indicate the position of the stick and the state of the buttons (pressed or released).

Capturing Joystick Messages to a Window Function

You can capture joystick input to a window function; your application then receives joystick messages at specified intervals or when the user manipulates the joystick. The messages are described in "Processing Joystick Messages," following this section.

Only one application can capture joystick messages from a given joystick. Capturing joystick messages does not, however, prevent your application (or other applications) from polling the joystick using **joyGetPos**. If **joyGetPos** is called while joystick input is captured, joystick events occurring close to the time of the **joyGetPos** call might not be accurately reported to the capture window.

Capturing Joystick Input

The **joySetCapture** function captures joystick input to a window function you specify. To release the joystick, call the **joyReleaseCapture** function. The **joySetCapture** function has the following syntax:

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

Specify the handle of the window to receive the messages in the *hWnd* parameter. For *wJoyID*, specify which joystick to capture; use the constants JOYSTICKID1 or JOYSTICKID2. The *wPeriod* parameter specifies the frequency, in milliseconds, of the joystick messages, and the *bChanged* parameter specifies whether messages are to be sent only when the stick position or button states change. The joystick messages are described in the next section.

To capture messages from two joysticks attached to the system, you must call **joySetCapture** twice, once for each joystick. Your window then receives separate messages for each device.

You cannot capture an unplugged joystick. The **joySetCapture** function returns zero if successful; it returns JOYERR_UNPLUGGED if the specified device is unplugged.

Note The joystick services set up a Windows timer event with each call to **joySetCapture**.

Specifying the Resolution and Threshold

Assign the *wPeriod* parameter a value that falls within the minimum and maximum resolution range for the joystick. To determine the minimum and maximum resolution of the joystick, call the **joyGetDevCaps** function, which fills the **wPeriodMin** and **wPeriodMax** fields in the **JOYCAPS** structure.

If the *wPeriod* value is outside the range of valid resolution values for the joystick, the joystick services use the minimum or maximum resolution value, whichever is closer to the *wPeriod* value.

The *bChanged* parameter controls when the window receives joystick movement messages. If *bChanged* is set to FALSE, these messages occur approximately every *wPeriod* milliseconds, regardless of whether the position has changed since the last time the joystick was polled. If *bChanged* is set to TRUE, messages are sent when the position of a joystick axis changes by a value greater than the movement threshold of the device. To change the movement threshold, use the **joySetThreshold** function, discussed in "Setting the Movement Threshold," later in this chapter.

Processing Joystick Messages

The following joystick messages can be sent to a window function. Numerals 1 and 2 in these messages correspond to the joystick initiating the message. MM_JOY1 messages are sent to the window function if your application requests input from the first joystick, and MM_JOY2 messages are sent if your application requests input from the second joystick. All messages report nonexistent buttons as released.

Messages	Description
MM_JOY1MOVE MM_JOY2MOVE	Report a change in the *x*-axis and/or *y*-axis position of the joystick. The *wParam* parameter contains a combination of JOY_BUTTON bit flags specifying which buttons were pressed. The low-order word of *lParam* contains the *x*-position of the joystick, and the high-order word contains the *y*-position.
MM_JOY1ZMOVE MM_JOY2ZMOVE	Report a change in the *z*-axis position of a 3-axis joystick. The *wParam* parameter contains a combination of JOY_BUTTON bit flags specifying which buttons were pressed. The low-order word of *lParam* contains the *x*-position of the joystick, and the high-order word contains the *y*-position.
MM_JOY1BUTTONUP MM_JOY2BUTTONUP MM_JOY1BUTTONDOWN MM_JOY2BUTTONDOWN	Report that a joystick button has been pressed or released. The *wParam* parameter contains one JOY_BUTTONCHG bit flag specifying which button changed state and a combination of JOY_BUTTON bit flags specifying the current button states. The low-order word of *lParam* contains the *x*-position of the joystick, and the high-order word contains the *y*-position.

Using the Button Flags

The joystick services use the following bit flags, passed in the *wParam* parameter of the window function, to report the state of the joystick buttons:

Flag	Description
JOY_BUTTON1	Set when button 1 is pressed.
JOY_BUTTON1CHG	Set when button 1 has changed state.
JOY_BUTTON2	Set when button 2 is pressed.
JOY_BUTTON2CHG	Set when button 2 has changed state.
JOY_BUTTON3	Set when button 3 is pressed.
JOY_BUTTON3CHG	Set when button 3 has changed state.
JOY_BUTTON4	Set when button 4 is pressed.
JOY_BUTTON4CHG	Set when button 4 has changed state.

The MM_JOYMOVE messages use the JOY_BUTTON flags to report the state (pressed or released) of all buttons on the specified joystick.

The MM_JOYBUTTONUP and MM_JOYBUTTONDOWN messages use the JOY_BUTTON flags to report the state (pressed or released) of all buttons on the specified joystick. Also, they use JOY_BUTTONCHG flags to indicate which button changed state, thereby generating the message.

For example, if the user presses and holds buttons 1 and 2 and then moves the stick, a window function might receive the following messages:

Message	wParam Flags
MM_JOY1BUTTONDOWN	JOY_BUTTON1 | JOY_BUTTON1CHG
MM_JOY1BUTTONDOWN	JOY_BUTTON1 | JOY_BUTTON2 | JOY_BUTTON2CHG
MM_JOY1MOVE	JOY_BUTTON1 | JOY_BUTTON2

A window function might receive the following messages when the user presses and releases button 3 without moving the stick:

Message	wParam Flags
MM_JOY1BUTTONDOWN	JOY_BUTTON3 | JOY_BUTTON3CHG
MM_JOY1BUTTONUP	JOY_BUTTON3CHG

Releasing the Joystick

When your application no longer needs to receive periodic joystick messages, it should release the joystick using the **joyReleaseCapture** function. If your application does not release the joystick before ending, the joystick is released shortly after the capture window is destroyed.

The **joyReleaseCapture** function has the following syntax:

UINT joyReleaseCapture(*wJoyID*)

The *wJoyID* parameter is the joystick ID of the captured joystick. You can use the constants JOYSTICKID1 or JOYSTICKID2.

Setting the Movement Threshold

You can change the movement threshold of the joystick by calling the **joySetThreshold** function. The movement threshold is the number of device units that the stick must be moved before an MM_JOYMOVE message is sent to the window that has captured the device. The **joySetThreshold** function has the following syntax:

UINT joySetThreshold(*wJoyID, wThreshold*)

The two parameters, *wJoyID* and *wThreshold*, identify the joystick device and specify the movement threshold. You can get the minimum resolution of the joystick by calling the **joyGetDevCaps** function.

Polling the Joystick

You can poll the joystick for position and button information. For example, an application might poll the joystick to get baseline position values; the Joystick Control Panel applet uses this technique when calibrating the joystick. The **joyGetPos** function allows you to poll the joystick for position and button information. It has the following syntax:

UINT joyGetPos(*wJoyID, lpJoyInfo*)

The *wJoyID* parameter identifies the joystick. The *lpJoyInfo* parameter is a far pointer to a JOYINFO structure that is filled by the function.

Note Calling **joyGetPos** while joystick input is captured can prevent the joystick services from accurately reporting joystick events to the capture window.

The JOYINFO Structure

The JOYINFO structure has the following form:

```
typedef struct joyinfo_tag {
    UINT    wXpos;
    UINT    wYpos;
    UINT    wZpos;
    UINT    wButtons;
} JOYINFO;
```

The **wXpos**, **wYpos**, and **wZpos** fields specify the current *x*-, *y*-, and *z*-position of the joystick.

The **wButtons** field specifies the button states. This can be any combination of the JOY_BUTTON bit flags. For example, the following expression evaluates to TRUE if button 1 is pressed:

```
joyinfo.wButtons & JOY_BUTTON1
```

See "Processing Joystick Messages," earlier in this chapter, for an explanation of the JOY_BUTTON flags.

