Introduction to Multimedia Drivers

The following topics provide an introduction to writing installable multimedia drivers for Windows NT®:

- Types of Multimedia Devices
- User-Mode Multimedia Drivers
- Kernel-Mode Multimedia Drivers
- Multimedia Dynamic-Link Libraries
- Designing Multimedia Drivers
- Building Multimedia Drivers
- Installing Multimedia Drivers
- Configuring Multimedia Drivers
- Multimedia Driver Reference

These topics provide information that pertains to all types of multimedia devices and drivers. For specific information about writing drivers for a certain type of multimedia device, refer to the topics listed in Types of Multimedia Devices.

Types of Multimedia Devices

Each piece of multimedia hardware can be classified as being either an audio device, a video capture device, or a positioning device. Each of these classifications can be broken down further, as follows:

- Audio devices include waveform devices, MIDI devices, mixers, and auxiliary audio devices.

  For information about writing drivers for audio devices, see Audio Device Drivers.

  For information about writing drivers for compressing audio data, see Audio Compression Manager Drivers.
- Video capture devices capture video images that can be stored in a disk file and played back later.

  For information about writing drivers for video capture devices, see Video Capture Device Drivers.

  For information about writing drivers for compressing video data, see Video Compression Manager Drivers.
- Positioning devices, such as joysticks, light pens, and touch screens, are devices that can establish a screen position.

  For information about writing drivers for positioning devices, see Joystick Drivers.

For information about writing drivers for the Media Control Interface (MCI), which is a high-level application interface to all types of multimedia devices, see MCI Drivers.

User-Mode Multimedia Drivers

User-mode multimedia drivers executing under Windows NT have the following characteristics:

- They execute in user mode.
- They export user-mode driver entry points that are called by applications and other clients to request I/O operations.
- They communicate with kernel-mode multimedia drivers by calling Win32 functions which, in turn, call functions in the Windows NT Executive. The Windows NT Executive functions provide the context switch from user mode to kernel mode.

Typically, user-mode drivers call **CreateFile** to open a device instance. Then they make numerous calls to **DeviceIoControl** to send I/O control codes to the kernel-mode driver. Some drivers also call the **ReadFileEx** and **WriteFileEx** functions to transfer data blocks. Calling **CloseHandle** closes the device instance. (All of these functions are described in the Win32 SDK.)

User-mode drivers do not generally call these Win32 functions directly. Instead, they typically call functions in support libraries, which in turn call the Win32 functions. The support libraries are described in the chapters discussing the various driver types. The chapters are listed in Types of Multimedia Devices.

User-mode drivers do not necessarily communicate with a kernel-mode driver — some communicate with other user-mode drivers. For example some MCI drivers call Win32 Audio API functions to communicate with user-mode audio device drivers.

Because user-mode Windows NT drivers run under the Windows NT Win32 Subsystem and call Win32 API functions described in the Win32 SDK, these drivers are sometimes called *Win32-based drivers*.

For information about designing a user-mode multimedia driver, see Designing a User-Mode Multimedia Driver.

Kernel-Mode Multimedia Drivers

Kernel-mode multimedia drivers executing under Windows NT have the following characteristics:

- They execute in kernel mode.
- They export kernel-mode driver entry points that are called by the Windows NT Executive for performing I/O operations requested by user-mode multimedia drivers.
- They are implemented as services under the control of the Windows NT Service Control Manager.
- They communicate with device hardware by calling functions in the Windows NT Executive. The I/O Manager and the Hardware Abstraction Layer (HAL), both of which are parts of the Windows NT Executive, provide driver compatibility across the various hardware platforms supported by Windows NT.

  Kernel-mode drivers often do not call Windows NT Executive functions directly. Instead, they might call functions in support libraries, which in turn call the Executive functions. The support libraries are described in the chapters discussing the various driver types. The chapters are listed in Types of Multimedia Devices.

For information about designing a kernel-mode multimedia driver, see Designing a Kernel-Mode Multimedia Driver.

Multimedia Dynamic-Link Libraries

The following dynamic-link libraries export the APIs that clients use to access multimedia drivers.

winmm.dll

The *winmm.dll* dynamic-link library exports several Win32 multimedia APIs, described in the Win32 SDK, including the following:

- Functions for accessing audio device drivers (**wave**-prefixed, **midi**-prefixed, **mixer**-prefixed, and **aux**-prefixed functions).
- Functions for accessing MCI drivers (**mci**-prefixed functions).
- Functions for accessing joystick drivers (**joy**-prefixed functions).
- Functions for sending specific messages to a user-mode driver's **DriverProc** function (**OpenDriver**, **SendDriverMessage**, and **CloseDriver**).

Additionally, *winmm.dll* exports **DefDriverProc**, which user-mode drivers can call for default processing of the standard driver messages.

msacm32.dll

The *msacm32.dll* dynamic-link library exports the audio compression functions that are described in the Win32 SDK and are used to send messages to Audio Compression Manager drivers.

avicap32.dll

The *avicap32.dll* dynamic-link library exports the AVI capture window class, which is described in the Win32 SDK and which is used to send messages to video capture device drivers.

msvfw32.dll

The *msvfw32.dll* dynamic-link library exports the following multimedia software components:

- Functions described in the Video for Windows Development Kit, which are used to send messages to video capture device drivers.
- The Video Compression Manager, which sends messages to Video Compression Manager drivers.
- The MCIWnd window class, which is described in the Win32 SDK and which controls multimedia devices by sending messages to MCI drivers.
- The DRAWDIB functions, which are described in the Win32 SDK and which provide capabilities for high-performance drawing of device-independent bitmaps (DIBs).

Designing Multimedia Drivers

This section provides the following topics about designing user-mode multimedia drivers and kernel-mode multimedia drivers:

- Do you Need a New Driver?
- Designing a User-Mode Multimedia Driver
- Designing a Kernel-Mode Multimedia Driver

Do you Need a New Driver?

For a new piece of multimedia hardware, you must decide if you need both a new user-mode multimedia driver and a new kernel-mode multimedia driver. If a currently available user-mode driver allows applications to access all of the functionality provided by the new hardware, then you do not need to write a new user-mode driver. For example, the standard audio driver, *mmdrv.dll*, might provide all the necessary interfaces to access all the features of a new sound card. In this case, only a new kernel-mode driver is necessary. More typically, new hardware requires development of a new user-mode driver in order to handle the hardware's unique configuration requirements. New user-mode drivers can be easily written by modifying sample drivers provided with this DDK.

New hardware almost always requires a new kernel-mode driver, because the kernel-mode driver contains information about a device's registers and hardware buffers. Occasionally, a new user-mode driver is written that does not require support from a new kernel-mode driver. For example, the MCICDA CD audio driver uses the standard CD-ROM file system, so it does not require a unique underlying kernel-mode driver.

If your device can be connected to several different buses, you do not need a different kernel-mode driver for each bus. The Windows NT Hardware Abstraction Layer (HAL) insulates the kernel-mode driver from the bus. The *Kernel-Mode Drivers Design Guide* provides extensive general information about the design of kernel-mode drivers. Additionally, specific information about designing kernel-mode multimedia drivers is provided in the chapters discussing the multimedia driver types. The chapters are listed in Types of Multimedia Devices.

Designing a User-Mode Multimedia Driver

This section provides the following topics about designing a user-mode multimedia driver:

- User-Mode Driver Entry Points
- Standard Driver Messages
- Customized Driver Messages
- Driver Instances
- Character Strings

For additional information about designing user-mode drivers, see the chapters discussing the various driver types. The chapters are listed in Types of Multimedia Devices.

User-Mode Driver Entry Points

Applications access user-mode driver capabilities by passing messages to standard driver entry points.

All Win32-based user-mode drivers must export an entry point function called **DriverProc**. This function receives a set of messages known as the standard driver messages. Generally, a user-mode driver's **DriverProc** function also recognizes additional customized driver messages.

A default message handler, called **DefDriverProc,** is provided within *winmm.dll*. Every user-mode driver should call **DefDriverProc** from within its **DriverProc** function, if it receives an unrecognized or unsupported message.

User-mode audio device drivers provide additional entry point functions.

All driver entry points must be exported in a module definition (*.def*) file.

Standard Driver Messages

Every user-mode driver must export a **DriverProc** function. Each **DriverProc** function must recognize a set of standard driver messages, which are defined in *mmsystem.h*. A driver receives the following standard messages, in the order listed, when an application uses the driver to perform input or output operations.

Message	Operation Performed by Driver
DRV_LOAD	Performs post-load operations.
DRV_ENABLE	No operations performed under Windows NT.
DRV_OPEN	Opens a driver instance.
DRV_CLOSE	Closes a driver instance.
DRV_DISABLE	No operations performed under Windows NT.
DRV_FREE	Performs pre-unload operations.

Additionally, a driver can receive the following standard messages, which are typically sent from a Control Panel application during installation and configuration operations.

Message	Operation Performed by Driver
DRV_INSTALL	Installs the kernel-mode driver.
DRV_PNPINSTALL	Installs a kernel-mode driver, using Plug and Play configuration information.
DRV_CONFIGURE	Obtains configuration parameters.
DRV_QUERYCONFIGURE	Indicates whether configuration parameters can be modified.

Although applications can send standard driver messages directly by calling **SendDriverMessage**, described in the Win32 SDK, typically they do not. Instead, they call functions provided by higher level multimedia APIs. These APIs in turn act as clients to the

user-mode drivers and send messages by:

- Calling **SendDriverMessage**, to directly send any of the standard messages.
- Calling **OpenDriver**, described in the Win32 SDK. This function calls **SendDriverMessage** to send DRV_LOAD and DRV_ENABLE messages, if the driver has not been previously loaded, and then to send DRV_OPEN.
- Calling **CloseDriver**, described in the Win32 SDK. This function calls **SendDriverMessage** to send DRV_CLOSE and, if there are no other open instances of the driver, to also send DRV_DISABLE and DRV_FREE.

Besides supporting the standard driver messages, a user-mode multimedia driver's **DriverProc** function generally also supports a set of customized driver messages.

Customized Driver Messages

Besides supporting the standard driver messages, a user-mode driver's **DriverProc** function often supports additional customized messages that are specific to each multimedia device type. (User-mode audio device drivers support customized messages by providing additional entry point functions.)

Although applications can send customized driver messages directly by calling **SendDriverMessage**, described in the Win32 SDK, typically they do not. Instead, they call functions provided by higher level multimedia APIs. These APIs in turn act as clients to the user-mode drivers and send customized messages by calling **SendDriverMessage**.

Customized driver messages are described in the chapters discussing the various driver types. The chapters are listed in Types of Multimedia Devices.

Driver Instances

The Win32 multimedia APIs and the multimedia dynamic-link libraries allow multiple clients to simultaneously open a user-mode driver. You can choose whether to allow multiple open driver instances, as follows:

- If you want to allow multiple open instances, your user-mode driver should define a linked list of dynamically allocated structures. All instance data should be stored in this list, and you should not use any static or global variables. When the driver's **DriverProc** function receives a DRV_OPEN message, it should:

  1. Allocate memory space for a structure instance.
  2. Add the structure instance to the linked list.
  3. Store instance data in the new list entry.
  4. Specify the entry's number or address as the return value for the **DriverProc** function.

  Subsequent calls to **DriverProc** will include the list entry's identifier as its *dwDriverID* argument. The sample audio device drivers use this technique, although they use the customized audio driver entry points and messages instead of **DriverProc** and DRV_OPEN.

  If a user-mode driver allows multiple instances, the kernel-mode driver is usually responsible for rejecting conflicting requests for access to the hardware.
- If you do not want to allow multiple open instances, your driver can set a flag the first time it receives a DRV_OPEN message. When subsequent DRV_OPEN messages are received, the driver can provide an error return value for **DriverProc** until a DRV_CLOSE message is received, at which point it can clear the flag. The sample video capture device drivers use this technique.

Character Strings

Under Windows NT, character strings consist of Unicode characters. All strings passed between clients and drivers are Unicode strings.

The Win32 SDK provides numerous articles on defining and using Unicode strings.

Designing a Kernel-Mode Multimedia Driver

This section provides the following topic for designing a kernel-mode multimedia driver:

- Kernel-Mode Driver Entry Points

For additional information about designing kernel-mode drivers, see the chapters discussing the various driver types. The chapters are listed in Types of Multimedia Devices.

Kernel-Mode Driver Entry Points

All kernel-mode multimedia drivers must export a **DriverEntry** function. They must also fill in the Windows NT device object's dispatch table.

DriverEntry in Kernel-Mode Multimedia Drivers

All kernel-mode multimedia drivers must export a **DriverEntry** function, which is the first function executed after the driver is loaded. (The driver is loaded as a result of being installed by a user-mode driver. See Installing a Kernel-Mode Multimedia Driver.)

DriverEntry for multimedia drivers should perform such installation-time operations as obtaining hardware configuration parameter values from the registry, reserving system resources, and verifying that device hardware is accessible.

For more information about the **DriverEntry** function, see the chapters discussing the various driver types. The chapters are listed in Types of Multimedia Devices.

The Driver Object's Dispatch Table

Before Windows NT calls a kernel-mode driver's **DriverEntry** function, it creates a driver object. It then passes the driver object's address as an input argument to **DriverEntry** for multimedia drivers. The kernel-mode driver is responsible for filling in the driver object's **MajorFunction** member, which is a dispatch table representing the various I/O control codes that a user-mode driver can send to a kernel-mode driver. For more information about driver objects, which are defined by the DRIVER_OBJECT structure, see the *Kernel-Mode Drivers Reference*.

Kernel-mode multimedia drivers do not always fill in the dispatch table directly. Sometimes, multimedia driver support libraries take care of this operation. The support libraries are described in the chapters discussing the various driver types. The chapters are listed in Types of Multimedia Devices.

Building Multimedia Drivers

To build a multimedia driver, you should use the BUILD utility, which is described in the *Programmer's Guide*. For more information, see the following topics:

- Building a User-Mode Multimedia Driver
- Building a Kernel-Mode Multimedia Driver

Building a User-Mode Multimedia Driver

To build user-mode drivers, refer to the description of the BUILD utility in the *Programmer's Guide*. You must provide a file named *makefile* and a file named *sources*, and place them in the directory path containing your source files. Refer to the *makefile* and *sources* files provided with the source code for the sample drivers in this DDK.

All user-mode multimedia drivers must be linked with *winmm.dll*. Additional libraries are provided for the various device and driver types and are referred to in the chapters describing each driver type. The chapters are listed in Types of Multimedia Devices.

All user-mode multimedia drivers must include the *mmystem.h* header file. Additional header files are provided for the various device and driver types and are referred to in the chapters describing each driver type.

Because user-mode drivers are dynamic-link libraries, their file extension should generally be *.dll*. (This is not a requirement and, for example, Audio Compression Manager drivers have an extension of *.acm*.) If you provide both a 16-bit and a 32-bit version of your driver, append *32* to the file name for the 32-bit version.

Building a Kernel-Mode Multimedia Driver

To build kernel-mode drivers, refer to the description of the BUILD utility in the *Programmer's Guide*. You must provide a file named *makefile* and a file named *sources*, and place them in the directory path containing your source files. Refer to the *makefile* and *sources* files provided with the source code for the sample drivers in this DDK.

Libraries and header files are provided for the various device and driver types and are referred to in the chapters describing each driver type. The chapters are listed in Types of Multimedia Devices.

Kernel-mode multimedia drivers for Windows NT must have a file extension of *.sys*.

Installing Multimedia Drivers

Driver installation procedures are described in the Windows NT DDK *Programmer's Guide*. For multimedia drivers not supplied by Microsoft, installation is accomplished by running the Multimedia applet in the Control Panel. This section provides the following topics:

- Installing a User-Mode Multimedia Driver
- Installing a Kernel-Mode Multimedia Driver

Installing a User-Mode Multimedia Driver

Windows NT users with Administrator privilege can install non-Microsoft multimedia drivers by running the Control Panel's Multimedia applet. The Multimedia applet reads *oemsetup.inf* files to determine which files to install. To allow the Multimedia applet to install your user-mode driver, you must provide an *oemsetup.inf* file identifying the user-mode driver. For more information, see Using *oemsetup.inf* Files with Multimedia Drivers.

Using *oemsetup.inf* Files with Multimedia Drivers

The driver's installation medium must include an *oemsetup.inf* file. A general discussion of *oemsetup.inf* files is provided in the *Programmer's Guide*. When you use the Multimedia applet in the Control Panel to install a multimedia driver, the Multimedia applet reads the *oemsetup.inf* file in order to determine which driver files to install. To understand how *oemsetup.inf* files must be constructed for multimedia drivers, look at the following example:

```
[Source Media Descriptions]
    1  = "Sound Blaster Driver"  , TAGFILE = disk1

[Installable.Drivers]
soundblaster = 1:sndblst.dll, "wave,MIDI,aux,mixer", "Creative Labs Sound Blaster 1

[soundblaster]
1:sndblst.sys
```

The **[Source Media Descriptions]** section identifies the load medium, as described in the *Programmer's Guide*.

Next is the **[Installable.Drivers]** section, which is required for multimedia drivers installed using the Multimedia Applet. Each line under this section is a driver *profile*. Each driver profile describes

a user-mode driver. A driver profile consists of six fields, separated by commas. The first three fields are:

1. Source medium and user-mode driver filename
2. List of supported device types, also known as *aliases*
3. Driver description

In the example, the "1:sndblst.dll" field indicates that the driver's file name is *sndblst.dll*, located on source medium 1. This file will be copied to the system's *\system32* subdirectory.

The next field lists all of the device types (aliases) supported by the driver. The Multimedia applet creates a registry entry for each device type. It would create the following entries for the example *.inf* file:

```
wave : REG_SZ : sndblst.dll
MIDI : REG_SZ : sndblst.dll
aux : REG_SZ : sndblst.dll
mixer : REG_SZ : sndblst.dll
```

If the alias already exists in the registry path, a number is appended to the alias string. The entries are placed in the registry path **HKEY_LOCAL_MACHINE \SOFTWARE \Microsoft \Windows NT \CurrentVersion \Drivers32**.

The third field is the driver's description, which is the description that is displayed by the Multimedia Applet. The remaining fields are not used.

The **[soundblaster]** section name matches the name given to the profile entry. This section lists additional files that need to be copied. Include your kernel-mode driver in this section. The Multimedia Applet copies files with an extension of *.sys* into the *\system32\drivers* subdirectory, and copies all other files into the *\system32* subdirectory.

After the files have been copied, the Multimedia applet sends DRV_LOAD, DRV_ENABLE, DRV_OPEN, DRV_INSTALL, DRV_QUERYCONFIGURE, DRV_CONFIGURE, DRV_CLOSE, DRV_DISABLE, and DRV_FREE messages to the user-mode driver, in that order.

Sample *oemsetup.inf* files are included with the source code provided with this DDK.

Installing a Kernel-Mode Multimedia Driver

If you list your kernel-mode driver in your *oemsetup.inf* file, the Multimedia applet installs it for you by calling the Win32 **CreateService** function. For more information about *oemsetup.inf* files, see Using *oemsetup.inf* Files with Multimedia Drivers.

Additionally, the user-mode driver can install the kernel-mode driver by either calling **CreateService**, or by calling one of the functions in the driver support libraries. These libraries are described in the chapters discussing the various driver types. The chapters are listed in Types of Multimedia Devices.

Configuring Multimedia Drivers

Drivers typically store configuration information in the Windows NT Registry. Multimedia drivers can optionally require two types of configuration parameters — hardware configuration parameters and user configuration parameters.

Hardware configuration parameters represent values that a kernel-mode driver needs in order to access device hardware. For information about storing these parameters, see Storing Hardware Configuration Parameters.

User configuration parameters represent values that a user-mode or kernel-mode driver needs in order to support user-controllable runtime options. For information about storing these parameters, see Storing User Configuration Parameters.

Storing Hardware Configuration Parameters

Kernel-mode drivers generally require hardware configuration parameters, such a device's interrupt number and DMA channel, that must be supplied by a system administrator. The user-mode driver obtains values for these parameters by displaying a dialog box in response to a DRV_CONFIGURE message. The user-mode driver stores the parameter values in the Windows NT Registry, where they are accessible to the kernel-mode driver.

Because kernel-mode drivers are treated as services under Windows NT, their parameters are stored under the registry's **\Services** key. The path to the **\Services** key is **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services**.

Under **\Services**, there is a subkey for each installed kernel-mode driver. For multimedia drivers, the registry structure is as follows:

```
HKEY_LOCAL_MACHINE
    SYSTEM
        CurrentControlSet
            Services
                DriverName
                        Type
                        Group
                        ErrorControl
                        Start
                        Tag
                    Parameters
                        Device0
                                Interrupt
                                Port
                                DMA Channel
                                (Etc.)
                        Device1
                                Interrupt
                                Port
                                DMA Channel
                                (Etc.)
```

The Service Control Manager creates and maintains *DriverName*, along with the **Type**, **Group**, **ErrorControl**, **Start**, and **Tag** subkeys. User-mode multimedia drivers provide the **Parameters** and underlying subkeys, as necessary. Use the **Interrupt**, **Port**, and **DMA Channel** value names to store interrupt levels, port addresses, and DMA channels.

Drivers can create and modify registry key values by calling Win32 functions, or by using functions provided in the driver support libraries. The support libraries are described in the chapters discussing the various driver types. The chapters are listed in Types of Multimedia Devices.

Storing User Configuration Parameters

Some multimedia drivers allow each user to specify configuration options. For example, a video capture driver might allow each user to specify video display color values. User-mode drivers should obtain user configuration parameter values and store them in the registry under a subkey of the registry path **HKEY_CURRENT_USER\Software\Microsoft\Multimedia**. For more information about storing user configuration parameters, see the chapters discussing the various driver types. Those chapters are listed in Types of Multimedia Devices.

Multimedia Driver Reference

This section provides the following topics:

- Messages, Multimedia Drivers
- Structures, Multimedia Drivers
- Functions, Multimedia Drivers

The messages, structures, and functions described in this section are common to multimedia drivers for all device types. For information about additional messages, structures, and functions for specific device types, see the chapters discussing the various driver types. Those chapters are listed in Types of Multimedia Devices.

Messages, Multimedia Drivers

This section describes the standard driver messages that are received by user-mode multimedia drivers. The messages are listed in alphabetic order, and are defined in *mmsystem.h* or *mmddk.h*.

DRV_CLOSE

The DRV_CLOSE message requests a user-mode multimedia driver to close the specified driver instance.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
DRV_CLOSE

lParam1
Contains the *lParam1* parameter from the **CloseDriver** function. Currently not used. Set to zero.

lParam2
Contains the *lParam2* parameter from the **CloseDriver** function. Currently not used. Set to zero.

Return Value

The driver should return a nonzero value if the operation succeeds. Otherwise it should return zero.

Comments

The DRV_CLOSE message is one of the standard driver messages. A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameter values.

When a driver receives a DRV_CLOSE message, it should close the specified driver instance. Other driver instances might still be open.

DRV_CONFIGURE

The DRV_CONFIGURE message requests a user-mode multimedia driver to display a dialog box that allows administrators to modify the driver's hardware configuration parameters.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
DRV_CONFIGURE
lParam1
Handle to the parent window the driver should use when creating a configuration dialog box.
lParam2
If not null, specifies the address of a DRVCONFIGINFO structure.

Return Value

The driver should provide one of the following return values:

DRVCNF_CANCEL	The user canceled the configuration dialog box.
DRVCNF_OK	The configuration operation was successful.
DRVCNF_RESTART	The configuration operation was successful. The new configuration does not take effect until Windows NT is restarted.

Comments

The DRV_CONFIGURE message is one of the standard driver messages. A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Typically, this message is sent by the Control Panel's Multimedia applet.

Drivers display a dialog box to obtain configuration parameters from the system administrator. Your driver must confirm that the client has Administrator privilege.

Configuration parameters typically include information the kernel-mode driver needs in order to access the hardware, such as an interrupt number, DMA channel, and port address. After obtaining this information from the dialog box, the user-mode driver stores it in the registry, as described in Storing Hardware Configuration Parameters, where it is accessible to the kernel-mode driver.

Drivers receive a DRV_OPEN message before receiving DRV_CONFIGURE.

Some drivers combine installation and configuration operations into one step and perform them upon receipt of either a DRV_INSTALL or a DRV_CONFIGURE message.

If the driver returns DRVCNF_RESTART, you can assume that the caller will display a message telling the administrator to restart Windows NT.

You can assume that the Control Panel's Multimedia applet will not install a driver that cannot be configured. When installing a driver, the Multimedia applet sends a DRV_CONFIGURE message immediately after sending DRV_INSTALL. If the driver returns DRVCNF_CANCEL in response to DRV_CONFIGURE, the driver is not installed.

DRV_DISABLE

The DRV_DISABLE message causes a Windows NT user-mode multimedia driver to just return a nonzero value (see the following **Comments** section).

Parameters

dwDriverID
Driver instance identifier. This value is zero for the DRV_DISABLE message.
hDriver
Driver handle.
uMsg
DRV_DISABLE
lParam1
Not used. Set to zero.
lParam2
Not used. Set to zero.

Return Value

The driver should return a nonzero value.

Comments

The DRV_DISABLE message is one of the standard driver messages. A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameter values.

Windows 95 drivers respond to the DRV_DISABLE message by disabling hardware. Because hardware access under Windows NT is controlled by kernel-mode drivers, Windows NT user-mode drivers do not perform any operations when they receive a DRV_DISABLE message.

DRV_ENABLE

The DRV_ENABLE message causes a Windows NT user-mode multimedia driver to just return a nonzero value (see **Comments** section below).

Parameters

dwDriverID
Driver instance identifier. This value is zero for the DRV_ENABLE message.

hDriver
Driver handle.

uMsg
DRV_ENABLE

lParam1
Not used. Set to zero.

lParam2
Not used. Set to zero.

Return Value

The driver should return a nonzero value.

Comments

The DRV_ENABLE message is one of the standard driver messages. A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameter values.

Windows 95 drivers respond to the DRV_ENABLE message by enabling hardware. Because hardware access under Windows NT is controlled by kernel-mode drivers, Windows NT user-mode drivers do not perform any operations when they receive a DRV_ENABLE message.

DRV_FREE

The DRV_FREE message requests a user-mode multimedia driver to perform pre-unload operations.

Parameters

dwDriverID
Driver instance identifier. This value is zero for the DRV_FREE message.

hDriver
Driver handle.

uMsg
DRV_FREE

lParam1
Not used. Set to zero.

lParam2
Not used. Set to zero.

Return Value

The driver should return a nonzero value if the operation succeeds. Otherwise it should return zero.

Comments

The DRV_FREE message is one of the standard driver messages. A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameter values.

User-mode drivers receive the DRV_FREE message just prior to being freed (unloaded) from memory. It is the last message a driver receives before being unloaded. The driver should release acquired system resources.

Drivers receive DRV_CLOSE and DRV_DISABLE messages before receiving DRV_FREE.

DRV_INSTALL

The DRV_INSTALL message requests a user-mode multimedia driver to allow a system administrator to perform installation operations.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
DRV_INSTALL

lParam1
Specifies the handle to the parent window the driver should use when creating a configuration dialog box.

lParam2
If not null, specifies the address of a DRVCONFIGINFO structure.

Return Value

Drivers provide one of the following return values:

DRVCNF_CANCEL	The installation operation should be canceled.
DRVCNF_OK	The installation operation was successful.
DRVCNF_RESTART	The installation operation was successful. The installation does not take effect until Windows NT is restarted.

Comments

The DRV_INSTALL message is one of the standard driver messages. A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Typically, this message is sent by the Control Panel's Multimedia applet.

Drivers receive DRV_LOAD, DRV_ENABLE, and DRV_OPEN messages before receiving DRV_INSTALL.

Installation operations include installing a kernel-mode driver, and creating Windows NT Registry keys along with their default values. Windows NT only allows users with Administrator privilege to install kernel-mode drivers, as discussed in Installing a Kernel-Mode Multimedia Driver.

Some drivers combine installation and configuration operations into one step and perform them upon receipt of either a DRV_INSTALL or a DRV_CONFIGURE message.

If the driver returns DRVCNF_RESTART, you can assume that the caller will display a message telling the administrator to restart Windows NT.

For more information about driver installation, see Installing Multimedia Drivers.

DRV_LOAD

The DRV_LOAD message requests a user-mode multimedia driver to perform post-load operations.

Parameters

dwDriverID
Specifies the driver instance identifier. This value is zero for the DRV_LOAD message.

hDriver
Driver handle.

uMsg
DRV_LOAD

lParam1
Not used. Set to zero.

lParam2
Not used. Set to zero.

Return Value

The driver should return a nonzero value if the operation succeeds. Otherwise it should return zero, which causes *winmm.dll* to unload the driver.

Comments

The DRV_LOAD message is one of the standard driver messages. A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameter values.

A user mode driver is not loaded until the first time a client attempts to open a driver instance. Immediately after being loaded, a driver receives a DRV_LOAD message so it can handle load-time activities. These activities might include initializing libraries or loading additional resources.

After a user-mode driver instance has been opened, the driver remains loaded until all instances have been closed.

See Also

DRV_FREE

DRV_OPEN

The DRV_OPEN message requests a user-mode multimedia driver to open a driver instance.

Parameters

dwDriverID
Specifies the driver instance identifier. This value is zero for the DRV_OPEN message.

hDriver
Driver handle.

uMsg
DRV_OPEN

lParam1
Address of a string containing any characters following the driver's file name in the Windows NT Registry.

lParam2
Typically an interface-specific data structure. For example, MCI drivers received the address of an MCI data structure.

Return Value

The driver should return a nonzero value if the operation succeeds. Otherwise it should return

zero. (See the following **Comments** section.)

Comments

The DRV_OPEN message is one of the standard driver messages. A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameter values.

The nonzero return value is saved by *winmm.dll* and used as the *dwDriverID* input value for subsequent calls to **DriverProc**. Because multiple driver instances can be opened simultaneously, drivers typically use this value to identify driver instance data. For example, the value could be an index into a driver-defined array, where each array element is a local, dynamically allocated structure containing driver instance data. When the driver receives subsequent calls to **DriverProc** for the open instance, it can use the *dwDriverID* value to determine which set of instance data to use.

Drivers receive DRV_LOAD and DRV_ENABLE messages before receiving DRV_OPEN.

Audio device drivers receive customized driver messages for opening driver instances. These drivers can ignore DRV_OPEN.

DRV_PNPINSTALL

The DRV_PNPINSTALL message requests a user-mode multimedia driver to allow a system administrator to perform installation operations, using Plug and Play configuration information.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
DRV_PNPINSTALL

lParam1
Handle to a device information set. The handle type is HDEVINFO, which is defined in *setupapi.h*.

lParam2
Pointer to an SP_DEVINFO_DATA structure.

Return Value

Drivers provide one of the following return values:

DRVCNF_CANCEL	The installation operation should be canceled.
DRVCNF_OK	The installation operation was successful.
DRVCNF_RESTART	The installation operation was successful. The installation does not take effect until Windows NT is restarted.

Comments

The DRV_PNPINSTALL message is one of the standard driver messages. A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Typically, this message is sent by the Media Class Installer, which is included in the Control Panel's Multimedia applet.

If the system provides Plug and Play capabilities, the driver receives this message instead of DRV_INSTALL. The driver uses the received *lParam1* and *lParam2* values as inputs to the **SetupDi**-prefixed device installation functions provided by *setupapi.dll*. For descriptions of the device installation functions, see the *Programmer's Guide*.

For more information about responding to the DRV_PNPINSTALL and DRV_CONFIGURE messages on a Windows NT system providing Plug and Play capabilities, see the user-mode

driver source code for the Creative Labs Sound Blaster, which is one of the sample audio drivers.

DRV_QUERYCONFIGURE

The DRV_QUERYCONFIGURE message requests a user-mode multimedia driver to return a value indicating whether it provides modifiable operating parameters.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
DRV_QUERYCONFIGURE

lParam1
Not used. Set to zero.

lParam2
Not used. Set to zero.

Return Value

If the driver supports modifiable parameters, it should return a nonzero value. Otherwise it should return zero.

Comments

The DRV_QUERYCONFIGURE message is one of the standard driver messages. A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Typically, this message is sent by the Control Panel's Multimedia applet.

Drivers receive a DRV_OPEN message before receiving DRV_QUERYCONFIGURE.

If a driver provides modifiable parameters, it displays a dialog box when it receives a DRV_CONFIGURE command.

DRV_REMOVE

The DRV_REMOVE message requests a user-mode multimedia driver to allow a system administrator to perform removal operations.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
DRV_REMOVE

lParam1
Not used. Set to zero.

lParam2
Not used. Set to zero.

Return Value

Drivers provide one of the following return values:

DRVCNF_CANCEL	The removal operation failed.
DRVCNF_OK	The removal operation was successful.

DRVCNF_RESTART	The removal operation was successful. The removal does not take effect until Windows NT is restarted.

Comments

The DRV_REMOVE message is one of the standard driver messages. A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Typically, this message is sent by the Control Panel's Multimedia applet.

Removal is the opposite of installation. Removal operations might include removing a kernel-mode driver and deleting Windows NT Registry keys that were created during installation.

Drivers receive a DRV_OPEN message before receiving DRV_REMOVE.

If the driver returns DRVCNF_RESTART, you can assume that the caller will display a message telling the administrator to restart Windows NT.

See Also

DRV_INSTALL

Structures, Multimedia Drivers

This section describes the structures used by user-mode multimedia drivers.

DRVCONFIGINFO

```
typedef struct tagDRVCONFIGINFO {
    DWORD dwDCISize;
    LPCWSTR lpszDCISectionName;
    LPCWSTR lpszDCIAliasName;
} DRVCONFIGINFO;
```

The DRVCONFIGINFO structure, defined in *mmsystem.h*, is used as an input argument with the DRV_INSTALL and DRV_CONFIGURE messages.

Members

dwDCISize
Specifies the size of the DRVCONFIGINFO structure.

lpszDCISectionName
Specifies a registry key name. This is always **Drivers32**.

lpszDCIAliasName
Specifies a driver alias.

Comments

For information about the **\Drivers32** registry key name and driver aliases, see Using *oemsetup.inf* Files with Multimedia Drivers.

The alias is used in the registry as a value name. The value assigned to this value name is the user-mode driver's file name.

Functions, Multimedia Drivers

This section describes functions that are either called by or exported by all multimedia drivers. The functions are listed in alphabetic order.

DefDriverProc

LRESULT WINAPI
DefDriverProc (

DWORD *dwDriverID*,
HDRVR *hDriver*,
UINT *uMsg*,
LPARAM *lParam1*,
LPARAM *lParam2*
);

The **DefDriverProc** function is called by user-mode multimedia drivers to handle messages not processed by the driver's **DriverProc** function. The function is defined in *winmm.dll*.

Parameters

dwDriverID
Value passed to the driver as the *dwDriverID* argument to **DriverProc**.

hDriver
Value passed to the driver as the *hDriver* argument to **DriverProc**.

uMsg
Value passed to the driver as the *uMsg* argument to **DriverProc**.

lParam1
Value passed to the driver as the *lParam1* argument to **DriverProc**.

lParam2
Value passed to the driver as the *lParam2* argument to **DriverProc**.

Return Value

The **DefDriverProc** function returns a value that is based on the received message. Return values are shown in the following table.

Message	Return Value
DRV_LOAD	1
DRV_FREE	1
DRV_ENABLE	1
DRV_DISABLE	1
DRV_INSTALL	DRV_OK
DRV_REMOVE	DRV_OK
All other messages.	0

Comments

A user-mode driver calls **DefDriverProc** by passing it the same arguments the driver received as input to its **DriverProc** function.

Typically, a user-mode driver's **DriverProc** function assigns supported *uMsg* values to C-language **case** statement arguments, and associates **DefDriverProc** with a **default** statement. Refer to the sample drivers for examples. The value returned from **DefDriverProc** should be used as the return value for **DriverProc**.

DriverCallback

BOOL APIENTRY
DriverCallback(
DWORD *dwCallBack*,
DWORD *dwFlags*,
HDRVR *hDriver*,
DWORD *dwMsg*,
DWORD *dwInstance*,
DWORD *dwParam1*,
DWORD *dwParam2*
);

The **DriverCallback** function is used by user-mode drivers to send a callback message to a client application.

Parameters

dwCallBack

The target for the callback message. Can be one of:

- An event handle
- A callback function address
- A thread identifier
- A window handle

The application specifies the type of callback target when opening a driver instance. The *dwFlags* parameter indicates the type of value stored in *dwCallback*.

dwFlags

One of the following flags, defined in *mmddk.h*, indicating the type of callback target:

DCB_EVENT	Equivalent to HIWORD(CALLBACK_EVENT). Indicates *dwCallback* contains an event handle. Code in *winmm.dll* calls the Win32 **SetEvent** function.
DCB_FUNCTION	Equivalent to HIWORD(CALLBACK_FUNCTION). Indicates *dwCallback* contains a function address. Code in *winmm.dll* calls the function.
DCB_TASK	Equivalent to HIWORD(CALLBACK_THREAD). Indicates *dwCallback* contains a thread identifer. Code in *winmm.dll* calls the Win32 **PostThreadMessage** function to post a WM_USER message.
DCB_WINDOW	Equivalent to HIWORD(CALLBACK_WINDOW). Indicates *dwCallback* contains a window handle. Code in *winmm.dll* calls the Win32 **PostMessage** function.

The CALLBACK_EVENT, CALLBACK_FUNCTION, CALLBACK_THREAD, and CALLBACK_WINDOW flags, referred to in the preceding table, are longword values used by applications when calling Win32 API functions that open multimedia drivers, such as **midiOutOpen**, **waveOutOpen**, or **videoStreamInit**.

hDriver

The driver handle that the driver received with the DRV_OPEN message.

dwMsg

A message to send to the application. Ignored if *dwFlags* is DCB_EVENT or DCB_TASK. The messages that can be sent are unique for each type of multimedia device and are listed in the chapters for each device type.

dwInstance

An application-supplied value to be passed to a callback function. Ignored if *dwFlags* is DCB_EVENT, DCB_TASK, or DCB_WINDOW. Applications specify this value when calling Win32 API functions that open multimedia drivers, such as **midiOutOpen**, **waveOutOpen**, or **videoStreamInit**.

dwParam1

A message-dependent parameter. Ignored if *dwFlags* is DCB_EVENT or DCB_TASK.

dwParam2

A message-dependent parameter. Ignored if *dwFlags* is DCB_EVENT, DCB_TASK, or DCB_WINDOW.