Using Joystick Messages

The remainder of this chapter presents code fragments from a simple joystick game that performs the useful function of shooting holes in the desktop: it gets position and button-state information from the joystick services and, when a user presses the joystick buttons, plays waveform resources and paints bullet holes on the screen.

Most of the joystick-control code is in the main window function. In the following WM_CREATE case of the message handler, the application captures input from joystick 1:

```
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;
```

In response to the MM_JOY1MOVE messages, the application changes the position of the cursor and, if either button is pressed, draws a hole in the desktop:

```
case MM_JOY1MOVE :
    if((UINT) wParam & (JOY_BUTTON1 | JOY_BUTTON2))
        DrawFire(hWnd);
    DrawSight(lParam);                    // Calculate new cursor position
    break;
```

In response to the MM_JOY1BUTTONDOWN messages, the application uses **sndPlaySound** to play a waveform audio file:

```
case MM_JOY1BUTTONDOWN :
    if((UINT) wParam & JOY_BUTTON1)
    {
        sndPlaySound(lpButton1, SND_LOOP | SND_ASYNC | SND_MEMORY);
        DrawFire(hWnd);
    }
    else if(wParam & JOY_BUTTON2)
    {
        sndPlaySound(lpButton2, SND_ASYNC | SND_MEMORY |  SND_LOOP);
        DrawFire(hWnd);
    }
    break;
```

By specifying the SND_LOOP and SND_ASYNC flags with **sndPlaySound**, the JOYTOY application repeats the waveform playback until the button is released.

When a button is released, the window function receives a MM_JOY1BUTTONUP message, which it handles as follows:

```
case MM_JOY1BUTTONUP :
    sndPlaySound(NULL, 0);
    break;
```

This sequence stops the waveform-audio playback.

Chapter 7

Multimedia File I/O Services

Most multimedia applications require file I/O—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 IBM/Microsoft Resource Interchange File Format (RIFF) files. The services are extensible with custom I/O procedures that can be shared among applications.

This chapter covers the following topics:

- Using basic file I/O services
- Performing buffered file I/O
- Working with RIFF files
- Directly accessing a file I/O buffer
- Using memory files
- Writing a custom I/O procedure

About the Multimedia File I/O Services

The multimedia file I/O services provide support for the following file I/O operations:

- Basic unbuffered and buffered file I/O
- RIFF file I/O
- Direct access to the file I/O buffer
- Memory files
- Custom storage system I/O using application-supplied I/O procedures

Most applications only need 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 a 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* can provide their own I/O procedure that reads and writes elements of the storage system. A storage system is a method of physically storing data in a file, such as a file archival system or a database storage system.

Comparison with MS-DOS, C Run-time, and Windows File I/O

You might ask why you need another set of file I/O services, when you already have the services of MS-DOS, the C run-time libraries, and Windows. The multimedia file I/O services offer the following advantages over other file I/O services:

- They provide more functionality and are easier to use than the MS-DOS services
- They are a part of the system software, so they don't increase the size of your application, like linking to the C run-time libraries
- They provide more functionality than the Windows services, such as support for buffered I/O, RIFF files, memory files, and custom storage systems

In addition, the multimedia file I/O services are optimized for performance-intensive applications. The CPU overhead of using these services versus going directly to MS-DOS is very low.

Function Prefixes

All multimedia file I/O function names begin with the **mmio** prefix. Similarly, all multimedia file I/O message names begin with the **MMIOM_** prefix.

Data Types

The MMSYSTEM.H header file defines data types and function prototypes for all multimedia file I/O functions. You must include this header file in any source module that uses these functions. MMSYSTEM.H depends on declarations made in WINDOWS.H, so you must first include the WINDOWS.H header file. MMSYSTEM.H defines the following data types for the multimedia file I/O functions:

FOURCC	A four-character code identifying an element of a RIFF file.
HMMIO	A handle to an open file.
MMCKINFO	A data structure containing information about a chunk in a RIFF file.
MMIOINFO	A data structure used to maintain the current state of a file accessed using the multimedia file I/O services.
MMIOPROC	A custom multimedia file I/O procedure.

Performing Basic File I/O

Using the basic I/O services is similar to using the C run-time file I/O services. Files must be opened before they can be read or written. After reading or writing, the file must be closed. You can change the current read/write location by seeking to a specified position in an open file. The following table lists the basic file I/O functions:

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

These file I/O functions provide the core of the multimedia file I/O services—you can use them for buffered and unbuffered I/O, as well as for I/O to RIFF files, memory files, and custom storage systems.

Opening a File

Before doing any I/O operations to a file, you must first open the file using the **mmioOpen** function. The **mmioOpen** function returns a file handle which you use to identify the open file when calling other file I/O functions. The **mmioOpen** function has the following syntax:

HMMIO mmioOpen(*szFileName, lpmmioinfo, dwFlags*)

The *szFileName* parameter points to a null-terminated string containing the path of the file to open.

The *lpmmioinfo* parameter is a far pointer to an MMIOINFO structure containing additional parameters. For basic file I/O services, this parameter should be NULL.

The *dwFlags* parameter specifies options for opening the file. The most commonly used flags for basic file I/O are MMIO_READ, MMIO_WRITE, and MMIO_CREATE.

The return value is a file handle of type HMMIO. Use this file handle to identify the open file when calling other file I/O functions. If the file cannot be opened, the return value is NULL.

Warning An HMMIO file handle is not a MS-DOS file handle. Do not use HMMIO file handles with MS-DOS, Windows, or C run-time file I/O functions.

There are options you can use with the **mmioOpen** function for operations beyond basic file I/O. By specifying an MMIOINFO structure with the *lpmmioinfo* parameter, you can open memory files, specify a custom I/O procedure, or supply a buffer for buffered I/O. These topics are discussed later in this chapter. First, this chapter discusses the most basic use of **mmioOpen**—opening files for basic unbuffered file I/O.

Opening a File

To open a file for basic I/O operations, set the *lpmmioinfo* parameter of **mmioOpen** to NULL. For example, the following code fragment opens a file named "C:\SAMPLES\SAMPLE1.TXT" for reading, and checks the return value for errors:

```
HMMIO hFile;
.
.
.
if ((hFile = mmioOpen("C:\\SAMPLES\\SAMPLE1.TXT", NULL, MMIO_READ)) != NULL)
    /* File opened successfully */

else
    /* File cannot be opened */
```

Options for Opening a File

When you open a file, you must specify whether you are opening the file for reading, writing, or both reading and writing. In addition, you can specify other options, such as to create or delete a new file. Use the *dwFlags* parameter of **mmioOpen** to specify options for opening a file.

Basic Options The following table lists the basic options for opening a file using **mmioOpen**:

Flag	Description
MMIO_READ	Opens a file for reading only.
MMIO_WRITE	Opens a file for writing only.
MMIO_READWRITE	Opens a file for reading and writing.
MMIO_CREATE	Creates a new file (if the file already exists, it truncates it to zero length).
MMIO_DELETE	Deletes a file.
MMIO_ALLOCBUF	Opens a file for buffered I/O.

The MMIO_READ, MMIO_WRITE, and MMIO_READWRITE flags are read/write privilege flags. These flags are mutually exclusive—specify only one when opening a file. If you don't specify one of these flags, **mmioOpen** opens the file for reading only.

The MMIO_CREATE and MMIO_DELETE flags are also mutually exclusive. You can specify one of the read/write privilege flags with the MMIO_CREATE flag. You can't specify any additional flags with the MMIO_DELETE flag.

For information on using the MMIO_ALLOCBUF flag, see "Performing Buffered File I/O," later in this chapter.

For a complete list of the **mmioOpen** flags, see the **mmioOpen** function entry in the *Multimedia Programmer's Reference*.

Sharing Options

In addition to the basic options for opening a file, there are sharing options you can use for opening MS-DOS files so they can be opened and accessed by more than one process. The following table lists the sharing options for opening a file using **mmioOpen**.

Flag	Description
MMIO_COMPAT	Opens a file in compatibility mode.
MMIO_EXCLUSIVE	Opens a file in exclusive mode.
MMIO_DENYWRITE	Opens a file and denies other processes write access to the file.
MMIO_DENYREAD	Opens a file and denies other processes read access to the file.
MMIO_DENYNONE	Opens a file without denying other processes read or write access to the file.

The sharing options are rarely used by applications, and are provided only because they are available through MS-DOS. They are not available for memory files or for files opened using custom I/O procedures. For more information on sharing options, see the *Microsoft MS-DOS Programmer's Reference* or the *Microsoft MS-DOS Encyclopedia.*

Note Sharing options are ignored unless the user has run the MS-DOS **share** command.

Creating and Deleting Files

To create a new file, specify the MMIO_CREATE option with the **mmioOpen** function. For example, the following code fragment creates a new file and opens it for both reading and writing:

```
HMMIO hFile;
.
.
.
hFile = mmioOpen("NEWFILE.TXT", NULL, MMIO_CREATE | MMIO_READWRITE);
if (hFile != NULL)
    /* File created successfully */

else
    /* File could not be created */
```

Note If the file you are creating already exists, it will be truncated to zero length.

To delete a file, specify the MMIO_DELETE flag with the **mmioOpen** function. Once you delete a file, it can't be recovered (except possibly by using certain MS-DOS based file recovery utilities). If the file deletion is the result of a request from a user, you should always query the user to be sure the user wants to delete the specified file before actually deleting it.

Reading and Writing a File

To read and write to open files, use the **mmioRead** and **mmioWrite** functions. Each of these functions takes an HMMIO file handle, a pointer to a buffer, and a parameter specifying the number of bytes to read or write. The read and write operations are not limited to 64K—the buffer pointers are huge pointers.

See "Example of RIFF File I/O," later in this chapter, for an example using **mmioRead** to read from a file.

Seeking to a New Position in a File

The *current position* or *file pointer* in a file is the location where the next read or write operation will occur. To change the current position in an open file, use the **mmioSeek** function. The **mmioSeek** function has the following syntax:

LONG mmioSeek(*hmmio, lOffset, iOrigin*)

The *hmmio* parameter specifies the file handle for the file.

The *lOffset* parameter specifies an offset for changing the current position in the file.

The *iOrigin* parameter specifies how the offset given by *lOffset* is interpreted. If *iOrigin* is SEEK_SET, the offset is from the beginning of the file. If it is SEEK_CUR, the offset is from the current position. If it is SEEK_END, the offset is from the end of the file.

The return value is the new position, specified in bytes from the beginning of the file. If an error occurs, the return value is –1. If you seek to an invalid location in a file, such as past the end of the file, **mmioSeek** might not return an error, but subsequent I/O operations can fail.

Examples Using mmioSeek

To seek to the beginning of an open file, use the following:

```
mmioSeek(hFile, 0L, SEEK_SET);
```

To seek to the end of an open file, use the following:

```
mmioSeek(hFile, 0L, SEEK_END);
```

To seek to a position ten bytes from the end of an open file, use the following:

```
mmioSeek(hFile, -10L, SEEK_END);
```

Performing Buffered File I/O

Most of the overhead in file I/O involves accessing the media (the physical device). If you are reading or writing many small blocks of information, the media device can spend a lot of time seeking to find the physical location on the media for each read or write operation. In this case, better performance is achieved by using buffered file I/O. 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 only accesses the media when the buffer must be filled from or written to the disk.

Deciding When to Use Buffered File I/O

It's difficult to provide exact metrics telling you when you need to use buffered I/O. It depends on how many read and write operations you perform on a file, and on the size of these read and write operations. A general guideline is if you are doing a lot of I/O operations less than 2K each, then use buffered I/O. But this rule isn't absolute—it's best to understand exactly how your program uses file I/O and experiment to optimize file I/O for your program's requirements.

Opening a File for Buffered File I/O

The multimedia file I/O manager provides several ways to set up and use buffered file I/O. The main distinction between these different approaches is whether the file I/O manager or the application allocates the buffer—either the file I/O manager or the application can allocate the I/O buffer. Unless you want to directly access the I/O buffer or open a memory file, you should let the file I/O manager allocate the buffer. For more information on directly accessing an I/O buffer and using memory files, see "Directly Accessing a File I/O Buffer" and "Performing File I/O on Memory Files," both later in this chapter.

A buffer allocated by the file I/O manager is called an *internal 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. Once 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.

I/O Buffer Control Functions

The multimedia file I/O services also include some functions giving you more control over the file I/O buffer. Using the following functions, you can force the contents of an I/O buffer to be written to disk, enable buffered I/O on a file opened for unbuffered I/O, change the size of an I/O buffer, and supply your own I/O buffer:

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

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

Flushing an I/O Buffer

Flushing an I/O buffer means writing the contents of the buffer to disk. You don't have to call **mmioFlush** to flush an I/O buffer—the buffer is automatically flushed when you close a file using **mmioClose**. If you don't close a file immediately after writing to it, you should flush the buffer to ensure the information is written to disk.

Note If you run out of disk space, **mmioFlush** might fail, even if the preceding **mmioWrite** calls were successful. Similarly, **mmioClose** might fail when it is flushing its I/O buffer.

Changing the Size of the Internal I/O Buffer

The default size of the internal I/O buffer is 8K. If this size is not adequate, you can use **mmioSetBuffer** to change the size of the buffer. You can also use **mmioSetBuffer** to enable buffering on a file opened for unbuffered I/O. The **mmioSetBuffer** function has the following syntax:

UINT mmioSetBuffer(*hmmio, pchBuffer, cchBuffer, wFlags*)

The *hmmio* parameter specifies the file handle for the file associated with the buffer.

The *pchBuffer* parameter specifies a pointer to a user-supplied buffer. For an internal buffer, set this parameter to NULL.

The *cchBuffer* parameter specifies the size of the buffer.

The *wFlags* parameter is unused and should be zero.

The return value is zero if the function is successful; otherwise, the return value specifies an error code.