Return Value

Returns FALSE if *dwCallback* is NULL, if *dwFlags* is invalid, or if the message cannot be queued. Otherwise returns TRUE.

Comments

User-mode drivers call the **DriverCallback** function to deliver callback messages to applications that have requested them. Applications request delivery of callback messages by specifying a callback target when they open a driver instance. Win32 API functions that allow applications to specify a callback target include **midiOutOpen**, **waveOutOpen**, **videoStreamInit**, and others.

DriverEntry for Multimedia Drivers

NTSTATUS
DriverEntry(
PDRIVER_OBJECT *pDriverObject*,
PUNICODE_STRING *RegistryPathName*
);

The **DriverEntry** function is a kernel-mode driver's entry point. For kernel-mode multimedia drivers, the **DriverEntry** function's two parameters are defined as shown. (Other types of kernel-mode drivers might define the function's parameters differently.)

Parameters

pDriverObject
Pointer to a DRIVER_OBJECT structure.

RegistryPathName
Pointer to the registry path that leads to the kernel-mode driver's subkey. See the following **Comments** section.

Return Value

Returns STATUS_SUCCESS if initialization operations succeed. Otherwise the function returns one of the error codes defined in *ntstatus.h* that fail the NT_SUCCESS macro.

Comments

Windows NT calls a kernel-mode driver's **DriverEntry** function immediately after it has loaded the driver.

The registry path pointed to by *RegistryPathName* is **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***drivername*, where *drivername* is the name that the user-mode driver specified when it installed the kernel-mode driver. Drivers use this path for storing hardware configuration parameters.

For more information about **DriverEntry**, see **DriverEntry** in Kernel-Mode Multimedia Drivers.

DriverProc

LRESULT
DriverProc (
DWORD *dwDriverID*,
HDRVR *hDriver*,
UINT *uMsg*,
LPARAM *lParam1*,
LPARAM *lParam2*
);

The **DriverProc** function is a user-mode driver's entry point.

Parameters

dwDriverID
Instance identifier. This value is message-dependent.

hDriver
Driver handle.

uMsg
One of the standard driver messages or one of the customized driver messages.

lParam1
Message-dependent parameter.
lParam2
Message-dependent parameter.

Return Value

DriverProc returns a value that is dependent upon the message.

Comments

All user-mode multimedia drivers must define a **DriverProc** function. The **DriverProc** function name must be exported in a module definition (*.def*) file.

The meanings of the *dwDriverID*, *lParam1*, and *lParam2* parameters are dependent upon the message that the driver receives as the *uMsg* parameter value. Refer to the description of each message to determine the meanings of these parameters.

For more information about **DriverProc**, see User-Mode Driver Entry Points.

MCI Drivers

The following topics explain how to write a Windows NT® multimedia driver that supports the Media Control Interface (MCI):

- Introduction to MCI
- Introduction to MCI Drivers
- Designing an MCI Driver
- Creating Customized MCI Commands
- MCI Reference

For a general discussion of multimedia drivers, see Introduction to Multimedia Drivers.

Introduction to MCI

The Media Control Interface (MCI) provides a convenient, common API that client applications can use for controlling all multimedia hardware. MCI is made up of the following components:

- An API available to applications, consisting of callable functions plus a window class and associated messages.
- MCI command parsing and dispatching routines.
- A set of MCI drivers.

Under Windows NT, the command parsing and dispatching functions reside within *winmm.dll*. Client applications access these functions by calling **mci**-prefixed API functions provided by *winmm.dll*. Applications can also use the MCIWnd window class and its associated messages, which provide a simpler, higher-level interface than the **mci**-prefixed functions. (The MCI functions and window messages are described in the Win32 SDK.)

Usually when an application calls an MCI function, *winmm.dll* calls an MCI driver to perform the specified operation. Applications primarily call the **mciSendCommand** and **mciSendString** functions. Even when an application uses the MCIWnd window class, it is actually making indirect calls to these two functions. The **mciSendCommand** and **mciSendString** functions are described in the Win32 SDK.

When an application calls **mciSendCommand**, it specifies a device and an MCI *command message*, which is simply a predefined constant value, such as MCI_PLAY. The **mciSendCommand** function calls the MCI driver for the specified device and passes the command message. The driver processes the command and returns.

When an application calls **mciSendString**, it specifies an MCI *command string*, which is a text string describing the command and device, such as "play videodisc1". In this case, an MCI command parser searches a set of MCI command tables to verify that the command is valid for the specified device. If the command is valid, the parser converts the command string to a command message, and the message is passed to the appropriate MCI driver. The driver processes the command and returns.

A unique data structure is defined for each MCI command. These MCI data structures have a standardized format and are used for passing information to and from MCI drivers. Applications calling **mciSendCommand** reference the structures directly. Applications calling **mciSendString** do not reference the structures. Instead, *winmm.dll* converts input strings into structure member values. Before the call returns, *winmm.dll* extracts output values from the structure and converts them to output strings.

Introduction to MCI Drivers

MCI drivers are installable, user-mode device drivers that process MCI commands. Like all other user-mode drivers under Windows NT, MCI drivers are DLLs. For more information about user-mode drivers under Windows NT, see Introduction to Multimedia Drivers.

MCI drivers can initiate hardware operations by either of two methods:

1. They can call kernel-mode drivers using Win32 API functions such as **CreateFile**, **ReadFile**, and **WriteFile**, as described in Introduction to Multimedia Drivers.
2. They can call other user-mode drivers, using Win32 multimedia functions described in the Win32 SDK. For example, a driver implementing an MCI interface for MIDI sequencers can call the Win32 MIDI functions, causing a user-mode audio driver to be accessed.

Like all other types of installable drivers, MCI drivers must define a **DriverProc** function. This function must handle the standard driver messages. Additionally, it must handle MCI messages. For more information, see **DriverProc** in MCI Drivers.

The following topics provide more introductory information about MCI drivers:

- MCI Command Tables
- MCI Command Types
- MCI Device Types
- Simple and Compound Devices
- Sample MCI Driver

MCI Command Tables

Client applications can specify MCI commands as either strings or messages, but an MCI driver's **DriverProc** function only accepts messages. Therefore, it is necessary to provide command tables, which the MCI command parser uses to translate command strings into messages. Furthermore, most commands can accept modifiers, so the MCI parser translates string modifiers into message arguments that MCI drivers can read.

There are three types of command tables:

- *Core command table*

  There is one core command table. It contains the commands that all MCI drivers must support. These include the *required* and *basic* MCI command types. The core command table is included in *winmm.dll*.

- *Device-type command tables*

  There are command tables for several MCI device types. Each of these tables contains commands that are common to a particular multimedia device type, such as "videodisc" or "waveaudio". Several device-type tables are included in *winmm.dll*. Others can be included as separate files, with a filename extension of *.mci*.

- *Device-specific command tables*

  There can be any number of device-specific command tables. If a particular device requires its own set of commands, a unique command table must be provided. These tables can exist as separate files, with a filename extension of *.mci*. They can also be defined within a particular driver's resource file and linked to the driver.

Command tables are compiled by the Microsoft Windows Resource Compiler, which is described in the Win32 SDK.

Device-type command tables take precedence over the core command table. Likewise, device-specific command tables take precedence over device-type command tables. And finally, device-specific tables that reside in separate files take precedence over those that are linked to drivers. This scheme allows the core commands and device-type commands to be modified. In fact, it is common for a device-type or a device-specific table to redefine a core command in order to provide an expanded set of command modifiers.

For details about the contents of command tables and their associated data structures, see Creating Customized MCI Commands.

MCI Command Types

There are four types of MCI commands:

1. *System* commands
2. *Required* commands
3. *Basic* commands
4. *Extended* commands

System commands are not passed to an MCI driver. They are processed within *winmm.dll*.

Required commands are defined in the core command table. *MCI drivers must recognize and process all required commands*.

Basic commands are also defined in the core command table. *MCI drivers must recognize all basic commands*. However, if a basic command is not relevant to a particular device, the driver can simply return MCI_UNSUPPORTED_FUNCTION for that command.

Extended commands are those that are included in the device-type or device-specific command tables. Extended commands can either be modifications of the basic and required commands, or they can be new commands. An MCI driver only needs to recognize extended commands that are relevant for its device.

The following table lists the system, required, and basic commands. Extended commands are described within the Win32 SDK.

Command Type	Command String	Command Message
System Commands	**break**	MCI_BREAK
	sound	MCI_SOUND
	sysinfo	MCI_SYSINFO
Required Commands	**capability**	MCI_GETDEVCAPS
	close	MCI_CLOSE (Driver receives MCI_CLOSE_DRIVER.)
	info	MCI_INFO
	open	MCI_OPEN (Driver receives

		MCI_OPEN_DRIVER.)
	status	MCI_STATUS
Basic Commands	**load**	MCI_LOAD
	pause	MCI_PAUSE
	play	MCI_PLAY
	record	MCI_RECORD
	resume	MCI_RESUME
	save	MCI_SAVE
	seek	MCI_SEEK
	set	MCI_SET
	stop	MCI_STOP

All MCI commands are described in the Win32 SDK. To find the definition of a command, use the Win32 SDK keyword index to search for the command message (for example, MCI_SYSINFO).

MCI Device Types

Devices with similar properties are grouped together in categories known as *device types*. All of the devices belonging to a particular type respond to a common set of MCI commands. Each set of commands is contained in a separate MCI command table.

The following table lists device types defined by Microsoft. The table includes both string names and constants. Constants are defined in *mmsystem.h*.

String	Constant
animation	MCI_DEVTYPE_ANIMATION
cdaudio	MCI_DEVTYPE_CD_AUDIO
dat	MCI_DEVTYPE_DAT
digitalvideo	MCI_DEVTYPE_DIGITAL_VIDEO
overlay	MCI_DEVTYPE_OVERLAY
scanner	MCI_DEVTYPE_SCANNER
sequencer	MCI_DEVTYPE_SEQUENCER
vcr	MCI_DEVTYPE_VCR
videodisc	MCI_DEVTYPE_VIDEODISC
waveaudio	MCI_DEVTYPE_WAVEFORM_AUDIO

An MCI driver assigns a device to its appropriate type when it receives a DRV_OPEN message. (See Opening an MCI Driver.) This tells *winmm.dll* which command table to use. The driver uses the MCI_DEVTYPE_OTHER type for devices not belonging to any of the predefined types.

To find which MCI commands a driver must support for a particular device type, see the Win32 SDK. The Win32 SDK lists the command set associated with each device type. (Use the keyword index to search for a device type string, such as "videodisc".)

Simple and Compound Devices

For the purposes of MCI, devices are classified as either:

Simple devices
: Simple devices do not require a data file for playback. Videodisc players and compact disc (CD) audio players are examples of simple devices.

Compound 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 waveform files.

Sample MCI Driver

Source code for a sample MCI driver is included with this DDK. The MCIPIONR driver is the MCI device driver for the Pioneer 4200 videodisc player. Its source files can be found in *\ddk\src\mmedia\mcipionr*.

Designing an MCI Driver

This topic provides the following subtopics, which cover information you need to design an MCI driver:

- **DriverProc** in MCI Drivers
- Handling Standard **Driver** Messages
- Handling the MCI_NOTIFY Flag
- Handling the MCI_WAIT Flag
- Handling the MCI_TEST Flag
- Opening an MCI Driver
- Sharing A Device
- Storing Instance-Specific Information
- Providing Device Information
- Closing an MCI Driver
- Guidelines for Handling MCI Commands

If you are defining customized command tables, also see Creating Customized MCI Commands.

DriverProc in MCI Drivers

Like all other types of installable drivers, MCI drivers must define a **DriverProc** entry point. This function must handle the standard driver messages. Additionally, it must handle MCI messages. Within *winmm.dll*, application calls to **mciSendCommand** and **mciSendString** become calls to **DrvSendMessage**, which is described in the Win32 SDK and is the standard method for calling a driver's **DriverProc** function.

Like the standard messages, MCI messages are defined as constant values that can be used in a C-language **case** statement. These constants are defined in *mmsystem.h*.

This section also provides information on **DriverProc** parameters and **DriverProc** return values for MCI drivers.

DriverProc Parameters

The **DriverProc** function is defined as follows:

```
LRESULT WINAPI DriverProc (
    DWORD dwDriverID,
    HDRVR hDriver,
    UINT uMsg,
    LPARAM lParam1,
    LPARAM lParam2
    );
```

When *winmm.dll* passes MCI messages to **DriverProc**, the function parameters are used as follows:

dwDriverID
Contains the driver identifier created by MCI.

hDriver
Contains a handle to the device driver.

uMsg
Contains an MCI message constant value.

lParam1
Contains message-specific MCI flags.

lParam2
Points to a message-specific MCI data structure.

The *lParam1* and *lParam2* parameters generally are used to represent MCI command arguments. Some arguments can simply be represented as flags in *lParam1*. Other arguments are passed as members of a data structure pointed to by *lParam2*. In this latter case, *lParam1* flags are used to indicate the presence of valid members within the structure pointed to by *lParam2*. This structure is also used for returning information to the application.

There is a unique *lParam2* data structure for each MCI message. Additionally, driver developers can define customized structures for extended commands. All of the data structures provided by Microsoft are defined in *mmsystem.h* and described in the Win32 SDK. Refer to Creating New MCI Command Structures to learn how to create custom structures.

DriverProc Return Values

When **DriverProc** receives an MCI message, its double-word return value must be assigned as follows:

- If no errors occur, return zero.
- If an error occurs, return one of the MCI error return values in the low word. (If the error is device-specific, *winmm.dll* places the device ID in the high word.)

The MCI error return values are defined in *mmsystem.h*, and are prefixed with "MCIERR_". For a definition of each MCI error return value, see the Win32 SDK. Note that an additional set of error return values is defined for **mciSendString**. These additional values are returned by *winmm.dll*, not by MCI drivers.

When **DriverProc** receives one of the standard driver messages, its return value must be zero *if an error occurs*. Note that this is opposite to the situation for MCI messages.

Handling Standard Driver Messages

Handling standard driver messages is described in Introduction to Multimedia Drivers. Also see this chapter's sections entitled Opening an MCI Driver and Closing an MCI Driver, which discuss some of the standard messages.

Handling the MCI_NOTIFY Flag

All MCI command tables must include the MCI_NOTIFY flag for all commands. Applications use this flag to request notification when an operation has completed.

Normally when an MCI driver receives a command, it should initiate the associated operation and then return control to the calling application. For example, if an application sends an MCI_SEEK command, the driver should start the seek operation and immediately return. The application is not notified when the operation completes.

As an option, an application sending any MCI command can request to be notified when the command operation has completed. To do this, the application specifies the following arguments along with the command:

- The MCI_NOTIFY flag. The driver receives this flag in the *lParam1* argument to **DriverProc**.
- A window handle. The driver receives this handle as the **dwCallback** member of any structure whose address is passed as the *lParam2* argument to **DriverProc**.

When a driver receives an MCI command that includes the MCI_NOTIFY flag, the driver must start the specified operation and then return. When the operation completes, the driver must call **mciDriverNotify** in *winmm.dll*. The **mciDriverNotify** function enqueues an MM_MCINOTIFY message to the window specified in **dwCallback**. The application's window procedure must check for and process the MM_MCINOTIFY message.

Handling the MCI_WAIT Flag

All MCI command tables must include the MCI_WAIT flag for all commands. Applications use this flag to request the driver to complete the operation before returning control to the application.

When an MCI driver receives a command, by default it should start the operation and then return control to the calling application. The driver should not wait for the operation to complete before returning. For example, if an application sends an MCI_SEEK command, the driver should start the seek operation and immediately return.

Optionally, an application sending any MCI command can request the driver to wait until the associated operation is complete before returning. The application makes this request by including the MCI_WAIT flag as a command argument.

The driver receives the MCI_WAIT flag in the *lParam1* argument to **DriverProc**. If the flag is present, then the driver must initiate the requested operation and then wait for it to complete before returning.

Yielding

Since the waiting time can be potentially long, a user must be allowed to interrupt the operation. By default, MCI provides a *yield* routine that checks for a break key. The default break key is CTRL+BREAK. An application can change the break key by sending the MCI_BREAK command. It can also replace the default yield routine with a customized one by calling the **mciSetYieldProc** function in *winmm.dll*. The yield routine returns a nonzero value if the driver should terminate the current operation.

While the driver is waiting for the requested operation to complete, it must periodically call the **mciDriverYield** function in *winmm.dll*. This function calls the currently selected yield routine and returns its return value. If this value is nonzero, the driver must stop the operation.

Handling the MCI_TEST Flag

The MCI command tables for some device types, including MCI_DEVTYPE_VCR and MCI_DEVTYPE_DIGITAL_VIDEO types, provide support for the MCI_TEST flag. The driver receives the MCI_TEST flag in the *lParam1* argument to **DriverProc**.

If an application includes this flag with a command, the driver does not initiate the specified operation. Instead, it determines if the operation is currently available. If the operation is available, the driver returns zero. Otherwise it returns MCIERR_NONAPPLICABLE_FUNCTION. For example, a VCR driver might not allow an MCI_INDEX command while a seek operation is in progress. (Of course, in such a case the driver should return MCIERR_NONAPPLICABLE_FUNCTION even if the MCI_TEST flag is not specified.)

Opening an MCI Driver

A client application using MCI opens a driver by sending an MCI_OPEN message. This message is intercepted by *winmm.dll*, which first loads the appropriate MCI driver into the application's address space, and then sends it the following messages, in the order listed:

1. DRV_LOAD
2. DRV_ENABLE
3. DRV_OPEN
4. MCI_OPEN_DRIVER

Note that the driver does *not* receive the MCI_OPEN message. Also be aware that DRV_LOAD and DRV_ENABLE are sent only if the driver was not previously loaded.

For more information, see the following topics:

- Handling DRV_LOAD
- Handling DRV_ENABLE
- Handling DRV_OPEN
- Handling MCI_OPEN_DRIVER

Handling DRV_LOAD

If your driver provides a customized command table, load it by calling **mciLoadCommandResource** when **DriverProc** receives the DRV_LOAD message. For more information on handling DRV_LOAD, see Introduction to Multimedia Drivers.

Handling DRV_ENABLE

For information on handling DRV_ENABLE, see Introduction to Multimedia Drivers.

Handling DRV_OPEN

When *winmm.dll* sends the DRV_OPEN message, it sets the **DriverProc** parameters as follows:

dwDriverID
Zero.

hDriver
The driver's handle.

uMsg
DRV_OPEN.

lParam1
Contains a pointer to a zero-terminated string. The string contains any characters that follow the filename in the system registry.

lParam2
Pointer to an MCI_OPEN_DRIVER_PARMS structure.

The value of the **lpstrParams** member of MCI_OPEN_DRIVER_PARMS is the same as the value of *lParam1*.

Before **DriverProc** returns, you must:

- Set the **wCustomCommandTable** member of MCI_OPEN_DRIVER_PARMS. If the driver is using a custom command table, this member must contain the handle returned by **mciLoadCommandResource**. Otherwise use MCI_NO_COMMAND_TABLE, defined in *mmddk.h*.
- Set the **wType** member of MCI_OPEN_DRIVER_PARMS to one of the defined MCI device types. If the device does not belong to any of the defined types, use MCI_DEVTYPE_OTHER.
- Assign the **DriverProc** function's return value to be the contents of the **wDeviceID** member of MCI_OPEN_DRIVER_PARMS. However, if you encounter errors during the process of opening the device, you should assign a return value of zero instead.

The driver can also perform instance-specific operations for the device being opened.

Handling MCI_OPEN_DRIVER

The first MCI message a driver receives is MCI_OPEN_DRIVER. For this message, **DriverProc** parameters are set as follows:

dwDriverID

The driver ID value specified as the **DriverProc** return value for the DRV_OPEN message.

hDriver
The driver's handle.

uMsg
MCI_OPEN_DRIVER.

lParam1
Flags. (See MCI_OPEN in Win32 SDK.)

lParam2
Pointer to an MCI_OPEN_PARMS structure.

An application specifies the contents of *lParam1* and *lParam2* when it sends an MCI_OPEN command.

For a compound device, the driver should test the MCI_OPEN_ELEMENT flag. If set, it indicates that a pointer to an element pathname is available in the **lpstrElementName** member of MCI_OPEN_PARMS.

Drivers for compound devices sometimes receive MCI_OPEN_DRIVER messages with the MCI_OPEN_ELEMENT flag cleared. This situation can occur if the application opens the device only for the purpose of querying the device with an MCI_GETDEVCAPS or MCI_INFO command. Therefore, drivers for compound devices must allow the MCI_OPEN_DRIVER command to succeed if it is received without an element name, but they must provide a failure return for any command that requires an element, if the element name is missing.

The MCI_OPEN_ELEMENT_ID flag indicates that the **lpstrElementName** member of MCI_OPEN_PARMS contains a DWORD value instead of a string pointer. This allows you to define an element specifier as being something other than a file pathname. Use of this flag is not recommended.

Drivers can ignore the MCI_OPEN_TYPE, MCI_OPEN_TYPE_ID, and MCI_OPEN_ALIAS flags. These are handled within *winmm.dll*.

Drivers that support sharing a device must test the MCI_OPEN_SHAREABLE flag.

The driver can also perform instance-specific operations for the device or element being opened, such as storing instance-specific information.

When handling the MCI_NOTIFY flag with the MCI_OPEN_DRIVER command, a driver should return to the application only after it verifies that the open operation will probably succeed. For example, suppose that opening a device element requires reading a large file from a CD-ROM. Before returning to the application, the driver should confirm that the file is accessible and that enough memory can be allocated to load it. This avoids providing a successful (zero) return value to **DriverProc**, only to later have to deliver an MM_MCINOTIFY message with a status of MCI_NOTIFY_FAILURE.

Sharing A Device

To share a simple device, a driver creates a single context that can be shared by multiple applications. This context allows each application to reference and modify device characteristics set up by other applications. For example, a shared driver for a CD player should allow an application to issue a command ("resume", for example) that is based on the context of the last command received from any other application ("stop", for example).

Because each instance of a shared driver is created by a separate DRV_OPEN command, a separate driver ID is assigned to each instance.

For a compound device, the shareable object is the device element. Each element has a unique device ID. The driver allows multiple applications to share an element's context. Each application can reference and modify characteristics of the element that have been set up by other applications.

If a driver *can* share a device (or object), the decision about whether it *should* share the device (or

object) is based on the first MCI_OPEN_DRIVER command. If this command includes the MCI_OPEN_SHAREABLE flag, then the driver should mark the device (or object) as being shared. Each subsequent MCI_OPEN_DRIVER command received must also include the MCI_OPEN_SHAREABLE flag, or else the MCIERR_MUST_USE_SHAREABLE error value must be returned.

On the other hand, if the first MCI_OPEN_DRIVER command does not include the MCI_OPEN_SHAREABLE flag, then no subsequent MCI_OPEN_DRIVER command can include the MCI_OPEN_SHAREABLE flag either. If it does, the MCIERR_MUST_USE_SHAREABLE error value must be returned.

If an application specifies the MCI_OPEN_SHAREABLE flag but the driver does not allow device sharing, then the driver must return MCIERR_UNSUPPORTED_FUNCTION.

For compound devices, drivers can alternatively allow several applications to open a single element without specifying the MCI_OPEN_SHAREABLE flag. In this case, each application should possess its own context for the element. As a result, each application makes its own modifications to the element context without affecting any other application's context. Each application can move independently within a single element. Also, a single application can open a single element multiple times, providing multiple contexts for one element.

Storing Instance-Specific Information

It is sometimes necessary for a driver to save instance-specific information. Two functions supplied by *winmm.dll*, **mciSetDriverData** and **mciGetDriverData**, are useful for associating instance-specific information with driver ID's. You can use these functions to store and retrieve a longword for each assigned driver ID (that is, for each opened device or element). Drivers typically cast the longword to a pointer to a locally allocated structure.

Providing Device Information

This section discusses the MCI commands that request information. It also explains the process of returning information to applications. Finally, it provides guidelines for returning information.

MCI Commands that Request Information

Four MCI commands allow applications to obtain information about a device. Three of these commands are sent to MCI drivers. They are:

1. MCI_GETDEVCAPS
2. MCI_INFO
3. MCI_STATUS

A fourth command, MCI_SYSINFO, is handled within *winmm.dll.* This command is not sent to MCI drivers.

The MCI_GETDEVCAPS, MCI_INFO, and MCI_STATUS commands are similar to each other in the following ways:

- The driver receives a flag in the *lParam1* parameter of **DriverProc**, indicating the type of information being requested.
- The driver receives the address of a data structure in the *lParam2* parameter of **DriverProc**. This structure contains either a location or a pointer that the driver uses for returning the requested information.

An application can only request, and the driver can only return, one type of information per call. The MCI_GETDEVCAPS and MCI_STATUS commands return information as integer values. The MCI_INFO command returns information as strings (such as pathnames).

The types of information provided vary with the device type. See the descriptions of MCI_GETDEVCAPS, MCI_INFO, and MCI_STATUS in the Win32 SDK for information on the

flags that have been defined with each command, for each device type. Drivers that define extended command sets can define new flags appropriate to their devices. Drivers can also define additional commands for returning other types of information. For example, drivers for video device types support an MCI_WHERE command, which returns a RECT data type for describing a rectangle.

Returning Information to Applications

There are two standardized methods for returning information to applications. One method is used for returning integer values and the other is used for returning strings.

When a driver receives an MCI command that requests information, it also receives the address of a data structure in the *lParam2* parameter to **DriverProc**. A different structure is defined for each MCI command. Customized structures are defined for some device types. The structure's definition dictates which method is used for returning the requested information.

Following is the structure definition for MCI_STATUS_PARMS, which is used with the MCI_STATUS command to return an integer value:

```
typedef struct tagMCI_STATUS_PARMS {
    DWORD   dwCallback;
    DWORD   dwReturn;
    DWORD   dwItem;
    DWORD   dwTrack;
} MCI_STATUS_PARMS, *PMCI_STATUS_PARMS, FAR * LPMCI_STATUS_PARMS;
```

The MCI_STATUS_PARMS structure defines a DWORD-sized member called **dwReturn**. To return an integer value for MCI_STATUS, the driver places a longword value into **dwReturn**.

Following is the structure definition for MCI_INFO_PARMS, used with the MCI_INFO command to return a string:

```
typedef struct tagMCI_INFO_PARMS {
    DWORD   dwCallback;
    LPSTR   lpstrReturn;
    DWORD   dwRetSize;
} MCI_INFO_PARMS, FAR * LPMCI_INFO_PARMS;
```

The MCI_INFO_PARMS structure defines two members, **lpstrReturn** and **dwRetSize**. These members are used for returning a string value. In this case, the application places a string buffer pointer in **lpstrReturn** and a buffer size in **dwRetSize**. The driver copies the return string into the buffer.

Returning String Resource Indentifiers

It is important to remember that applications can communicate with MCI by using either strings or command constants. When an application calls **mciSendString**, it specifies commands in string form and expects information to be returned in string form. When an application calls **mciSendCommand** it specifies command constants and flag constants, and expects information to be returned in a command-specific data structure, which it references. MCI drivers sometimes need to return information in both formats, in order to support both interfaces.

Suppose an application uses the MCI_STATUS command to request the device's current mode, and it happens that the device is currently stopped. An application using **mciSendCommand** should be able to test the MCI_MODE_STOP flag value stored in the **dwReturn** member of an MCI_STATUS_PARMS structure. An application using **mciSendString** should receive the string "stopped" in its string buffer. The driver is responsible for returning both the flag and a string resource identifier. (String resources are discussed in the Win32 SDK.)

The proper way to return both a flag and a string resource identifier is to combine them in the **dwReturn** member, using the **MAKEMCIRESOURCE** macro. This macro concatenates two integers to make a single long integer. The resource identifier must be placed in the high word of the long integer.

If a driver returns a resource identifier, it must set the **DriverProc** return value to MCI_RESOURCE_RETURNED, as shown in the following example. (Notice that, in this case, the flag constant value and the string resource ID value are the same.)

```
wResource = MCI_MODE_STOP;
lpStatus->dwReturn = (DWORD)MAKEMCIRESOURCE(wResource, wResource);
dReturn = MCI_RESOURCE_RETURNED;
```

If the string resource is defined in a driver-specific resource file, the driver must also set MCI_RESOURCE_DRIVER in the return value, as shown in the following example of returning the "audio" status from an AVI driver. (Notice that, in this case, the flag constant value and the string resource ID value are not the same.)

```
lpStatus->dwReturn = (npMCI->dwFlags & MCIAVI_PLAYAUDIO) ?
                        (MAKEMCIRESOURCE(MCI_ON, MCI_ON_S)) :
                        (MAKEMCIRESOURCE(MCI_OFF, MCI_OFF_S));
return MCI_RESOURCE_RETURNED | MCI_RESOURCE_DRIVER;
```

Both MCI_RESOURCE_RETURNED and MCI_RESOURCE_DRIVER set bits in the return value's high word.

If the application used **mciSendString** to send the command, then *winmm.dll* checks the high word of the **DriverProc** return value. If MCI_RESOURCE_RETURNED is set, *winmm.dll* loads the string associated with the resource identifier and places it in the application's return buffer.

If the application used **mciSendCommand** to send the command and MCI_RESOURCE_RETURNED is set in the high word of the **DriverProc** return value, then *winmm.dll* just clears the high word before passing the return value to the application.

The following table contains the resource strings provided by *winmm.dll* for use by drivers. The resource identifiers and constants are defined within *mmsystem.h* and *mmddk.h*. (Sometimes the constant and resource ID are the same.) The strings are defined in a resource file that is part of *winmm.dll*. Driver developers can define additional strings within a driver-specific resource file. The driver calls **mciLoadCommandResource** to register the resource file with *winmm.dll*.

Constant	Resource ID	String
MCI_FALSE	MCI_FALSE	**false**
MCI_TRUE	MCI_TRUE	**true**
MCI_DEVTYPE_ANIMATION	MCI_DEVTYPE_ANIMATION	**animation**
MCI_DEVTYPE_CD_AUDIO	MCI_DEVTYPE_CD_AUDIO	**cdaudio**
MCI_DEVTYPE_DAT	MCI_DEVTYPE_DAT	**dat**
MCI_DEVTYPE_DIGITAL_VIDEO	MCI_DEVTYPE_DIGITAL_VIDEO	**digitalvideo**
MCI_DEVTYPE_OTHER	MCI_DEVTYPE_OTHER	**other**
MCI_DEVTYPE_OVERLAY	MCI_DEVTYPE_OVERLAY	**overlay**
MCI_DEVTYPE_SCANNER	MCI_DEVTYPE_SCANNER	**scanner**
MCI_DEVTYPE_SEQUENCER	MCI_DEVTYPE_SEQUENCER	**sequencer**
MCI_DEVTYPE_VCR	MCI_DEVTYPE_VCR	**vcr**
MCI_DEVTYPE_VIDEODISC	MCI_DEVTYPE_VIDEODISC	**videodisc**
MCI_DEVTYPE_WAVEFORM_AUDIO	MCI_DEVTYPE_WAVEFORM_AUDIO	**waveaudio**
MCI_FORMAT_BYTES	MCI_FORMAT_BYTES_S	**bytes**
MCI_FORMAT_FRAMES	MCI_FORMAT_FRAMES_S	**frames**
MCI_FORMAT_HMS	MCI_FORMAT_HMS_S	**hms**
MCI_FORMAT_MILLISECONDS	MCI_FORMAT_MILLISECONDS_S	**milliseconds**
MCI_FORMAT_MSF	MCI_FORMAT_MSF_S	**msf**
MCI_FORMAT_SAMPLES	MCI_FORMAT_SAMPLES_S	**samples**
MCI_FORMAT_SMPTE_24	MCI_FORMAT_SMPTE_24_S	**smpte 24**

MCI_FORMAT_SMPTE_25	MCI_FORMAT_SMPTE_25_S	**smpte 25**
MCI_FORMAT_SMPTE_30	MCI_FORMAT_SMPTE_30_S	**smpte 30**
MCI_FORMAT_SMPTE_30DROP	MCI_FORMAT_SMPTE_30DROP_S	**smpte 30 drop**
MCI_FORMAT_TMSF	MCI_FORMAT_TMSF_S	**tmsf**
MCI_MODE_NOT_READY	MCI_MODE_NOT_READY	**not ready**
MCI_MODE_OPEN	MCI_MODE_OPEN	**open**
MCI_MODE_PAUSE	MCI_MODE_PAUSE	**paused**
MCI_MODE_PLAY	MCI_MODE_PLAY	**playing**
MCI_MODE_RECORD	MCI_MODE_RECORD	**recording**
MCI_MODE_SEEK	MCI_MODE_SEEK	**seeking**
MCI_MODE_STOP	MCI_MODE_STOP	**stopped**
MCI_SEQ_DIV_PPQN	MCI_SEQ_DIV_PPQN	**PPQN**
MCI_SEQ_DIV_SMPTE_24	MCI_SEQ_DIV_SMPTE_24	**SMPTE 24 Frame**
MCI_SEQ_DIV_SMPTE_25	MCI_SEQ_DIV_SMPTE_25	**SMPTE 25 Frame**
MCI_SEQ_DIV_SMPTE_30	MCI_SEQ_DIV_SMPTE_30	**SMPTE 30 Frame**
MCI_SEQ_DIV_SMPTE_30DROP	MCI_SEQ_DIV_SMPTE_30DROP	**SMPTE 30 Drop Frame**
MCI_SEQ_FILE	MCI_SEQ_FILE_S	**file**
MCI_SEQ_FORMAT_SONGPTR	MCI_SEQ_FORMAT_SONGPTR_S	**song pointer**
MCI_SEQ_MIDI	MCI_SEQ_MIDI_S	**midi**
MCI_SEQ_NONE	MCI_SEQ_NONE_S	**none**
MCI_SEQ_SMPTE	MCI_SEQ_SMPTE_S	**smpte**
MCI_VD_FORMAT_TRACK	MCI_VD_FORMAT_TRACK_S	**track**
MCI_VD_MEDIA_CAV	MCI_VD_MEDIA_CAV	**CAV**
MCI_VD_MEDIA_CLV	MCI_VD_MEDIA_CLV	**CLV**
MCI_VD_MEDIA_OTHER	MCI_VD_MEDIA_OTHER	**other**
MCI_VD_MODE_PARK	MCI_VD_MODE_PARK	**parked**
MIDIMAPPER	MIDIMAPPER_S	**mapper**
WAVE_FORMAT_PCM	WAVE_FORMAT_PCM_S	**pcm**
WAVE_MAPPER	WAVE_MAPPER_S	**mapper**

Returning Integers

If an application requests information that is represented as an integer value, the driver just places the integer in the **dwReturn** member of the command's data structure. If the application used **mciSendString** to request the information, then *winmm.dll* converts the integer to a string and places the string in the application's return buffer.

The driver can request *winmm.dll* to insert colons into the integer string. A typical reason for inserting colons is returning time values. To request colon insertion, a driver assigns MCI_COLONIZED3_RETURN or MCI_COLONIZED4_RETURN to the **DriverProc** return value, as shown in the next example. These constants set bits in the return value's high word.

```
DWORD dwSeconds = dwFrames / CAV_FRAMES_PER_SECOND;
lpStatus->dwReturn = MCI_MAKE_HMS(dwSeconds / 3600,
                                  (dwSeconds % 3600) / 60,
                                  dwSeconds % 60);
return MCI_COLONIZED3_RETURN;
```

If the application used **mciSendString** to request the information, then *winmm.dll* treats each byte of the return value as a separate integer and inserts a colon between each integer in the string that is returned to the application. For an integer value of 0x01020304, specifying MCI_COLONIZED4_RETURN returns the string "4:3:2:1", and specifying MCI_COLONIZED3_RETURN returns "4:3:2".

One other special **DriverProc** return value, MCI_INTEGER_RETURNED, forces a command's returned information to be an integer even though the command's data structure defines the return type as a string. Microsoft uses this flag within *winmm.dll*, for support of the MCI_SYSINFO command. This command's MCI_SYSINFO_PARMS structure defines a string return, but if the requested information type is MCI_SYSINFO_QUANTITY, then an integer value is placed in the structure. (For more information, see the Win32 SDK.) If the application requested the information by calling **mciSendString**, *winmm.dll* converts the integer to a string.

Guidelines for Returning Information

Use the following guidelines for returning information when developing a new driver:

- Implement the MCI_GETDEVCAPS, MCI_INFO, and MCI_STATUS commands as they are described in the Win32 SDK. Be consistent with the implementations already provided with existing drivers.
- Remember that the MCI_GETDEVCAPS and MCI_STATUS commands return integer values, while MCI_INFO returns string values.
- Place strings requiring language translation in resource files. Return their resource identifiers as integer values, using MCI_GETDEVCAPS, MCI_STATUS, or a customized extended command. The strings returned by MCI_INFO contain information that does not require foreign language translation, such as file and path names, or media identifier strings.

Closing an MCI Driver

A client application using MCI closes a driver by sending an MCI_CLOSE message. This message is intercepted by *winmm.dll*, which sends the following messages to the appropriate MCI driver, in the order listed:

1. MCI_CLOSE_DRIVER
2. DRV_CLOSE
3. DRV_DISABLE
4. DRV_FREE

Note that the driver does *not* receive the MCI_CLOSE message. Also be aware that DRV_DISABLE and DRV_FREE are sent only if the calling application is the only one using the driver. After DRV_FREE is sent, *winmm.dll* unloads the driver from the application's address space.

For more information, see the following topics:

- Handling MCI_CLOSE_DRIVER
- Handling DRV_CLOSE
- Handling DRV_DISABLE
- Handling DRV_FREE

Handling MCI_CLOSE_DRIVER

When a driver receives an MCI_CLOSE_DRIVER command, it should close the object that is associated with the driver identifier received in *dwDriverID*. For a nonshared simple device, the object is the device. For a compound device, the object is an element.

If a device or element is being shared, then reception of an MCI_CLOSE_DRIVER message only

means that the driver should no longer allow the caller access to the device or element. The device or element should be closed only after there are no longer any applications using it.

Handling DRV_CLOSE

A driver should close a device only after all applications have finished using it. You might want to ignore the DRV_CLOSE command (just return with a nonzero return value) if the application that caused it to be sent is not the only one still using the driver.

Often, all of the steps required to close a device can be handled when a driver processes the MCI_CLOSE_DRIVER command. In such cases the driver can simply return a nonzero value when DRV_CLOSE is received.

Handling DRV_DISABLE

For information on handling DRV_DISABLE, see Introduction to Multimedia Drivers.

Handling DRV_FREE

If your driver provides a custom string table, unload it by calling **mciFreeCommandResource** when **DriverProc** receives the DRV_FREE message. For more information on handling DRV_FREE, see Introduction to Multimedia Drivers.

Guidelines for Handling MCI Commands

A developer's most important guideline is to *make sure the driver design adheres to the command descriptions and flag descriptions provided by the Win32 SDK*. For example, the description of MCI_PLAY says that the starting position is the current position and the ending position is the end of the medium, unless the application includes MCI_FROM and MCI_TO flags.

If you need to provide extensions to the basic commands, do so in a manner that is consistent with extended command sets that already exist.

If an application sends an MCI_PLAY command with MCI_NOTIFY set, subsequent MCI_PAUSE and MCI_RESUME commands should not cancel the notification request. The application should still be notified when the original play operation has completed. The MCI_PAUSE and MCI_RESUME commands just delay the completion of the MCI_PLAY operation; they do not cancel or modify it.

Drivers for compound devices can allow MCI_RECORD commands to be sent with the the MCI_OPEN_ELEMENT flag set, but with the element file name specified as a null-terminated, zero-length string. In this case the recorded data should be saved in temporary storage. On receipt of a subsequent MCI_SAVE command, the driver should move the temporary data into a permanent file. Alternatively, drivers can disallow MCI_RECORD commands without a valid filename, and return MCIERR_FILENAME_REQUIRED if the filename is not included.

Creating Customized MCI Commands

If MCI's required and basic command sets do not provide all of the functionality needed by applications to fully interact with a device, then a driver developer can create customized MCI commands by doing one or both of the following:

- Extending the required and basic commands as necessary, by adding additional input flags, redefining the command data structures, or both.
- Extending the command set by defining new commands and data structures.

You should match your MCI command extensions as closely as possible to existing command extensions. Refer to the Win32 SDK for descriptions of existing command extensions.

You define new or modified MCI commands by creating new MCI command tables. Use these

tables to define new commands and flags, and to redefine or extend the required or basic commands.

Creating new MCI command structures is also sometimes necessary. You must define a new structure if you are creating a new command, or if you are modifying an existing command in a way that requires additional structure members.

Creating New MCI Command Tables

You create a new MCI command table by:

- Defining MCI message constants for your commands, if they do not already exist.
- Defining flag constants for your command flags, if they do not already exist.
- Constructing a command table, using message strings, message constants, and flag constants.

Defining MCI Message Constants

If an MCI message constant does not already exist for your command in *mmsystem.h*, you must define one. Place this definition in an include file that is available both to your driver and to application developers. MCI message constants must be assigned values between the constants MCI_USER_MESSAGES and MCI_LAST, which are defined in *mmsystem.h*.

Note that message constants *cannot* be defined using an expression, such as "MCI_USER_MESSAGES + 1", because you are using the constant in a command table. Command tables are defined within resource files, using RCDATA type, and this type does not accept expressions. Define each messageconstant as a numeric value, such as:

```
#define MCI_MYCOMMAND  0xA01
```

Defining Flag Constants

You need to define new flag constants whenever you create new command modifiers, either for existing commands or for new commands.

Flags are passed to MCI drivers in the *lParam1* parameter to **DriverProc**. This is a DWORD parameter. The first 16 bits (0 through 15) of the DWORD are reserved by MCI. Driver developers can use bits 16 through 31 for customized flags. This means up to 16 different customized flags can be defined for each command. You should be sure that new flag definitions do not conflict with existing definitions.

Place new flag definitions in an include file that is available both to your driver and to application developers. When naming flags, include the command and a representation of your device type in the name. For example, flags for animation extensions to the "play" command are named as follows:

```
#define MCI_ANIM_PLAY_SPEED             0x00010000L
#define MCI_ANIM_PLAY_REVERSE           0x00020000L
#define MCI_ANIM_PLAY_FAST              0x00040000L
#define MCI_ANIM_PLAY_SLOW              0x00080000L
#define MCI_ANIM_PLAY_SCAN              0x00100000L
```

Constructing a Command Table

Command tables are defined as raw data resources which must be compiled using the Microsoft Windows Resource Compiler, discussed in the Win32 SDK. The command table can be created as a separate DLL file, with a file extension of *.mci*, or it can be linked to the driver.

If you place the command table in a separate DLL, you can also include all of the driver's string resources in the same file. This provides a convenient means for translating the driver's text into additional languages by simply replacing this one file.

Following is a small segment, including the first and last entries, of the core command table provided in *winmm.dll*. This segment provides an illustration of how command tables are constructed.

```
core RCDATA
BEGIN
  L"open\0",            MCI_OPEN, 0,                  MCI_COMMAND_HEAD,
  L"\0",                MCI_INTEGER, 0,               MCI_RETURN,
  L"notify\0",          MCI_NOTIFY,                   MCI_FLAG,
  L"wait\0",            MCI_WAIT,                     MCI_FLAG,
  L"type\0",            MCI_OPEN_TYPE,                MCI_STRING,
  L"element\0",         MCI_OPEN_ELEMENT,             MCI_STRING,
  L"alias\0",           MCI_OPEN_ALIAS,               MCI_STRING,
  L"shareable\0",       MCI_OPEN_SHAREABLE,           MCI_FLAG,
  L"\0",                0L,                           MCI_END_COMMAND,
  L"close\0",           MCI_CLOSE, 0,                 MCI_COMMAND_HEAD,
  L"notify\0",          MCI_NOTIFY,                   MCI_FLAG,
  L"wait\0",            MCI_WAIT,                     MCI_FLAG,
  L"\0",                0L,                           MCI_END_COMMAND,
  L"play\0",            MCI_PLAY, 0,                  MCI_COMMAND_HEAD,
  L"notify\0",          MCI_NOTIFY,                   MCI_FLAG,
  L"wait\0",            MCI_WAIT,                     MCI_FLAG,
  L"from\0",            MCI_FROM,                     MCI_INTEGER,
  L"to\0",              MCI_TO,                       MCI_INTEGER,
  L"\0",                0L,                           MCI_END_COMMAND,
                       .
                       .
                       .
  L"status\0",          MCI_STATUS, 0,                MCI_COMMAND_HEAD,
  L"\0",                MCI_INTEGER, 0,               MCI_RETURN,
  L"notify\0",          MCI_NOTIFY,                   MCI_FLAG,
  L"wait\0",            MCI_WAIT,                     MCI_FLAG ,
  L"\0",                MCI_STATUS_ITEM,              MCI_CONSTANT,
  L"position\0",        MCI_STATUS_POSITION,          MCI_INTEGER,
  L"length\0",          MCI_STATUS_LENGTH,            MCI_INTEGER,
  L"number of tracks\0",MCI_STATUS_NUMBER_OF_TRACKS,  MCI_INTEGER,
  L"ready\0",           MCI_STATUS_READY,             MCI_INTEGER,
  L"mode\0",            MCI_STATUS_MODE,              MCI_INTEGER,
  L"time format\0",     MCI_STATUS_TIME_FORMAT,       MCI_INTEGER,
  L"current track\0",   MCI_STATUS_CURRENT_TRACK,     MCI_INTEGER,
  L"\0",                0L,                           MCI_END_CONSTANT,
  L"track\0",           MCI_TRACK,                    MCI_INTEGER,
  L"start\0",           MCI_STATUS_START,             MCI_FLAG,
  L"\0",                0L,                           MCI_END_COMMAND,
                       .
                       .
                       .
  L"set\0",             MCI_SET, 0,                   MCI_COMMAND_HEAD,
  L"notify\0",          MCI_NOTIFY,                   MCI_FLAG,
  L"wait\0",            MCI_WAIT,                     MCI_FLAG ,
  L"time format\0",     MCI_SET_TIME_FORMAT,          MCI_CONSTANT,
  L"milliseconds\0",    MCI_FORMAT_MILLISECONDS, 0,   MCI_INTEGER,
  L"ms\0",              MCI_FORMAT_MILLISECONDS, 0,   MCI_INTEGER,
  L"\0",                0L,                           MCI_END_CONSTANT,
  L"door open\0",       MCI_SET_DOOR_OPEN,            MCI_FLAG,
  L"door closed\0",     MCI_SET_DOOR_CLOSED,          MCI_FLAG,
  L"audio\0",           MCI_SET_AUDIO,                MCI_CONSTANT,
  L"all\0",             MCI_SET_AUDIO_ALL,            MCI_INTEGER,
  L"left\0",            MCI_SET_AUDIO_LEFT,           MCI_INTEGER,
```

```
  L"right\0",           MCI_SET_AUDIO_RIGHT,          MCI_INTEGER,
  L"\0",                0L,                           MCI_END_CONSTANT,
  L"video\0",           MCI_SET_VIDEO,                MCI_FLAG,
  L"on\0",              MCI_SET_ON,                   MCI_FLAG,
  L"off\0",             MCI_SET_OFF,                  MCI_FLAG,
  L"\0",                0L,                           MCI_END_COMMAND,
                       .
                       .
                       .
  L"resume\0",          MCI_RESUME, 0,                MCI_COMMAND_HEAD,
  L"notify\0",          MCI_NOTIFY,                   MCI_FLAG,
  L"wait\0",            MCI_WAIT,                     MCI_FLAG,
  L"\0",                0L,                           MCI_END_COMMAND,
  L"\0",                0L,                         MCI_END_COMMAND_LIST
END
```

As shown in the previous example, a command table is defined by using an RCDATA statement, which is described in the Win32 SDK.. The RCDATA declaration includes the table's name. For the core table, the table's name is "core". For a device-type table, the table's name is the device type, such as "videodisc".

Command tables contain command table entries. Each entry contains:

- A null-terminated string identifying a command or command modifier. This is the text that applications specify with calls to **mciSendString**. Each string is prefaced with "L" to create wide-character storage for UNICODE characters.
- A longword value containing a command constant or modifier flag constant value. Mostly, these are the constants an application uses with the **mciSendCommand** function. Drivers receive the constant values as *umsg* and *lParam1* arguments to **DriverProc**. Note that some predefined constants, such as MCI_INTEGER, only define single word values. These single-word constants must be padded with an extra word to create a longword.
- A single word value representing the entry type.

The table must end with an entry of type MCI_END_COMMAND_LIST.

Example

As an example of how to define a command, look at the "play" command contained in the core table. While the "play" command in the core command table is fairly simple, it illustrates several of the entry types that can be used for creating a command description.

To define a command in a command table

1. Begin with an MCI_COMMAND_HEAD entry. This entry has the following format:

```
L"play\0",            MCI_PLAY, 0,                  MCI_COMMAND_HEAD,
```

   First, the entry contains a null-terminated, wide-character string containing the command's string name. Next, a longword value is specified, with the MCI_PLAY bit set. (Because MCI_PLAY is defined as a 16-bit constant, an additional word of zero is added as the highword value for the longword.) Finally, the entry type is specified as MCI_COMMAND_HEAD, which labels this entry as a new command.

2. Include two entries indicating that the command accepts the MCI_NOTIFY and MCI_WAIT modifiers. (*All* command definitions must contain these two entries.)

```
L"notify\0",          MCI_NOTIFY,                   MCI_FLAG,
L"wait\0",            MCI_WAIT,                     MCI_FLAG,
```

   The MCI_FLAG entry type is used for defining command modifiers that do not require either input or output members within the command's data structure (in this case, MCI_PLAY_PARMS). (See Creating New MCI Command Structures.)

3. Include command-specific modifier definitions. The "play" command accepts two modifiers, "from" and "to".

```
L"from\0",              MCI_FROM,                   MCI_INTEGER,
L"to\0",                MCI_TO,                     MCI_INTEGER,
```

The "from" and "to" modifiers both accept numeric arguments, such as "play videodisc1 from 10 to 100". The MCI_INTEGER type indicates that for each of these arguments, an integer-typed member exists in the "play" command's MCI_PLAY_PARMS structure. Following is a list of recognized data types:

MCI_INTEGER	Integer
MCI_HWND	Window handle
MCI_HPAL	Palette handle
MCI_HDC	Device Context handle
MCI_RECT	Rectangle
MCI_STRING	String

For a complete list of entry types, see Command Table Entry Types.

4. End the command definition with an MCI_END_COMMAND entry.

```
L"\0",                  0L,                         MCI_END_COMMAND,
```

The entry must contain a null string, followed by a null longword.

Example

As another example, look at the "status" command. This command provides a return value, so its command description includes an MCI_RETURN entry. This entry must appear immediately after the MCI_COMMAND_HEAD entry.

```
L"status\0",            MCI_STATUS, 0,              MCI_COMMAND_HEAD,
L"\0",                  MCI_INTEGER, 0,             MCI_RETURN,
```

An MCI_RETURN entry must contain a null string, followed by a longword specifiying the data type of the command's return value. Any of the MCI data types can be used. In this case, the command returns an integer value. The command's data structure must contain an integer-typed member for storing the return value. (See Creating New MCI Command Structures.)

Defining Constant Values

Sometimes it is necessary to define a set of constant values that can be assigned to a data structure member. Constant values are delimited by MCI_CONSTANT and MCI_END_CONSTANT entries. Here is a section of the "set" command description:

```
L"audio\0",             MCI_SET_AUDIO,              MCI_CONSTANT,
L"all\0",               MCI_SET_AUDIO_ALL,          MCI_INTEGER,
L"left\0",              MCI_SET_AUDIO_LEFT,         MCI_INTEGER,
L"right\0",             MCI_SET_AUDIO_RIGHT,        MCI_INTEGER,
L"\0",                  0L,                         MCI_END_CONSTANT,
```

An MCI_CONSTANT entry must contain a string, followed by a longword containing a flag value. A command's data structure must reserve a DWORD-sized member for each MCI_CONSTANT entry in the command table. Entries following the MCI_CONSTANT entry define a set of constant values that can be set in the data structure member. The set of constant entries ends with an MCI_END_CONSTANT entry, which must contain a null string and a null longword.

If an application uses **mciSendString** to specify the command "set vcr1 audio left", then *winmm.dll* sets the MCI_SET_AUDIO flag in *laram1* to indicate the **dwAudio** member for the MCI_SET_PARMS structure is valid, and sets MCI_SET_AUDIO_LEFT in **dwAudio**.

Here is another section of the "status" description:

```
L"\0",                  MCI_STATUS_ITEM,            MCI_CONSTANT,
L"position\0",          MCI_STATUS_POSITION,        MCI_INTEGER,
L"length\0",            MCI_STATUS_LENGTH,          MCI_INTEGER,
L"number of tracks\0",MCI_STATUS_NUMBER_OF_TRACKS,  MCI_INTEGER,
```

```
L"ready\0",           MCI_STATUS_READY,           MCI_INTEGER,
L"mode\0",            MCI_STATUS_MODE,            MCI_INTEGER,
L"time format\0",     MCI_STATUS_TIME_FORMAT,     MCI_INTEGER,
L"current track\0",   MCI_STATUS_CURRENT_TRACK,   MCI_INTEGER,
L"\0",                0L,                         MCI_END_CONSTANT,
```

For this constant list, the MCI_CONSTANT entry contains a null string. A null string is allowed if the command description defines only one modifier type and, hence, one structure input member. If the description defines more than one modifier, a string must be included so the command parser can identify which command structure member to use.

For the "status" command, an application can specify "status vcr1 position", for example, to obtain the current position, and "status vcr1 mode" to obtain the current mode. Both "position" and "mode" use the **dwItem** member in MCI_STATUS_PARMS. In constrast, for the "set" command an application must specify "set vcr1 time format milliseconds" to set the time format, and "set vcr1 audio all" to set all audio channels. Here, "time format" and "audio" differentiate between the **dwTimeFormat** and **dwAudio** members of MCI_SET_PARMS.

Command Table Entry Types

Following is a list of command table entry types that can be used for constructing a command table. All of these types are defined as single word values.

Entry Type	Purpose
MCI_COMMAND_HEAD	First entry for each command. String identifies command. Longword contains MCI message constant.
MCI_CONSTANT	Introduces a set of constant command modifiers. String and longword identify structure member in which constant value is stored.
MCI_END_COMMAND	Last entry for each command. String and longword values are null.
MCI_END_COMMAND_LIST	Last entry in a command table. String and longword values are null.
MCI_END_CONSTANT	Ends a constant set. String and longword values are null.
MCI_FLAG	Defines a command modifier that does not require associated data structure storage. String and longword values identify the modifier.
MCI_HDC	Defines a command modifier that accepts an HDC argument. String defines modifier name. Longword identifies data structure member in which HDC value is stored. Also used to specify return type with MCI_RETURN.
MCI_HPAL	Defines a command modifier that accepts an HPAL argument. String defines modifier name. Longword identifies data structure member in which HPAL value is stored. Also used to specify return type with MCI_RETURN.
MCI_HWND	Defines a command modifier that accepts an HWND argument. String defines modifier name. Longword identifies data structure member in which HWND value is stored. Also used to specify return type with MCI_RETURN.
MCI_INTEGER	Defines a command modifier that accepts an integer argument. String defines modifier name. Longword identifies data structure member in which integer is stored. Also used to specify return type with

	MCI_RETURN.
MCI_RECT	Defines a command modifier that accepts a RECT argument. String defines modifier name. Longword identifies data structure member in which RECT value is stored. Also used to specify return type with MCI_RETURN.
MCI_RETURN	Declares that command returns a value. String value is null. Longword identifies the return value's type, which can be MCI_INTEGER, MCI_HDC, MCI_HPAL, MCI_HWND, MCI_RECT, or MCI_STRING.
MCI_STRING	Defines a command modifier that accepts a string pointer argument. String in command table defines modifier name. Longword identifies data structure member in which string pointer argument is stored. Also used to specify return type with MCI_RETURN.

Loading and Unloading a Command Table

A driver loads a customized command table by calling **mciLoadCommandResource**, as follows:

```
uCommandTable = mciLoadCommandResource(hModuleInstance, "mcivcr", 0);
```

The call should be made when **DriverProc** receives a DRV_LOAD message. In the example, the command table's name is "mcivcr", which means "mcivcr" is the *nameID* argument to the table's RCDATA statement. Additionally, if you place the command table in a separate file, the file must be named *mcivcr.mci*. The file must reside in a directory that is accessible by Win32's **CreateFile** function.

A driver must unload its customized command table by calling **mciFreeCommandResource**. This call should be made when **DriverProc** receives a DRV_FREE message.

Every driver that defines a customized command table must use **mciLoadCommandResource** and **mciFreeCommandResource** to load and unload the table. There is one exception to this rule — if the command table defines a new device type *and* if the table resides in a separate file, then the driver does not need to explicitly load and unload the table. In this case, *winmm.dll* uses the device type name that the application specifies with the MCI_OPEN message to locate and load the command table.

In other words, a driver must explicitly load and unload its customized command table if the table is device-specific (as opposed to being for a device type) *or* if the table is linked to the driver.

If you create a customized command table for a one of the device types provided by Microsoft, and if you put the table in a separate file, do not use the device type name as the file name. Otherwise, other drivers will not be able to access the device type table provided by Microsoft because your table will override it.

How the MCI Parser Uses Command Tables

The MCI parser, within *winmm.dll*, is invoked when an application calls **mciSendString**. The parser reads the command from the command string and attempts to find the command in one of the command tables.

Code in *winmm.dll* searches for a customized table. A customized table is one that the driver has loaded by calling **mciLoadCommandResource**. First, *winmm.dll* uses the table name specified with **mciLoadCommandResource** and attempts to locate a file with that name and an extension of *.mci*. If a separate file is not found, the driver DLL is searched for a resource with the specified name.

This scheme of searching for a separate file before searching inside the driver facilitates creation of location-specific command tables. Tables for various languages can be created without affecting the driver. If you place the command table in a separate file, you can also include all of

the driver's string resources in the same file. When *winmm.dll* attempts to locate a string resource it also searchs in the *.mci* file first.

If a customized table is found, the parser searches the table for the command. If a customized table does not exist, or if the parser cannot find the command in a customized table, then the parser uses the tables defined in *winmm.dll*.

The parser compares the command string with the string specified in each MCI_COMMAND_HEAD entry of the table until it finds a match. When a match is found, the parser extracts the value stored in the longword following the string and uses it as the *umsg* argument for **DriverProc**. The parser then tries to match command modifiers included in the command string with entries in the command description. For each match, the parser extracts the value stored in the longword following the matched string and OR's it into the *lParam1* argument for **DriverProc**. For all modifier types except MCI_FLAG, the parser also takes the modifier value from the command string and stores it in the command's data structure. The parser allows command modifiers to be included in random order in the command string, so "play vcr1 to 100 from 10" is equivalent to "play vcr1 from 10 to 100".

Every time the parser receives a new command, it checks for the command tables in the order described. This means that a driver's customized command table does not need to include all of the commands the driver supports, if it only extends some of them. If, for example, a driver adds new flags to the "play" command but supports only the default (core) behavior for all other commands, then the custom table only needs to contain the extended "play" command. The parser uses the core table to parse the rest of the commands.

Creating New MCI Command Structures

You must create new MCI command structures whenever you extend existing commands or define new ones. For every command description you place in a customized command table, you must define a customized data structure. Place this definition in an include file that is available both to your driver and to application developers. Structure definitions for the core commands reside in *mmsystem.h*.

Command structure definitions follow a standardized format. A typical structure is the following MCI_STATUS_PARMS structure, used with the "status" command.

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

There are rules you should follow for naming command structures, laying out command structures, and referencing command structures.

Naming Command Structures

When naming a command structure, include the command in the structure name. Also include the device type, in order to differentiate customized structures from the structures defined for the core commands. An example is the following structure, used with the extension to the play command for animation devices:

```
typedef struct tagMCI_ANIM_PLAY_PARMS {
    DWORD   dwCallback;
    DWORD   dwFrom;
    DWORD   dwTo;
    DWORD   dwSpeed;
} MCI_ANIM_PLAY_PARMS;
```

Laying Out Command Structures

The first member of a command structure must be a DWORD for storing a window handle. The window handle is provided by the application and is used by *winmm.dll* for delivering command notification messages associated with the MCI_NOTIFY flag. Since all commands must support MCI_NOTIFY, all structures must contain this member. By convention, this member is called **dwCallback**.

Following the **dwCallback** member, each additional structure member must be matched to an entry in the command's description within the command table. The order and data type of each member must match those in the command description.

If a return value is associated with the command, then the second structure member is used for storing the return value. For all return types except MCI_STRING and MCI_RECT, the return member must be DWORD-sized. If the command returns a string, then the structure must provide storage for a string buffer address and a string buffer length, both of which are supplied by the application. For commands whose return type is MCI_RECT, the structure must provide a RECT-sized return member.

The "info" command's return value is a string, and its structure follows:

```
typedef struct tagMCI_INFO_PARMS {
    DWORD   dwCallback;
    LPSTR   lpstrReturn;
    DWORD   dwRetSize;
} MCI_INFO_PARMS;
```

If a command does not provide a return value, then the command's structure does not include return members. For example, the "play" command does not provide a return value. Following is its structure:

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

After the return value member, the rest of the structure members are used for storing input or output arguments. Following is the "play" command description, so you can see how it matches the MCI_PLAY_PARMS structure. Entries with the MCI_FLAG type do not require space in the data structure.

```
L"play\0",          MCI_PLAY, 0,              MCI_COMMAND_HEAD,
L"notify\0",        MCI_NOTIFY,               MCI_FLAG,
L"wait\0",          MCI_WAIT,                 MCI_FLAG,
L"from\0",          MCI_FROM,                 MCI_INTEGER,
L"to\0",            MCI_TO,                   MCI_INTEGER,
L"\0",              0L,                       MCI_END_COMMAND,
```

Here's the "open" command's description:

```
L"open\0",          MCI_OPEN, 0,              MCI_COMMAND_HEAD,
L"\0",              MCI_INTEGER, 0,           MCI_RETURN,
L"notify\0",        MCI_NOTIFY,               MCI_FLAG,
L"wait\0",          MCI_WAIT,                 MCI_FLAG,
L"type\0",          MCI_OPEN_TYPE,            MCI_STRING,
L"element\0",       MCI_OPEN_ELEMENT,         MCI_STRING,
L"alias\0",         MCI_OPEN_ALIAS,           MCI_STRING,
L"shareable\0",     MCI_OPEN_SHAREABLE,       MCI_FLAG,
L"\0",              0L,                       MCI_END_COMMAND,
```

And here is the MCI_OPEN_PARMS structure:

```
typedef struct tagMCI_OPEN_PARMS {
    DWORD       dwCallback;
    MCIDEVICEID wDeviceID;
    WORD        wReserved0;
    LPCSTR      lpstrDeviceType;
    LPCSTR      lpstrElementName;
    LPCSTR      lpstrAlias;
} MCI_OPEN_PARMS;
```

Note the difference between string return values, shown previously in MCI_INFO_PARMS, and string arguments, illustrated in MCI_OPEN_PARMS. For string return values, the structure must contain members to hold both a string buffer address and a string buffer size. For string arguments, only a string pointer is stored.

Referencing Command Structures

Drivers should obey the following rules when referencing command data structures:

- Test for null structure pointers. For some commands, it is acceptable for an application to send the command without the structure. For example, if an application sends the "play" command without modifiers, it does not need the structure.
- Before referencing a structure's input members, check the *lParam1* argument to **DriverProc** to see if the member's flag is set. Data in the structure member is only valid if the approriate flag has been set. Following is an example:

```
if (lparam1 & MCI_FROM)
       dwFrom = lparam2->dwFrom;
```

MCI Reference

This section describes the functions, messages, structures, and macros used by MCI drivers. The following topics are provided:

- MCI Functions
- MCI Messages
- MCI Structures
- MCI Macros

MCI Functions

This section describes the functions defined by *winmm.dll* that are used by MCI drivers.

mciDriverNotify

BOOL APIENTRY
mciDriverNotify (
HANDLE *hwndCallback*,
MCIDEVICEID *wDeviceID*,
UINT *uStatus*
);

MCI drivers call **mciDriverNotify** to post an MM_MCINOTIFY message to an application.

Parameters

hwndCallback
Specifies the handle of the window to notify. The handle is obtained from the **dwCallback** member of the structure pointed to by the **DriverProc** *lParam2* parameter.

wDeviceID
Specifies the device ID. This is the device ID received from **DriverProc**.

uStatus
Specifies the status of the operation requested by the application. Can be one of the following values:

MCI_NOTIFY_SUCCESSFUL	Driver successfully completed the requested operation.
MCI_NOTIFY_SUPERSEDED	Application sent an MCI message with the MCI_NOTIFY flag set, then sent a second message with MCI_NOTIFY set before the first operation completed. Driver calls **mciNotifyDrive**r with MCI_NOTIFY_SUPERSEDED status, then calls **mciNotifyDriver** again when the second operation completes.
MCI_NOTIFY_ABORTED	Application sent a command that prevents the notification condition from being satisfied. For example, the command "stop vcr1" cancels a pending notification for "play vcr1 to 500 notify".
MCI_NOTIFY_FAILURE	A device error prevented the notification condition from being satisfied.

Return Value
If a notification is successfully sent, **mciDriverNotify** returns TRUE; otherwise, it returns FALSE.

Comments
Drivers call **mciDriverNotify** after an operation has completed, if the application that requested the operation included the MCI_NOTIFY flag with the command. See Handling the MCI_NOTIFY Flag.

mciDriverYield

UINT APIENTRY
mciDriverYield (
MCIDEVICEID *wDeviceID*
);

The **mciDriverYield** function calls the application's yield function. The yield function checks to see if the user has pressed the break key.

Parameters
wDeviceID
Specifies the device ID. This is the device ID received from **DriverProc**.

Return Value
Returns the value returned from the yield function. If the break key was pressed, the return value should be a nonzero value. Otherwise returns zero.

Comments
Drivers call **mciDriverYield** while waiting for a requested operation to complete, if the application included the MCI_WAIT flag with the command. See Handling the MCI_WAIT Flag.

mciFreeCommandResource

BOOL APIENTRY

mciFreeCommandResource(
UINT *wTable*
);

The **mciFreeCommandResource** function frees from memory a command table that was loaded with **mciLoadCommandResource**.

Parameters

wTable
The table index number returned from a previous call to **mciLoadCommandResource**.

Return Value

Returns FALSE if the table index number is invalid. Oherwise, returns TRUE.

Comments

For more information see Loading and Unloading a Command Table.

mciGetDriverData

DWORD APIENTRY
mciGetDriverData(
MCIDEVICEID *wDeviceID*
);

The **mciGetDriverData** function returns instance-specific information that was set with **mciSetDriverData**.

Parameters

wDeviceID
Specifies the MCI device ID.

Return Value

Returns the driver instance information. Returns zero if an error occurs.

Comments

A driver can test for an error return only when the valid value is known to be nonzero, if, for example, the driver had previously called **mciSetDriverData** and specified a pointer value. If the driver has set the instance information to zero with **mciSetDriverData**, then it cannot test for an error return from **mciGetDriverData**.

mciLoadCommandResource

UINT APIENTRY
mciLoadCommandResource(
HANDLE *hInstance*,
LPCWSTR *lpResName*,
UINT *wType*
);

The **mciLoadCommandResource** function loads the specified resource and registers it as an MCI command table.

Parameters

hInstance
Specifies the driver module's instance handle. This argument is ignored if the resource resides in an external file.

lpResName
Points to a string representing the name of the resource.

wType
Specifies the device type. Customized device-specific tables must specify a type of zero.

Return Value

If an error occurs, returns MCI_NO_COMMAND_TABLE. Otherwise, returns a command table index number.

Comments

The **mciLoadCommandResource** function first attempts to locate a file, using the *lpResName* argument as the filename, with an extension of *.mci*. If a file is found, the command table is loaded from the file. This file can also contain string resources.

If a file is not found, then *lpResName* is assumed to be the name of a resource that is linked to the driver.

For more information see Loading and Unloading a Command Table.

mciSetDriverData

BOOL APIENTRY
mciSetDriverData(
MCIDEVICEID *wDeviceID*,
DWORD *dwData*
);

The **mciSetDriverData** function is used for storing instance-specific information for an MCI driver.

Parameters

wDeviceID
Specifies a device ID.

dwData
Specifies the information to assign.

Return Value

Returns FALSE if the device ID is invalid. Otherwise returns TRUE.

Comments

Drivers can call **mciSetDriverData** to associate a driver-defined DWORD value with the current instance of the driver. Typically, an MCI driver calls this function while processing the MCI_OPEN_DRIVER message, specifying, for example, a pointer to an instance-specific data structure. The driver might call the function again when processing an MCI_CLOSE_DRIVER message, specifying a *dwData* value of zero. The driver can retrieve the stored value by calling **mciGetDriverData**.

MCI Messages

This section describes the command messages that MCI drivers are required to support. For more information on the types of MCI command messages, see MCI Command Types.

For descriptions of all other MCI commands, refer to the Win32 SDK.

The command messages are defined in *mmsystem.h*.

MCI_CLOSE

MCI drivers do not receive the MCI_CLOSE command. Instead, they receive MCI_CLOSE_DRIVER. See Closing an MCI Driver.

MCI_CLOSE_DRIVER

The MCI_CLOSE_DRIVER message requests an MCI driver to close a driver instance that was previously opened with an MCI_OPEN_DRIVER message.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
MCI_CLOSE_DRIVER

lParam1
Contains the flags specified by the application, with the MCI_CLOSE command.

lParam2
Specifies a pointer to an MCI_GENERIC_PARMS structure, or to a customized structure.

Return Value

If the close operation succeeds, the driver returns zero. Otherwise, the driver returns one of tbe MCIERR error codes defined in *mmsystem.h*.

Comments

A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameters. When an application sends an MCI_CLOSE message, *winmm.dll* intercepts it and sends MCI_CLOSE_DRIVER to the driver.

See Also

Closing an MCI Driver, MCI_OPEN_DRIVER

MCI_GETDEVCAPS

The MCI_GETDEVCAPS message requests an MCI driver to return device capabilities information.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
MCI_GETDEVCAPS

lParam1
Contains the flags specified by the application.

lParam2
Specifies a pointer to an MCI_GETDEVCAPS_PARMS structure, or to a customized structure.

Return Value

If the operation succeeds, the driver returns zero. Otherwise, the driver returns one of tbe MCIERR error codes defined in *mmsystem.h*.

Comments

A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameters.

A driver tests *lParam1* to determine the type of information to return. The driver returns the requested information in the structure pointed to by *lParam2*. Refer to the Win32 SDK description

of MCI_GETDEVCAPS to determine the types of information the driver should return.

See Also

Providing Device Information

MCI_INFO

The MCI_INFO message requests an MCI driver to return device or driver information.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
MCI_INFO

lParam1
Contains the flags specified by the application.

lParam2
Specifies a pointer to an MCI_INFO_PARMS structure, or to a customized structure.

Return Value

If the operation succeeds, the driver returns zero. Otherwise, the driver returns one of tbe MCIERR error codes defined in *mmsystem.h*.

Comments

A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameters.

A driver tests *lParam1* to determine the type of information to return. The driver returns the requested information as a string in a buffer. The buffer's address and size are contained in the structure pointed to by *lParam2*. Refer to the Win32 SDK description of MCI_INFO to determine the types of information the driver should return.

If the application includes the MCI_INFO_PRODUCT flag, the returned string should include the manufacturer of the hardware and, if possible, model identification information. If a driver applies to a device type instead of a specific device, it should return the string name of the device type.

If a driver extends the MCI_INFO command to request additional information, it should not return information that is returned for the MCI_STATUS command.

See Also

Providing Device Information

MCI_OPEN

MCI drivers do not receive the MCI_OPEN command. Instead, they receive MCI_OPEN_DRIVER. See "Opening an MCI Driver".

MCI_OPEN_DRIVER

The MCI_OPEN_DRIVER message requests an MCI driver to open a driver instance.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
MCI_OPEN_DRIVER

lParam1
Contains the flags specified by the application, with the MCI_OPEN command.

lParam2
Specifies a pointer to an MCI_OPEN_PARMS structure, or to a customized structure.

Return Value

If the operation succeeds, the driver returns zero. Otherwise, the driver returns one of the MCIERR error codes defined in *mmsystem.h*.

Comments

A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameters. When an application sends an MCI_OPEN message, *winmm.dll* intercepts it and sends MCI_OPEN_DRIVER to the driver.

See Also

Opening an MCI Driver, MCI_CLOSE_DRIVER

MCI_STATUS

The MCI_STATUS message requests an MCI driver to return device status information.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
MCI_STATUS

lParam1
Contains the flags specified by the application.

lParam2
Specifies a pointer to an MCI_STATUS_PARMS structure, or to a customized structure.

Return Value

If the operation succeeds, the driver returns zero. Otherwise, the driver returns one of tbe MCIERR error codes defined in *mmsystem.h*.

Comments

A client sends the message by calling the driver's **DriverProc** entry point, passing the specified parameters.

A driver tests *lParam1* to determine the type of information to return. The driver returns the requested information in the structure pointed to by *lParam2*. Refer to the Win32 SDK description of MCI_STATUS to determine the types of information the driver should return.

See Also

Providing Device Information

MCI Structures

This section describes the structures associated with the command messages that MCI drivers

are required to support.

For descriptions of all other MCI command structures, refer to the Win32 SDK.

MCI_GENERIC_PARMS

```
typedef struct tagMCI_GENERIC_PARMS {
    DWORD   dwCallback;
} MCI_GENERIC_PARMS;
```

The MCI_GENERIC_PARMS structure is the data structure associated with the MCI_CLOSE_DRIVER command. This structure can be used with any command requiring only a minimal structure definition.

Members

dwCallback
Contains handle of window to receive MM_MCINOTIFY message.

MCI_GETDEVCAPS_PARMS

```
typedef struct tagMCI_GETDEVCAPS_PARMS {
    DWORD   dwCallback;
    DWORD   dwReturn;
    DWORD   dwItem;
} MCI_GETDEVCAPS_PARMS;
```

The MCI_GETDEVCAPS_PARMS structure is the standard data structure associated with the MCI_GETDEVCAPS command. To find out if customized versions of this structure exist, see the description of MCI_GETDEVCAPS in the Win32 SDK.

Members

dwCallback
Contains handle of window to receive MM_MCINOTIFY message.

dwReturn
Receives information returned by the driver.

dwItem
Contains a flag indicating the type of information requested. For a list of valid flags, see the description of MCI_GETDEVCAPS in the Win32 SDK. The contents of **dwItem** are valid only if MCI_GETDEVCAPS_ITEM is set in the *lParam1* argument to **DriverProc**.

MCI_INFO_PARMS

```
typedef struct tagMCI_INFO_PARMS {
    DWORD   dwCallback;
    LPSTR   lpstrReturn;
    DWORD   dwRetSize;
} MCI_INFO_PARMS;
```

The MCI_INFO_PARMS structure is the standard data structure associated with the MCI_INFO command. To see if customized versions of this structure exist, refer to the description of MCI_INFO in the Win32 SDK.

Members

dwCallback
Contains handle of window to receive MM_MCINOTIFY message.

lpstrReturn
Contains pointer to buffer that the driver fills with return string.

dwRetSize
Contains size of return buffer.