Changing the I/O Buffer Size

For example, the following code fragment 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 */
```

Supplying Your Own I/O Buffer

You can also use **mmioSetBuffer** to supply your own buffer for use as a memory file. For details on using memory files, see "Performing File I/O on Memory Files," later in this chapter.

Working with RIFF Files

The preferred format for multimedia files is the Resource Interchange File Format (RIFF). The RIFF format is a tagged-file structure, and is described in detail in the file formats chapter in the *Multimedia Programmer's Reference.*

The multimedia file I/O services provide the following functions to support file I/O to RIFF files:

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

mmioCreateChunk
Creates a new chunk in a RIFF file.

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

mmioFOURCC
Converts four individual characters into a four-character code.

mmioStringToFOURCC
Converts a null-terminated string into a four-character code.

These functions work with the basic buffered and unbuffered file I/O services—you can open, read, and write RIFF files the same as other file types.

About RIFF Files

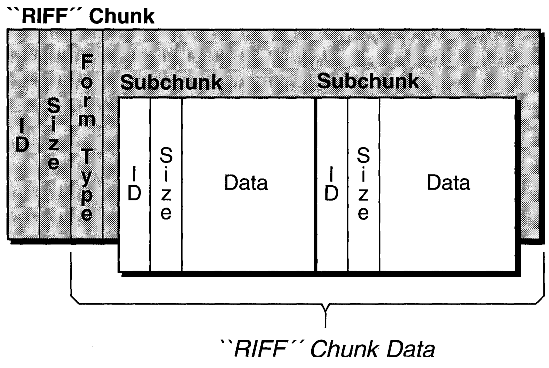
The basic building block of a RIFF file is called a *chunk*. Each chunk consists of the following fields:

- A four-character code specifying the chunk ID
- A DWORD specifying the size of the data field in the chunk
- A data field

The only chunks allowed to contain other chunks (subchunks) are those with a chunk ID of "RIFF" or "LIST". 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

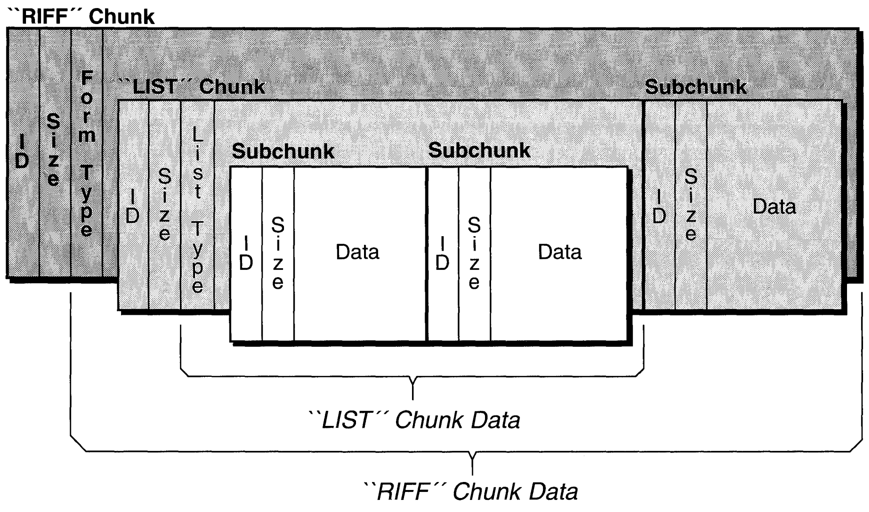
"RIFF" chunks include an additional field in the first four bytes of the data field. This additional field provides the form type of the field. The *form type* is a four-character code identifying the format of the data stored in the file. For example, Microsoft waveform audio files (WAVE files) have a form type of "WAVE". The following illustration shows a "RIFF" chunk containing two subchunks:



A "RIFF" chunk containing two subchunks.

"LIST" Chunks

"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. The following illustration shows a "RIFF" chunk containing a "LIST" chunk and one other subchunk (the "LIST" chunk contains two subchunks):



A "RIFF" chunk containing a "LIST" subchunk.

The MMCKINFO Structure

Several multimedia file I/O functions use the MMCKINFO structure to specify and retrieve information about RIFF chunks. The MMSYSTEM.H header file defines the MMCKINFO structure as follows:

```
typedef struct _MMCKINFO
{
    FOURCC      ckid;           // chunk ID
    DWORD       cksize;         // chunk size
    FOURCC      fccType;        // form type or list type
    DWORD       dwDataOffset;   // offset of data portion of chunk
    DWORD       dwFlags;        // flags
} MMCKINFO;
```

Generating Four-Character Codes

A *four-character code* is a 32-bit quantity representing a sequence of one to four ASCII alphanumeric characters, padded on the right with blank characters. The data type for a four-character code is FOURCC. Use the **mmioFOURCC** function to convert four characters to a four-character code, as shown in the following code fragment, which generates a four-character code for "WAVE":

```
FOURCC fourccID;
.
.
.
fourccID = mmioFOURCC('W', 'A', 'V', 'E');
```

To convert a null-terminated string into a four-character code, use **mmioStringToFOURCC**, as shown in the following code fragment, which also generates a four-character code for "WAVE":

```
FOURCC fourccID;
.
.
.
fourccID = mmioStringToFOURCC("WAVE", 0);
```

The second parameter in **mmioStringToFOURCC** specifies options 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 (four-character codes identifying file-extension names must be all uppercase).

Creating RIFF Chunks

To create a new chunk, use **mmioCreateChunk** to write a chunk header at the current position in an open file. The **mmioCreateChunk** function has the following syntax:

UINT mmioCreateChunk(*hmmio, lpmmckinfo, wFlags*)

The *hmmio* parameter specifies the file handle for an open RIFF file.

The *lpmmckinfo* parameter specifies a far pointer to an MMCKINFO structure containing information about the new chunk.

The *wFlags* parameter specifies option flags for creating the new chunk. To create a "RIFF" chunk, specify the MMIO_CREATERIFF flag. To create a "LIST" chunk, specify the MMIO_CREATELIST flag.

The return value is zero if the chunk is successfully created; otherwise, if there is an error creating the chunk, the return value specifies an error code.

Creating a "RIFF" Chunk

The following example creates a new chunk with a chunk ID 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 in the **fccType** field 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** field in 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 new chunk using **mmioCreateChunk**, 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).

Navigating RIFF Files

RIFF files consist of nested chunks of data. Multimedia file I/O services include two functions you can use to navigate between chunks in a RIFF file: **mmioAscend** and **mmioDescend**. You might think of 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.

Descending Into a Chunk

The **mmioDescend** function descends into a chunk or searches for a chunk, beginning at the current file position. The **mmioDescend** function has the following syntax:

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

The *hmmio* parameter specifies the file handle for an open RIFF file.

The *lpck* parameter specifies a far pointer to an MMCKINFO structure that **mmioDescend** fills with information on the current chunk. The structure can also contain additional parameters, depending on the *wFlags* parameter.

The *lpckParent* parameter specifies a far pointer to an MMCKINFO structure describing the parent or enclosing chunk. If there is no parent chunk, this parameter should be NULL.

The *wFlags* parameter specifies options for searching for a chunk. Valid flags are MMIO_FINDCHUNK, MMIO_FINDRIFF, and MMIO_FINDLIST. If no flags are specified, **mmioDescend** descends into the chunk at the current file position.

The return value is zero if the operation is successful; otherwise, the return value specifies an error code.

The **mmioDescend** function fills an MMCKINFO structure with information on the chunk. This information includes the chunk ID, the size of the data field, and the form type, or list type if the chunk is a "RIFF" or "LIST" chunk.

Searching for a Chunk

To search for a chunk in an open RIFF file, specify the MMIO_FINDCHUNK flag in the *wFlags* parameter of **mmioDescend**. Set the **ckid** field of the MMCKINFO structure referenced by *lpck* to the four-character code of the chunk you want to search for.

If you are searching for a "RIFF" or "LIST" chunk, you don't need to set the **ckid** field of the MMCKINFO structure—**mmioDescend** sets this field for you. Set the **fccType** field to the four-character code of the form type or list type of the chunk.

Searching for a "RIFF" Chunk

The following code fragment searches for a "RIFF" chunk with a form type of "WAVE" to verify that the file that has just been opened is a WAVE 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 WAVE file
 */
mmckinfoParent.fccType = mmioFOURCC('W', 'A', 'V', 'E');
if (mmioDescend(hmmio, (LPMMCKINFO) &mmckinfoParent, NULL, MMIO_FINDRIFF))
    /* The file is not a WAVE file. */
else
    /* The file is a WAVE file */
```

If the chunk you are searching for is a subchunk enclosed by a parent chunk (as are all chunks other than "RIFF" chunks), you should identify the parent chunk with the *lpckParent* parameter. In this case, **mmioDescend** will only search within the specified parent chunk.

Searching for a Subchunk

The following code fragment searches for the "fmt " chunk in the "RIFF" chunk descended into by 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 */
```

If you do not specify a parent chunk, the current file position should be at the beginning of a chunk before you call **mmioDescend** to search for a chunk. If you do specify a parent chunk, the current file position can be anywhere in the parent chunk.

If the search for a subchunk fails, the current file position is undefined. You can use **mmioSeek** and the **dwDataOffset** field of the MMCKINFO structure for the enclosing parent chunk to seek back to the beginning of the parent chunk, as in the following example:

```
mmioSeek(hmmio, mmckinfoParent.dwDataOffset + 4, SEEK_SET);
```

Since the **dwDataOffset** field specifies the offset to the beginning of the data portion of the chunk, you must seek four bytes past **dwDataOffset** to set the file position to be after the form type.

Ascending Out of a Chunk

After you descend into a chunk and read the data in the chunk, you can move the file position to the beginning of the next chunk by ascending out of the chunk by using the **mmioAscend** function. The **mmioAscend** function has the following syntax:

UINT mmioAscend(*hmmio, lpck, wFlags*)

The *hmmio* parameter specifies the file handle for an open RIFF file.

The *lpck* parameter specifies a far pointer to an MMCKINFO structure identifying a chunk. The function ascends to the location following the end of this chunk.

The *wFlags* parameter is not used and should be set to zero.

The return value is zero if the operation is successful; otherwise, the return value specifies an error code.

Ascending Out of a Subchunk

For example, the following statement ascends out of the "fmt " subchunk descended into by the previous example, illustrating searching for a subchunk:

```
/* Ascend out of the "fmt " subchunk
 */
mmioAscend(hmmio, &mmckinfoSubchunk, 0);
```

Example of RIFF File I/O

The following code fragment 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 WAVE waveform audio file backwards
 */
void ReversePlay()
{
    char        szFileName[128];    // filename of file to open
    HMMIO       hmmio;              // file handle for open file
    MMCKINFO    mmckinfoParent;     // parent chunk information structure
    MMCKINFO    mmckinfoSubchunk;   // subchunk information structure
    DWORD       dwFmtSize;          // size of "fmt " chunk
    DWORD       dwDataSize;         // size of "data" chunk
    WAVEFORMAT  *pFormat;           // pointer to memory for "fmt " chunk
    HPSTR       lpData;             // pointer to memory for "data" chunk
    ...

    /* Get the filename from the edit control
     */
     ...

    /* Open the given file for reading with buffered I/O
     * 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 WAVE file
     */
    mmckinfoParent.fccType = mmioFOURCC('W', 'A', 'V', 'E');
    if (mmioDescend(hmmio, (LPMMCKINFO) &mmckinfoParent, NULL,
                    MMIO_FINDRIFF)){
        Error("This is not a WAVE file.");
        mmioClose(hmmio, 0);
        return;
    }
```

```
/* Find the "fmt " chunk (form type "fmt "); it must be
 * a subchunk of the "RIFF" parent chunk
 */
mmckinfoSubchunk.ckid = mmioFOURCC('f', 'm', 't', ' ');
if (mmioDescend(hmmio, &mmckinfoSubchunk, &mmckinfoParent,
                MMIO_FINDCHUNK)){
    Error("WAVE 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("WAVE 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 output device
     */
    ...

    /* Allocate and lock memory for the waveform data
     */
    ...

    /* Read the waveform 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
     */
    ...

}
```

The MMIOINFO Structure

The multimedia file I/O manager uses the MMIOINFO data structure to maintain state information on an open file. The MMIOINFO data structure is defined in the MMSYSTEM.H header file as follows:

```
typedef struct _MMIOINFO
{
    /* general fields */
    DWORD       dwFlags;        // general status flags
    FOURCC      fccIOProc;      // ptr. to I/O procedure
    LPMMIOPROC  pIOProc;        // ptr. to I/O procedure
    UINT        wErrorRet;      // location for error to be returned
    HTASK       htask;          // alternate local task

    /* fields maintained by MMIO functions during buffered I/O */
    LONG        cchBuffer;      // size of I/O buffer (or 0L)
    HPSTR       pchBuffer;      // start of I/O buffer (or NULL)
    HPSTR       pchNext;        // ptr. to next byte to read/write
    HPSTR       pchEndRead;     // ptr. to last valid byte to read
    HPSTR       pchEndWrite;    // ptr. to last available byte to write
    LONG        lBufOffset;     // disk offset of start of buffer

    /* fields maintained by I/O procedure */
    LONG        lDiskOffset;    // disk offset of next read/write
    DWORD       adwInfo[3];     // data specific to MMIOPROC type

    /* other fields maintained by MMIO */
    DWORD       dwReserved1;    // reserved for internal use
    DWORD       dwReserved2;    // reserved for internal use
    HMMIO       hmmio;          // handle to open file
} MMIOINFO;
```

For more information and examples using the MMIOINFO structure, see "Directly Accessing a File I/O Buffer," "Performing File I/O on Memory Files," and "Using Custom I/O Procedures," all later in this chapter.

Directly Accessing a File I/O Buffer

Applications that are performance sensitive, such as applications that must stream data in real time from a CD-ROM, can optimize file I/O performance by directly accessing the file I/O buffer. Care should be exercised if you choose to do this—by accessing the file I/O buffer directly, you bypass some of the safeguards and error checking provided by the file I/O manager.

The multimedia file I/O services provide the following functions to support direct I/O buffer access:

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

mmioGetInfo
Retrieves information on the file I/O buffer of a file opened for buffered I/O.

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

To directly access a file I/O buffer, open the file for buffered I/O, as described in "Performing Buffered File I/O," earlier in this chapter. You can use the internal file I/O buffer or supply your own buffer with **mmioSetBuffer**.

Getting Information on the File I/O Buffer

Use the **mmioGetInfo** function to get information on a file I/O buffer, such as the buffer size and address. The **mmioGetInfo** function has the following syntax:

UINT mmioGetInfo(*hmmio, lpmmioinfo, wFlags*)

The *hmmio* parameter specifies the file handle for an open file.

The *lpmmioinfo* parameter specifies a far pointer to a MMIOINFO structure that **mmioGetInfo** fills with information on the file I/O buffer.

The *wFlags* parameter specifies options for the operation. Currently, there are no options for **mmioGetInfo**.

The return value is zero if the operation is successful; otherwise, the return value specifies an error code.

Reading and Writing the File I/O Buffer

There are three fields in the MMIOINFO structure used for reading and writing the file I/O buffer: **pchNext**, **pchEndRead**, and **pchEndWrite**. The **pchNext** field points to the next location in the buffer to read or write. You must increment **pchNext** as you read and write the buffer. The **pchEndRead** field identifies the last valid character you can read from the buffer. Likewise, **pchEndWrite** identifies the last location in the buffer you can write. To be precise, both **pchEndRead** and **pchEndWrite** point to the memory location *following* the last valid data in the buffer.

Advancing the File I/O Buffer

When you reach the end of the file I/O buffer, you must *advance* the buffer to fill it from the disk (if you are reading), and flush it to the disk (if you are writing). Use the **mmioAdvance** function to advance a file I/O buffer. The **mmioAdvance** function has the following syntax:

UINT mmioAdvance(*hmmio, lpmmioinfo, wFlags*)

The *hmmio* parameter specifies the file handle for a file opened for buffered I/O.

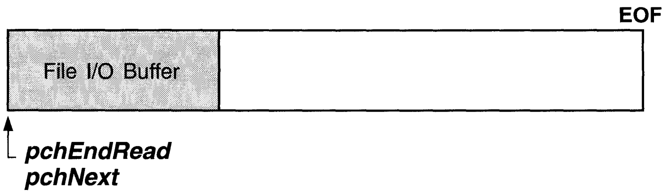
The *lpmmioinfo* parameter specifies a far pointer to an MMIOINFO structure containing information on the I/O buffer for the file.

The *wFlags* parameter specifies options for the operation. To fill an I/O buffer, use the MMIO_READ flag. To flush an I/O buffer, use the MMIO_WRITE flag. To flush the current buffer and fill it with more data from the file, use both flags.

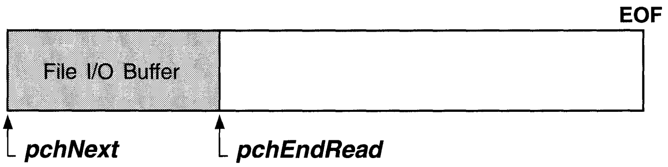
The return value is zero if the operation is successful; otherwise, the return value specifies an error code.

The following illustrations show how the file I/O buffer is advanced as a file is read or written.