MCI_OPEN_DRIVER_PARMS

```
typedef struct {
    MCIDEVICEID   wDeviceID;
    LPCWSTR  lpstrParams;
    UINT  wCustomCommandTable;
    UINT  wType;
} MCI_OPEN_DRIVER_PARMS;
```

The MCI_OPEN_DRIVER_PARMS structure is the data structure associated with the DRV_OPEN command for MCI drivers. See Opening an MCI Driver.

Members

wDeviceID
Contains the MCI device ID.

lpstrParams
Contains a pointer to a zero-terminated string. The string contains any characters that follow the filename in the system registry.

wCustomCommandTable
Receives a handle returned from **mciLoadCommandResource**, or MCI_NO_COMMAND_TABLE.

wType
Receives one of the following defined MCI device types:

- MCI_DEVTYPE_ANIMATION
- MCI_DEVTYPE_CD_AUDIO
- MCI_DEVTYPE_DAT
- MCI_DEVTYPE_DIGITAL_VIDEO
- MCI_DEVTYPE_OVERLAY
- MCI_DEVTYPE_SEQUENCER
- MCI_DEVTYPE_SCANNER
- MCI_DEVTYPE_VCR
- MCI_DEVTYPE_VIDEODISC
- MCI_DEVTYPE_WAVEFORM_AUDIO

If the driver does not support any of the defined types, it should set **wType** to MCI_DEVTYPE_OTHER.

MCI_OPEN_PARMS

```
typedef struct tagMCI_OPEN_PARMS {
    DWORD       dwCallback;
    MCIDEVICEID wDeviceID;
    WORD        wReserved0;
    LPCSTR      lpstrDeviceType;
    LPCSTR      lpstrElementName;
    LPCSTR      lpstrAlias;
} MCI_OPEN_PARMS;
```

The MCI_OPEN_PARMS structure is the standard data structure associated with the MCI_OPEN_DRIVER command. To find out if customized versions of this structure exist, see the description of MCI_OPEN in the Win32 SDK.

Members

dwCallback
Contains handle of window to receive MM_MCINOTIFY message.

wDeviceID
Contains the device ID.

wReserved0
Reserved.

lpstrDeviceType
Contains device type string, if MCI_OPEN_TYPE is set in the *lParam1* argument to **DriverProc**.

lpstrElementName
Contains element name string, if MCI_OPEN_ELEMENT is set in the *lParam1* argument to **DriverProc**.

lpstrAlias
Contains alias string, if MCI_OPEN_ALIAS is set in the *lParam1* argument to **DriverProc**.

MCI_STATUS_PARMS

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

The MCI_STATUS_PARMS structure is the standard data structure associated with the MCI_STATUS command. To find out if customized versions of this structure exist, see the description of MCI_STATUS in the Win32 SDK.

Members

dwCallback
Contains handle of window to receive MM_MCINOTIFY message.

dwReturn
Receives driver-supplied integer return value.

dwItem
Contains a flag indicating the type of information requested. For a list of valid flags, see the description of MCI_STATUS in the Win32 SDK. The contents of **dwItem** is valid only if MCI_STATUS_ITEM is set in the *lParam1* argument to **DriverProc**.

MCI Macros

This topic describes macros available to MCI drivers.

MAKEMCIRESOURCE

LRESULT
MAKEMCIRESOURCE(
WORD *wReturn*,
WORD *wResource*
);

The **MAKEMCIRESOURCE** macro concatenates two word values to create a longword value. Its purpose is to create a longword value that contains both a constant value and a resource ID.

Parameters

wReturn
Specifies the constant value.

wResource
Specifies the resource ID.

Return Value

Returns the longword result of the concatenation.

Comments

The **MAKEMCIRESOURCE** macro is used for constructing a longword value that can be returned in a command structure's **dwReturn** member. Structures for the MCI_GETDEVCAPS and MCI_STATUS commands contain this member. Extended commands can also include this structure member. The **MAKEMICRESOURCE** macro is used in combination with the MCI_RESOURCE_RETURNED and MCI_RESOURCE_DRIVER return values for **DriverProc**. See Returning Information to Applications.

Time Formatting Macros

The following time formatting macros are available to both MCI drivers and applications. The macros are defined in *mmsystem.h* and are described in the Win32 SDK.

MCI_HMS_HOUR
: Retrieves the hours component from a parameter containing packed hours/minutes/seconds (HMS) information.

MCI_HMS_MINUTE
: Retrieves the minutes component from a parameter containing packed hours/minutes/seconds (HMS) information.

MCI_HMS_SECOND
: Retrieves the seconds component from a parameter containing packed hours/minutes/seconds (HMS) information.

MCI_MAKE_HMS
: Creates a time value in packed hours/minutes/seconds (HMS) format from the given hours, minutes, and seconds values.

MCI_MAKE_MSF
: Creates a time value in packed minutes/seconds/frames (MSF) format from the given minutes, seconds, and frame values.

MCI_MAKE_TMSF
: Creates a time value in packed tracks/minutes/seconds/frames (TMSF) format from the given tracks, minutes, seconds, and frames values.

MCI_MSF_FRAME
: Creates the frames component from a parameter containing packed minutes/seconds/frames (MSF) information.

MCI_MSF_MINUTE
: Creates the minutes component from a parameter containing packed minutes/seconds/frames (MSF) information.

MCI_MSF_SECOND
: Creates the seconds component from a parameter containing packed minutes/seconds/frames (MSF) information.

MCI_TMSF_FRAME
: Retrieves the frames component from a parameter containing packed tracks/minutes/seconds/frames (TMSF) information.

MCI_TMSF_MINUTE
: Retrieves the minutes component from a parameter containing packed tracks/minutes/seconds/frames (TMSF) information.

MCI_TMSF_SECOND
: Retrieves the seconds component from a parameter containing packed tracks/minutes/seconds/frames (TMSF) information.

MCI_TMSF_TRACK
: Retrieves the tracks component from a parameter containing packed tracks/minutes/seconds/frames (TMSF) information.

Audio Device Drivers

The following topics explain how to write audio device drivers for Windows NT®:

- Introduction to Audio Device Drivers
- Designing a User-Mode Audio Driver
- Designing a Kernel-Mode Audio Driver
- Audio Driver Reference

For a general discussion of multimedia device drivers, refer to Introduction to Multimedia Drivers.

Introduction to Audio Device Drivers

The following topics provide an introductory information about audio device drivers:

- Types of Audio Devices
- Audio Software Components
- The Standard Audio Driver, *mmdrv.dll*
- Sample Audio Drivers

Types of Audio Devices

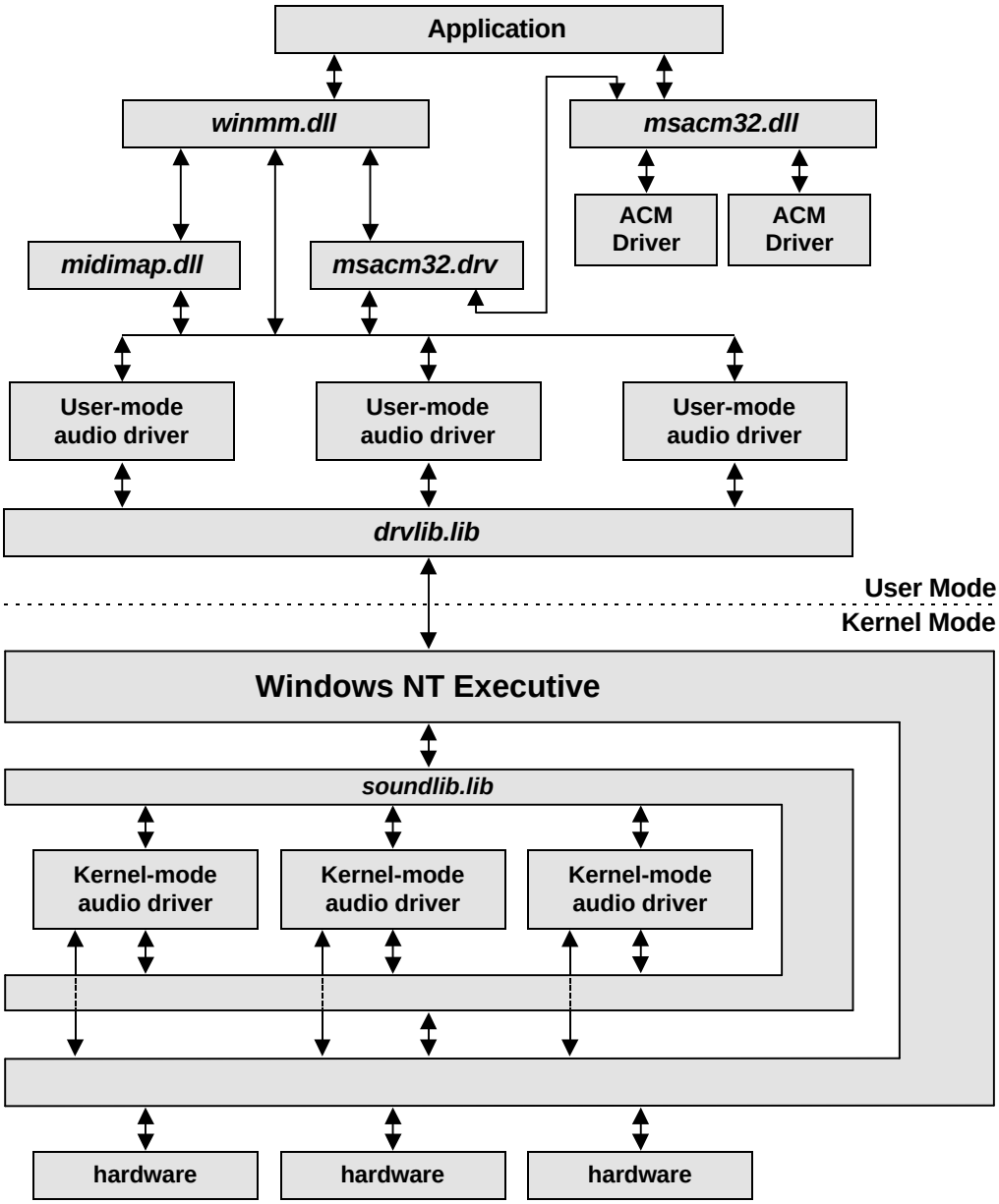
The following types of audio devices can be accessed by applications using API functions provided by the Windows NT Win32 subsystem:

- *Waveform input devices* that convert analog audio signals into digital information.
- *Waveform output devices* that convert digital audio information into analog audio signals.
- *MIDI input ports* that receive MIDI messages from external MIDI devices, such as keyboards.
- *MIDI output ports* that send MIDI messages to external MIDI devices, such as keyboards and drum machines.
- *Internal MIDI synthesizers*, which are internal MIDI output devices that synthesize music from MIDI messages sent by applications.
- *Auxiliary audio devices* such as CD players, which do not require data transfers but whose output can be mixed with MIDI and waveform audio, and whose volume can be controlled by a driver.
- *Mixer devices* that control multiple audio lines.

Combination drivers are audio device drivers that can support several of these types of audio devices. The standard audio driver, *mmdrv.dll*, is a combination driver.

Audio Software Components

The following diagram illustrates the relationship of the major Windows NT audio software components.



The components in the diagram include:

- Application

  Any user-mode, Win32-based application that calls the audio API functions described in the Win32 SDK.

- *winmm.dll*

  A dynamic-link library that exports the waveform, MIDI, mixer, and auxiliary audio functions described in the Win32 SDK.

- *msacm32.dll*

  A dynamic-link library that contains the Audio Compression Manager (ACM). For information about the ACM and ACM drivers, see Audio Compression Manager Drivers.

- *midimap.dll*

  The Windows NT MIDI Mapper.

- *msacm32.drv*

  The Windows NT Wave Mapper.

- User-mode drivers

Dynamic-link libraries that communicate with kernel-mode drivers (or sometimes other user-mode drivers).

- ACM drivers

  Dynamic-link libraries under the control of the ACM. See *msacm32.dll*, above.

- *drvlib.lib*

  A library used as a basis for user-mode audio drivers. For more information, see Using *drvlib.lib*.

- Kernel-mode drivers

  Kernel-mode code that communicates with the Windows NT Executive in order to access device hardware.

- *soundlib.lib*

  A library used as a basis for kernel-mode audio drivers. For more information, see Using *soundlib.lib*.

The Standard Audio Driver, *mmdrv.dll*

Microsoft provides a standard user-mode audio driver for Windows NT, called *mmdrv.dll*. It is a combination driver that supports waveform input and output devices, MIDI input and output devices, and auxiliary audio devices. It provides the user-mode audio driver entry points and support functions necessary for handling all of these device types.

If you are providing driver support for new audio hardware, you might be able to use *mmdrv.dll* as your user-mode driver. You can use *mmdrv.dll* if it provides access to all of the functionality provided by your kernel-mode driver. If you are able to use *mmdrv.dll* as your user-mode driver, then you only need to write a kernel-mode driver for your hardware.

If you want to use *mmdrv.dll* functionality but need to provide customized installation and configuration operations, consider using *drvlib.lib*, which is a library you can use to construct a customized user-mode driver, instead of using *mmdrv.dll*.

You can even use *mmdrv.dll* and *drvlib.lib* in combination. The user-mode driver for the Creative Labs Sound Blaster™ does this. See Sample Audio Drivers and Examining *sndblst.sys*.

The MS-DOS device names supported by *mmdrv.dll* are defined in the *ntddwave.h*, *ntddmidi.h*, and *ntddaux.h* files. MS-DOS names are prepended with "\\.\", as in "\\.\WaveOut0".

Source code for *mmdrv.dll* is included with this DDK.

Sample Audio Drivers

This DDK includes the source code for the following audio drivers and libraries:

Driver or Library	Location of Source Files
User mode audio driver library (see Using *drvlib.lib*.)	*ddk**src**mmedia**drvlib*
Standard user-mode audio driver	*ddk**src**mmedia**mmdrv*
Media Vision ProAudio Spectrum 16 driver	*ddk**src**mmedia**mvaudio*
Creative Labs Sound Blaster driver	*ddk**src**mmedia**sndblst*
Windows sound system driver	*ddk**src**mmedia**sndsys*
Kernel-mode driver library (see Using *soundlib.lib*.)	*ddk**src**mmedia**soundlib*
Ad Lib and OPL3 MIDI synthesizer driver	*ddk**src**mmedia**synth*
User-mode synthesizer driver library (see Using *synthlib.lib*.)	*ddk**src**mmedia**synthlib*

For the driver samples, code for both the user-mode and the kernel-mode driver is provided.

Under the listed driver directory, a *\dll* subdirectory contains the user-mode driver sources, and a *\driver* subdirectory contains the kernel-mode driver sources.

Within this chapter frequent reference is made to the user-mode and kernel-mode drivers for the Creative Labs Sound Blaster card. The kernel-mode driver is *sndblst.sys*. Two user-mode drivers call *sndblst.sys*: the standard audio driver, *mmdrv.dll*, handles MIDI input and output operations; a customized driver, *sndblst.dll*, handles waveform, mixer, synthesizer, and auxiliary I/O operations.

Designing a User-Mode Audio Driver

User-mode audio drivers are implemented as dynamic-link libraries. This section contains the following topics to assist you in designing a user-mode audio driver:

- User-Mode Audio Driver Entry Points
- User-Mode Audio Driver Messages
- Controlling Waveform and MIDI Devices
- Controlling Auxiliary Audio Devices
- Controlling Mixers
- Notifying Clients from Audio Drivers
- Using *drvlib.lib*
- Using *synthlib.lib*
- General Design Guidelines

User-Mode Audio Driver Entry Points

Like all installable user-mode drivers, user-mode audio drivers must export a **DriverProc** entry point and support the standard driver messages. Audio drivers typically ignore the DRV_OPEN standard message.

Audio drivers must also export one or more of the following entry points:

auxMessage	Entry point for auxiliary audio drivers.
midMessage	Entry point for MIDI input drivers.
modMessage	Entry point for MIDI output drivers.
mxdMessage	Entry point for mixer drivers.
widMessage	Entry point for waveform input drivers.
wodMessage	Entry point for waveform output drivers.

Like the **DriverProc** entry point, these additional entry points are implemented as functions that receive and process messages. Applications control multimedia device operations by calling multimedia API functions, which are described in the Win32 SDK and defined in *winmm.dll*. Code within *winmm.dll* translates each API function call into a call to a driver entry point, and includes one of the user-mode audio driver messages. For example, when an application calls the Win32 **WaveOutOpen** function, *winmm.dll* calls the appropriate user-mode driver's **wodMessage** entry point, passing a WODM_OPEN message.

User-Mode Audio Driver Messages

The following table lists the messages that each driver entry point can receive. Messages are divided into those that the driver is required to support, and those that the driver can optionally support. Message definitions are contained in *mmddk.h*.

Entry Points	Required Messages	Optional Messages
auxMessage	AUXDM_GETDEVCAPS	AUXDM_GETVOLUME
	AUXDM_GETNUMDEVS	AUXDM_SETVOLUME

midMessage	MIDM_ADDBUFFER	MIDM_PREPARE
	MIDM_CLOSE	MIDM_UNPREPARE
	MIDM_GETDEVCAPS	
	MIDM_GETNUMDEVS	
	MIDM_OPEN	
	MIDM_RESET	
	MIDM_START	
	MIDM_STOP	
modMessage	MODM_CACHEDRUMPATCHES	MODM_GETVOLUME
	MODM_CACHEPATCHES	MODM_PREPARE
	MODM_CLOSE	MODM_SETVOLUME
	MODM_DATA	MODM_UNPREPARE
	MODM_GETDEVCAPS	
	MODM_GETNUMDEVS	
	MODM_LONGDATA	
	MODM_OPEN	
	MODM_RESET	
mxdMessage	MXDM_CLOSE	MXDM_INIT
	MXDM_GETCONTROLDETAILS	
	MXDM_GETDEVCAPS	
	MXDM_GETLINECONTROLS	
	MXDM_GETLINEINFO	
	MXDM_GETNUMDEVS	
	MXDM_OPEN	
	MXDM_SETCONTROLDETAILS	
widMessage	WIDM_ADDBUFFER	WIDM_LOWPRIORITY
	WIDM_CLOSE	WIDM_PREPARE
	WIDM_GETDEVCAPS	WIDM_UNPREPARE
	WIDM_GETNUMDEVS	
	WIDM_GETPOS	
	WIDM_OPEN	
	WIDM_RESET	
	WIDM_START	
	WIDM_STOP	
wodMessage	WODM_BREAKLOOP	WODM_GETPITCH
	WODM_CLOSE	WODM_GETPLAYBACKRATE
	WODM_GETDEVCAPS	WODM_GETVOLUME
	WODM_GETNUMDEVS	WODM_PREPARE
	WODM_GETPOS	WODM_SETPITCH
	WODM_OPEN	WODM_SETPLAYBACKRATE
	WODM_PAUSE	WODM_SETVOLUME
	WODM_RESET	WODM_UNPREPARE
	WODM_RESTART	
	WODM_WRITE	

Note: If you are writing a driver for a waveform device that supports compressed data formats, your driver must also support some of the Audio Compression Manager (ACM) messages. For further information, see Providing ACM Support in Device Drivers, which is contained in the Audio Compression Manager Drivers chapter.

Controlling Waveform and MIDI Devices

The following topics describe the methods used for controlling input and output data transfers for both waveform and MIDI devices:

- Introduction to Transferring Audio Data
- Transferring Waveform Input Data

- Transferring Waveform Output Data
- Transferring MIDI Input Data
- Transferring MIDI Output Data

Within these topics, the term *client* is used to refer to any higher-level software that is making calls into the user-mode driver. This higher-level software could be an application or, more likely, it could be an intermediate interface called by an application and implemented in *winmm.dll*.

Introduction to Transferring Audio Data

A primary responsibility of user-mode waveform and MIDI drivers is to pass streams of data between clients and kernel-mode drivers. For both input and output operations, clients first allocate data buffers and request the user-mode driver to prepare the buffers for use.

For input operations, the client passes empty buffers to the user-mode driver, which then requests data from the kernel-mode driver. The kernel-mode driver reads data from the device and passes it back the user-mode driver, which then fills the buffers. The user-mode driver notifies the client each time a buffer is filled, so that the client can copy the data from the buffer. The client can then re-use the buffer or, if all data has been received, it can request the user-mode driver to remove the buffer's preparation, and then deallocate the buffer.

For output operations, the client fills the buffers with data and begins passing them to the user-mode driver. The user-mode driver reads the data and passes it to the kernel-mode driver, which in turn sends the data to the device. The user-mode driver notifies the client when each buffer has been read. The client can then re-use the buffer or, if all data has been sent, it can request the user-mode driver to remove the buffer's preparation, and then deallocate the buffer. (Some MIDI output operations do not use buffers.)

For more details about transferring audio data, see:

- Transferring Waveform Input Data
- Transferring Waveform Output Data
- Transferring MIDI Input Data
- Transferring MIDI Output Data

The algorithms described in these topics are implemented in *drvlib.lib* and *mmdrv.dll*.

Transferring Waveform Input Data

For waveform input operations, clients call the user-mode driver's **widMessage** function. The user-mode driver should expect the client to first send a WIDM_OPEN message to open a driver instance. Next, the client allocates memory for one or more data buffers and sends WIDM_PREPARE messages to prepare the buffers for use. The client then sends a WIDM_ADDBUFFER message for each buffer, which passes the address of the empty buffer to the user-mode driver. The user-mode driver keeps a list of available empty buffers.

To start the read operation, the client sends WIDM_START. The user-mode driver then uses a separate thread to begin requesting data from the kernel-mode driver, typically by calling **ReadFile** or **ReadFileEx**, and filling the empty buffers. Each time a buffer has been filled, the user-mode driver notifies the client by sending it a WIM_DATA callback message. The client copies the data from the buffer and re-adds the buffer to the user-mode driver's buffer list by sending another WIDM_ADDBUFFER message.

When the client has finished the input operation, it sends WIDM_STOP. It can also send WIDM_RESET, which indicates to the user-mode driver that it should not fill any remaining data buffers. The client can then send a WIDM_UNPREPARE message for each buffer and deallocate the buffers. Finally, the driver should expect the client to close the instance by sending WIDM_CLOSE.

Transferring Waveform Output Data

For waveform output operations, clients call the user-mode driver's **wodMessage** function. User-mode drivers should expect the client to first send a WODM_OPEN message to open a driver instance. Next, the client allocates memory for one or more data buffers and sends WODM_PREPARE messages to prepare the buffers for use.

The client then begins filling the buffers and sending WODM_WRITE messages, which include a buffer address. The user-mode driver keeps a queue of buffer addresses. When the user-mode driver starts receiving WODM_WRITE messages, it uses a separate thread to begin sending the received data to the kernel-mode driver, typically by calling **WriteFile** or **WriteFileEx**. Each time the user-mode driver reads a buffer and sends the data to the kernel-mode driver, it dequeues the buffer and sends the client a WOM_DONE callback message to notify it that the buffer has been read. The client can then re-use the buffer by specifying its address with another WODM_WRITE message.

When the client has finished the output operation, it stops sending WODM_WRITE messages. It can also send WODM_RESET, which indicates to the user-mode driver that it should not dequeue any remaining data buffers. The client can then send a WODM_UNPREPARE message for each buffer and deallocate the buffers. Finally, the driver should expect the client to close the instance by sending WODM_CLOSE.

Transferring MIDI Input Data

For MIDI input operations, clients call the user-mode driver's **midMessage** function. The user-mode driver should expect the client to first send a MIDM_OPEN message to open a driver instance. Next, the client allocates memory for one or more data buffers and sends MIDM_PREPARE messages to prepare the buffers for use. The client then sends a MIDM_ADDBUFFER message for each buffer, which passes the address of the empty buffer to the user-mode driver. The user-mode driver keeps a queue of available empty buffers.

To start the read operation, the client sends MIDM_START. The user-mode driver then uses a separate thread to begin requesting data from the kernel-mode driver, typically by calling **ReadFileEx**. The user-mode driver receives a buffer of MIDI data that can consist of a combination of short MIDI messages, or single *events*, and long MIDI messages, or *system-exclusive events*. (For descriptions of MIDI events, see the *Standard MIDI Files 1.0* specification.) The user-mode driver must parse the bytes received and do the following:

- Create a time stamp (see *Adding Time Stamps*, below).
- If the received event is system-exclusive, place the event's bytes in the next available buffer from the queue of client buffers. If the buffer becomes full, notify the client with a MIM_LONGDATA callback message. The client can read the buffer and re-use it by sending another MIDM_ADDBUFFER message.
- If the received event is not system-exclusive, check the message to see if running status (see *Running Status*, below) is in effect and if so, add the previous status byte to the event. (All events passed to clients must include a status byte.) Then pass the event's bytes to the client with a MIM_DATA callback message.

When the client has finished the input operation, it sends MIDM_STOP. It can also send MIDM_RESET, which indicates to the user-mode driver that it should not fill any remaining data buffers. The client can then send a MIDM_UNPREPARE message for each buffer and deallocate the buffers. Finally, the driver should expect the client to close the instance by sending MIDM_CLOSE.

Adding Time Stamps

All MIDI events that are returned to a client must include a time stamp. The time stamp represents the number of milliseconds that have passed since input began. When a client sends the MIDM_START message, the kernel-mode driver saves the current system time to use as a reference time. Then each time the kernel-mode driver reads an event, it saves a time stamp value equal to the difference between the current time and the reference time.

Running Status

MIDI events might or might not include a status byte. If the status byte is not included, then the client is employing *running status*. This means that the last status byte sent is still in effect and need not be re-sent. When a user-mode driver receives a MODM_DATA message it checks the status byte and if no value is present, it does not pass the byte to the kernel-mode driver.

Handling MIDI Thru

Code within *winmm.dll* supports MIDI thru-ing to the extent that it will connect one MIDI input driver to one MIDI output driver. You can write a thru-ing driver by responding to DRVM_ADD_THRU and DRVM_REMOVE_THRU messages within **midMessage** and **modMessage**. For more information, see the discussion of managing MIDI thru-ing and the description of **midiConnect** in the Win32 SDK.

Transferring MIDI Output Data

Clients can send output data to user-mode MIDI drivers as either short MIDI messages or long MIDI messages. A short MIDI message contains a single MIDI *event*. The event is passed as an argument to the user-mode driver's **modMessage** function. Long MIDI messages consist of a buffer of MIDI events, including MIDI *system exclusive* events. The buffer address is passed as an argument to the user-mode driver's **modMessage** function. (For descriptions of MIDI events, see the *Standard MIDI Files 1.0* specification.)

For MIDI output operations, clients call the user-mode driver's **modMessage** function. User-mode drivers should expect the client to first send a MODM_OPEN message to open a driver instance. If the client will be sending long MIDI messages, it allocates memory for one or more data buffers and sends MODM_PREPARE messages to prepare the buffers for use. The client then begins sending either short messages or long messages.

To send a short message, the client sends a MODM_DATA message and includes the message data. Since MIDI events can be one, two, or three bytes in length, the user-mode driver must examine the event's status field to determine how many bytes are valid. It then passes the proper number of bytes to the kernel-mode driver by calling **DeviceIoControl**.

To send a long message, the client places the message in a buffer and sends a MODM_LONGDATA message, which includes the buffer's address as an argument. The user-mode driver does not examine the contents of a long message; it just passes the buffer contents to the kernel-mode driver by calling **DeviceIoControl**.

When the user-mode driver begins receiving MODM_DATA and MODM_LONGDATA messages, it uses a separate thread to begin sending the received data to the kernel-mode driver, typically by calling **DeviceIoControl**. After the data has been sent to the kernel-mode driver, the user-mode driver notifies the client by sending it a MOM_DONE callback message. Clients can re-use a MODM_LONGDATA buffer by refilling it and including it with a subsequent MODM_LONGDATA message.

When the client has finished the output operation, it stops sending MODM_DATA and MODM_LONGDATA messages. It can also send MODM_RESET, which indicates to the user-mode driver that it should not dequeue any remaining data buffers. The client can then send a MODM_UNPREPARE message for each buffer and deallocate the buffers. Finally, the driver should expect the client to close the instance by sending MODM_CLOSE.

MIDI Output Streams

Clients create MIDI streams by using the MIDI stream functions. These functions are defined within *winmm.dll* and described in the Win32 SDK. Code in *winmm.dll* translates the MIDI stream functions into MODM_DATA and MODM_LONGDATA messages. Therefore, MIDI drivers are not aware of stream operations. Each MODM_LONGDATA message buffer contains a single MIDI event that can be passed directly to the driver by calling **DeviceIoControl**.

Controlling Auxiliary Audio Devices

To control auxiliary audio device operations, clients call the user-mode driver's **auxMessage**

function. Clients do not need to open or close auxiliary audio devices. The user-mode driver should expect to receive AUXDM_GETDEVCAPS, AUXDM_GETNUMDEVS, AUXDM_GETVOLUME, and AUXDM_SETVOLUME messages in any order.

Controlling Mixers

To control mixer operations, clients call the user-mode driver's **mxdMessage** function. The user-mode driver should first expect the client to send a MXDM_OPEN message to open a driver instance. Then the driver can expect to receive MXDM_GETLINEINFO, MXDM_GETLINECONTROLS, MXDM_GETCONTROLDETAILS, and MXDM_SETCONTROLDETAILS messages from the opened the instance, until the client sends MXDM_CLOSE.

Notifying Clients from Audio Drivers

User-mode drivers are responsible for notifying clients upon the completion of various audio operations. Clients indicate the type of notification, if any, they expect when they open a driver instance for waveform, MIDI, or mixer operations. (Refer to MIDM_OPEN, MODM_OPEN, MXDM_OPEN, WIDM_OPEN, and WODM_OPEN messages.) Clients that request notification can specify any of the following notification targets:

- A callback function
- A window handle
- An event handle
- A thread identifier

Mixer drivers accept only window handles.

User-mode drivers notify clients by calling the **DriverCallback** function in *winmm.dll*. This function delivers a message to the client's notification target. The function also delivers message parameters, if the target type accepts parameters.

Following are the messages user-mode drivers must send to a client if the client has requested notification:

Operations	Messages
Waveform Input	WIM_CLOSE WIM_DATA WIM_OPEN
Waveform Output	WOM_CLOSE WOM_DONE WOM_OPEN
MIDI Input	MIM_CLOSE MIM_DATA MIM_ERROR MIM_LONGDATA MIM_LONGERROR MIM_MOREDATA MIM_OPEN
MIDI Output	MOM_CLOSE MOM_DONE MOM_OPENMOM_POSITIONCB
Mixer	MM_MIXM_LINE_CHANGE MM_MIXM_CONTROL_CHANGE

Using *drvlib.lib*

The user-mode library *drvlib.lib* provides the following:

- A standard set of user-mode audio driver entry points for waveform, MIDI, mixer, and auxiliary user-mode drivers.
- A communication path to the device's kernel-mode driver. This path consists of:
  - Calling the Windows NT I/O Manager (by means of **DeviceIoControl**, described in the Win32 SDK) to send I/O control codes to the kernel-mode driver.
  - Calling **ReadFileEx** and **WriteFileEx** (described in the Win32 SDK) to pass data between the user-mode and kernel-mode drivers.

  For information on I/O control codes for multimedia drivers, see the *Kernel-Mode Drivers Reference*.
- A set of functions for creating and managing a kernel-mode driver as a Windows NT service, and for easily accessing driver keys within the Windows NT Registry. These functions are useful for installation and configuration operations, and are listed in Installing and Configuring your Driver, Using *drvlib.lib*.

The entry points, support functions, and communications path to the kernel-mode driver are identical to those provided by the standard audio driver, *mmdrv.dll*. This library is particularly useful if your user-mode driver requires only the functionality of *mmdrv.dll*, with the addition of customized installation and configuration operations. In such a case, you would create a module containing a customized **DriverProc** function, and then link it with *drvlib.lib*. To initialize the library, call **DrvLibInit** when your driver is loaded.

If your driver requires customized audio entry point functions, you might still want to base your functions on the those contained in *drvlib.lib*. Sources for *drvlib.lib* are provided with this DDK, in the directory path listed in Sample Audio Drivers.

Installing and Configuring your Driver, Using *drvlib.lib*

User-mode drivers can call functions in *drvlib.lib* to start the kernel-mode driver and to modify registry keys. The most commonly used functions include:

DrvLibInit
Initializes *drvlib.lib* for use with your user-mode driver.

DrvCreateServicesNode
Creates a connection to the Windows NT service control manager and, optionally, creates a service object for the kernel-mode driver.

DrvConfigureDriver
Opens a connection to the service control manager, creates a kernel-mode driver service for the specified driver, and loads the kernel-mode driver. This function is typically called when **DriverProc** receives a DRV_INSTALL or DRV_CONFIGURE message.

DrvCloseServiceManager
Closes a connection to the service control manager.

DrvCreateDeviceKey
Creates a device subkey under the driver's **\Parameters** registry key. You should store a device's configuration parameters under this key.

GetInterruptsAndDma
Examines the registry to determine which interrupt numbers and DMA channels are assigned to devices.

DrvSetDeviceIdParameter
Assigns a value to a value name in the registry, under a device's **\Parameters** key. Use this function to save configuration parameters.

DrvQueryDeviceIdParameter
Reads the value associated with a value name, under a device's **\Parameters** registry key. Use this function to read configuration parameters you have stored in the registry.

DrvRemoveDriver

Unloads the kernel-mode driver and marks the kernel-mode driver service for deletion. Typically called when **DriverProc** receives a DRV_REMOVE message.

Other installation and configuration functions provided by *drvlib.lib* are **DrvSaveParametersKey**, **DrvRestoreParametersKey**, **DrvDeleteServicesNode**, **DrvLoadKernelDriver**, **DrvUnloadKernelDriver**, **DrvIsDriverLoaded**, **DrvNumberOfDevices**, and **DrvSetMapperName**.

Using *synthlib.lib*

The *synthlib.lib* library provides support for internal MIDI synthesizers by converting MIDI output messages into calls to MIDI FM synthesis functions. The library supports ADLIB and OPL3 synthesizers.

The library contains an alternate **modMessage** entry point function called **modsynthMessage**. If you choose to use this library, link it with your user-mode driver. The driver's module definition file must contain the following export line:

```
modMessage = modsynthMessage
```

Source code for *synthlib.lib* is provided with this DDK, at the directory path listed in Sample Audio Drivers.

General Design Guidelines

You most important guideline is to *make sure the driver design adheres to the function descriptions provided by the Win32 SDK*. Oftentimes the messages, entry point function parameters, and return codes described within this chapter correlate to Win32 API functions, parameters, and return codes described in the Win32 SDK. This chapter provides cross references to function descriptions in the Win32 SDK.

For MIDI file format descriptions, see the *MIDI 1.0 Detailed Specification* and *Standard MIDI Files 1.0.*

Designing a Kernel-Mode Audio Driver

Kernel-mode audio drivers are responsible for accessing audio hardware. They are implemented as services under the control of the Windows NT service control manager.

The following topics are provided to assist you in designing a kernel-mode audio driver:

- Using *soundlib.lib*
- Examining *sndblst.sys*
- Initializing and Configuring a Driver
- Synchronizing Driver Activities
- Supporting Waveform Devices
- Supporting MIDI Devices
- Supporting Mixer Devices
- Supporting Auxiliary Audio Devices

For general information on the structure of kernel-mode drivers, refer to the *Kernel-Mode Drivers Design Guide* and *Kernel-Mode Drivers Reference* in this DDK.

Using *soundlib.lib*

Kernel-mode audio drivers can use the *soundlib.lib* library. For waveform, MIDI, auxiliary audio, and mixer drivers, *soundlib.lib* provides the following:

- Functions that the kernel-mode driver can call for initialization, configuration, and synchronization activities.
- A set of dispatch functions that the driver can use for handling IRPs and I/O control codes, which are received from the Windows NT I/O Manager as a result of calls from user-mode drivers.

For information on IRPs and I/O control codes for multimedia drivers, see the *Kernel-Mode Drivers Reference*.

You can use *soundlib.lib* to make creating a new kernel-mode driver easier. To use *soundlib.lib*, link it with your kernel-mode driver. You'll need to include some or all of the following header files:

devices.h
midi.h
mixer.h
mmsystem.h
ntddk.h
ntddaux.h
ntddmidi.h
mtddmix.h
ntddwave.h
soundcfg.h
soundlib.h
synthdrv.h
wave.h

If you include *soundlib.h*, it includes many, but not all, of the other header files listed.

Source code for *soundlib.lib* is included with this DDK, at the directory path listed in Sample Audio Drivers.

Descriptions of the functions and structures provided by *soundlib.lib* can be found in the Audio Driver Reference.

Examining *sndblst.sys*

To illustrate both the construction of a kernel-mode driver and the use of *soundlib.lib*, the following sections examine the sample kernel-mode driver *sndblst.sys*. This driver supports the Creative Labs Sound Blaster card. It is called from the user-mode driver *sndblst.dll*, for waveform, mixer, and auxiliary I/O operations. For MIDI operations, *sndblst.sys* is called from the standard audio driver, *mmdrv.dll*. Source code for *sndblst.sys, sndblst.dll*, and *mmdrv.dll* are included with this DDK, at the directory paths listed in Sample Audio Drivers. When you read the topics within this section, it is helpful to refer to the source code for *sndblst.sys*.

Initializing and Configuring a Driver

The following topics examine how the kernel-mode audio driver, *sndblst.sys*, uses *soundlib.lib* functions to perform initialization and configuration operations:

- Examining **DriverEntry** in *sndblst.sys*
- Hardware and Driver Initialization
- Handling System Shutdown
- Using I/O Ports
- Using Interrupts
- Using DMA Channels

Examining DriverEntry in *sndblst.sys*

Like all kernel-mode drivers, *sndblst.sys* supplies an initialization function called **DriverEntry** to handle initialization and configuration operations. Because this code is only executed once, it is located in the driver's INIT data segment, which is marked as discardable.