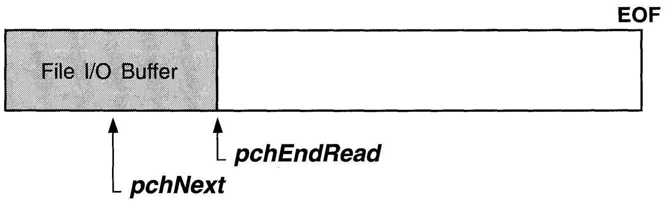
The application opens the file for buffered I/O. The buffer is initially empty, so ***mmioOpen*** *sets* ***pchNext*** *and* ***pchEndRead*** *to point to the beginning of the file I/O buffer.*



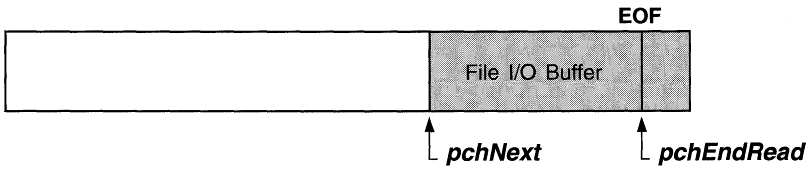
The application calls ***mmioAdvance*** *to fill the I/O buffer.* ***mmioAdvance*** *fills the buffer and sets* ***pchEndRead*** *to point to the end of the buffer.*



The application reads from the I/O buffer and increments ***pchNext****.*

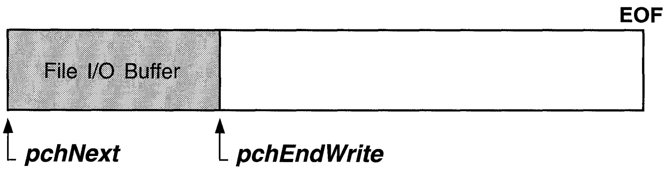


The application continues to read the buffer and call ***mmioAdvance*** *to refill the buffer when it's empty. When* ***mmioAdvance*** *reaches the end of the file, there is not enough information to fill the buffer.* ***mmioAdvance*** *sets* ***pchEndRead*** *to point to the end of valid data in the buffer.*

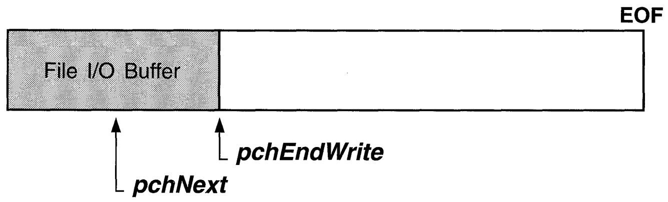


Advancing a file I/O buffer for reading.

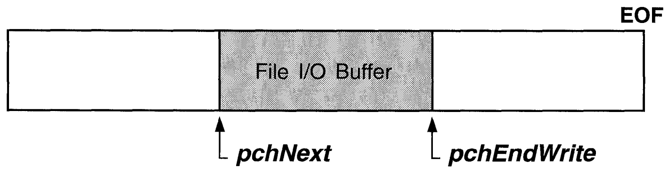
The application opens the file for buffered I/O. ***mmioOpen*** *sets* ***pchNext*** *to point to the beginning of the file I/O buffer and* ***pchEndWrite*** *to point to the end of the buffer.*



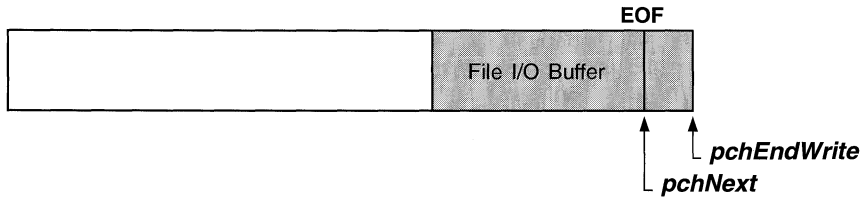
The application writes to the I/O buffer and increments ***pchNext****.*



Once the application fills the buffer, it calls ***mmioAdvance*** *to flush the buffer to disk.* ***mmioAdvance*** *resets* ***pchNext*** *to point to the beginning of the buffer.*



The application continues to write to the buffer and call ***mmioAdvance*** *to flush the buffer when it's full. At the end of the file, there is not enough information to fill the buffer. When the application calls* ***mmioAdvance*** *to flush the buffer,* ***pchNext*** *points to the end of valid data in the buffer.*



Advancing a file I/O buffer for writing.

Using the mmioAdvance Function

To fill a file I/O buffer from disk, call **mmioAdvance** with the MMIO_READ flag. If there is not enough data remaining in the file to fill the buffer, the **pchEndRead** field in the MMIOINFO structure points to the location following the last valid byte in the buffer.

To flush a buffer to disk, first set the MMIO_DIRTY flag in the **dwFlags** field of the MMIOINFO structure. Then call **mmioAdvance** with the MMIO_WRITE flag.

Example of Accessing a File I/O Buffer

The following code fragment is based on the ReversePlay example function discussed in "Example of RIFF File I/O," earlier in this chapter. In this example, direct-buffer access is used to read waveform data from a file.

```
HMMIO       hmmio;
MMIOINFO    mmioinfo;
DWORD       dwDataSize;
DWORD       dwCount;
HPSTR       hptr;
.
.
.

/* Get information on 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);
```

Ending Direct Access of a File I/O Buffer

When you finish accessing a file I/O buffer, pass the MMIOINFO structure filled by **mmioGetInfo** to **mmioSetInfo** to terminate direct-buffer access. If you wrote to the buffer, set the MMIO_DIRTY flag in the **dwFlags** field of the MMIOINFO structure before calling **mmioSetInfo**. Otherwise, the buffer will not be flushed to disk.

Performing File I/O on Memory Files

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 file memory images as if they were disk-based files. You can also use memory files with the Clipboard.

Opening Memory Files

Like 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 non-expandable. 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 **mmioOpen** 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 as follows:

- Set the **pIOProc** field to NULL.
- Set the **fccIOProc** field to FOURCC_MEM.
- Set the **pchBuffer** field to point to the memory block. To request that the file I/O manager allocate the memory block, set **pchBuffer** to NULL.
- Set the **cchBuffer** field to the initial size of the memory block.
- Set the **adwInfo[0]** field to the minimum expansion size of the memory block. For a non-expandable memory file, set **adwInfo[0]** to NULL.
- Set all other fields to zero.

Allocating Memory for Memory Files

There are no restrictions on allocating memory for use as a non-expandable memory file. You can use static memory or stack memory, or you can use locally allocated or globally allocated memory. For expandable memory files, you must use memory allocated using **GlobalAlloc** and locked using **GlobalLock**.

Using Custom I/O Procedures

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. There are predefined I/O procedures for standard MS-DOS files and for memory files. In the future, there will be a predefined I/O procedure for accessing elements of *compound files*. Compound files consist of a number of individual files, called *file elements*, bound together in one physical file.

You can supply a custom I/O procedure for accessing a unique storage system such as a database or file archive. This I/O procedure can be private to your application or it can be shared with other applications.

The multimedia file I/O services provide the following functions to support custom I/O procedures:

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

Opening a File Using a Custom I/O Procedure

Before learning about how to write an I/O procedure, it's helpful to understand how to use one. To open a file using a custom I/O procedure, use **mmioOpen** as you would to open any other file. Use a plus sign (+) in the filename to separate the name of the physical file from the name of the element of the file you want to open. For example, the following statement 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 an plus sign in a filename, it looks at the preceding filename extension to determine which I/O procedure to associate with the file. In the previous example, the file I/O manager will attempt to use the I/O procedure associated with the .ARC filename extension. If no I/O procedure is installed, the **mmioOpen** function returns an error.

Writing an I/O Procedure

An I/O procedure is a message-processing function supplied by an application. An I/O procedure function has the following syntax:

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

The *lpmmioinfo* parameter specifies a far pointer to an MMIOINFO structure associated with the file being accessed.