For more information about **DriverEntry**, see **DriverEntry** for Multimedia Drivers.

The **DriverEntry** function for *sndblst.sys* is located in *\src\mmedia\sndblst\driver\init.c*. As a first step, the function initializes the received device object's dispatch table. Drivers using *soundlib.lib* must specify **SoundDispatch** as the dispatcher for the IRP_MJ_CLEANUP, IRP_MJ_CLOSE, IRP_MJ_CREATE, IRP_MJ_DEVICE_CONTROL, IRP_MJ_READ ,and IRP_MJ_ WRITE control codes.

The **DriverEntry** function calls **SoundEnumSubkeys** in *soundlib.lib* to locate the registry entry for each card. For each card found, **SoundEnumSubkeys** calls back into **SoundCardInstanceInit**, also located in *init.c.* If any callback to **SoundCardInstanceInit** returns a failure, **SoundEnumSubkeys** returns immediately. After **SoundEnumSubkeys** returns, **DriverEntry** calls **SoundWriteRegistryDWORD** to write each card's initialization status into the registry.

Note: The **DriverEntry** function for *sndblst.sys* searches for multiple cards. Both *soundlib.lib* and the registry's multimedia device keys can handle multiple sound cards. However, the current implementation of *sndblst.sys* references only one card for I/O operations.

Hardware and Driver Initialization

The **DriverEntry** function in *sndblst.sys* calls **SoundEnumSubkeys**, which in turn calls the **SoundCardInstanceData** function in *sndblst.sys* (defined in *\src\mmedia\sndblst\driver\init.c*) for each card described in the registry. For each card, **SoundCardInstanceInit** is responsible for:

- Saving the registry path
- Allocating device-specific memory
- Finding the bus number and type
- Obtaining configuration parameters
- Creating device objects
- Acquiring hardware resources
- Updating configuration information in the registry.

Saving the Registry Path

When **SoundCardInstanceInit** is called, it receives a pointer to a registry path. This path is the full path to a key representing a hardware device, such as **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***\Parameters\Device***Number*. The driver saves this path in device-specific storage for later use with *soundlib.lib* functions requiring a registry path as input.

Note: Drivers that do not support multiple hardware devices do not call **SoundEnumSubkeys**. Instead, they call **SoundSaveRegistryPath** to store the registry path for later use with *soundlib.lib* functions requiring a registry path as input.

Allocating Device-Specific Memory

Waveform output and MIDI input drivers use interrupts and therefore provide interrupt service routines (ISRs) and deferred procedure calls (DPCs), as discussed in the *Kernel-Mode Drivers Design Guide*. All data that these routines access must be nonpaged. The driver allocates card-specific memory from nonpaged pool in a GLOBAL_DEVICE_INFO structure and adds the structure to a linked list.

Finding the Bus Number and Type

Calls to **SoundGetBusNumber** are used to determine which buses exist. Code in **SoundCardInstanceData** checks for ISA, EISA, and Microchannel buses, in that order. The code assumes that the Sound Blaster card is connected to the first of these buses that it finds.

This code can be modified to provide a more sophisticated bus search.

Different buses do not require different kernel-mode drivers. The Windows NT Hardware

Abstraction Layer (HAL) insulates the kernel-mode driver from the bus.

Obtaining Configuration Parameters

Another task of **SoundCardInstanceInit** is copying device configuration information from the registry into the device's GLOBAL_DEVICE_INFO structure. Calling **RtlQueryRegistryValues** is an easy way to read registry values. The registry path to the card's subkeys are passed to **SoundCardInstanceInit** by **SoundEnumSubKeys**.

For the Sound Blaster card, configuration information stored in the registry includes the card I/O address, interrupt number, and DMA channel number. Usually, configuration values stored in the registry are values that can be set on the card. They can also be read-only values. Values that can be referenced only indirectly, such as those on the Sound Blaster Pro Card that can only be accessed through a device interrupt, can also be stored in the registry.

Creating Device Objects

SoundCreateDevice is used to create a new sound device object. **SoundSaveDeviceName** stores the device name under the registry path **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***Parameters\Device***Number***De**
Code in *drvlib.lib* finds the name there when the user-mode driver opens the device. The **\Devices** subkey is volatile and thus isn't written to disk.

Acquiring Hardware Resources

Before a kernel-mode driver can use hardware resources, such as DMA channels, I/O ports, and interrupts, the resources must be acquired for use. The **SoundReportResourceUsage** function determines if specified resources are already assigned to another piece of hardware and, if they are not, assigns them to the caller.

Updating Configuration Information in the Registry

The **SoundWriteRegistryDWORD** function is called if it is necessary to update hardware configuration information stored in the registry.

Handling System Shutdown

For cards with mixer devices, call **IoRegisterShutdownNotification** so that the IRP_MJ_SHUTDOWN entry point in the driver object is called when the system is shut down. Within the shutdown routine, save the mixer settings in the registry.

Using I/O Ports

After *sndblst.sys* acquires an I/O port address range by calling **SoundReportResourceUsage**, it calls **SoundMapPortAddress** to map the physical address range to a virtual range. The virtual addresses can be used as input to HAL functions, such as **READ_PORT_UCHAR**, and **WRITE_PORT_UCHAR**, as described in the *Kernel-Mode Drivers Reference*.

Drivers call also access an address by first calling **HalTranslateBusAddress** (or a related function) to map the address, and then calling **READ_REGISTER_UCHAR, WRITE_REGISTER_UCHAR** and related functions.

Drivers must not access port addresses directly, or they will not be portable.

Using Interrupts

After *sndblst.sys* calls **SoundReportResourceUsage** to acquire an interrupt number, it calls **SoundConnectInterrupt** to connect an interrupt service routine to the interrupt. After this call is made, interrupts can be received.

Using DMA Channels

After *sndblst.sys* calls **SoundReportResourceUsage** to acquire a DMA channel, it calls **HalGetAdapter** to acquire an adapter object.

Since *sndblst.sys* uses auto-initialize DMA for transferring data to and from waveform devices, it must acquire an adapter object and allocate a buffer for DMA transfers. The buffer must be noncached and sharable between the driver and the HAL. So, *sndblst.sys* calls the **SoundGetCommonBuffer** function in *soundlib.lib*, which in turn calls **HalGetAdapter** to acquire the adapter object and **HalAllocateCommonBuffer** to allocate the buffer. (**HalAllocateCommonBuffer** is more commonly used for bus-mastering devices.) It also allocates a memory descriptor list.

Since *sndblst.sys* requires a second DMA channel without an additional DMA buffer, it calls **HalGetAdapter** directly for the second DMA Channel.

Synchronizing Driver Activities

The kernel-mode audio drivers, including the Sound Blaster driver, do not provide a **StartIo** function, so they must manage their own queues of I/O Request Packets (IRPs). You must consider synchronization issues under the following circumstances:

- Opening and closing devices.
- Handling I/O requests, which are sent when a client calls **ReadFile**, **WriteFile**, or **DeviceIoControl**, and which are received in IRPs.
- Sharing hardware. For instance, MIDI input and output hardware sometimes uses the same I/O ports as waveform input and output hardware.
- Synchronizing with Deferred Procedure Call routines (DPCs).
- Synchronizing with Interrupt Service Routines (ISRs).

Most of the time, kernel-mode audio drivers execute at PASSIVE_LEVEL priority. Synchronization can be achieved by:

- Using exclusion routines
- Using spin locks
- Using **KeSynchronizeExecution**
- Using system worker threads

Using Exclusion Routines

Drivers using *soundlib.lib* must define an exclusion routine that accepts input messages. The exclusion routine is called under the following circumstances:

- When a device is opened for writing.
  The exclusion routine receives a **SoundExcludeOpen** message.
- When a device, previously opened for writing, is closed.
  The exclusion routine receives a **SoundExcludeClose** message.
- When a request is directed to a device opened for writing.
  The exclusion routine receives a **SoundExcludeEnter** message.
- When a request to a device opened for writing is complete.
  The exclusion routine receives a **SoundExcludeLeave** message.
- When the caller wants to confirm that the drive is open.
  The exclusion routine receives a **SoundExcludeQueryOpen** message.

The exclusion routine is specified as the **ExclusionRoutine** member of a SOUND_DEVICE_INIT structure. Exclusion messages are defined in *devices.h.* In *sndblst.sys*, the exclusion routine is called **SoundExcludeRoutine** and is contained in the file *\src\mmedia\sndblst\driver\init.c*. The function's purpose is to implement mutual exclusion during write operations. For *sndblst.sys*, **SoundExcludeRoutine** does this by calling **KeWaitForSingleObject** when it receives the **SoundExcludeEnter** message and **KeReleaseMutex** when it receives **SoundExcludeLeave**.

For information on mutexes, see the *Kernel-Mode Drivers Design Guide*.

Using Spin Locks

Code that references the same objects that the driver's deferred procedure call (DPC) function references must be synchronized to avoid simultaneous attempts at referencing the same object. Drivers can acquire a spin lock at a specified IRQL in order to prevent other processors, or other code running at a lower IRQL, from simultaneously referencing an object. To synchronize access to objects referenced by a DPC function, the spin lock is obtained at an IRQL of DISPATCH_LEVEL.

Spin locks are used within *soundlib.lib.*

The *Kernel-Mode Drivers Design Guide* provides an extensive discussion of spin locks.

Using KeSynchronizeExecution

Code that references the same objects that the driver's interrupt service routine (ISR) references must be synchronized to avoid simultaneous attempts at referencing the same object. Drivers must use **KeSynchronizeExecution** to achieve this synchronization. For example, *sndblst.sys* provides synchronization routines that set up the hardware for either DSP or mixer operations. Then a single ISR is provided to handle interrupts. The synchronization routines provided by *sndblst.sys* are contained in *\src\mmedia\sndblst\driver\hardware.c.*

Using System Worker Threads

It is sometimes useful to delegate the completion of some non-time-critical tasks to a system worker thread. An example is the quiescing of some older, slower sound cards after wave output has completed.

For wave devices, *soundlib.lib* queues a worker routine that you specify as the **HwStopDMA** member in a WAVE_INFO structure. This routine is called after your DPC completes. (For more information, see SOUND_DEVICE_INIT.) Code in *soundlib.lib* calls **ExInitializeWorkItem** and **ExQueueWorkItem** to cause a system worker thread to execute the function pointed to by **HwStopDMA** in WAVE_INFO (which, in *sndblst.sys*, happens to be a function called **HwStopDMA**).

Supporting Waveform Devices

Of the types of operations supported by *sndblst.sys*, waveform I/O operations are the most complex. Waveform operations are complex because they require the use of both interrupts and auto-initialize DMA. Functions within *soundlib.lib* support both interrupt and DMA operations for waveform devices. To use *soundlib.lib* for handling waveform devices, you must:

- Within the driver object received by **DriverEntry**, assign **SoundDispatch** to be the driver's main dispatcher for IRP control codes. See "Examining **DriverEntry** in *sndblst.sys*."
- Define a SOUND_DEVICE_INIT structure for waveform input, and another for waveform output. The structures' **DispatchRoutine** members must be set to the address of **SoundWaveDispatch**, which is the waveform dispatch routine within *soundlib.lib* for **DeviceIoControl** messages.
- Define a WAVE_INFO structure for each waveform device that can be in operation simultaneously. If your device can only support waveform input or output at one time, then only one WAVE_INFO structure is needed. If input and output can occur simultaneously, two structures must be defined. In *sndblst.sys*, only one WAVE_INFO structure is defined because input and output cannot occur simultaneously.
- Call **SoundCreateDevice** once for each SOUND_DEVICE_INIT structure you have defined.

Supporting MIDI Devices

Functions in *soundlib.lib* provide support for MIDI synthesizers and external MIDI devices. Both Ad Lib and OPL3 synthesizer types are supported. External MIDI devices are supported in UART mode. All of these capabilities are utilized by *sndblst.sys*.

Supporting MIDI Synthesizers

To make use of the synthesizer functions provided in *soundlib.lib*, do the following:

- Within the driver object received by **DriverEntry**, assign **SoundDispatch** to be the driver's main dispatcher for IRP control codes. See "Examining **DriverEntry** in *sndblst.sys*."
- Call the **SynthInit** function for each card, during hardware and driver initialization.
- Call the **SynthCleanup** function for each card, just before the driver is unloaded.

Additionally, if your hardware generates an interrupt for timer expiration, your driver must provide code to connect to the interrupt and to dismiss the interrupt. This interrupt is not handled by *soundlib.lib*.

Supporting External MIDI Devices

To use *soundlib.lib* for handling external MIDI devices, you must:

- Within the driver object received by **DriverEntry**, assign **SoundDispatch** to be the driver's main dispatcher for IRP control codes. See "Examining **DriverEntry** in *sndblst.sys*."
- Define a SOUND_DEVICE_INIT structure for MIDI input, and another for MIDI output. The structures' **DispatchRoutine** members must be set to the address of **SoundMidiDispatch**, which is the MIDI dispatch routine within *soundlib.lib* for **DeviceIoControl** messages.
- Define a MIDI_INFO structure. A single MIDI_INFO structure can support both MIDI output and MIDI input.
- Call **SoundCreateDevice** once for each SOUND_DEVICE_INIT structure you have defined.

Supporting Mixer Devices

User-mode mixer drivers send IOCTL_MIX_REQUEST_NOTIFY messages to kernel-mode drivers to request notification of changes to line and control information. If a user-mode driver uses *drvlib.lib*, code in *drvlib.lib* begins continually calling **DeviceIoControl**, sending IOCTL_MIX_REQUEST_NOTIFY messages, after MXDM_OPEN is received. (This assumes the client has stipulated change notification when sending MXDM_OPEN.) Functions are provided in *soundlib.lib* to assist kernel-mode drivers in responding to IOCTL_MIX_REQUEST_NOTIFY messages. Code in *soundlib.lib*'s **SoundMixerDispatch** queues the IRPs associated with these notification requests.

Kernel-mode drivers call the **SoundMixerChangedItem** function to queue information about line and control changes. This function, in turn, dequeues the queued IRPs, writes changed information into each IRP structure, and calls **IoCompleteRequest** to complete the I/O request and pass the changed information back to *drvlib.lib*, in user mode.

Kernel-mode drivers call **SoundSetLineNotify** to register a routine that *soundlib.lib* calls whenever the status of a line changes. For wave devices, *soundlib.lib* calls this routine whenever the device state should change from inactive to active, or vice versa. The routine typically sets hardware appropriately and calls **SoundMixerChangedItem**.

To use *soundlib.lib* for handling mixer devices, you must:

- Within the driver object received by **DriverEntry**, assign **SoundDispatch** to be the driver's main dispatcher for IRP control codes. See "Examining **DriverEntry** in *sndblst.sys*."
- Define a SOUND_DEVICE_INIT structure for each mixer device. Mixer drivers generally support one device instance per card. The structures' **DispatchRoutine** members must be set to the address of **SoundMixerDispatch**, which is the mixer dispatch routine within *soundlib.lib* for **DeviceIoControl** messages.
- Define a MIXER_INFO structure.

- Call **SoundCreateDevice** once for each SOUND_DEVICE_INIT structure you have defined.
- Assign the address of the mixer device's LOCAL_DEVICE_INFO structure to the **MixerDevice** member of every *other* device's LOCAL_DEVICE_INFO structure.

Mixer drivers should save their settings in the registry before system shutdown. To register for shutdown notification, call **IoRegisterShutdownNotification**. Drivers can save mixer settings in the registry in any format, but the most efficient registry data type for saving the settings is REG_BINARY. When the driver initializes, it can use either the stored settings or default settings.

Supporting Auxiliary Audio Devices

Kernel-mode drivers that support auxiliary audio devices can use the auxiliary audio message dispatcher, **SoundAuxDispatch**, provided by *soundlib.lib*. If you use *soundlib.lib*, the amount of customized code a driver needs to provide is minimal. If the hardware provides a mixer, then *soundlib.lib* calls the mixer's **HwSetControlData** and **HwGetControlData** functions to control volume (see MIXER_INFO). If the hardware does not provide mixer hardware, the driver needs to include a **HwSetVolume** function for the auxiliary audio device, to control volume (see SOUND_DEVICE_INIT). A **DevCapsRoutine** is the only other function you need to provide (see SOUND_DEVICE_INIT).

To use *soundlib.lib* for handling auxiliary audio devices, you must:

- Within the driver object received by **DriverEntry**, assign **SoundDispatch** to be the driver's main dispatcher for IRP control codes. See "Examining **DriverEntry** in *sndblst.sys*."
- Define a SOUND_DEVICE_INIT structure. The structures' **DispatchRoutine** members must be set to the address of **SoundAuxDispatch**, which is the auxiliary audio dispatch routine within *soundlib.lib* for **DeviceIoControl** messages.
- Call **SoundCreateDevice** once for each SOUND_DEVICE_INIT structure you have defined.

Audio Driver Reference

This section describes the functions, messages, structures, and types used by audio drivers.

Topics for user-mode drivers include:

- Entry Points, User-Mode Audio Drivers
- Messages, User-Mode Audio Drivers
- Structures, User-Mode Audio Drivers
- Functions and Macros, *drvlib.lib*
- Structures and Types, *drvlib.lib*

Topics for kernel-mode drivers include:

- Functions, *soundlib.lib*
- Structures and Types, *soundlib.lib*

Entry Points, User-Mode Audio Drivers

This section describes the entry point functions that user-mode audio drivers must provide.

auxMessage

DWORD APIENTRY
auxMessage (
UINT *uDeviceId*,
UINT *uMsg*,

DWORD *dwUser*,
DWORD *dwParam1*,
DWORD *dwParam2*
);

The **auxMessage** function is one of the user-mode audio driver entry points. It is the entry point for user-mode auxiliary audio drivers.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
Specifies one of the user-mode audio driver messages.

dwUser
Not used.

dwParam1
Specifies the first message parameter. Dependent on message type.

dwParam2
Specifies the second message parameter. Dependent on message type.

Return Value

The **auxMessage** function returns a value that is dependent upon the message. If the received message is not recognized, the function returns MMSYSERR_NOTSUPPORTED.

midMessage

DWORD APIENTRY
midMessage (
UINT *uDeviceId*,
UINT *uMsg*,
DWORD *dwUser*,
DWORD *dwParam1*,
DWORD *dwParam2*
);

The **midMessage** function is one of the user-mode audio driver entry points. It is the entry point for user-mode MIDI input drivers.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
Specifies one of the user-mode audio driver messages.

dwUser
Specifies a device instance identifier. For the MIDM_OPEN message, this is an *output* parameter. The driver creates the instance identifier and returns it in the address specified as the argument. For all other messages, this is an *input* parameter. The argument is the instance identifier.

dwParam1
Specifies the first message parameter. Dependent on message type.

dwParam2
Specifies the second message parameter. Dependent on message type.

Return Value

The **midMessage** function returns a value that is dependent upon the message. If the received message is not recognized, the function returns MMSYSERR_NOTSUPPORTED.

Comments

You can use *dwUser* in any manner you wish. Drivers that can support multiple clients return a different value for each MIDM_OPEN message, in order to keep track of which subsequent messages are being sent by which client.

modMessage

DWORD APIENTRY
modMessage (
UINT *uDeviceId*,
UINT *uMsg*,
DWORD *dwUser*,
DWORD *dwParam1*,
DWORD *dwParam2*
);

The **modMessage** function is one of the user-mode audio driver entry points. It is the entry point for user-mode MIDI output drivers.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
Specifies one of the user-mode audio driver messages.

dwUser
Specifies a device instance identifier. For the MODM_OPEN message, this is an *output* parameter. The driver creates the instance identifier and returns it in the address specified as the argument. For all other messages, this is an *input* parameter. The argument is the instance identifier.

dwParam1
Specifies the first message parameter. Dependent on message type.

dwParam2
Specifies the second message parameter. Dependent on message type.

Return Value

The **modMessage** function returns a value that is dependent upon the message. If the received message is not recognized, the function returns MMSYSERR_NOTSUPPORTED.

Comments

You can use *dwUser* in any manner you wish. Drivers that can support multiple clients return a different value for each MODM_OPEN message, in order to keep track of which subsequent messages are being sent by which client.

mxdMessage

DWORD APIENTRY
mxdMessage (
UINT *uDeviceId*,
UINT *uMsg*,
DWORD *dwUser*,
DWORD *dwParam1*,
DWORD *dwParam2*
);

The **mxdMessage** function is one of the user-mode audio driver entry points. It is the entry point for user-mode mixer drivers.

Parameters

uDeviceId
Device identifier (0, 1, 2, etc.) for the target device.

uMsg
Specifies one of the user-mode audio driver messages.

dwUser
Specifies a device instance identifier. For the MXDM_OPEN message, this is an *output* parameter. The driver creates the instance identifier and returns it in the address specified as the argument. For all other messages, this is an *input* parameter. The argument is the instance identifier.

dwParam1
Specifies the first message parameter. Dependent on message type.

dwParam2
Specifies the second message parameter. Dependent on message type.

Return Value

The **mxdMessage** function returns a value that is dependent upon the message. If the received message is not recognized, the function returns MMSYSERR_NOTSUPPORTED.

Comments

You can use *dwUser* in any manner you wish. Drivers that can support multiple clients return a different value for each MXDM_OPEN message, in order to keep track of which subsequent messages are being sent by which client.

widMessage

```
DWORD APIENTRY
    widMessage (
      UINT uDeviceId,
      UINT uMsg,
      DWORD dwUser,
      DWORD dwParam1,
      DWORD dwParam2
      );
```

The **widMessage** function is one of the user-mode audio driver entry points. It the entry point for user-mode waveform input drivers.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
Specifies one of the user-mode audio driver messages.

dwUser
Specifies a device instance identifier. For the WIDM_OPEN message, this is an *output* parameter. The driver creates the instance identifier and returns it in the address specified as the argument. For all other messages, this is an *input* parameter. The argument is the instance identifier.

dwParam1
Specifies the first message parameter. Dependent on message type.

dwParam2
Specifies the second message parameter. Dependent on message type.

Return Value

The **widMessage** function returns a value that is dependent upon the message. If the received message is not recognized, the function returns MMSYSERR_NOTSUPPORTED.

Comments

You can use *dwUser* in any manner you wish. Drivers that can support multiple clients return a different value for each WIDM_OPEN message, in order to keep track of which subsequent messages are being sent by which client.

wodMessage

DWORD APIENTRY
wodMessage (
UINT *uDeviceId*,
UINT *uMsg*,
DWORD *dwUser*,
DWORD *dwParam1*,
DWORD *dwParam2*
);

The **wodMessage** function is one of the user-mode audio driver entry points. It the entry point for user-mode waveform output drivers.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
Specifies one of the user-mode audio driver messages.

dwUser
Specifies a device instance identifier. For the WODM_OPEN message, this is an *output* parameter. The driver creates the instance identifier and returns it in the address specified as the argument. For all other messages, this is an *input* parameter. The argument is the instance identifier.

dwParam1
Specifies the first message parameter. Dependent on message type.

dwParam2
Specifies the second message parameter. Dependent on message type.

Return Value

The **wodMessage** function returns a value that is dependent upon the message. If the received message is not recognized, the function returns MMSYSERR_NOTSUPPORTED.

Comments

You can use *dwUser* in any manner you wish. Drivers that can support multiple clients return a different value for each WODM_OPEN message, in order to keep track of which subsequent messages are being sent by which client.

Messages, User-Mode Audio Drivers

This section describes the messages, listed in alphabetical order, that are received by user-mode audio drivers.

AUXDM_GETDEVCAPS

The AUXDM_GETDEVCAPS message requests an auxiliary audio driver to return the capabilities of the specified auxiliary audio device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
AUXDM_GETDEVCAPS

dwUser
Not used.

dwParam1
Pointer to an empty AUXCAPS structure. This structure is used to return the capabilities of the device. (The AUXCAPS structure is described in the Win32 SDK.)

dwParam2
Size of the AUXCAPS structure in bytes.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*.

Comments

A client sends the AUXDM_GETDEVCAPS message by calling the user-mode driver's **auxMessage** entry point, passing the specified parameters.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_AUX_GET_CAPABILITIES control code.

The user-mode driver fills the AUXCAPS structure.

AUXDM_GETNUMDEVS

The AUXDM_GETNUMDEVS message requests an auxiliary audio driver to return the number of auxiliary audio device instances that it supports.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
AUXDM_GETNUMDEVS

dwUser
Not used.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver returns the number of auxiliary audio device instances it supports.

Comments

A client sends the AUXDM_GETNUMDEVS message by calling the user-mode driver's **auxMessage** entry point, passing the specified parameters.

The driver should return the number of logical auxiliary audio devices that can be supported. Typically, for each physical device, a kernel-mode driver can support one or more logical devices of various types. For example, for each Creative Labs Sound Blaster card, there are MIDI, waveform input, mixer, and auxiliary audio devices. Kernel-mode drivers store logical device names and types in the registry under the path **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***Parameters\Device***Number***De**
To correctly return the number of logical devices, the user-mode driver should examine the **\Devices** subkey for each of the driver's **\Device***Number* keys, searching for logical devices of the desired type. (Code in *drvlib.lib* provides this capability.)

The AUXDM_GETVOLUME message requests a user-mode driver to return the current volume level setting for the specified auxiliary audio device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
AUXDM_GETVOLUME

dwUser
Not used.

dwParam1
Pointer to a DWORD location to receive the volume setting.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*.

Comments

A client sends the AUXDM_GETVOLUME message by calling the user-mode driver's **auxMessage** entry point, passing the specified parameters.

Support for this message by user-mode drivers is optional. If the driver supports AUXDM_SETVOLUME, it must support AUXDM_GETVOLUME.

The volume value is returned in the DWORD pointed to by *dwParam1* as follows:

Channel	**Portion of *dwParam1* Used**
Left channel	Low word
Right channel	High word
Single channel	Low word

A value of zero is silence, and a value of 0xFFFF is full volume.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_AUX_GET_VOLUME control code.

AUXDM_SETVOLUME

The AUXDM_SETVOLUME message requests a user-mode driver to set the volume level for the specified auxiliary audio device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
AUXDM_SETVOLUME

dwUser
Not used.

dwParam1
A DWORD containing the volume setting.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*.

Comments

A client sends the AUXDM_SETVOLUME message by calling the user-mode driver's **auxMessage** entry point, passing the specified parameters.

Support for this message by user-mode drivers is optional. If the driver supports AUXDM_SETVOLUME, it must support AUXDM_GETVOLUME.

The volume value is specified by *dwParam1* as follows.

Channel	Portion of *dwParam1* Used
Left channel	Low word
Right channel	High word
Single channel	Low word

A value of zero is silence, and a value of 0xFFFF is full volume.

The kernel-mode driver might not support the full 16 bits of volume control and can truncate the lower bits. However, the original value requested with AUXDM_SETVOLUME should be returned with AUXDM_GETVOLUME.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_AUX_SET_VOLUME control code.

MIDM_ADDBUFFER

The MIDM_ADDBUFFER message requests a user-mode MIDI input driver to add an empty input buffer to its input buffer queue.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
AUXDM_SETVOLUME

dwUser
Device instance identifier.

dwParam1
Pointer to a MIDIHDR structure identifying the buffer. (The MIDIHDR structure is described in the Win32 SDK.)

dwParam2
Size of the MIDIHDR structure in bytes.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiInAddBuffer** return values in the Win32 SDK.

Comments

A client sends the AUXDM_SETVOLUME message by calling the user-mode driver's **midMessage** entry point, passing the specified parameters.

If the MHDR_PREPARED flag is not set in the **dwFlags** member of the MIDIHDR structure, the driver should return MIDIERR_UNPREPARED. If the flag is set, the driver should:

- Clear the MHDR_DONE flag.
- Set the MHDR_INQUEUE flag.
- Place the empty buffer in its input queue.

- Return control to the client with a return value of MMSYSERR_NOERROR.

Only system-exclusive events (long messages) should be placed in the buffer. Other MIDI events (short messages) should be passed to the client with a MIM_DATA callback message.

The user-mode driver starts recording when it receives a MIDM_START message.

For additional information, see Transferring MIDI Input Data.

MIDM_CLOSE

The MIDM_CLOSE message requests a MIDI input driver to close a specified device instance that was previously opened with a MIDM_OPEN message.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MIDM_CLOSE

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiInClose** return values in the Win32 SDK.

Comments

A client sends the MIDM_CLOSE message by calling the user-mode driver's **midMessage** entry point, passing the specified parameters.

If the driver has not filled and returned all of the buffers received with MIDM_ADDBUFFER messages, it should not close the instance and should instead return MIDIERR_STILLPLAYING.

After the driver closes the device instance it should send a MIM_CLOSE callback message to the client.

For more information about closing a device instance, see Transferring MIDI Input Data.

MIDM_GETDEVCAPS

The MIDM_GETDEVCAPS message requests a MIDI input driver to return the capabilities of a specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MIDM_GETDEVCAPS

dwUser
Device instance identifier.

dwParam1
Pointer to a MIDIINCAPS structure. (The MIDIINCAPS structure is described in the Win32 SDK.)

dwParam2

Size of the MIDIINCAPS structure in bytes.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiInGetDevCaps** return values in the Win32 SDK.

Comments

A client sends the MIDM_GETDEVCAPS message by calling the user-mode driver's **midMessage** entry point, passing the specified parameters.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_MIDI_GET_CAPABILITIES control code.

The user-mode driver fills the MIDIINCAPS structure.

MIDM_GETNUMDEVS

The MIDM_GETNUMDEVS message requests a MIDI input driver to return the number of device instances that it supports.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MIDM_GETNUMDEVS

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver returns the number of MIDI input device instances it supports.

Comments

A client sends the MIDM_GETNUMDEVS message by calling the user-mode driver's **midMessage** entry point, passing the specified parameters.

The driver should return the number of logical MIDI input devices that can be supported. Typically, for each physical device, a kernel-mode driver can support one or more logical devices of various types. For example, for each Creative Labs Sound Blaster™ card, there are MIDI, waveform, mixer, and auxiliary audio devices. Kernel-mode drivers store logical device names and types in the registry under the path **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***\Parameters\Device***Number***\De**
To correctly return the number of logical devices, the user-mode driver should examine the **\Devices** subkey for each of the driver's **\Device***Number* keys, searching for logical devices of the desired type. (Code in *drvlib.lib* provides this capability.)

MIDM_OPEN

The MIDM_OPEN message is sent to a MIDI input driver to request it to open an instance of a specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MIDM_OPEN

dwUser
Pointer to location to receive device instance identifier.

dwParam1
Pointer to a MIDIOPENDESC structure, containing the client's device handle, notification target, and instance ID.

dwParam2
Contains flags. The following flags are defined.

Flag	Definition
CALLBACK_WINDOW	Indicates the **dwCallback** member of MIDIOPENDESC is a window handle.
CALLBACK_FUNCTION	Indicates the **dwCallback** member of MIDIOPENDESC is the address of a callback function.
CALLBACK_TASK	Indicates the **dwCallback** member of MIDIOPENDESC is a task handle.
MIDI_IO_STATUS	Indicates the client wants to receive MIM_MOREDATA callback messages.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiInOpen** return values in the Win32 SDK.

Comments

A client sends the MIDM_OPEN message by calling the user-mode driver's **midMessage** entry point, passing the specified parameters.

Typically, user-mode drivers connect to kernel-mode drivers by calling **CreateFile**, specifying the MS-DOS device name of one of the kernel-mode driver's devices.

The driver assigns a device instance identifier and returns it in the location pointed to by *dwUser*. The driver can expect to receive this value as the *dwUser* input argument to all other **midMessage** messages.

The driver determines the number of clients it allows to use a particular device. If a device is opened by the maximum number of clients, it returns MMSYSERR_ALLOCATED for subsequent open requests.

If the open operation succeeds, the driver should send the client a MIM_OPEN message by calling the **DriverCallback** function.

For additional information, see Transferring MIDI Input Data.

MIDM_PREPARE

The MIDM_PREPARE message requests a MIDI input driver to prepare a system-exclusive data buffer for input.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MIDM_PREPARE

dwUser
Device instance identifier.

dwParam1

Pointer to a MIDIHDR structure identifying the buffer. (The MIDIHDR structure is described in the Win32 SDK.)

dwParam2
Size of the MIDIHDR structure.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiInPrepareHeader** return values in the Win32 SDK.

Comments

A client sends the MIDM_PREPARE message by calling the user-mode driver's **midMessage** entry point, passing the specified parameters.

Support for this message by user-mode drivers is optional. If the driver supports MIDM_PREPARE, it must also support MIDM_UNPREPARE.

If the driver returns MMSYSERR_NOTSUPPORTED, *winmm.dll* prepares the buffer for use. For most drivers, this behavior is sufficient. If the driver does perform buffer preparation, it must set MHDR_PREPARED in the **dwFlags** member of MIDIHDR and return MMSYSERR_NOERROR.

For additional information, see Transferring MIDI Input Data.

MIDM_RESET

The MIDM_RESET message requests a MIDI input driver to stop recording and return all buffers in the input queue to the client.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MIDM_RESET

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiInReset** return values in the Win32 SDK.

Comments

A client sends the MIDM_RESET message by calling the user-mode driver's **midMessage** entry point, passing the specified parameters.

Typically, the user-mode driver stops recording by calling **DeviceIoControl,** sending the kernel-mode driver an IOCTL_MIDI_SET_STATE control code.

For each buffer remaining in the driver's input queue (see MIDM_ADDBUFFER), the driver should set MHDR_DONE and clear MHDR_INQUEUE in the **dwFlags** member of the buffer's MIDIHDR structure, and also set the structure's **dwBytesRecorded** member. Finally, a MOM_DONE callback message should be sent for each buffer.

For additional information, see Transferring MIDI Input Data.

The MIDM_START message requests a MIDI input driver to begin recording.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MIDM_START

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiInStart** return values in the Win32 SDK.

Comments

A client sends the MIDM_START message by calling the user-mode driver's **midMessage** entry point, passing the specified parameters.

This message resets the time stamp to zero. If the message is received after input has been started, the driver should return MMSYSERR_NOERROR.

Typically, the user-mode driver starts recording by calling **ReadFileEx**, to fill internal buffers with data returned by the kernel-mode driver, and by calling **DeviceIoControl**, to send the kernel-mode driver an IOCTL_MIDI_SET_STATE control code. When the kernel-mode driver returns a filled buffer, the user-mode driver should read the buffer data to differentiate short MIDI messages from long MIDI messages. The user-mode driver returns each short message to the client by means of a MIM_DATA callback message. It should copy long messages into the user-specified input buffers (see MIDM_ADDBUFFER) and, when a buffer is full, do the following:

- Set the **dwBytesRecorded** member in the buffer's MIDIHDR structure.
- Set the buffer's MHDR_DONE flag.
- Clear the buffer's MHDR_INQUEUE flag.
- Send a MIM_LONGDATA callback message to the client.

If the driver receives long messages with no buffers in its input queue, it should ignore the messages without notifying the client.

Recording should continue until the client sends MIDM_STOP or MIDM_RESET.

For additional information, see Transferring MIDI Input Data.

MIDM_STOP

The MIDM_STOP message requests a MIDI input driver to stop recording.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MIDM_STOP

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiInStop** return values in the Win32 SDK.

Comments

A client sends the MIDM_STOP message by calling the user-mode driver's **midMessage** entry point, passing the specified parameters.

If a buffer in the input queue (see MIDM_ADDBUFFER) has been partially filled, the driver should treat it as a full buffer and return it to the client (see MIDM_START). Empty buffers should remain in the queue.

While recording is stopped, the driver should maintain the current MIDI status byte for events using running status and the parsing state for multibyte events. If the driver receives a subsequent MIDM_START message, it should be able to resume recording from the point at which it was stopped.

If this message is received and recording is already stopped, the driver should return MMSYSERR_NOERROR.

Typically, the user-mode driver stops recording by calling **DeviceIoControl**, sending the kernel-mode driver an IOCTL_MIDI_SET_STATE control code.

For additional information, see Transferring MIDI Input Data.

MIDM_UNPREPARE

The MIDM_UNPREPARE message requests a MIDI input driver to remove the buffer preparation that was performed in response to a MIDM_PREPARE message.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MIDM_UNPREPARE

dwUser
Device instance identifier.

dwParam1
Specifies a pointer to MIDIHDR identifying the data buffer. (The MIDIHDR structure is described in the Win32 SDK.)

dwParam2
Specifies the size of MIDIHDR in bytes.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiInUnprepareHeader** return values in the Win32 SDK.

Comments

A client sends the MIDM_UNPREPARE message by calling the user-mode driver's **midMessage** entry point, passing the specified parameters.

Support for this message by user-mode drivers is optional. If the driver supports

MIDM_PREPARE, it must also support MIDM_UNPREPARE.

If the driver returns MMSYSERR_NOTSUPPORTED, *winmm.dll* removes the buffer preparation. For most drivers, this behavior is sufficient. If the driver does support MIDM_UNPREPARE, it must clear MHDR_PREPARED in the **dwFlags** member of MIDIHDR and return MMSYSERR_NOERROR.

For additional information, see Transferring MIDI Input Data.

MIM_CLOSE

The MIM_CLOSE callback message notifies a client that a user-mode driver has finished processing a MIDM_CLOSE message.

Parameters

dwMsg
MIM_CLOSE

dwParam1
NULL

dwParam2
NULL

Comments

A user-mode MIDI input driver sends a MIM_CLOSE message to its client, by means of a callback, when the driver finishes processing a MIDM_CLOSE message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the MIM_CLOSE message only if the client has previously specified a notification target with a MIDM_OPEN message.

Win32 SDK documentation states that clients receive an MM_MIM_CLOSE message if the notification target is a window handle. MIM_CLOSE and MM_MIM_CLOSE are equivalent.

For additional information, see Notifying Clients from Audio Drivers and Transferring MIDI Input Data.

MIM_DATA

The MIM_DATA callback message notifies a client that a user-mode driver has received a MIDI short message.

Parameters

dwMsg
MIM_DATA

dwParam1
MIDI short message contents (one to three bytes).

dwParam2
Time stamp. Number of milliseconds since MIDM_START was received.