The *wMsg* parameter is a UINT and specifies the message being sent by the file I/O manager to the I/O procedure.

The *lParam1* parameter is an LPARAM and specifies 32 bits of message-dependent information.

The *lParam2* parameter is an LPARAM and specifies 32 bits of message-dependent information.

The return value is message-dependent. If the I/O procedure does not recognize a message, it should return zero.

I/O Procedure Messages

I/O procedures must respond to the MMIOM_CLOSE, MMIOM_OPEN, MMIOM_READ, MMIOM_WRITE, MMIOM_SEEK, MMIOM_RENAME, and MMIOM_WRITEFLUSH messages. Each of these messages has two parameters. For details about these messages and their parameters, see the *Multimedia Programmer's Reference.*

You can also create custom messages and send them to your I/O procedure using the **mmioSendMessage** function. If you define your own messages, be sure they are defined at or above the MMIOM_USER message. For example, the following code fragment defines a message named MMIOM_MYMESSAGE:

```
#define MMIOM_MYMESSAGE     MMIOM_USER + 0
```

Using the MMIOINFO Structure

In addition to processing messages, an I/O procedure must maintain the **lDiskOffset** field in the MMIOINFO structure referenced by the *lpmmioinfo* parameter. The **lDiskOffset** field 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[]** field to maintain any required state information. The I/O procedure should not modify any other fields in the MMIOINFO structure.

Installing an I/O Procedure

Use **mmioInstallIOProc** to install, remove, or locate an I/O procedure. The **mmioInstallIOProc** function has the following syntax:

LPMMIOPROC mmioInstallIOProc(*fccIOProc, pIOProc, dwFlags*)

The *fccIOProc* parameter specifies the filename extension associated with the I/O procedure. Use a four-character code to specify the extension. All characters in the four-character code must be uppercase characters.

The *pIOProc* parameter specifies a far pointer to the I/O procedure being installed. If the I/O procedure resides in the application (rather than being in a DLL), use **MakeProcInstance** to get a procedure-instance address and specify this address for *pIOProc*. If you are removing or locating an I/O procedure, *pIOProc* should be NULL.

The *dwFlags* parameter specifies whether to install, remove, or locate the specified I/O procedure. Specify one of the following flags for *dwFlags*: MMIO_INSTALLPROC, MMIO_REMOVEPROC, or MMIO_FINDPROC.

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

For example, to install an I/O procedure associated with the filename extension "ARC", use the following statement:

```
mmioInstallIOProc (mmioFOURCC('A', 'R', 'C', ' '),
                   (LPMMIOPROC)lpmmioproc, MMIO_INSTALLPROC);
```

Note Be sure to remove any I/O procedures you've installed before exiting your application.

Installing an I/O Procedure Using mmioOpen

When you install an I/O procedure using **mmioInstallIOProc**, the procedure remains installed until you remove it. The I/O procedure will be used for any file you open if the file has the appropriate filename extension. You can also temporarily install an I/O procedure using **mmioOpen**. In this case, the I/O procedure is only used with a file opened by **mmioOpen** and is removed when the file is closed using **mmioClose**.

To specify an I/O procedure when you open a file using **mmioOpen**, use the *lpmmioinfo* parameter to reference an MMIOINFO structure as follows:

- Set the **fccIOProc** field to NULL.
- Set the **pIOProc** field to the procedure-instance address of the I/O procedure.
- Set all other fields to zero (unless you are opening a memory file, or directly reading or writing to the file I/O buffer).

Sharing an I/O Procedure with Other Applications

To share an I/O procedure with other applications, follow these guidelines:

- Put the code for the I/O procedure in a dynamic-link library (DLL).
- Create a function in the DLL that calls **mmioInstallIOProc** 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.

Glossary

A

ADPCM (Adaptive Differential Pulse Code Modulation) An audio-compression technique.

animation The display of a series of graphic images, simulating motion. Animation can be frame-based or cast-based.

auxiliary audio device Audio devices whose output is mixed with the MIDI and waveform output devices in a multimedia computer. An example of an auxiliary audio device is the compact disc audio output from a CD-ROM drive.

B

break key In MCI, a keystroke that interrupts a wait operation. By default, MCI defines this key as CTRL+BREAK. An application can redefine this key by using the MCI_BREAK command message.

C

CD-DA (Compact Disc-Digital Audio) An optical data-storage format that provides for the storage of up to 73 minutes of high-quality digital-audio data on a compact disc. Also known as Red Book audio.

CD-ROM (Compact Disc-Read Only Memory) An optical data-storage technology that allows large quantities of data to be stored on a compact disc.

CD-XA (CD-ROM Extended Architecture) An extension of the CD-ROM standard that provides for storage of compressed audio data along with other data on a CD-ROM disc. This standard also defines the way data is read from a disc. Audio signals are combined with text and graphic data on a single track so they can be read at virtually the same time.

channel MIDI provides a way to send messages to an individual device within a MIDI setup. There are 16 MIDI channel numbers. Devices in a MIDI setup can be directed to respond only to messages marked with a channel number specific to the device.

channel map The MIDI Mapper provides a channel map that can redirect MIDI messages from one channel to another.

chunk The basic building block of a RIFF file, consisting of an identifier (called a *chunk ID*), a chunk-size variable, and a chunk data area of variable size.

command message In MCI, a command message is a symbolic constant that represents a unique command for an MCI device. Command messages have associated data structures that provide information a device requires to carry out a request.

command string In MCI, a command string is a null-terminated character string that represents a command for an MCI device. The text string contains all the information that an MCI device needs to carry out a request. MCI parses the text string and translates it into an equivalent command message and data structure that it then sends to a MCI device driver.

compound device An MCI device that requires a *device element*, usually a data file. An example of a compound device is the MCI waveform-audio driver.

compound file A number of individual files bound together in one physical file. Each individual file in a compound file can be accessed as if it were a single physical file.

control change See MIDI control-change message.

D

device element Data required for operation of MCI compound devices. The device element is generally an input or output data file.

F

file element An complete file contained in a RIFF compound file.

FM (Frequency Modulation) synthesizer A synthesizer that creates sounds by combining the output of digital oscillators using a frequency modulation technique.

form type A four-character code (FOURCC) identifying the type of data contained in a RIFF chunk. For example, a RIFF chunk with a form type of WAVE contains waveform audio data.

FOURCC (Four-Character Code) A code used to identify RIFF chunks. A FOURCC is a 32-bit quantity represented as a sequence of one to four ASCII alphanumeric characters, padded on the right with blank characters.

G

General MIDI A synthesizer specification created by the MIDI Manufacturers Association (MMA) defining a common configuration and set of capabilities for consumer MIDI synthesizers.

H

HMS time format A time format used by MCI to express time in hours, minutes, and seconds. The HMS time format is used primarily by videodisc devices.

I

IMA (International MIDI Association) The non-profit organization that circulates information about the MIDI specification.

IMA (Interactive Multimedia Association) A professional trade association of companies, institutions, and individuals involved in producing and using interactive multimedia technology.

L

LIST chunk A RIFF chunk with a chunk ID of LIST. LIST chunks contain a series of subchunks.

list type A four-character code (FOURCC) identifying the type of data contained in a RIFF chunk with a chunk ID of LIST. For example, a LIST chunk with a list type of INFO contains a list of information about a file, such as the creation date and author.

M

Media Control Interface (MCI) High-level control software that provides a device-independent interface to multimedia devices and media files. MCI includes a command-message interface and a command-string interface.

MIDI (Musical Instrument Digital Interface) A standard protocol for communication between musical instruments and computers.

MIDI control-change message A MIDI message sent to a synthesizer to change different synthesizer control settings. An example of a control-change message is the volume controller message, which changes the volume of a specific MIDI channel.

MIDI file A file format for storing MIDI songs. In Windows with Multimedia, MIDI files have a .MID filename extension. RIFF MIDI files have a .RMI filename extension.