Comments

A user-mode MIDI input driver sends a MIM_DATA message to its client, by means of a callback, when the driver has received a MIDI short message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the MIM_DATA message only if the client has previously specified a notification target with a MIDM_OPEN message.

If the driver detects that the message is invalid, it should send MIM_ERROR instead of MIM_DATA.

Win32 SDK documentation states that clients receive an MM_MIM_DATA message if the notification target is a window handle. MIM_DATA and MM_MIM_DATA are equivalent.

For additional information, see Notifying Clients from Audio Drivers and Transferring MIDI Input Data.

MIM_ERROR

The MIM_ERROR callback message notifies a client that a user-mode driver has received an invalid MIDI short message.

Parameters

dwMsg
MIM_ERROR

dwParam1
Invalid MIDI short message contents (one to three bytes).

dwParam2
Time stamp. Number of milliseconds since MIDM_START was received.

Comments

A user-mode MIDI input driver sends a MIM_ERROR message to its client, by means of a callback, when the driver has received an invalid MIDI short message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the MIM_ERROR message only if the client has previously specified a notification target with a MIDM_OPEN message.

Win32 SDK documentation states that clients receive a MM_MIM_ERROR message if the notification target is a window handle. MIM_ERROR and MM_MIM_ERROR are equivalent.

For more information, see Notifying Clients from Audio Drivers and Transferring MIDI Input Data.

MIM_LONGDATA

The MIM_LONGDATA callback message notifies a client that a user-mode driver has received a MIDI system-exclusive (long) message.

Parameters

dwMsg
MIM_LONGDATA

dwParam1
Address of a MIDIHDR structure identifying a buffer containing the long message. (MIDIHDR is defined in the Win32 SDK.)

dwParam2
Time stamp. Number of milliseconds since MIDM_START was received.

Comments

A user-mode MIDI input driver sends a MIM_LONGDATA message to its client, by means of a callback, when the driver has received a MIDI system-exclusive (long) message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the MIM_LONGDATA message only if the client has previously specified a notification target with a MIDM_OPEN message.

If the driver detects that the message is invalid, it should send MIM_LONGERROR instead of MIM_LONGDATA.

Win32 SDK documentation states that clients receive a MM_MIM_LONGDATA message if the notification target is a window handle. MIM_LONGDATA and MM_MIM_LONGDATA are equivalent.

For additional information, see Notifying Clients from Audio Drivers and Transferring MIDI Input Data.

MIM_LONGERROR

The MIM_LONGERROR callback message notifies a client that a user-mode driver has received an invalid MIDI system-exclusive (long) message.

Parameters

dwMsg
MIM_LONGERROR

dwParam1
Address of a MIDIHDR structure identifying a buffer containing the invalid long message. (MIDIHDR is defined in the Win32 SDK.)

dwParam2
Time stamp. Number of milliseconds since MIDM_START was received.

Comments

A user-mode MIDI input driver sends a MIM_LONGERROR message to its client, by means of a callback, when the driver has received an invalid MIDI system-exclusive (long) message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the MIM_LONGERROR message only if the client has previously specified a notification target with a MIDM_OPEN message.

When a driver detects an invalid message long message, it should send MIM_LONGERROR instead of MIM_LONGDATA.

Win32 SDK documentation states that clients receive a MM_MIM_LONGERROR message if the notification target is a window handle. MIM_LONGERROR and MM_MIM_LONGERROR are equivalent.

For additional information, see Notifying Clients from Audio Drivers and Transferring MIDI Input Data.

MIM_MOREDATA

The MIM_MOREDATA callback message notifies a client that a user-mode driver has received a MIDI short message, and the client is not processing MIM_DATA messages fast enough to keep up with the driver.

Parameters

dwMsg
MIM_MOREDATA

dwParam1
MIDI short message contents (one to three bytes).

dwParam2
Time stamp. Number of milliseconds since MIDM_START was received.

Comments

A user-mode MIDI input driver sends a MIM_MOREDATA message to its client, by means of a callback, when the driver has received a MIDI short message and the client is not processing MIM_DATA messages fast enough to keep up with the driver. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the MIM_MOREDATA message only if the client has previously specified a notification target with a MIDM_OPEN message, *and* only if the MIDI_IO_STATUS flag was included with the MIDM_OPEN message.

Currently, *mmdrv.dll* and *drvlib.lib* do not send MIM_MOREDATA messages.

If the driver detects that the message is invalid, it should send MIM_ERROR instead of MIM_MOREDATA.

Win32 SDK documentation states that clients receive a MM_MIM_MOREDATA message if the notification target is a window handle. MIM_MOREDATA and MM_MIM_MOREDATA are equivalent.

For additional information, see Notifying Clients from Audio Drivers and Transferring MIDI Input Data.

MIM_OPEN

The MIM_OPEN callback message notifies a client that a user-mode driver has finished processing a MIDM_OPEN message.

Parameters

dwMsg
MIM_OPEN

dwParam1
NULL

dwParam2
NULL

Comments

A user-mode MIDI input driver sends a MIM_OPEN message to its client, by means of a callback, when the driver finishes processing a MIDM_OPEN message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the MIM_OPEN message only if the client has specified a notification target with the MIDM_OPEN message.

Win32 SDK documentation states that clients receive an MM_MIM_OPEN message if the notification target is a window handle. MIM_OPEN and MM_MIM_OPEN are equivalent.

For additional information, see Notifying Clients from Audio Drivers and Transferring MIDI Input Data.

MM_MIXM_CONTROL_CHANGE

The MM_MIXM_CONTROL_CHANGE callback message notifies a client that the value of a mixer control item has changed.

Parameters

dwMsg
MM_MIXM_CONTROL_CHANGE

dwParam1
Control ID. Must match control ID value returned to the client in response to an MXDM_GETLINECONTROLS message.

dwParam2
NULL

Comments

A user-mode mixer driver sends an MM_MIXM_CONTROL_CHANGE message to its client, by means of a callback, when the value of a mixer control item has changed. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the MM_MIXM_CONTROL_CHANGE message only if the client has specified a notification target with the MXDM_OPEN message.

For kernel-mode drivers using *soundlib.lib*, this value is stored in a MIXER_DATA_ITEM structure and passed to the user-mode driver upon receipt of an IOCTL_MIX_REQUEST_NOTIFY control code.

For additional information, see Notifying Clients from Audio Drivers.

MM_MIXM_LINE_CHANGE

The MM_MIXM_LINE_CHANGE callback message notifies a client that the value of a mixer line item has changed.

Parameters

dwMsg
MM_MIXM_LINE_CHANGE

dwParam1
Line ID. Must match line ID value returned to the client in response to an MXDM_GETLINEINFO message.

dwParam2
NULL

Comments

A user-mode mixer driver sends an MM_MIXM_LINE_CHANGE message to its client, by means of a callback, when the value of a mixer line item has changed. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the MM_MIXM_LINE_CHANGE message only if the client has specified a notification target with the MXDM_OPEN message.

For kernel-mode drivers using *soundlib.lib*, this value is stored in a MIXER_DATA_ITEM structure and passed to the user-mode driver upon receipt of an IOCTL_MIX_REQUEST_NOTIFY control code.

For additional information, see Notifying Clients from Audio Drivers.

MODM_CACHEDRUMPATCHES

The MODM_CACHEDRUMPATCHES message requests a MIDI output driver to load and cache a specified set of key-based percussion patches.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MODM_CACHEDRUMPATCHES

dwUser
Device instance identifier.

dwParam1
Pointer to an array of type KEYARRAY, which is described in the Win32 SDK.

dwParam2
Contains a DWORD value, defined as follows:

Low word	Flag values. (For flag descriptions, see **midiOutCacheDrumPatches** in the Win32 SDK.)
High word	Drum patch number.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See

midiOutCacheDrumPatches return values in the Win32 SDK.

Comments

A client sends the MODM_CACHEDRUMPATCHES message by calling the user-mode driver's **modMessage** entry point, passing the specified parameters.

Typically, the user-mode driver requests the kernel-mode driver to cache the patches by calling **DeviceIoControl** with an IOCTL_MIDI_CACHE_DRUM_PATCHES control code.

MODM_CACHEPATCHES

The MODM_CACHEPATCHES message requests a MIDI output driver to load and cache a specified bank of patches.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MODM_CACHEPATCHES

dwUser
Device instance identifier.

dwParam1
Pointer to an array of type PATCHARRAY, which is described in the Win32 SDK.

dwParam2
Contains a DWORD value, defined as follows:

Low word	Flag values. (For flag descriptions, see **midiOutCachePatches** in the Win32 SDK.)
High word	Patch bank number. Zero implies the default bank.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiOutCachePatches** return values in the Win32 SDK.

Comments

A client sends the MODM_CACHEPATCHES message by calling the user-mode driver's **modMessage** entry point, passing the specified parameters.

Typically, the user-mode driver requests the kernel-mode driver to cache the patch bank by calling **DeviceIoControl** with an IOCTL_MIDI_CACHE_PATCHES control code.

MODM_CLOSE

The MODM_CLOSE message requests a MIDI output driver to close a specified device instance that was previously opened with a MODM_OPEN message.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MODM_CLOSE

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2

Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiOutClose** return values in the Win32 SDK.

Comments

A client sends the MODM_CLOSE message by calling the user-mode driver's **modMessage** entry point, passing the specified parameters.

If the client has passed data buffers to the user-mode driver by means of MODM_LONGDATA messages, and if the user-mode driver hasn't finished sending the data to the kernel-mode driver, the user-mode driver should return MIDIERR_STILLPLAYING in response to MODM_CLOSE.

After the driver closes the device instance it should send a MOM_CLOSE callback message to the client.

For additional information, see Transferring MIDI Output Data.

MODM_DATA

The MODM_DATA message requests a MIDI output driver to send a single MIDI short message.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MODM_DATA

dwUser
Device instance identifier.

dwParam1
A MIDI short message. (See **Comments** section below.)

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiOutShortMsg** return values in the Win32 SDK.

Comments

A client sends the MODM_DATA message by calling the user-mode driver's **modMessage** entry point, passing the specified parameters.

This message is used for the output of all MIDI events, except system-exclusive events. System-exclusive events are output with the MODM_LONGDATA message.

Because the client can employ running status, and because MIDI short messages have varying lengths, the user-mode driver must parse the *dwParam1* parameter to determine the number of bytes to send to the kernel-mode driver. Unused bytes in *dwParam1* are not guaranteed to be zero.

The driver can be designed to run synchronously, not returning until it sends the message, or asynchronously, returning immediately and sending the MIDI data in the background, using a separate thread.

Typically, the user-mode driver sends the message to the kernel-mode driver by calling **DeviceIoControl** with an IOCTL_MIDI_PLAY control code.

The MODM_GETDEVCAPS message requests a MIDI output driver to return the capabilities of a specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MODM_GETDEVCAPS

dwUser
Device instance identifier.

dwParam1
Pointer to a MIDIOUTCAPS structure, which is described in the Win32 SDK.

dwParam2
Size of the MIDIOUTCAPS structure in bytes.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiOutGetDevCaps** return values in the Win32 SDK.

Comments

A client sends the MODM_GETDEVCAPS message by calling the user-mode driver's **modMessage** entry point, passing the specified parameters.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_MIDI_GET_CAPABILITIES control code.

The user-mode driver fills the MIDIOUTCAPS structure.

MODM_GETNUMDEVS

The MODM_GETNUMDEVS message requests a MIDI output driver to return the number of device instances that it supports.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MODM_GETNUMDEVS

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver returns the number of MIDI output device instances it supports.

Comments

A client sends the MODM_GETNUMDEVS message by calling the user-mode driver's **modMessage** entry point, passing the specified parameters.

The driver should return the number of logical MIDI output devices that can be supported. Typically, for each physical device, a kernel-mode driver can support one or more logical devices

of various types. For example, for each Creative Labs Sound Blaster card, there are MIDI, waveform, mixer, and auxiliary audio devices. Kernel-mode drivers store logical device names and types in the registry under the path **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***\Parameters\Device***Number***\De**
To correctly return the number of logical devices, the user-mode driver should examine the **\Devices** subkey for each of the driver's **\Device***Number* keys, searching for logical devices of the desired type. (Code in *drvlib.lib* provides this capability.)

MODM_GETVOLUME

The MODM_GETVOLUME message requests a MIDI output driver to return the current volume level setting for the specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MODM_GETVOLUME

dwUser
Device instance identifier.

dwParam1
Pointer to a DWORD location to receive the volume setting.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiOutGetVolume** return values in the Win32 SDK.

Comments

A client sends the MODM_GETVOLUME message by calling the user-mode driver's **modMessage** entry point, passing the specified parameters.

Support for this message by user-mode drivers is optional. If the driver supports MODM_SETVOLUME, it must support MODM_GETVOLUME.

The volume value is returned in the DWORD pointed to by *dwParam1* as follows:

Channel	**Portion of *dwParam1* Used**
Left channel	Low word
Right channel	High word
Single channel	Low word

A value of zero is silence, and a value of 0xFFFF is full volume.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_MIDI_GET_VOLUME control code.

Only drivers for internal synthesizer devices can support volume level changes. Drivers for MIDI output ports should return a MMSYSERR_NOTSUPPORTED error for this message.

MODM_LONGDATA

The MODM_LONGDATA message requests a MIDI output driver to send the contents of a specified output buffer containing one or more MIDI events, including system-exclusive events.

Parameters

uDeviceId

Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MODM_LONGDATA

dwUser
Device instance identifier.

dwParam1
Pointer to a MIDIHDR structure identifying the output buffer. (The MIDIHDR structure is described in the Win32 SDK.)

dwParam2
Size of the MIDIHDR structure.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiOutLongMsg** return values in the Win32 SDK.

Comments

A client sends the MODM_LONGDATA message by calling the user-mode driver's **modMessage** entry point, passing the specified parameters.

If the MHDR_PREPARED flag in the **dwFlags** member of MIDIHDR is not set, the driver should return MIDIERR_UNPREPARED.

The driver should clear the MHDR_DONE flag, set the MHDR_INQUEUE flag, and place the output buffer in its output queue. The driver returns control to the client by returning MMSYSERR_NOERROR.

When the buffer contents have been sent, the driver should set the MHDR_DONE flag, clear the MHDR_INQUEUE flag, and send the client a MOM_DONE callback message.

The driver can be designed to handle MODM_LONGDATA messages synchronously, not returning until the message has been sent to the kernel-mode driver, or asynchronously, returning immediately and using a separate thread to send the MIDI data in the background.

Typically, the user-mode driver sends the buffer to the kernel-mode driver by calling **DeviceIoControl** with an IOCTL_MIDI_PLAY control code.

If clients use high-level audio interfaces, *winmm.dll* guarantees that the input buffer contains only a single MIDI event, which can be either a short or long (system-exclusive) message. On the other hand, if clients call **midiOutLongMsg**, there is no such guarantee. If your user-mode driver is *mmdrv.dll* or is based on *drvlib.lib* functions, whatever is received in the input buffer is passed directly to the kernel-mode driver by means of the **DeviceIoControl** call.

MODM_OPEN

The MODM_OPEN message is sent to a MIDI output driver to request it to open an instance of a specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MODM_OPEN

dwUser
Pointer to location to receive device instance identifier.

dwParam1
Pointer to a MIDIOPENDESC structure, containing the client's device handle, notification target, and instance ID.

dwParam2

Contains flags. The following flags are defined.

Flag	Definition
CALLBACK_EVENT	Indicates **dwCallback** member of MIDIOPENDESC is an event handle.
CALLBACK_FUNCTION	Indicates **dwCallback** member of MIDIOPENDESC is the address of a callback function.
CALLBACK_TASK	Indicates **dwCallback** member of MIDIOPENDESC is a task handle.
CALLBACK_WINDOW	Indicates **dwCallback** member of MIDIOPENDESC is a window handle.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiOutOpen** return values in the Win32 SDK.

Comments

A client sends the MODM_OPEN message by calling the user-mode driver's **modMessage** entry point, passing the specified parameters.

Typically, user-mode drivers connect to kernel-mode drivers by calling **CreateFile**, specifying the MS-DOS device name of one of the kernel-mode driver's devices.

The driver assigns a device instance identifier and returns it in the location pointed to by *dwUser*. The driver can expect to receive this value as the *dwUser* input argument to all other **modMessage** messages.

The driver determines the number of clients it allows to use a particular device. If a device is opened by the maximum number of clients, it returns MMSYSERR_ALLOCATED for subsequent open requests.

If the open operation succeeds, the driver should send the client a MOM_OPEN message by calling the **DriverCallback** function.

For additional information, see Transferring MIDI Output Data.

MODM_PREPARE

The MODM_PREPARE message requests a MIDI output driver to prepare a system-exclusive data buffer for output.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MODM_PREPARE

dwUser
Device instance identifier.

dwParam1
Pointer to a MIDIHDR structure identifying the buffer. (The MIDIHDR structure is described in the Win32 SDK.)

dwParam2
Size of the MIDIHDR structure.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See

midiOutPrepareHeader return values in the Win32 SDK.

Comments

A client sends the MODM_PREPARE message by calling the user-mode driver's **modMessage** entry point, passing the specified parameters.

Support for this message by user-mode drivers is optional. If the driver supports MODM_PREPARE, it must also support MODM_UNPREPARE.

If the driver returns MMSYSERR_NOTSUPPORTED, *winmm.dll* prepares the buffer for use. For most drivers, this behavior is sufficient. If the driver does perform buffer preparation, it must set MHDR_PREPARED in the **dwFlags** member of MIDIHDR and return MMSYSERR_NOERROR.

For additional information, see Transferring MIDI Output Data.

MODM_RESET

The MODM_RESET message requests a MIDI output driver to stop sending output data and return all output buffers to the client.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MODM_RESET

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiOutReset** return values in the Win32 SDK.

Comments

A client sends the MODM_RESET message by calling the user-mode driver's **modMessage** entry point, passing the specified parameters.

If the driver's output queue contains any output buffers (see MODM_LONGDATA) whose contents have not been sent to the kernel-mode driver, the driver should set the MHDR_DONE flag and clear the MHDR_INQUEUE flag in each buffer's MIDIHDR structure, and then send the client a MOM_DONE callback message for each buffer.

Typically, the user-mode driver stops device output by calling **DeviceIoControl**, sending the kernel-mode driver an IOCTL_MIDI_SET_STATE control code.

If the device is an internal synthesizer, the driver should turn off all notes.

For additional information, see Transferring MIDI Output Data.

MODM_SETVOLUME

The MODM_SETVOLUME message requests a user-mode MIDI output driver to set the volume level for the specified device.

Parameters

uDeviceId

Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MODM_SETVOLUME

dwUser
Device instance identifier.

dwParam1
A DWORD containing the volume setting.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiOutSetVolume** return values in the Win32 SDK.

Comments

A client sends the MODM_SETVOLUME message by calling the user-mode driver's **modMessage** entry point, passing the specified parameters.

Support for this message by user-mode drivers is optional. If the driver supports MODM_SETVOLUME, it must support MODM_GETVOLUME.

The volume value is specified by *dwParam1* as follows:

Channel	**Portion of *dwParam1* Used**
Left channel	Low word
Right channel	High word
Single channel	Low word

A value of zero is silence, and a value of 0xFFFF is full volume.

The kernel-mode driver might not support the full 16 bits of volume control and can truncate the lower bits. However, the original value requested with AUXDM_SETVOLUME should be returned with AUXDM_GETVOLUME.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_MIDI_SET_VOLUME control code.

Only drivers for internal synthesizer devices can support volume level changes. Drivers for MIDI output ports should return a MMSYSERR_NOTSUPPORTED error for this message.

MODM_UNPREPARE

The MODM_UNPREPARE message requests a MIDI output driver to remove the buffer preparation that was performed in response to a MODM_PREPARE message.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MODM_UNPREPARE

dwUser
Device instance identifier.

dwParam1
Specifies a pointer to MIDIHDR identifying the data buffer. (The MIDIHDR structure is described in the Win32 SDK.)

dwParam2
Specifies the size of MIDIHDR in bytes.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIDIERR error codes defined in *mmsystem.h*. See **midiOutUnprepareHeader** return values in the Win32 SDK.

Comments

A client sends the MODM_UNPREPARE message by calling the user-mode driver's **modMessage** entry point, passing the specified parameters.

Support for this message by user-mode drivers is optional. If the driver supports MODM_PREPARE, it must also support MODM_UNPREPARE.

If the driver returns MMSYSERR_NOTSUPPORTED, *winmm.dll* removes the buffer preparation. For most drivers, this behavior is sufficient. If the driver does support MODM_UNPREPARE, it must clear MHDR_PREPARED in the **dwFlags** member of MIDIHDR and return MMSYSERR_NOERROR.

For additional information, see Transferring MIDI Output Data.

MOM_CLOSE

The MOM_CLOSE callback message notifies a client that a user-mode driver has finished processing a MODM_CLOSE message.

Parameters

dwMsg
MOM_CLOSE

dwParam1
NULL

dwParam2
NULL

Comments

A user-mode MIDI output driver sends a MOM_CLOSE message to its client, by means of a callback, when the driver finishes processing a MODM_CLOSE message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the MOM_CLOSE message only if the client has previously specified a notification target with a MODM_OPEN message.

Win32 SDK documentation states that clients receive a MM_MOM_CLOSE message if the notification target is a window handle. MOM_CLOSE and MM_MOM_CLOSE are equivalent.

For more information, see Notifying Clients from Audio Drivers and Transferring MIDI Output Data.

MOM_DONE

The MOM_DONE callback message notifies a client that a user-mode driver has finished processing a MODM_LONGDATA message.

Parameters

dwMsg
MOM_DONE

dwParam1
Address of the MDIHDR structure that was received with the MODM_LONGDATA message. (MIDIHDR is defined in the Win32 SDK.)

dwParam2
NULL

Comments

A user-mode MIDI output driver sends a MOM_DONE message to its client, by means of a callback, when the driver finishes processing a MODM_LONGDATA message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the MOM_DONE message only if the client has previously specified a notification target with a MODM_OPEN message.

Win32 SDK documentation states that clients receive a MM_MOM_DONE message if the notification target is a window handle. MOM_DONE and MM_MOM_DONE are equivalent.

For more information, see Notifying Clients from Audio Drivers and Transferring MIDI Output Data.

MOM_OPEN

The MOM_OPEN callback message notifies a client that a user-mode driver has finished processing a MODM_OPEN message.

Parameters

dwMsg
MOM_OPEN

dwParam1
NULL

dwParam2
NULL

Comments

A user-mode MIDI output driver sends a MOM_OPEN message to its client, by means of a callback, when the driver finishes processing a MODM_OPEN message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the MOM_OPEN message only if the client has specified a notification target with the MODM_OPEN message.

Win32 SDK documentation states that clients receive an MM_MOM_OPEN message if the notification target is a window handle. MOM_OPEN and MM_MOM_OPEN are equivalent.

For more information, see Notifying Clients from Audio Drivers and Transferring MIDI Output Data.

MOM_POSITIONCB

The MOM_POSITIONCB callback message notifies a client that a user-mode driver has encountered a MIDI event containing a MEVT_F_CALLBACK flag.

Parameters

dwMsg
MOM_POSITIONCB

dwParam1
NULL

dwParam2
NULL

Comments

User-mode drivers do not send MOM_POSITIONCB callback messages.

Clients using the MIDI output stream functions send streams of data contained in a series of MIDIEVENT structures (see the Win32 SDK). Code in *winmm.dll* examines each MIDIEVENT

structure and, if the MEVT_F_CALLBACK flag is set, sends a MOM_POSITIONCB message by calling **DriverCallback**, passing the specified parameters.

The *winmm.dll* code sends MOM_POSITIONCB messages only if the client has specified a notification target with the MODM_OPEN message.

Win32 SDK documentation states that clients receive a MM_MOM_POSITIONCB message if the notification target is a window handle. MOM_POSITIONCB and MM_MOM_POSITIONCB are equivalent.

For more information, see MIDI Output Streams, Notifying Clients from Audio Drivers and Transferring MIDI Output Data.

MXDM_CLOSE

The MXDM_CLOSE message requests a user-mode mixer driver to close the specified device instance that was opened with an MXDM_OPEN message.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MXDM_CLOSE

dwUser
Instance identifier of instance to close.

lParam1
Not used.

lParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIXERR error codes defined in *mmsystem.h*. See **mixerClose** return values in the Win32 SDK.

Comments

A client sends the MXDM_CLOSE message by calling the user-mode driver's **mxdMessage** entry point, passing the specified parameters.

Often, closing a driver instance simply involves removing a client-specific data structure.

MXDM_GETCONTROLDETAILS

The MXDM_GETCONTROLDETAILS message requests a user-mode mixer driver to return detailed information about the specified control on the specified audio line.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MXDM_GETCONTROLDETAILS

dwUser
Instance identifier associated with the caller.

lParam1
Pointer to a MIXERCONTROLDETAILS structure, which is described in the Win32 SDK.

lParam2
Contains flag values. For a list of valid flags, see the description of **mixerGetControlDetails** in the Win32 SDK.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIXERR error codes defined in *mmsystem.h*. See **mixerGetControlDetails** return values in the Win32 SDK.

Comments

A client sends the MXDM_GETCONTROLDETAILS message by calling the user-mode driver's **mxdMessage** entry point, passing the specified parameters.

The driver receives an empty MIXERCONTROLDETAILS structure and fills it in.

MXDM_GETDEVCAPS

The MXDM_GETDEVCAPS message requests a user-mode mixer driver to return capabilities information about the specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MXDM_GETDEVCAPS

dwUser
Instance identifier associated with the caller.

lParam1
Pointer to a MIXERCAPS structure, which is described in the Win32 SDK.

lParam2
Size of buffer pointed to by *lparam1*.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIXERR error codes defined in *mmsystem.h*. See **mixerGetDevCaps** return values in the Win32 SDK.

Comments

A client sends the MXDM_GETDEVCAPS message by calling the user-mode driver's **mxdMessage** entry point, passing the specified parameters.

The driver receives an empty MIXERCAPS structure and fills it in.

MXDM_GETLINECONTROLS

The MXDM_GETLINECONTROLS message requests a user-mode mixer driver to return information about specified controls connected to a specified audio line, for the specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MXDM_GETLINECONTROLS

dwUser
Instance identifier associated with the caller.

lParam1
Pointer to a MIXERLINECONTROLS structure, which is described in the Win32 SDK.

lParam2
Contains flag values. For a list of valid flags, see the description of **mixerGetLineControls** in the Win32 SDK.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIXERR error codes defined in *mmsystem.h*. See **mixerGetLineControls** return values in the Win32 SDK.

Comments

A client sends the MXDM_GETLINECONTROLS message by calling the user-mode driver's **mxdMessage** entry point, passing the specified parameters.

The driver receives an empty MIXERLINECONTROLS structure and fills it in.

MXDM_GETLINEINFO

The MXDM_GETLINEINFO message requests a user-mode mixer driver to return information about a specified audio line for the specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MXDM_GETLINEINFO

dwUser
Instance identifier associated with the caller.

lParam1
Pointer to a MIXERLINE structure, which is described in the Win32 SDK.

lParam2
Contains flag values. For a list of valid flags, see the description of **mixerGetLineInfo** in the Win32 SDK.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIXERR error codes defined in *mmsystem.h*. See **mixerGetLineInfo** return values in the Win32 SDK.

Comments

A client sends the MXDM_GETLINEINFO message by calling the user-mode driver's **mxdMessage** entry point, passing the specified parameters.

The driver receives an empty MIXERLINE structure and fills it in.

MXDM_GETNUMDEVS

The MXDM_GETNUMDEVS message requests a user-mode mixer driver to return the number of device instances it supports.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MXDM_GETNUMDEVS

dwUser
Not used.

lParam1
Not used.

lParam2
Not used.

Return Value

Returns the number of devices the driver supports.

Comments

A client sends the MXDM_GETNUMDEVS message by calling the user-mode driver's **mxdMessage** entry point, passing the specified parameters.

MXDM_INIT

A user-mode mixer driver's **mxdMessage** function receives a MXDM_INIT message while the driver is being installed.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MXDM_INIT

dwUser
Not used.

lParam1
Not used.

lParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIXERR error codes defined in *mmsystem.h*.

Comments

A client sends the MXDM_INIT message by calling the user-mode driver's **mxdMessage** entry point, passing the specified parameters.

Support for this message is optional. The message allows mixer drivers to perform additional installation activities after DRV_INSTALL has been received.

MXDM_OPEN

The MXDM_OPEN message requests a user-mode mixer driver to open an instance of the specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MXDM_OPEN

dwUser
Address of location into which driver places instance identifier.

lParam1
Pointer to a MIXEROPENDESC structure.

lParam2
Contains flag values. This is always CALLBACK_FUNCTION. (See **Comments** section below.)

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIXERR error codes defined in *mmsystem.h*. See **mixerOpen**

return values in the Win32 SDK.

Comments

A client sends the MXDM_OPEN message by calling the user-mode driver's **mxdMessage** entry point, passing the specified parameters.

Often, creating a driver instance simply entails creating an instance-specific data structure. The instance identifier can be a handle to this structure.

Even though the description of **mixerOpen** in the Win32 SDK lists numerous flag values, these flags are handled within *winmm.dll*.

According to the description of **mixerOpen** in the Win32 SDK, the only acceptable callback target is a window handle. Code within *winmm.dll* pre-empts this callback target by placing the address of a local callback function in the MIXEROPENDESC structure, and setting the CALLBACK_FUNCTION flag in *lParam2*. The driver calls **DriverCallback** at the appropriate times, specifying the callback function. The callback function, within *winmm.dll*, then sends a callback message to the client-specified window handle.

MXDM_SETCONTROLDETAILS

The MXDM_SETCONTROLDETAILS message requests a user-mode mixer driver to set information about specified controls connected to a specified audio line for a specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
MXDM_SETCONTROLDETAILS

dwUser
Instance identifier associated with caller.

lParam1
Pointer to a MIXERCONTROLDETAILS structure, which is described in the Win32 SDK.

lParam2
Contains flag values. For a list of valid flags, see the description of **mixerGetLineControls** in the Win32 SDK.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or MIXERR error codes defined in *mmsystem.h*. See **mixerSetControlDetails** return values in the Win32 SDK.

Comments

A client sends the MXDM_SETCONTROLDETAILS message by calling the user-mode driver's **mxdMessage** entry point, passing the specified parameters.

The driver receives a MIXERCONTROLDETAILS structure containing information to be set.

WIDM_ADDBUFFER

The WIDM_ADDBUFFER requests a user-mode waveform input driver to add an empty input buffer to its input buffer queue.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WIDM_ADDBUFFER

dwUser
Device instance identifier.

dwParam1
Pointer to a WAVEHDR structure identifying the buffer. (The WAVEHDR structure is described in the Win32 SDK.)

dwParam2
Size of the WAVEHDR structure in bytes.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveInAddBuffer** return values in the Win32 SDK.

Comments

A client sends the message by calling the user-mode driver's **widMessage** entry point, passing the specified parameters.

If the WHDR_PREPARED flag is not set in the **dwFlags** member of the WAVEHDR structure, the driver should return WAVERR_UNPREPARED. If the flag is set, the driver should:

- Clear the WHDR_DONE flag.
- Set the WHDR_INQUEUE flag.
- Place the empty buffer in its input queue.
- Return control to the client with a return value of MMSYSERR_NOERROR.

The user-mode driver starts recording when it receives a WIDM_START message.

For additional information, see Transferring Waveform Input Data.

WIDM_CLOSE

The WIDM_CLOSE message requests a waveform input driver to close a specified device instance that was previously opened with a WIDM_OPEN message.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WIDM_CLOSE

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveInClose** return values in the Win32 SDK.

Comments

A client sends the WIDM_CLOSE message by calling the user-mode driver's **widMessage** entry point, passing the specified parameters.

If the driver has not filled and returned all of the buffers received with WIDM_ADDBUFFER messages, it should not close the instance and should instead return WAVERR_STILLPLAYING.

After the driver closes the device instance it should send a WIM_CLOSE callback message to the client.

For additional information, see Transferring Waveform Input Data.

WIDM_GETDEVCAPS

The WIDM_GETDEVCAPS message requests a waveform input driver to return the capabilities of a specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WIDM_GETDEVCAPS

dwUser
Device instance identifier.

dwParam1
Pointer to a WAVEINCAPS structure. (The WAVEINCAPS structure is described in the Win32 SDK.)

dwParam2
Size of the WAVEINCAPS structure in bytes.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveInGetDevCaps** return values in the Win32 SDK.

Comments

A client sends the WIDM_GETDEVCAPS message by calling the user-mode driver's **widMessage** entry point, passing the specified parameters.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_WAVE_GET_CAPABILITIES control code.

The user-mode driver fills the WAVEINCAPS structure.

WIDM_GETNUMDEVS

The WIDM_GETNUMDEVS message requests a waveform input driver to return the number of device instances that it supports.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WIDM_GETNUMDEVS

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver returns the number of waveform input device instances it supports.

Comments

A client sends the WIDM_GETNUMDEVS message by calling the user-mode driver's **widMessage** entry point, passing the specified parameters.

The driver should return the number of logical waveform input devices that can be supported. Typically, for each physical device, a kernel-mode driver can support one or more logical devices of various types. For example, for each Creative Labs Sound Blaster™ card, there are MIDI, waveform, mixer, and auxiliary audio devices. Kernel-mode drivers store logical device names and types in the registry under the path **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***Parameters\Device***Number***De**
To correctly return the number of logical devices, the user-mode driver should examine the **\Devices** subkey for each of the driver's **\Device***Number* keys, looking for logical devices of the desired type. (Code in *drvlib.lib* provides this capability.)

WIDM_GETPOS

The WIDM_GETPOS message requests a waveform input driver to return the current input position within a waveform. The input position is relative to the first recorded sample of the waveform.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WIDM_GETPOS

dwUser
Device instance identifier.

dwParam1
Pointer to an MMTIME structure. (The MMTIME structure is defined in the Win32 SDK.)

dwParam2
Size of the MMTIME in bytes.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveInGetPos** return values in the Win32 SDK.

Comments

A client sends the WIDM_GETPOS message by calling the user-mode driver's **widMessage** entry point, passing the specified parameters.

The **wType** member of the MMTIME structure indicates the time format requested by the client. If the driver cannot support the requested format, it should return the position in a format that it does support, and change the **wType** member accordingly.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_WAVE_GET_POSITION control code.

WIDM_LOWPRIORITY

The WIDM_LOWPRIORITY message requests a waveform input driver to run at low priority.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WIDM_LOWPRIORITY

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*.

Comments

A client sends the WIDM_LOWPRIORITY message by calling the user-mode driver's **widMessage** entry point, passing the specified parameters.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_WAVE_SET_LOW_PRIORITY control code. Kernel-mode drivers using *soundlib.lib* allow only one client to be running at low priority.

Support for this message by user-mode drivers is optional.

WIDM_OPEN

The WIDM_OPEN message is sent to a waveform input driver to request it to open an instance of a specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WIDM_OPEN

dwUser
Pointer to location to receive device instance identifier.

dwParam1
Pointer to a WAVEOPENDESC structure, containing the client's device handle, notification target, and instance ID.

dwParam2
Contains flags. The following flags are defined:

Flag	Definition
CALLBACK_EVENT	Indicates the **dwCallback** member of WAVEOPENDESC is an event handle.
CALLBACK_FUNCTION	Indicates the **dwCallback** member of WAVEOPENDESC is the address of a callback function.
CALLBACK_TASK	Indicates the **dwCallback** member of WAVEOPENDESC is a task handle.
CALLBACK_WINDOW	Indicates the **dwCallback** member of WAVEOPENDESC is a window handle.
WAVE_FORMAT_DIRECT	Data compression/decompression operations should take place in hardware. See **Comments** section below.
WAVE_FORMAT_QUERY	The driver should indicate whether or not it supports the specified format. See **Comments** section below.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveInOpen** return values in the Win32 SDK.

Comments

A client sends the WIDM_OPEN message by calling the user-mode driver's **widMessage** entry point, passing the specified parameters.

Typically, user-mode drivers connect to kernel-mode drivers by calling **CreateFile**, specifying the MS-DOS device name of one of the kernel-mode driver's devices.

The driver assigns a device instance identifier and returns it in the location pointed to by *dwUser*. The driver can expect to receive this value as the *dwUser* input argument to all other **widMessage** messages.

The driver determines the number of clients it allows to use a particular device. If a device is opened by the maximum number of clients, it returns MMSYSERR_ALLOCATED for subsequent open requests.

The WAVE_FORMAT_DIRECT flag is meant for use with a wave mapper. If the flag is set in *dwParam2*, the driver should not call the Audio Compression Manager to handle compression/decompression operations; the caller wants the hardware to perform these operations directly. If the hardware is not capable of performing compression/decompression operations, the driver should return MMSYSERR_NOTSUPPORTED when WAVE_FORMAT_DIRECT is set.

If the WAVE_FORMAT_QUERY flag is set in *dwParam2*, the driver should not open the device, but should instead determine whether it supports the format specified by the WAVEOPENDESC structure's **lpFormat** member. If the driver supports the requested format, it should return MMSYSERR_NOERROR. Otherwise it should return WAVERR_BADFORMAT.

If the open operation succeeds, the driver should send the client a WIM_OPEN message by calling the **DriverCallback** function.

For additional information, see Transferring Waveform Input Data.

WIDM_PREPARE

The WIDM_PREPARE message requests a waveform input driver to prepare a system-exclusive data buffer for input.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WIDM_PREPARE

dwUser
Device instance identifier.

dwParam1
Pointer to a WAVEHDR structure identifying the buffer. (The WAVEHDR structure is described in the Win32 SDK.)

dwParam2
Size of the WAVEHDR structure.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveInPrepareHeader** return values in the Win32 SDK.

Comments

A client sends the WIDM_PREPARE message by calling the user-mode driver's **widMessage** entry point, passing the specified parameters.

Support for this message by user-mode drivers is optional. If the driver supports WIDM_PREPARE, it must also support WIDM_UNPREPARE.

If the driver returns MMSYSERR_NOTSUPPORTED, *winmm.dll* prepares the buffer for use. For most drivers, this behavior is sufficient. If the driver does perform buffer preparation, it must set WHDR_PREPARED in the **dwFlags** member of WAVEHDR and return MMSYSERR_NOERROR.

For additional information, see Transferring Waveform Input Data.

WIDM_RESET

The WIDM_RESET message requests a waveform input driver to stop recording and return all buffers in the input queue to the client.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WIDM_RESET

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveInReset** return values in the Win32 SDK.

Comments

A client sends the WIDM_RESET message by calling the user-mode driver's **widMessage** entry point, passing the specified parameters.

Typically, the user-mode driver stops recording by calling **DeviceIoControl**, sending the kernel-mode driver an IOCTL_WAVE_SET_STATE control code.

For each buffer remaining in the driver's input queue (see WIDM_ADDBUFFER), the driver should set WHDR_DONE and clear WHDR_INQUEUE in the **dwFlags** member of the buffer's WAVEHDR structure, and also set the structure's **dwBytesRecorded** member. Finally, a WIM_DATA callback message should be sent for each buffer.

For additional information, see Transferring Waveform Input Data.

WIDM_START

The WIDM_START message requests a waveform input driver to begin recording.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WIDM_START

dwUser
Device instance identifier.

dwParam1
Not used.
dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveInStart** return values in the Win32 SDK.

Comments

A client sends the WIDM_START message by calling the user-mode driver's **widMessage** entry point, passing the specified parameters.

If the message is received after input has been started, the driver should return MMSYSERR_NOERROR.

User-mode waveform input drivers should handle input asynchronously, by creating a separate thread to handle communication with the kernel-mode driver. Typically, the new thread starts recording by calling **DeviceIoControl** to send the kernel-mode driver an IOCTL_WAVE_SET_STATE control code, and by calling **ReadFileEx** to request the kernel-mode driver to fill client-supplied buffers (see WIDM_ADDBUFFER). When the kernel-mode driver returns a filled buffer, the user-mode driver should:

- Set the **dwBytesRecorded** member in the buffer's WAVEHDR structure.
- Set the buffer's WHDR_DONE flag.
- Clear the buffer's WHDR_INQUEUE flag.
- Send a WIM_DATA callback message to the client.

To avoid unnecessarily locking too much memory, do not send the kernel-mode driver too many buffers at once, or buffers that are excessively large.

Recording should continue until the client sends WIDM_STOP or WIDM_RESET.

For additional information, see Transferring Waveform Input Data.

WIDM_STOP

The WIDM_STOP message requests a waveform input driver to stop recording.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.
uMsg
WIDM_STOP
dwUser
Device instance identifier.
dwParam1
Not used.
dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveInStop** return values in the Win32 SDK.

Comments

A client sends the WIDM_STOP message by calling the user-mode driver's **widMessage** entry point, passing the specified parameters.

If a buffer in the input queue (see WIDM_ADDBUFFER) has been partially filled, the driver should treat it as a full buffer and return it to the client (see WIDM_START). Empty buffers should remain in the queue.

If this message is received and recording is already stopped, the driver should return MMSYSERR_NOERROR.

Typically, the user-mode driver stops recording by calling **DeviceIoControl**, sending the kernel-mode driver an IOCTL_WAVE_SET_STATE control code.

For additional information, see Transferring Waveform Input Data.

WIDM_UNPREPARE

The WIDM_UNPREPARE message requests a waveform input driver to remove the buffer preparation that was performed in response to a WIDM_PREPARE message.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WIDM_UNPREPARE

dwUser
Device instance identifier.

dwParam1
Pointer to a WAVEHDR structure identifying the data buffer. (The WAVEHDR structure is described in the Win32 SDK.)

dwParam2
Size, in bytes, of the WAVEHDR structure.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveInUnprepareHeader** return values in the Win32 SDK.

Comments

A client sends the WIDM_UNPREPARE message by calling the user-mode driver's **widMessage** entry point, passing the specified parameters.

Support for this message by user-mode drivers is optional. If the driver supports WIDM_PREPARE, it must also support WIDM_UNPREPARE.

If the driver returns MMSYSERR_NOTSUPPORTED, *winmm.dll* removes the buffer preparation. For most drivers, this behavior is sufficient. If the driver does support WIDM_UNPREPARE, it must clear WHDR_PREPARED in the **dwFlags** member of WAVEHDR and return MMSYSERR_NOERROR.

For additional information, see Transferring Waveform Input Data.

WIM_CLOSE

The WIM_CLOSE callback message notifies a client that a user-mode driver has finished processing a WIDM_CLOSE message.

Parameters

dwMsg
WIM_CLOSE

dwParam1
NULL
dwParam2
NULL

Comments

A user-mode waveform input driver sends a WIM_CLOSE message to its client, by means of a callback, when the driver finishes processing a WIDM_CLOSE message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the WIM_CLOSE message only if the client has previously specified a notification target with a WIDM_OPEN message.

Win32 SDK documentation states that clients receive an MM_WIM_CLOSE message if the notification target is a window handle. WIM_CLOSE and MM_WIM_CLOSE are equivalent.

For additional information, see Notifying Clients from Audio Drivers and Transferring Waveform Input Data.

WIM_DATA

The WIM_DATA callback message notifies a client that a user-mode driver has filled a buffer with waveform data.

Parameters

dwMsg
WIM_DATA
dwParam1
Address of a WAVEHDR structure identifying a buffer containing the message. (WAVEHDR is defined in the Win32 SDK.)
dwParam2
NULL

Comments

A user-mode waveform input driver sends a WIM_DATA message to its client, by means of a callback, when the driver has filled a buffer with waveform data. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The WAVEHDR structure is one that was received along with a WIDM_ADDBUFFER message.

The driver sends the WIM_DATA message only if the client has previously specified a notification target with a WIDM_OPEN message.

Win32 SDK documentation states that clients receive an MM_WIM_DATA message if the notification target is a window handle. WIM_DATA and MM_WIM_DATA are equivalent.

For additional information, see Notifying Clients from Audio Drivers and Transferring Waveform Input Data.

WIM_OPEN

The WIM_OPEN callback message notifies a client that a user-mode driver has finished processing a WIDM_OPEN message.

Parameters

dwMsg
WIM_OPEN
dwParam1
NULL
dwParam2

NULL

Comments

A user-mode waveform output driver sends a WIM_OPEN message to its client, by means of a callback, when the driver finishes processing a WIDM_OPEN message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the WIM_OPEN message only if the client has specified a notification target with the WIDM_OPEN message.

Win32 SDK documentation states that clients receive an MM_WIM_OPEN message if the notification target is a window handle. WIM_OPEN and MM_WIM_OPEN are equivalent.

For additional information, see Notifying Clients from Audio Drivers and Transferring Waveform Input Data.

WODM_BREAKLOOP

The WODM_BREAKLOOP message requests a waveform output driver to break an output loop that was created with a WODM_WRITE message.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_BREAKLOOP

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutBreakLoop** return values in the Win32 SDK.

Comments

A client sends the WODM_BREAKLOOP message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

The driver should stop output of the loop buffers at the end of the next loop iteration.

If the driver receives this message and a loop is not in progress, it should return MMSYSERR_NOERROR.

WODM_CLOSE

The WODM_CLOSE message requests a waveform output driver to close a specified device instance that was previously opened with a WODM_OPEN message.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_CLOSE

dwUser

Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutClose** return values in the Win32 SDK.

Comments

A client sends the WODM_CLOSE message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

If the client has passed data buffers to the user-mode driver by means of WODM_WRITE messages, and if the user-mode driver hasn't finished sending the data to the kernel-mode driver, the user-mode driver should return WAVERR_STILLPLAYING in response to WODM_CLOSE.

After the driver closes the device instance it should send a WOM_CLOSE callback message to the client.

For additional information, see Transferring Waveform Output Data.

WODM_GETDEVCAPS

The WODM_GETDEVCAPS message requests a waveform output driver to return the capabilities of a specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_GETDEVCAPS

dwUser
Device instance identifier.

dwParam1
Pointer to a WAVEOUTCAPS structure, which is described in the Win32 SDK.

dwParam2
Size of the WAVEOUTCAPS structure in bytes.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutGetDevCaps** return values in the Win32 SDK.

Comments

A client sends the WODM_GETDEVCAPS message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_WAVE_GET_CAPABILITIES control code.

The user-mode driver fills the WAVEOUTCAPS structure.

WODM_GETNUMDEVS

The WODM_GETNUMDEVS message requests a waveform output driver to return the number of

device instances that it supports.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_GETNUMDEVS

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver returns the number of waveform output device instances it supports.

Comments

A client sends the WODM_GETNUMDEVS message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

The driver should return the number of logical waveform output devices that can be supported. Typically, for each physical device, a kernel-mode driver can support one or more logical devices of various types. For example, for each Creative Labs Sound Blaster card, there are MIDI, waveform, mixer, and auxiliary audio devices. Kernel-mode drivers store logical device names and types in the registry under the path **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***\Parameters\Device***Number***\De**
To correctly return the number of logical devices, the user-mode driver should examine the **\Devices** subkey for each of the driver's **\Device***Number* keys, searching for logical devices of the desired type. (Code in *drvlib.lib* provides this capability.)

WODM_GETPITCH

The WODM_GETPITCH message requests a waveform output driver to return the specified device's current pitch multiplier value.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_GETPITCH

dwUser
Device instance identifier.

dwParam1
Pointer to a DWORD location used to return the current pitch multiplier value. This is specified as a fixed-point value, where the high-order word of the DWORD contains the signed integer part of the number, and the low-order word contains the fractional part. The fraction consists of a WORD value, for which 0x8000 represents one half, and 0x4000 represents one quarter. For example, the value 0x00010000 specifies a multiplier of 1.0 (no pitch change), and a value of 0x000F8000 specifies a multiplier of 15.5.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See

waveOutGetPitch return values in the Win32 SDK.

Comments

A client sends the WODM_GETPITCH message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

Support for the WODM_GETPITCH message by user-mode drivers is optional. If a driver supports the WODM_SETPITCH message, it must also support WODM_GETPITCH.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_WAVE_GET_PITCH control code.

WODM_GETPLAYBACKRATE

The WODM_GETPLAYBACKRATE message requests a waveform output driver to return the current playback rate multiplier value for the specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, etc.) for the target device.

uMsg
WODM_GETPLAYBACKRATE

dwUser
Device instance identifier.

dwParam1
Pointer to a DWORD location used to return the current playback rate multiplier value. This is specified as a fixed-point value, where the high-order word of the DWORD contains the signed integer part of the number, and the low-order word contains the fractional part. The fraction consists of a WORD value, for which 0x8000 represents one half, and 0x4000 represents one quarter. For example, the value 0x00010000 specifies a multiplier of 1.0 (no rate change), and a value of 0x000F8000 specifies a multiplier of 15.5.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutGetPlaybackRate** return values in the Win32 SDK.

Comments

A client sends the WODM_GETPLAYBACKRATE message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

Support for the WODM_GETPLAYBACKRATE message by user-mode drivers is optional. If a driver supports the WODM_SETPLAYBACKRATE message, it must also support WODM_GETPLAYBACKRATE.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_WAVE_GET_PLAYBACK_RATE control code.

WODM_GETPOS

The WODM_GETPOS message requests a waveform output driver to return the current input position within a waveform. The input position is relative to the beginning of the waveform.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_GETPOS

dwUser
Device instance identifier.

dwParam1
Pointer to an MMTIME structure, which is described in the Win32 SDK.

dwParam2
Size of the MMTIME structure in bytes.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutGetPos** return values in the Win32 SDK.

Comments

A client sends the WODM_GETPOS message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

The **wType** member of the MMTIME structure indicates the time format requested by the client. If the driver cannot support the requested format, it should return the position in a format that it does support, and change the **wType** member accordingly.

The position should be reset to zero when the driver receives a WODM_OPEN or WODM_RESET message.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_WAVE_GET_POSITION control code.

WODM_GETVOLUME

The WODM_GETVOLUME message requests a waveform output driver to return the current volume level setting for the specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_GETVOLUME

dwUser
Device instance identifier.

dwParam1
Pointer to a DWORD location to receive the volume setting.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutGetVolume** return values in the Win32 SDK.

Comments

A client sends the WODM_GETVOLUME message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

Support for this message by user-mode drivers is optional. If the driver supports WODM_SETVOLUME, it must support WODM_GETVOLUME.

The volume value is returned in the DWORD pointed to by *dwParam1* as follows.

Channel	Portion of *dwParam1* Used
Left channel	Low word
Right channel	High word
Single channel	Low word

A value of zero is silence, and a value of 0xFFFF is full volume.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_WAVE_GET_VOLUME control code.

WODM_OPEN

The WODM_OPEN message requests a waveform output driver to open an instance of a specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_OPEN

dwUser
Pointer to location to receive device instance identifier.

dwParam1
Pointer to a WAVEOPENDESC structure, containing the client's device handle, notification target, and instance ID.

dwParam2
Contains flags. The following flags are defined:

Flag	Definition
CALLBACK_EVENT	Indicates the **dwCallback** member of WAVEOPENDESC is an event handle.
CALLBACK_FUNCTION	Indicates the **dwCallback** member of WAVEOPENDESC is the address of a callback function.
CALLBACK_TASK	Indicates the **dwCallback** member of WAVEOPENDESC is a task handle.
CALLBACK_WINDOW	Indicates the **dwCallback** member of WAVEOPENDESC is a window handle.
WAVE_FORMAT_DIRECT	Data compression/decompression operations should take place in hardware. See **Comments** section below.
WAVE_FORMAT_QUERY	The driver should indicate whether or not it supports the specified format. See comments.
WAVE_ALLOWSYNC	Indicates the driver should allow opening of a synchronous device. *Ignored by* mmdrv.dll *and* drvlib.lib.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutOpen** return values in the Win32 SDK.

Comments

A client sends the WODM_OPEN message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

Typically, user-mode drivers connect to kernel-mode drivers by calling **CreateFile**, specifying the

MS-DOS device name of one of the kernel-mode driver's devices.

The driver assigns a device instance identifier and returns it in the location pointed to by *dwUser*. The driver can expect to receive this value as the *dwUser* input argument to all other **wodMessage** messages.

The driver determines the number of clients it allows to use a particular device. If a device is opened by the maximum number of clients, it returns MMSYSERR_ALLOCATED for subsequent open requests.

The WAVE_FORMAT_DIRECT flag is meant for use with a wave mapper. If the flag is set in *dwParam2*, the driver should not call the Audio Compression Manager to handle compression/decompression operations; the caller wants the hardware to perform these operations directly. If the hardware is not capable of performing compression/decompression operations, the driver should return MMSYSERR_NOTSUPPORTED when WAVE_FORMAT_DIRECT is set.

If the WAVE_FORMAT_QUERY flag is set in *dwParam2*, the driver should not open the device, but should instead determine whether it supports the format specified by the WAVEOPENDESC structure's **lpFormat** member. If the driver supports the requested format, it should return MMSYSERR_NOERROR. Otherwise it should return WAVERR_BADFORMAT.

If the open operation succeeds, the driver should send the client a WOM_OPEN message by calling the **DriverCallback** function.

For additional information, see Transferring Waveform Output Data.

WODM_PAUSE

The WODM_PAUSE message requests a waveform output driver to pause playback of a waveform.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_PAUSE

dwUser
Pointer to location to receive device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutPause** return values in the Win32 SDK.

Comments

A client sends the WODM_PAUSE message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

The driver should stop playing the waveform and should save the current position. Playback should continue from this position when a WODM_RESTART message is received. Output buffers received with the WODM_WRITE message while playback is paused should be placed in the output queue.

If the driver receives this message while output is already paused, it should return MMSYSERR_NOERROR.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_WAVE_SET_STATE control code.

WODM_PREPARE

The WODM_PREPARE message requests a waveform output driver to prepare a system-exclusive data buffer for output.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_PREPARE

dwUser
Device instance identifier.

dwParam1
Pointer to a WAVEHDR structure identifying the buffer. (The WAVEHDR structure is described in the Win32 SDK.)

dwParam2
Size of the WAVEHDR structure.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutPrepareHeader** return values in the Win32 SDK.

Comments

A client sends the WODM_PREPARE message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

Support for this message by user-mode drivers is optional. If the driver supports WODM_PREPARE, it must also support WODM_UNPREPARE.

If the driver returns MMSYSERR_NOTSUPPORTED, *winmm.dll* prepares the buffer for use. For most drivers, this behavior is sufficient. If the driver does perform buffer preparation, it must set WHDR_PREPARED in the **dwFlags** member of WAVEHDR and return MMSYSERR_NOERROR.

For additional information, see Transferring Waveform Output Data.

WODM_RESET

The WODM_RESET message requests a waveform output driver to stop sending output data and return all output buffers to the client.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_RESET

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutReset** return values in the Win32 SDK.

Comments

A client sends the WODM_RESET message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

If the driver's output queue contains any output buffers (see WODM_WRITE) whose contents have not been sent to the kernel-mode driver, the driver should set the WHDR_DONE flag and clear the WDR_INQUEUE flag in each buffer's WAVEHDR structure The driver should then send the client a WOM_DONE callback message for each buffer.

The driver should reset its position count to zero. If playback is paused, the driver should also take itself out of the paused state.

Typically, the user-mode driver stops device output by calling **DeviceIoControl**, sending the kernel-mode driver an IOCTL_WAVE_SET_STATE control code.

For additional information, see Transferring Waveform Output Data.

WODM_RESTART

The WODM_RESTART message requests a waveform output driver to continue playback of a waveform after playback has been paused with the WODM_PAUSE message.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_RESET

dwUser
Device instance identifier.

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutRestart** return values in the Win32 SDK.

Comments

A client sends the WODM_RESET message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

Data output should resume from the position that was saved when the WODM_PAUSE message was received.

If the driver receives a WODM_RESTART message and output is not in a paused state, it should to nothing except return MMSYSERR_NOERROR.

Typically, the user-mode driver resumes device output by calling **DeviceIoControl**, sending the kernel-mode driver an IOCTL_WAVE_SET_STATE control code.

WODM_SETPITCH

The WODM_SETPITCH message requests a waveform output driver to set the specified device's pitch multiplier value.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_SETPITCH

dwUser
Device instance identifier.

dwParam1
A DWORD containing the pitch multiplier value. This is specified as a fixed-point value, where the high-order word of the DWORD contains the signed integer part of the number, and the low-order word contains the fractional part. The fraction consists of a WORD value, for which 0x8000 represents one half, and 0x4000 represents one quarter. For example, the value 0x00010000 specifies a multiplier of 1.0 (no pitch change), and a value of 0x000F8000 specifies a multiplier of 15.5.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutSetPitch** return values in the Win32 SDK.

Comments

A client sends the WODM_SETPITCH message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

Support for the WODM_SETPITCH message by user-mode drivers is optional. If a driver supports the WODM_SETPITCH message, it must also support WODM_GETPITCH. Additionally, in response to a WODM_GETDEVCAPS message, it must set WAVECAPS_PITCH in the **dwSupport** member of the WAVEOUTCAPS structure.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_WAVE_SET_PITCH control code.

Note: The kernel-mode driver library, *soundlib.lib*, does not support pitch changes.

WODM_SETPLAYBACKRATE

The WODM_SETPLAYBACKRATE message requests a waveform output driver to set the playback rate multiplier value for the specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_SETPLAYBACKRATE

dwUser
Device instance identifier.

dwParam1
A DWORD containing the playback rate multiplier value. This is specified as a fixed-point value, where the high-order word of the DWORD contains the signed integer part of the number, and the low-order word contains the fractional part. The fraction consists of a WORD value, for which 0x8000 represents one half, and 0x4000 represents one quarter. For example, the value 0x00010000 specifies a multiplier of 1.0 (no rate change), and a value of 0x000F8000 specifies a multiplier of 15.5.

dwParam2

Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutSetPlaybackRate** return values in the Win32 SDK.

Comments

A client sends the WODM_SETPLAYBACKRATE message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

Support for the WODM_SETPLAYBACKRATE message by user-mode drivers is optional. If a driver supports the WODM_SETPLAYBACKRATE message, it must also support WODM_GETPLAYBACKRATE. Additionally, in response to a WODM_GETDEVCAPS message, it must set WAVECAPS_PLAYBACKRATE in the **dwSupport** member of the WAVEOUTCAPS structure.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_WAVE_SET_PLAYBACK_RATE control code. Kernel-mode drivers can implement playback rate changes by skipping or repeating samples. For example, if the playback rate is 2.0, the driver would play every second sample at the original playback rate.

Note: The kernel-mode driver library, *soundlib.lib*, does not support playback rate changes.

WODM_SETVOLUME

The WODM_SETVOLUME message requests a waveform output driver to set the volume level for the specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_SETVOLUME

dwUser
Device instance identifier.

dwParam1
Pointer to a DWORD location to receive the volume setting.

dwParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutSetVolume** return values in the Win32 SDK.

Comments

A client sends the WODM_SETVOLUME message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

Support for this message by user-mode drivers is optional. If the driver supports WODM_SETVOLUME, it must support WODM_GETVOLUME.

The volume value is returned in the DWORD pointed to by *dwParam1* as follows.

Channel	**Portion of *dwParam1* Used**
Left channel	Low word
Right channel	High word

Single channel	Low word

A value of zero is silence, and a value of 0xFFFF is full volume.

Typically, the user-mode driver calls **DeviceIoControl** to send the kernel-mode driver an IOCTL_WAVE_SET_VOLUME control code.

WODM_UNPREPARE

The WODM_UNPREPARE message requests a waveform output driver to remove the buffer preparation that was performed in response to a WODM_PREPARE message.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_UNPREPARE

dwUser
Device instance identifier.

dwParam1
Pointer to a WAVEHDR structure identifying the data buffer. (The WAVEHDR structure is described in the Win32 SDK.)

dwParam2
Size of the WAVEHDR structure in bytes.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutUnprepareHeader** return values in the Win32 SDK.

Comments

A client sends the WODM_UNPREPARE message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

Support for this message by user-mode drivers is optional. If the driver supports WODM_PREPARE, it must also support WODM_UNPREPARE.

If the driver returns MMSYSERR_NOTSUPPORTED, *winmm.dll* removes the buffer preparation. For most drivers, this behavior is sufficient. If the driver does support WODM_UNPREPARE, it must clear WHDR_PREPARED in the **dwFlags** member of WAVEHDR and return MMSYSERR_NOERROR.

For additional information, see Transferring Waveform Output Data.

WODM_WRITE

The WODM_WRITE message requests a waveform output driver to write a waveform data block to the specified device.

Parameters

uDeviceId
Device identifier (0, 1, 2, and so on) for the target device.

uMsg
WODM_WRITE

dwUser
Device instance identifier.

dwParam1
Pointer to a WAVEHDR structure identifying the data buffer. (The WAVEHDR structure is

described in the Win32 SDK.)

dwParam2

Size of the WAVEHDR structure in bytes.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR or WAVERR error codes defined in *mmsystem.h*. See **waveOutWrite** return values in the Win32 SDK.

Comments

A client sends the WODM_WRITE message by calling the user-mode driver's **wodMessage** entry point, passing the specified parameters.

If the WHDR_PREPARED flag in the **dwFlags** member of WAVEHDR is not set, the driver should return WAVERR_UNPREPARED.

Unless the device has been paused with a WODM_PAUSE message, the driver should begin playback the first time it receives a WODM_WRITE message.

User-mode waveform output drivers should handle output asynchronously, by creating a separate thread to handle communication with the kernel-mode driver. Typically, the original thread queues the output buffer, sets its WHDR_INQUEUE flag and clears its WHDR_DONE flag in the WAVEHDR structure, and returns control to the client.

Meanwhile, the new thread starts the output operation by calling **DeviceIoControl** to send the kernel-mode driver an IOCTL_WAVE_SET_STATE control code, and by calling **WriteFileEx** to send the kernel-mode driver the client-supplied data. When the kernel-mode driver finishes using a buffer, this thread should set the buffer's WHDR_DONE flag. clear the buffer's WHDR_INQUEUE flag, and send a WOM_DONE callback message to the client.

To avoid unnecessarily locking too much memory, do not send the kernel-mode driver too many buffers at once, or buffers that are excessively large.

The driver should continue sending buffers to the kernel-mode driver until the client sends WODM_PAUSE or WIDM_RESET.

The user-mode driver is usually responsible for implementing waveform looping. The driver should check each buffer's WAVEHDR structure for WHDR_BEGINLOOP and WHDR_ENDLOOP flags, along with an iteration count in the structure's **dwLoops** member.

For additional information, see Transferring Waveform Output Data.

WOM_CLOSE

The WOM_CLOSE callback message notifies a client that a user-mode driver has finished processing a WODM_CLOSE message.

Parameters

dwMsg

WOM_CLOSE

dwParam1

NULL

dwParam2

NULL

Comments

A user-mode waveform input driver sends a WOM_CLOSE message to its client, by means of a callback, when the driver finishes processing a WODM_CLOSE message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the WOM_CLOSE message only if the client has previously specified a

notification target with a WODM_OPEN message.

Win32 SDK documentation states that clients receive an MM_WOM_CLOSE message if the notification target is a window handle. WOM_CLOSE and MM_WOM_CLOSE are equivalent.

For additional information, see Notifying Clients from Audio Drivers and Transferring Waveform Output Data.

WOM_DONE

The WOM_DONE callback message notifies a client that a user-mode driver has finished processing a WODM_WRITE message.

Parameters

dwMsg
WOM_DONE

dwParam1
Address of the WAVEHDR structure that was received with the WODM_WRITE message. (WAVEIHDR is defined in the Win32 SDK.)

dwParam2
NULL

Comments

A user-mode waveform output driver sends a WOM_DONE message to its client, by means of a callback, when the driver finishes processing a WODM_WRITE message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the WOM_DONE message only if the client has previously specified a notification target with a WODM_OPEN message.

Win32 SDK documentation states that clients receive an MM_WOM_DONE message if the notification target is a window handle. WOM_DONE and MM_WOM_DONE are equivalent.

For additional information, see Notifying Clients from Audio Drivers and Transferring Waveform Output Data.

WOM_OPEN

The WOM_OPEN callback message notifies a client that a user-mode driver has finished processing a WODM_OPEN message.

Parameters

dwMsg
WOM_OPEN

dwParam1
NULL

dwParam2
NULL

Comments

A user-mode waveform output driver sends a WOM_OPEN message to its client, by means of a callback, when the driver finishes processing a WODM_OPEN message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the WOM_OPEN message only if the client has specified a notification target with the WODM_OPEN message.

Win32 SDK documentation states that clients receive an MM_WOM_OPEN message if the notification target is a window handle. WOM_OPEN and MM_WOM_OPEN are equivalent.

For additional information, see Notifying Clients from Audio Drivers and Transferring Waveform Output Data.

Structures, User-Mode Audio Drivers

This section describes the structures used by user-mode audio drivers. These structures are defined in *mmddk.h*.

MIDIOPENDESC

```
typedef struct midiopendesc_tag {
    HMIDI hMidi;
    DWORD dwCallback;
    DWORD dwInstance;
    DWORD cIds;
    MIDIOPENSTRMID rgIds[1];
} MIDIOPENDESC;
```

The MIDIOPENDESC structure contains information needed by user-mode MIDI input and MIDI output drivers for sending callback messages to clients. The structure is created by *winmm.dll* and passed to the user-mode driver along with a MODM_OPEN or MIDM_OPEN message.

Members

hMidi
Specifies the client's handle to the device, as assigned by *winmm.dll*. User-mode drivers specify this handle as the *hDriver* parameter to **DriverCallback**, when sending a callback message.

dwCallback
Specifies either the address of a callback function, a window handle, an event handle, or a task handle, depending on the flag specified in the *dwParam2* parameter of the MODM_OPEN or MIDM_OPEN message.

dwInstance
Contains the *dwCallbackInstance* argument that the client specified when calling the **midiInOpen** or **midiOutOpen** function. This value is returned to the client as the *dwInstance* parameter to **DriverCallback**.

cIds
Number of **rgIds** array elements.

rgIds
Array of stream identifiers, containing one element for each open stream.

MIXEROPENDESC

```
typedef struct tMIXEROPENDESC
{
    HMIXER          hmx;
    LPVOID          pReserved0;
    DWORD           dwCallback;
    DWORD           dwInstance;
} MIXEROPENDESC;
```

The MIXEROPENDESC structure contains information needed by user-mode mixer drivers. The structure is created by *winmm.dll* and passed to the driver with an MXDM_OPEN message.

Members

hmx
Specifies the client's handle to the device, as assigned by *winmm.dll*. User-mode drivers

specify this handle as the *hDriver* parameter to **DriverCallback**, when sending a callback message.

dwCallback
Specifies the address of a callback function. (See the description of the MXDM_OPEN message.)

dwInstance
Contains the *dwCallbackInstance* argument that the client specified when calling the **mixerOpen** function. This value is returned to the client as the *dwInstance* parameter to **DriverCallback**.

WAVEOPENDESC

```
typedef struct waveopendesc_tag {
    HWAVE hWave;
    LPWAVEFORMAT lpFormat;
    DWORD dwCallback;
    DWORD dwInstance;
    UINT uMappedDeviceID;
} WAVEOPENDESC;
```

The WAVEOPENDESC structure contains information needed by user-mode waveform input and output drivers. The structure is created by *winmm.dll* and passed to the driver with a WODM_OPEN or WIDM_OPEN message.

Members

hWave
Specifies the client's handle to the device, as assigned by *winmm.dll*. User-mode drivers specify this handle as the *hDriver* parameter to **DriverCallback**, when sending a callback message.

lpFormat
Points to a WAVEFORMATEX structure, indicating the waveform data format requested by the client. (The WAVEFORMATEX structure is described in the Win32 SDK.)

dwCallback
Specifies either the address of a callback function, a window handle, an event handle, or a task handle, depending on the flag contained in the *dwParam2* parameter of the WODM_OPEN or WIDM_OPEN message.

dwInstance
Contains the *dwCallbackInstance* argument that the client specified when calling the **waveInOpen** or **waveOutOpen** function. This value is returned to the client as the *dwInstance* parameter to **DriverCallback**.

uMappedDeviceID
For wave mapper, contains device identifier of mapped device.

Functions and Macros, *drvlib.lib*

This section describes the functions and macros available to user-mode audio drivers using *drvlib.lib*. Function prototypes and macros are defined in the file *registry.h*.

DrvAccess

DrvAccess(
PREG_ACCESS *RegAccess*
);

The **DrvAccess** macro determines if access to the service control manager has been granted by a previous call to **DrvCreateServicesNode**.

Parameters

RegAccess
Pointer to a globally-defined structure of type REG_ACCESS.

Return Value

The macro equals 1 if access was granted. Otherwise the macro equals 0.

Comments

The structure pointed to by *RegAccess* must be a single, globally-defined REG_ACCESS structure that the driver uses with all calls to *drvlib.lib* functions requiring a *RegAccess* parameter.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

DrvCloseServiceManager

```
VOID
    DrvCloseServiceManager(
      PREG_ACCESS RegAccess
      );
```

The **DrvCloseServiceManager** function closes the connection to the service control manager that was created by calling **DrvCreateServicesNode**.

Parameters

RegAccess
Pointer to a globally-defined structure of type REG_ACCESS.

Return Value

None.

Comments

The structure pointed to by *RegAccess* must be a single, globally-defined REG_ACCESS structure that the driver uses with all calls to *drvlib.lib* functions requiring a *RegAccess* parameter.

If **DrvSaveParametersKey** has been called to save registry contents in a temporary file, the file is deleted.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

DrvConfigureDriver

```
BOOL
    DrvConfigureDriver(
      PREG_ACCESS RegAccess,
      LPTSTR DriverName,
      SOUND_KERNEL_MODE_DRIVER_TYPE DriverType,
      BOOL (*SetParms)(PVOID),
      PVOID Context
      );
```

The **DrvConfigureDriver** function opens a connection to the service control manager, creates a kernel-mode driver service for the specified driver, and loads the kernel-mode driver.

Parameters

RegAccess
Pointer to a globally-defined structure of type REG_ACCESS.

DriverName
Pointer to the driver name. Must match *DriverName* in registry path.

DriverType

Driver type, used for indicating the kernel-mode driver's load group. The type is SOUND_KERNEL_MODE_DRIVER_TYPE. The value can be one of the following.

Value	Definition
SoundDriverTypeNormal	Adds kernel-mode driver to "base" load group.
SoundDriverTypeSynth	Adds kernel-mode driver to "Synthesizer Drivers" load group.

The "Synthesizer Drivers" group is unknown to Windows NT and therefore is guaranteed to be loaded last.

SetParms
Pointer to a driver-supplied function that is called before the kernel-mode driver is reloaded. Can be NULL.

Context
Pointer to a driver-defined structure that is passed as input to the function pointed to by *SetParms*. Can be NULL.

Return Value
Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments
The structure pointed to by *RegAccess* must be a single, globally-defined REG_ACCESS structure that the driver uses with all calls to *drvlib.lib* functions requiring a *RegAccess* parameter.

The function performs the following operations, in order:

1. Calls **DrvCreateServicesNode**, specifying TRUE for the *Create* parameter.
2. Calls **DrvUnloadKernelDriver** to unload the kernel-mode driver.
3. Calls the driver-supplied function specified by the *SetParms* parameter. Generally, drivers use this function to modify configuration parameters in the registry.
4. Calls **DrvLoadKernelDriver** to reload and restart the kernel-mode driver.

Typically, a user-mode driver calls **DrvConfigureDriver** from its **DriverProc** function when processing a DRV_CONFIGURE or DRV_INSTALL command, after obtaining user-specified configuration parameters from a dialog box.

After **DrvConfigureDriver** returns, call **DrvCloseServiceManager**.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

DrvCreateDeviceKey

```
HKEY
    DrvCreateDeviceKey(
      LPCTSTR DriverName
);
```

The **DrvCreateDeviceKey** function creates a device subkey under the driver's **\Parameters** key. The registry path to the created key is **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***\Parameters\Device***Number*.

Parameters
DriverName
Pointer to the driver name. Must match *DriverName* in registry path.

Return Value
Returns a handle to the device subkey, if the operation succeeds. Otherwise returns NULL.

Comments

The function creates the **\Parameters** key if it does not exist.

Device subkey names are assigned as **\Device0**, **\Device1**, **\Device2**, and so on.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

DrvCreateServicesNode

BOOL
DrvCreateServicesNode(
PTCHAR *DriverName*,
SOUND_KERNEL_MODE_DRIVER_TYPE *DriverType*,
PREG_ACCESS *RegAccess*,
BOOL *Create*
);

The **DrvCreateServicesNode** function creates a connection to the service control manager and, optionally, creates a service object for the kernel-mode driver. The caller must have Administrator's privilege.

Parameters

DriverName
Pointer to the driver name. Must match *DriverName* in registry path.

DriverType
Driver type, used for indicating the kernel-mode driver's load group. The type is SOUND_KERNEL_MODE_DRIVER_TYPE. The value can be one of the following.

Value	Definition
SoundDriverTypeNormal	Adds kernel-mode driver to "base" load group.
SoundDriverTypeSynth	Adds kernel-mode driver to "Synthesizer Drivers" load group.

The "Synthesizer Drivers" group is unknown to Windows NT and therefore is guaranteed to be loaded last.

RegAccess
Pointer to a globally-defined structure of type REG_ACCESS.

Create
If TRUE, the function creates a service object for the kernel-mode driver, if it does not already exist. If FALSE, the function does not create the service object.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

The structure pointed to by *RegAccess* must be a single, globally-defined REG_ACCESS structure that the driver uses with all calls to *drvlib.lib* functions requiring a *RegAccess* parameter.

Under Windows NT, kernel-mode drivers are considered to be services under the control of the service control manager. The **DrvCreateServicesNode** function calls **OpenSCManager** to create a connection to the local service control manager. **OpenSCManager** is called with a desired access type of SC_MANAGER_ALL_ACCESS, which requires Administrators privilege. (For information about **OpenSCManager**, see the Win32 SDK.) The service manager handle returned by **OpenSCManager** is stored in the REG_ACCESS structure.

If the *Create* parameter is TRUE, the **DrvCreateServicesNode** function calls **CreateService** to create the kernel-mode driver service and obtain a service handle. The **DrvCreateServicesNode** function sets the service's start type to SERVICE_DEMAND_START, so it will not automatically reload when the system is restarted. (For information about **CreateService**, see the Win32 SDK.)

A result of calling **CreateService** is the creation of a driver subkey under the **\Services** registry key. The path to the subkey is

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services*DriverName*.

If the *Create* parameter is FALSE, the function just creates a connection to the service control manager and saves the service manager handle returned by **OpenSCManager** in the REG_ACCESS structure. Anytime after calling **DrvCreateServicesNode**, you can call the **DrvAccess** macro to determine if access to the service control manager was granted.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

DrvDeleteServicesNode

BOOL
DrvDeleteServicesNode(
PREG_ACCESS *RegAccess*
);

The **DrvDeleteServicesNode** function marks for deletion a kernel-mode driver service that was created by calling **DrvCreateServicesNode**.

Parameters

RegAccess
Pointer to a globally-defined structure of type REG_ACCESS.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. To obtain an error code value, call **GetLastError**, which is described in the Win32 SDK.

Comments

The structure pointed to by *RegAccess* must be a single, globally-defined REG_ACCESS structure that the driver uses with all calls to *drvlib.lib* functions requiring a *RegAccess* parameter.

The function calls **DeleteService**, described in the Win32 SDK, to mark the kernel-mode driver service for deletion.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

DrvIsDriverLoaded

BOOL
DrvIsDriverLoaded(
PREG_ACCESS *RegAccess*
);

The **DrvIsDriverLoaded** function determines if the kernel-mode driver is currently loaded and running.

Parameters

RegAccess
Pointer to a globally-defined structure of type REG_ACCESS.

Return Value

Returns TRUE if the kernel-mode driver is loaded and running. Otherwise returns FALSE.