MIDI Mapper Windows systems software that modifies MIDI output messages and redirects them to a MIDI output device using values stored in a MIDI setup map. The MIDI Mapper can change the destination channel and output device for a message, as well as modify program-change messages, volume values, and key values.

The Control Panel includes a MIDI Mapper applet that allows a user to create and edit MIDI setup maps.

MIDI mapping The process of translating and redirecting MIDI messages according to data defined in a MIDI map setup.

MIDI program-change message A MIDI message sent to a synthesizer to change the patch on a specific MIDI channel.

MIDI sequence Time-stamped MIDI data that can be played by a MIDI sequencer.

MIDI sequencer A program that creates or plays songs stored as MIDI files. When a sequencer plays MIDI files, it sends MIDI data from the file to a MIDI synthesizer, which produces the sounds. Windows provides a MIDI sequencer, accessible through MCI, that plays MIDI files.

MIDI setup map A complete set of data for the MIDI Mapper to use when redirecting MIDI messages. Only one setup map can be in effect at a given time, but the user can have several setup maps available and can choose between them using the MIDI Mapper Control Panel option.

MIDI time code (MTC) MIDI messages used for synchronizing MIDI sequences with external devices. The MCI MIDI sequencer does not support any type of synchronization.

MMA (MIDI Manufacturers Association) A collective organization composed of MIDI instrument manufacturers and MIDI software companies. The MMA works with the MIDI Standard Committee to maintain the MIDI specification.

MSF time format A time format used by MCI to express time in minutes, seconds, and frames. The number of frames in a second depends on the device type being used. Compact disc audio devices use 75 frames per second. The MSF time format is used primarily by compact disc audio devices.

MSCDEX (Microsoft Compact Disc Extensions) A terminate-and-stay-resident (TSR) program that makes CD-ROM drives appear to MS-DOS as network drives. MSCDEX uses hardware-dependent drivers to communicate with a CD-ROM drive.

P

patch A particular setup of a MIDI synthesizer that results in a particular sound, usually a sound simulating a specific musical instrument. Patches are also called *programs*. A MIDI program-change message changes the patch setting in a synthesizer. Patch also refers to the connection or connections between MIDI devices.

patch caching Some internal MIDI synthesizer device drivers can preload, or cache, their patch data. Patch caching reduces the delay between the moment that the synthesizer receives a MIDI program-change message and when it plays a note using the new patch. Patch caching also ensures that required patches are available (the synthesizer might load only a subset of its patches).

pitch scale factor An application can request that a waveform audio driver scale the pitch by a specified factor. A scale factor of two results in a one-octave increase in pitch. Pitch scaling requires specialized hardware. The playback rate and sample rate are not changed.

playback rate scale factor In waveform audio, an application can request that the waveform audio driver scale the playback rate by a specified factor. Playback scaling is accomplished through software; 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.

PPQN (Parts Per Quarter Note) A time format used for MIDI sequences. PPQN is the most common time format used with standard MIDI files.

preimaging The process of building a movie frame in a memory buffer before it is displayed.

R

Red Book audio See CD-DA.

Resolution For joysticks, resolution refers to the minimum and maximum intervals between joystick messages sent for a captured joystick.

For timers, resolution refers to the accuracy of the timer event. A resolution value of zero means that the event must occur at the exact time requested, while a resolution value of ten means that the event must occur within ten milliseconds of the requested time.

RIFF (Resource Interchange File Format) A tagged-file specification used to define standard formats for multimedia files. Tagged-file structure helps prevent compatibility problems that often occur when file-format definitions change over time. Because each piece of data in the file is identified by a standard header, an application that does not recognize a given data element can skip over the unknown information.

RIFF chunk A chunk with chunk ID RIFF that includes an identifying code and zero or more subchunks, the contents of which depend on the form type.

RIFF file A file whose format complies with one of the published RIFF forms.

Examples of RIFF files include WAVE files for waveform audio data, RMID files for MIDI sequences, and RDIB files for bitmaps.

RIFF form A file-format specification based on the RIFF standard.

S

sample A discrete piece of waveform data represented by a single numerical value. Sampling is the process of converting analog data to digital data by taking samples of the analog waveform at regular intervals.

sampling rate The rate at which a waveform audio driver performs audio-to-digital or digital-to-audio conversion. For CD-DA, the sampling rate is 44.1 kHz.

seek With file I/O, seek means to change the current position in the file. The current position is the location where the next read or write operation will take place. With a media device (such as a hard disk), seek means to position the media so a certain sector can be accessed. The seek involves a physical movement of the device, so the time it takes can often be perceived by the user.

sequence See MIDI sequence.

sequencer See MIDI sequencer.

simple device An MCI device that does not require a device element (data file) for playback. The MCI compact-disc audio driver is an example of a simple device.

SMPTE (Society of Motion Picture and Television Engineers) An association of engineers involved in movie, television, and video production. SMPTE also refers to SMPTE time, the timing standard that this group adopted.

SMPTE division type One of four SMPTE timing formats. SMPTE time is expressed in hours, minutes, seconds, and frames. The SMPTE division type specifies the frames-per-second value corresponding to a given SMPTE time. For example, a SMPTE time of one hour, 30 minutes, 24 seconds, and 15 frames is useful only if the frames-per-second value, or SMPTE division type, is known.

SMPTE offset A MIDI event that designates the SMPTE time at which playback of a MIDI file is to start. SMPTE offsets are used only with MIDI files using SMPTE division type.

SMPTE time A standard representation of time developed for the video and film industries. SMPTE time is used with MIDI audio because many people use MIDI to score films and video. SMPTE time is an absolute time format expressed in hours, minutes, seconds, and frames. Standard SMPTE division types are 24, 25, and 30 frames per second.

square-wave synthesizer A synthesizer that produces sound by adding square waves of various frequencies. A square wave is a rectangular waveform.

streaming The process of transferring information from a storage device, such as a hard disk or CD-ROM, to a device driver. Rather than transferring all the information in a single data copy, the information is transferred in smaller parts over a period of time, typically while the application is performing other tasks.

system-exclusive data In MIDI, messages understood only by MIDI devices from a specific manufacturer. System-exclusive data provides a way for MIDI-device manufacturers to define custom messages that can be exchanged between their MIDI devices. The standard MIDI specification defines only a framework for system-exclusive messages.

T

tagged file format A file format in which data is tagged using standard headers that identify information type and length.

tempo With the MIDI sequencer, tempo is the speed that a MIDI file is played. It is measured in beats per minute (BPM). A typical MIDI tempo is 120 BPM.

threshold For the joystick interface, the movement threshold is the distance in device units that the coordinates must change before the application is notified of the movement. Setting a high threshold reduces the number of joystick messages sent to your application, however, it also reduces the sensitivity of the joystick.

time stamp With recorded MIDI data (such as MIDI files), MIDI messages are tagged with a time stamp so that a MIDI sequencer can replay the data at the proper moment.

TMSF time format A time format used by MCI to express time in tracks, minutes, seconds, and frames. The number of frames in a second depends on the device type being used. Compact disc audio devices use 75 frames per second. The TMSF time format is used primarily by compact disc audio devices.

track A sequence of sound on a CD-DA disc. A CD-DA track usually corresponds to a song.

With a MIDI file, information can be separated into tracks, which are defined by the creator of the MIDI file. MIDI file tracks can correspond to MIDI channels, or they can correspond to parts of a song (such as melody or chorus).

V

volume scalar A component of a MIDI Mapper patch map that adjusts the volume of a patch on a synthesizer. For example, if the bass patch on a synthesizer is too loud compared to its piano patch, the volume scalar can reduce the volume for the bass or increase the volume for the piano.

Applications playing waveform audio can also adjust the output volume.

W

WAVE file A Microsoft standard file format for storing waveform audio data. WAVE files have a .WAV filename extension.

waveform audio A technique of recreating an audio waveform from digital samples of the waveform.

Index

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-639

Microsoft®

0392 Part No. 30253