Comments

The structure pointed to by *RegAccess* must be a single, globally-defined REG_ACCESS structure that the driver uses with all calls to *drvlib.lib* functions requiring a *RegAccess* parameter.

Drivers must call **DrvCreateServicesNode** before calling **DrvIsDriverLoaded**.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

BOOL
DrvLibInit(
HINSTANCE *hModule*,
ULONG *Reason*,
PCONTEXT *pContext*
);

The **DrvLibInit** function initializes *drvlib.lib* for use with the calling user-mode audio driver. User-mode audio drivers call this function before they begin calling other functions in *drvlib.lib*. They call the function again prior to being unloaded.

Parameters

hModule
Instance handle of the module that opened the user-mode driver. Obtained by calling **GetDriverModuleHandle**, which is described in the Win32 SDK.

Reason
Set to one of the following values.

- DLL_PROCESS_ATTACH, if attaching to *drvlib.lib*.
- DLL_PROCESS_DETACH, if detaching from *drvlib.lib*.

pContext
Not used. Should be set to NULL.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

When the driver's **DriverProc** function receives a DRV_LOAD message, it should call **DrvLibInit** with a *Reason* value of DLL_PROCESS_ATTACH.

When the driver's **DriverProc** function receives a DRV_FREE message, it should call **DrvLibInit** with a *Reason* value of DLL_PROCESS_DETACH.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

DrvLoadKernelDriver

BOOL
DrvLoadKernelDriver(
PREG_ACCESS *RegAccess*
);

The **DrvLoadKernelDriver** function loads and starts the kernel-mode driver.

Parameters

RegAccess
Pointer to a globally-defined structure of type REG_ACCESS.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. To obtain an error code value, call **GetLastError**. which is described in the Win32 SDK.

Comments

The structure pointed to by *RegAccess* must be a single, globally-defined REG_ACCESS structure that the driver uses with all calls to *drvlib.lib* functions requiring a *RegAccess* parameter.

The function sets the kernel-mode driver service's start type to SERVICE_SYSTEM_START, so it

will automatically reload and restart when the system is restarted. (For more information, **ChangeServiceConfig** in the Win32 SDK.)

Drivers must call **DrvCreateServicesNode** before calling **DrvLoadKernelDriver**.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

DrvNumberOfDevices

LONG
DrvNumberOfDevices(
PREG_ACCESS *RegAccess*,
LPDWORD *NumberOfDevices*
);

The **DrvNumberOfDevices** function determines the number of device subkeys existing under the driver's **\Parameters** key in the registry.

Parameters

RegAccess
Pointer to a globally-defined structure of type REG_ACCESS.

NumberOfDevices
Address of a location to receive the number of devices.

Return Value

Returns ERROR_SUCCESS if the operation succeeds. Otherwise returns one of the error codes defined in *winerror.h*.

Comments

The structure pointed to by *RegAccess* must be a single, globally-defined REG_ACCESS structure that the driver uses with all calls to *drvlib.lib* functions requiring a *RegAccess* parameter.

Drivers create device subkeys by calling **DrvCreateDeviceKey**.

Drivers must call **DrvCreateServicesNode** before calling **DrvNumberOfDevices**.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

DrvQueryDeviceIdParameter

LONG
DrvQueryDeviceIdParameter(
PREG_ACCESS *RegAccess*,
UINT *DeviceNumber*,
PTCHAR *ValueName*,
PDWORD *pValue*);

The **DrvQueryDeviceIdParameter** function reads the value associated with the specified value name, under the specified driver's **\Parameters** registry key.

Parameters

RegAccess
Pointer to a globally-defined structure of type REG_ACCESS.

DeviceNumber
Device number. Used as an index to the device subkeys under the **\Parameters** key. See the **Comments** section below.

ValueName
Pointer to a string representing the value name.

pValue
Pointer to a DWORD to receive the requested value.

Return Value

Returns ERROR_SUCCESS if the operation succeeds. Otherwise returns one of the error codes defined in *winerror.h*. See the **Comments** section below.

Comments

The structure pointed to by *RegAccess* must be a single, globally-defined REG_ACCESS structure that the driver uses with all calls to *drvlib.lib* functions requiring a *RegAccess* parameter.

The value name and value are read from the registry path **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***\Parameters\Device***Number*, where *Number* represents the number supplied by the *DeviceNumber* parameter.

The function can only return a value that is stored as a REG_DWORD type.

The function calls **RegQueryValueEx** and returns its return value. (See the Win32 SDK.)

To store values under a driver's **\Parameters** registry key, call **DrvSetDeviceIdParameter**.

Drivers must call **DrvCreateServicesNode** before calling **DrvQueryDeviceIdParameter**.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

DrvRemoveDriver

```
LRESULT
    DrvRemoveDriver(
      PREG_ACCESS RegAccess
      );
```

The **DrvRemoveDriver** function unloads the kernel-mode driver and marks the kernel-mode driver service for deletion.

Parameters

RegAccess

Pointer to a globally-defined structure of type REG_ACCESS.

Return Value

Returns one of the following values.

Value	Definition
DRVCNF_OK	Kernel-mode driver service has been marked for deletion. Driver was not loaded prior to call.
DRVCNF_RESTART	Kernel-mode driver service has been marked for deletion. Driver was loaded prior to call.
DRVCNF_CANCEL	Attempt to mark kernel-mode driver service for deletion failed.

See the **Comments** section below.

Comments

The structure pointed to by *RegAccess* must be a single, globally-defined REG_ACCESS structure that the driver uses with all calls to *drvlib.lib* functions requiring a *RegAccess* parameter.

The **DrvRemoveDriver** function first checks to see if the kernel-mode driver is loaded and, if it is, calls **DrvUnloadKernelDriver**. Then it calls **DrvDeleteServicesNode**.

Typically, a user-mode driver calls **DrvRemoveDriver** from its **DriverProc** function, when processing a DRV_REMOVE command. The return values provided by **DrvRemoveDriver** match those specified for DRV_REMOVE.

The driver must call **DrvCreateServicesNode** before calling **DrvRemoveDriver**. After **DrvRemoveDriver** returns, the driver should call **DrvCloseServiceManager**.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

DrvRestoreParametersKey

BOOL
DrvRestoreParametersKey(
PREG_ACCESS *RegAccess*
);

The **DrvRestoreParametersKey** function restores registry information that was saved when the driver called **DrvSaveParametersKey**.

Parameters

RegAccess
Pointer to a globally-defined structure of type REG_ACCESS.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

The structure pointed to by *RegAccess* must be a single, globally-defined REG_ACCESS structure that the driver uses with all calls to *drvlib.lib* functions requiring a *RegAccess* parameter.

The **DrvRestoreParametersKey** function deletes the temporary file in which the key information was written by **DrvSaveParametersKey**, and removes the file name from the REG_ACCESS structure.

Drivers must call **DrvCreateServicesNode** before calling **DrvRestoreParametersKey**.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

DrvSaveParametersKey

BOOL
DrvSaveParametersKey(
PREG_ACCESS *RegAccess*
);

The **DrvSaveParametersKey** function saves the current contents of the driver's **\Parameters** registry key, along with its subkeys, in a temporary file.

Parameters

RegAccess
Pointer to a globally-defined structure of type REG_ACCESS.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

The structure pointed to by *RegAccess* must be a single, globally-defined REG_ACCESS structure that the driver uses with all calls to *drvlib.lib* functions requiring a *RegAccess* parameter.

If the **DrvSaveParametersKey** function succeeds, the REG_ACCESS structure contains the name of the temporary file.

To restore the saved registry information, call **DrvRestoreParametersKey**.

Drivers must call **DrvCreateServicesNode** before calling **DrvSaveParametersKey**.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

LONG
DrvSetDeviceIdParameter(
PREG_ACCESS *RegAccess*,
UINT *DeviceNumber*,
PTCHAR *ValueName*,
DWORD *Value*
);

The **DrvSetDeviceIdParameter** function assigns the specified value to the specified value name in the registry, under the specified device's **\Parameters** key.

Parameters

RegAccess
Pointer to a globally-defined structure of type REG_ACCESS.

DeviceNumber
Device number. Used as an index to the device subkeys under the **\Parameters** key. See the **Comments** section below.

ValueName
Pointer to a string representing the value name.

Value
Value to be written.

Return Value

Returns ERROR_SUCCESS if the operation succeeds. Otherwise returns one of the error codes defined in *winerror.h*. See the **Comments** section below.

Comments

The structure pointed to by *RegAccess* must be a single, globally-defined REG_ACCESS structure that the driver uses with all calls to *drvlib.lib* functions requiring a *RegAccess* parameter.

Before a driver can call the **DrvSetDeviceIdParameter** function, it must call **DrvCreateDeviceKey**.

The value name and value are written to the registry path **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***\Parameters\Device***Number*, where *Number* represents the number supplied by the *DeviceNumber* parameter.

The specified value is stored as a REG_DWORD type.

The function calls **RegSetValueEx**, which is described in the Win32 SDK, and returns its return value.

To retrieve a registry value written with **DrvSetDeviceIdParameter**, call **DrvQueryDeviceIdParameter**.

Drivers must call **DrvCreateServicesNode** before calling **DrvSetDeviceIdParameter**.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

DrvSetMapperName

VOID
DrvSetMapperName(
LPTSTR *SetupName*
);

The **DrvSetMapperName** function specifies the name of the map that the MIDI Mapper should use.

Parameters

SetupName
Pointer to a string representing the name of the mapping to use.

Return Value
None.

Comments
The functions assigns the specified string to be the value of "Mapping Name", under the registry key **HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Midimap**.

Drivers must call **DrvCreateServicesNode** before calling **DrvSetMapperName**.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

DrvUnloadKernelDriver

BOOL
DrvUnloadKernelDriver(
PREG_ACCESS *RegAccess*
);

The **DrvUnloadKernelDriver** function unloads the kernel mode driver.

Parameters
RegAccess
Pointer to a globally-defined structure of type REG_ACCESS.

Return Value
Returns TRUE if the operation succeeds. Otherwise returns FALSE. To obtain an error code value, call **GetLastError**, which is described in the Win32 SDK.

Comments
The structure pointed to by *RegAccess* must be a single, globally-defined REG_ACCESS structure that the driver uses with all calls to *drvlib.lib* functions requiring a *RegAccess* parameter.

The function sets the kernel-mode driver service's start type to SERVICE_DEMAND_START, so it will not automatically reload when the system is restarted. (For more information, **ChangeServiceConfig** in the Win32 SDK.)

Drivers must call **DrvCreateServicesNode** before calling **DrvUnLoadKernelDriver**.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

GetInterruptsAndDMA

BOOL
GetInterruptsAndDMA(
LPDWORD *InterruptsInUse*,
LPDWORD *DmaChannelsInUse*,
LPCTSTR *IgnoreDriver*
);

The **GetInterruptsAndDMA** function examines the registry to determine which interrupt numbers and DMA channels are assigned to devices.

Parameters
InterruptsInUse
Pointer to a DWORD. Receives a bit array of interrupts in use. If interrupt *i* is in use, then bit *i* is set.

DMAChannelsInUse
Pointer to a DWORD. Receives a bit array of DMA channels in use. If channel *i* is in use, then bit *i* is set.

IgnoreDriver
Pointer to the name of a kernel-mode driver. The interrupt numbers and DMA channels of the devices controlled by this driver are not included in the returned *InterruptsInUse* and *DMAChannelsInUse* bit arrays. Can be NULL.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

Typically, a user-mode driver calling this function specifies the name of its kernel-mode driver for the *IgnoreDriver* parameter.

For additional information, see Installing and Configuring your Driver, Using *drvlib.lib*.

Structures and Types, *drvlib.lib*

This section describes the structures and types available to user-mode audio drivers using *drvlib.lib*. The structures and types are defined in the file *registry.h*.

REG_ACCESS

```
typedef struct {
    SC_HANDLE     ServiceManagerHandle;
    LPTSTR        DriverName;
    TCHAR         TempKeySaveFileName[MAX_PATH];
} REG_ACCESS, *PREG_ACCESS;
```

The REG_ACCESS structure contains registry access information.

Members

ServiceManagerHandle
Contains a handle to the local service control manager.

DriverName
Pointer to a driver name. This is the *DriverName* argument to **DrvCreateServicesNode**.

TempKeySaveFileName
Name of a file created by **DrvSaveParametersKey**.

Comments

User-mode drivers using *drvlib.lib* functions to access the registry must declare a single global variable of this type for use with all function calls.

Drivers do not reference the structure contents.

The structure is initialized by **DrvCreateServicesNode**.

SOUND_KERNEL_MODE_DRIVER_TYPE

```
typedef enum {
    SoundDriverTypeNormal = 1,
    SoundDriverTypeSynth          /* Go in the synth group */
} SOUND_KERNEL_MODE_DRIVER_TYPE;
```

SOUND_KERNEL_MODE_DRIVER_TYPE is an enumeration type used to differentiate drivers that belong to the "base" load group from those that do not.

Specifying **SoundDriverTypeNormal** assigns a driver to the "base" group.

Specifying **SoundDriverTypeSynth** assigns a driver to the "Synthesizer Drivers" group. This group is unknown to Windows NT and therefore is guaranteed to be loaded last.

The type is used with input parameters to **DrvCreateServicesNode** and **DrvConfigureDriver**. (See **CreateService** in the Win32 SDK for information about load groups.)

Functions, *soundlib.lib*

This section describes the functions available to kernel-mode audio drivers using *soundlib.lib*. Function prototypes are defined in *soundlib.h*, *devices.h*, *wave.h*, *midi.h*, *mixer.h*, and *synthdrv.h*.

SoundAddIrpToCancellableQ

VOID
SoundAddIrpToCancellableQ(
PLIST_ENTRY *QueueHead*,
PIRP *Irp*,
BOOLEAN *Head*
);

The **SoundAddIrpToCancellableQ** function adds an IRP to a queue and makes the IRP cancelable.

Parameters

QueueHead
Pointer to the head of a queue of IRPs.

Irp
Pointer to an IRP.

Head
TRUE or FALSE. If TRUE, add IRP to head of queue. If FALSE, add IRP to tail of queue.

Return Value

None.

Comments

Before an IRP can be added to the queue, the queue must be initialized by calling **InitializeListHead**.

Generally, drivers using *soundlib.lib* do not need to manipulate IRP queues because DPCs in *soundlib.lib* handle IRP completion. For drivers that do manipulate IRP queues, the **SoundAddIrpToCancellableQ**, **SoundRemoveFromCancellableQ**, **SoundMoveCancellableQueue**, **SoundFreePendingIrps**, and **SoundFreeQ** functions are provided.

SoundAuxDispatch

NTSTATUS
SoundAuxDispatch(
IN OUT PLOCAL_DEVICE_INFO *pLDI*,
IN PIRP *pIrp*,
IN PIO_STACK_LOCATION *IrpStack*
);

The **SoundAuxDispatch** function is the IRP control code dispatcher for auxiliary audio devices.

Parameters

pLDI

Pointer to a LOCAL_DEVICE_INFO structure.

pIrp
Pointer to an IRP.

IrpStack
Pointer to an I/O stack location.

Return Value

Returns STATUS_SUCCESS if the operation succeeds. Otherwise returns an NTSTATUS error code.

Comments

Kernel-mode auxiliary audio device drivers using *soundlib.lib* place the address of the **SoundAuxDispatch** function in the **DispatchRoutine** member of a SOUND_DEVICE_INIT structure. The function is called by *soundlib.lib*'s main dispatcher, **SoundDispatch**.

The function processes the following IRP control codes:

IRP_MJ_CLEANUP
IRP_MJ_CLOSE
IRP_MJ_CREATE
IRP_MJ_DEVICE_CONTROL

For IRP_MJ_DEVICE_CONTROL, the **SoundAuxDispatch** function processes the following I/O control codes:

IOCTL_AUX_GET_CAPABILITIES
IOCTL_AUX_GET_VOLUME
IOCTL_AUX_SET_VOLUME
IOCTL_SOUND_GET_CHANGED_VOLUME

Refer to the *Kernel-Mode Drivers Reference* for descriptions of PIRP and PIO_STACK_LOCATION types, along with IRP control codes and I/O control codes.

SoundConnectInterrupt

NTSTATUS
SoundConnectInterrupt(
IN ULONG *InterruptNumber*,
IN INTERFACE_TYPE *BusType*,
IN ULONG *BusNumber*,
IN PKSERVICE_ROUTINE *Isr*,
IN PVOID *ServiceContext*,
IN KINTERRUPT_MODE *InterruptMode*,
IN BOOLEAN *ShareVector*,
OUT PKINTERRUPT **Interrupt*
);

The **SoundConnectInterrupt** function creates an interrupt object and installs an interrupt handler.

Parameters

InterruptNumber
Interrupt number.

BusType
Bus type. INTERFACE_TYPE is defined in *ntddk.h*.

BusNumber
Bus number, as returned from **SoundGetBusNumber**.

Isr
Pointer to an interrupt service routine (ISR). PKSERVICE_ROUTINE is defined in *ntddk.h* as

follows:

```
typedef BOOLEAN
(*PKSERVICE_ROUTINE) (
    IN struct _KINTERRUPT *Interrupt,
    IN PVOID ServiceContext
    );
```

ServiceContext
Pointer to driver-specified information that is passed to the ISR.

InterruptMode
Interrupt mode (**Latched** or **LevelSensitive**). KINTERRUPT_MODE is defined in *ntddk.h*.

ShareVector
TRUE if interrupt is shareable, FALSE otherwise.

Interrupt
Pointer to one or more system-defined interrupt objects in nonpaged memory.

Return Value

Returns one of the following values.

Value	Definition
STATUS_SUCCESS	Operation succeeded.
STATUS_DEVICE_CONFIGURATION_ERROR	Improper input parameter.
STATUS_INSUFFICIENT_RESOURCES	Insufficient system resources.

Comments

Interrupt objects are of type _KINTERRUPT. These are a system-defined type that your driver doesn't reference directly. Save the interrupt object pointer received in *Interrupt*, because it must be passed as input to **KeSynchronizeExecution**.

The ISR is responsible for clearing device interrupts. The HAL handles controller interrupts. After interrupts are cleared, the ISR should call **IoRequestDPC** to queue a deferred procedure call to the DPC function pointed to by the **DeferredRoutine** member of the SOUND_DEVICE_INIT structure. The DPC function should finish processing the interrupt.

Prior to being unloaded, the driver must release the interrupt objects by calling **IoDisconnectInterrupt**.

For discussions of interrupt objects and deferred procedure calls, see the *Kernel-Mode Drivers Design Guide*.

SoundCreateDevice

NTSTATUS
SoundCreateDevice(
IN PCSOUND_DEVICE_INIT *DeviceInit*,
IN UCHAR *CreationFlags*,
IN PDRIVER_OBJECT *pDriverObject*,
IN PVOID *pGDI*,
IN PVOID *DeviceSpecificData*,
IN PVOID *pHw*,
IN int *i*,
OUT PDEVICE_OBJECT **ppDevObj*
);

The **SoundCreateDevice** function creates a device object and an associated LOCAL_DEVICE_INFO structure.

Parameters

DeviceInit
Pointer to an initialized SOUND_DEVICE_INIT structure.

CreationFlags
The following flag values are supported:

Flag	Definition
SOUND_CREATION_NO_NAME_RANGE	Don't append number to name prototype when creating device name.
SOUND_CREATION_NO_VOLUME	Volume setting not supported.

pDriverObject
Pointer to the driver object received as input to the **DriverEntry** function.

pGDI
Pointer to driver-specified information. Pointer is stored in the **pGlobalInfo** member of the device's LOCAL_DEVICE_INFO structure.

DeviceSpecificData
Pointer to one of the following device-type structures.

Structure	Usage
WAVE_INFO	For waveform devices
MIDI_INFO	For MIDI devices
MIXER_INFO	For mixer devices
NULL	For other devices

pHw
Pointer to driver-specified hardware context information. The pointer is stored in the **HwContext** member of the device's LOCAL_DEVICE_INFO structure.

i
Driver-specified index value. Stored in **DeviceIndex** member of the device's LOCAL_DEVICE_INFO structure.

ppDevObj
Address of a location to receive a pointer to a DEVICE_OBJECT structure, if the call succeeds.

Return Value

Returns STATUS_SUCCESS if device creation succeeds. Returns STATUS_INSUFFICIENT_RESOURCES if device creation fails.

Comments

The SOUND_DEVICE_INIT structure must be nonpaged.

The **SoundCreateDevice** function creates a device object by calling **IoCreateDevice**. If the object creation succeeds, the function returns the address of a DEVICE_OBJECT structure. The function also allocates a LOCAL_DEVICE_INFO structure and assigns its address to the **DeviceExtension** member of the DEVICE_OBJECT structure.

The function calls **IoCreateSymbolicLink** to map the NT generic name for the device object to the corresponding Win32 logical name.

For more information about device objects, see the *Kernel-Mode Drivers Design Guide*.

SoundCreateDeviceName

```
NTSTATUS
    SoundCreateDeviceName(
      PCWSTR PrePrefix,
      PCWSTR Prefix,
      UCHAR  Index,
      PUNICODE_STRING DeviceName
      );
```

The **SoundCreateDeviceName** function creates a device name from the specified component parts.

Parameters

PrePrefix
Pointer to a string to prepend to the string pointed to by *Prefix*. Typically, this string is "\Device\" or "\DosDevices\".

Prefix
Pointer to a string containing the main part of the name. Typically, this is the **PrototypeName** member of a SOUND_DEVICE_INIT structure.

Index
Number to append to the device name. Specify 0xFF if an index number should not be appended.

DeviceName
Pointer to a UNICODE_STRING structure, which receives the constructed name string.

Return Value

Returns one of the following values.

Value	Definition
STATUS_SUCCESS	Operation succeeded.
STATUS_INSUFFICIENT_RESOURCES	Could not allocate string buffer.

SoundDelay

```
VOID
    SoundDelay(
      IN ULONG Milliseconds
      );
```

The **SoundDelay** function delays execution of the calling thread for at least the specified number of milliseconds.

Parameters

Milliseconds
Number of milliseconds to delay.

Return Value

None.

Comments

If the IRQL is greater than or equal to DISPATCH_LEVEL, the function returns without waiting.

SoundDispatch

```
NTSTATUS
    SoundDispatch(
      IN PDEVICE_OBJECT pDO,
      IN PIRP pIrp
  );
```

The **SoundDispatch** function is the main dispatcher for IRP function codes within *soundlib.lib*

Parameters

pDO
Pointer to a device object.

pIrp

Pointer to an IRP.

Return Value

Returns STATUS_SUCCESS if the operation succeeds. Otherwise returns an NTSTATUS error code.

Comments

For more information about device objects and IRPs, see the *Kernel-Mode Drivers Design Guide*.

Within the DEVICE_OBJECT structure passed to a kernel-mode driver's **DriverEntry** function, **SoundDispatch** must be specified as the dispatcher for the following IRP control codes:

- IRP_MJ_CLEANUP
- IRP_MJ_CLOSE
- IRP_MJ_CREATE
- IRP_MJ_DEVICE_CONTROL
- IRP_MJ_READ
- IRP_MJ_WRITE

SoundDispatch does not process the IRPs. Instead, it calls the secondary dispatcher specified as the **DispatchRoutine** member of the appropriate SOUND_DEVICE_INIT structure. (A pointer to the SOUND_DEVICE_INIT structure is contained in the device object.) Secondary dispatchers provided by *soundlib.lib* are:

- **SoundAuxDispatch**
- **SoundMidiDispatch**
- **SoundMixerDispatch**
- **SoundWaveDispatch**

SoundEnter

VOID
SoundEnter(
PLOCAL_DEVICE_INFO *pLDI*,
BOOLEAN *Enter*
);

The **SoundEnter** function calls a device's exclusion routine, passing it either a **SoundExcludeEnter** or a **SoundExcludeLeave** message.

Parameters

pLDI
Pointer to a LOCAL_DEVICE_INFO structure.

Enter
TRUE or FALSE. If TRUE, the exclusion routine is called with a **SoundExcludeEnter** message. If FALSE, the exclusion routine is called with a **SoundExcludeLeave** message.

Return Value

None.

Comments

The **SoundEnter** function calls the exclusion routine that is specified as the **ExclusionRoutine** member of a SOUND_DEVICE_INIT structure. The SOUND_DEVICE-INIT structure's address is contained in the *pLDI* structure. See "Using Exclusion Routines".

For definitions of the **SoundExcludeEnter** and **SoundExcludeLeave** messages, see SOUND_EXCLUDE_CODE.

SoundEnumSubkeys

NTSTATUS
SoundEnumSubkeys(
IN PUNICODE_STRING *RegistryPathName*,
IN PWSTR *Subkey*,
IN PSOUND_REGISTRY_CALLBACK_ROUTINE *Callback*,
IN PVOID *Context*
);

The **SoundEnumSubkeys** function enumerates the subkeys under the specified subkey, which is in the specified registry path. For each subkey found under the specified subkey, **SoundEnumSubkeys** calls the specified callback function.

Parameters

RegistryPathName
Pointer to a string representing the registry path to the driver subkey. Use the path name received as the *RegistryPathName* argument to **DriverEntry**.

SubKey
Pointer to a string representing the subkey at end of the path specified by *RegistryPathName*. **SoundEnumSubkeys** enumerates subkeys under this subkey.

Callback
Pointer to a function to call for each subkey found under *Subkey*. The function type is SOUND_REGISTRY_CALLBACK_ROUTINE.

Context
Pointer to a driver-specified context parameter. This parameter is passed to the function pointed to by *Callback*.

Return Value

If the function detects a failure, it returns an NTSTATUS error code. Otherwise it returns the value returned by the callback function. If the callback function succeeds, it should return STATUS_SUCCESS. Otherwise it should return an NTSTATUS error code.

Comments

Use the **SoundEnumSubkeys** function when performing device initialization operations, if your driver supports multiple hardware devices. If you specify "Parameters" for the *Subkey* parameter, the function calls the callback function for each "device" subkey under **\Parameters**. For an example, see Examining **DriverEntry** in *sndblst.sys*. (The device entries under the **\Parameters** subkey are created by user-mode drivers, typically by calling **DrvCreateDeviceKey**.)

SoundEnumSubkeys passes a registry path name to the callback function (see SOUND_REGISTRY_CALLBACK_ROUTINE).This path name is the path name received as the *RegistryPathName* argument, with **\Parameters\Device***Number* appended.

The callback function returns an NTSTATUS value. If, for any call to the callback function, the return value is not STATUS_SUCCESS, then **SoundEnumSubkeys** immediately returns to its caller, passing back the return value received from the callback function.

Drivers which do not support multiple hardware devices do not call **SoundEnumSubkeys**. Instead, they should call **SoundSaveRegistryPath** as part of their initialization operation, in order to create a registry path name that can be used as input to other *soundlib.lib* functions.

SoundFreeCommonBuffer

VOID
SoundFreeCommonBuffer(
IN OUT PSOUND_DMA_BUFFER *SoundAutoData*
);

The **SoundFreeCommonBuffer** function frees a DMA buffer that was previously allocated by **SoundGetCommonBuffer**.

Parameters

SoundAutoData
Pointer to a SOUND_DMA_BUFFER structure.

Return Value

None.

Comments

The **SoundFreeCommonBuffer** function also frees the MDL associated with the buffer. (For information about MDLs, see the *Kernel-Mode Drivers Design Guide*.)

Drivers calling **SoundFreeCommonBuffer** must include *wave.h*.

SoundFreeDevice

```
VOID
    SoundFreeDevice(
      IN  PDEVICE_OBJECT DeviceObject
      );
```

The **SoundFreeDevice** function deletes a device object and frees its resources.

Parameters

DeviceObject
Pointer to a device object.

Return Value

None.

Comments

SoundFreeDevice frees resources that were reserved with **SoundReportResourceUsage**. It also frees resources associated with the *driver* object. The specified *device* object is deleted.

SoundFreePendingIrps

```
VOID
    SoundFreePendingIrps(
      PLIST_ENTRY QueueHead,
      PFILE_OBJECT FileObject
      );
```

The **SoundFreePendingIrps** function traverses a queue of IRPs. For each IRP associated with the specified file object, the function completes the interrupt request and removes the IRP from the queue.

Parameters

QueueHead
Pointer to the head of a queue of IRPs.

FileObject
Pointer to a file object.

Return Value

None.

Comments

The function completes the interrupt request by calling **IoCompleteRequest**, with a priority boost argument of IO_NO_INCREMENT.

SoundFreePendingIrps only works for IRP queues created by calling **SoundAddIrpToCancellableQ**. It makes each IRP noncancelable before calling **IoCompleteRequest**.

Generally, drivers using *soundlib.lib* do not need to manipulate IRP queues because DPCs in *soundlib.lib* handle IRP completion. For drivers that do manipulate IRP queues, the **SoundAddIrpToCancellableQ**, **SoundRemoveFromCancellableQ**, **SoundMoveCancellableQueue**, **SoundFreePendingIrps**, and **SoundFreeQ** functions are provided.

SoundFreeQ

VOID
SoundFreeQ(
PLIST_ENTRY *ListHead***,**
NTSTATUS *IoStatus*
);

The **SoundFreeQ** function traverses a queue of IRPs. For each IRP, the function completes the interrupt request and removes the IRP from the queue.

Parameters

ListHead
Pointer to the head of a queue of IRPs.

IoStatus
An NTSTATUS code to use as an IRP completion status. This value, which is placed in each IRP, is generally either STATUS_SUCCESS or STATUS_CANCELLED.

Return Value

None.

Comments

The function completes the interrupt request by calling **IoCompleteRequest**, with a priority boost argument of IO_SOUND_INCREMENT.

SoundFreeQ only works for IRP queues created by calling **SoundAddIrpToCancellableQ**. It dequeues entries by calling **SoundRemoveFromCancellableQ**.

Generally, drivers using *soundlib.lib* do not need to manipulate IRP queues because DPCs in *soundlib.lib* handle IRP completion. For drivers that do manipulate IRP queues, the **SoundAddIrpToCancellableQ**, **SoundRemoveFromCancellableQ**, **SoundMoveCancellableQueue**, **SoundFreePendingIrps**, and **SoundFreeQ** functions are provided.

SoundGetBusNumber

NTSTATUS
SoundGetBusNumber(
IN OUT INTERFACE_TYPE *InterfaceType***,**
OUT PULONG *BusNumber*
);

The **SoundGetBusNumber** function returns the number of the first bus of the specified bus type.

Parameters

InterfaceType

Type of bus for which to search. INTERFACE_TYPE is defined in *ntddk.h*.

BusNumber

Address of location to receive the bus number, if found.

Return Value

Returns STATUS_SUCCESS if successful. Otherwise returns STATUS_DEVICE_DOES_NOT_EXIST.

SoundGetCommonBuffer

NTSTATUS
SoundGetCommonBuffer(
IN PDEVICE_DESCRIPTION *DeviceDescription*,
IN OUT PSOUND_DMA_BUFFER *SoundAutoData*
);

The **SoundGetCommonBuffer** function allocates a buffer for use by waveform devices during auto-initialize DMA transfers, and maps the buffer so it is accessible to both the processor and the device.

Parameters

DeviceDescription

Pointer to an initialized DEVICE_DESCRIPTION structure. (For a description of this structure, see **HalGetAdapter**.)

SoundAutoData

Pointer to a SOUND_DMA_BUFFER structure. Generally this is the address of the **DMABuf** member of a WAVE_INFO structure.

Return Value

Returns one of the following values.

Value	Definition
STATUS_SUCCESS	Operation succeeded.
STATUS_DEVICE_CONFIGURATION_ERROR	Couldn't find adapter.
STATUS_INSUFFICIENT_RESOURCES	Not enough memory to allocate buffer.

Comments

The returned buffer is described by the contents of the SOUND_DMA_BUFFER structure.

SoundGetCommonBuffer calls **HalGetAdapter** to obtain an adapter object and **HalAllocateCommonBuffer** to get a buffer. If a buffer of the requested size is not available, a smaller one is returned. The smallest buffer that can be requested is 4 kilobytes.

After the buffer has been allocated, the function calls **IoAllocateMdl** and **MmBuildMdlForNonPagedPool** to build a memory descriptor list (MDL). A MDL is needed by *soundlib.lib* for calls to **IoMapTransfer**.

If the requested buffer size is large, *soundlib.lib* might not use the entire buffer. To find out how much of the buffer is actually used, call **SoundGetDMABufferSize**.

Prior to being unloaded, the driver must free the allocated buffer space by calling **SoundFreeCommonBuffer**.

Drivers calling **SoundGetCommonBuffer** must include *wave.h*.

SoundGetDMABufferSize

ULONG
SoundGetDMABufferSize(
IN PWAVE_INFO *WaveInfo*

```
);
```

The **SoundGetDMABufferSize** function returns the actual DMA buffer size used for DMA transfers by the specified waveform device.

Parameters

WaveInfo
Pointer to a WAVE_INFO structure.

Return Value

Buffer size in bytes.

Comments

This function returns the number of bytes that are actually used within the DMA buffer specified by the WAVE_INFO structure's **DMABuf** member. If the buffer is large enough, *soundlib.lib* tries to set a buffer size corresponding to 1/8 of a second of play or record time.

Generally, DMA buffers for waveform devices are allocated by calling **SoundGetCommonBuffer**.

Drivers calling **SoundGetDMABufferSize** must include *wave.h*.

SoundGetTime

```
LARGE_INTEGER
  SoundGetTime(
    VOID
    );
```

The **SoundGetTime** function returns a time value in 100-nanosecond units.

Parameters

None.

Return Value

Returns a time value in 100-nanosecond units.

Comments

This function calls **KeQueryPerformanceCounter**. Refer to the *Kernel-Mode Drivers Reference* for a discussion of **KeQueryPerformanceCounter** and restrictions on its use.

SoundInitDataItem

```
VOID
    SoundInitDataItem(
      PMIXER_INFO MixerInfo,
      PMIXER_DATA_ITEM MixerDataItem,
      USHORT Message,
      USHORT Id
      );
```

The **SoundInitDataItem** function adds a mixer data item to an internal list. Mixer data items represent mixer lines and controls. When items on the list change state, clients are notified of the change.

Parameters

MixerInfo
Pointer to a MIXER_INFO structure.

MixerDataItem
Pointer to an empty, globally-allocated MIXER_DATA_ITEM structure.

Message
Message to be sent to the client when an item changes. One of the following messages will be sent:

- MM_MIXM_LINE_CHANGE, for mixer line items.
- MM_MIXM_CONTROL_CHANGE, for mixer control items.

Id
Driver-defined ID value.

Return Value
None.

Comments
The pointer specified by *MixerDataItem* is added to the **ChangedItems** list for the specified MIXER_INFO structure. The driver calls **SoundMixerChangedItem** when an item on the **ChangedItems** list changes state.

Drivers calling **SoundInitDataItem** must include *mixer.h*.

SoundInitializeWaveInfo

VOID
SoundInitializeWaveInfo(
PWAVE_INFO *WaveInfo*,
UCHAR *DMAType*,
PSOUND_QUERY_FORMAT_ROUTINE *QueryFormat*,
PVOID *HwContext*
);

The **SoundInitializeWaveInfo** initializes a WAVE_INFO structure.

Parameters
WaveInfo
Pointer to a WAVE_INFO structure.

DMAType
Type of DMA to use. One of the following enumerated values:

```
enum {
    SoundNoDMA,
    SoundAutoInitDMA,             // Use auto-initialize
    SoundReprogramOnInterruptDMA, // Reprogram on interrupt
    Sound2ChannelDMA              // Keep 2 channels going
};
```

Sound2ChannelDMA is not currently supported in *soundlib.lib*.

QueryFormat
Pointer to a function of type SOUND_QUERY_FORMAT_ROUTINE.

HwContext
Pointer to a driver-defined structure containing hardware context information. Pointer is stored in the **HwContext** member of WAVE_INFO.

Return Value
None.

Comments
Before calling **SoundInitializeWaveInfo**, the driver must initialize the **HwSetupDMA**, **HwStopDMA**, and **HwSetWaveFormat** structure members and zero the rest of the structure.

Drivers calling **SoundInitializeWaveInfo** must include *wave.h*.

VOID
SoundInitMidiIn(
IN OUT PMIDI_INFO *pMidi*,
IN PVOID *HwContext*
);

The **SoundInitMidiIn** function initializes a MIDI_INFO structure.

Parameters

pMidi
Pointer to a MIDI_INFO structure.

HwContext
Pointer to a driver-defined structure containing hardware context information. The pointer is stored in the **HwContext** member of MIDI_INFO.

Return Value

None.

Comments

Before calling **SoundInitMidiIn**, the driver must initialize the **HwStartMidiIn**, **HwStopMidiIn**, **HwMidiRead**, and **HwMidiOut** structure members and zero the rest of the structure.

Drivers calling **SoundInitMidiIn** must include *midi.h*.

SoundInitMixerInfo

VOID
SoundInitMixerInfo(
IN OUT PMIXER_INFO *MixerInfo*,
PMIXER_DD_GET_SET_DATA *HwGetLineData*,
PMIXER_DD_GET_SET_DATA *HwGetControlData*,
PMIXER_DD_GET_SET_DATA *HwGetCombinedControlData*,
PMIXER_DD_GET_SET_DATA *HwGetSetControlData*
);

The **SoundInitMixerInfo** function initializes a MIXER_INFO structure.

Parameters

MixerInfo
Pointer to a MIXER_INFO structure.

HwGetLineData
Pointer to a function of type PMIXER_DD_GET_SET_DATA. The pointer is stored in the **HwGetLineData** member of MIXER_INFO.

HwGetControlData
Pointer to a function of type PMIXER_DD_GET_SET_DATA. The pointer is stored in the **HwGetControlData** member of MIXER_INFO.

HwGetCombinedControlData
Pointer to a function of type PMIXER_DD_GET_SET_DATA. The pointer is stored in the **HwGetCombinedControlData** member of MIXER_INFO.

HwGetSetControlData
Pointer to a function of type PMIXER_DD_GET_SET_DATA. The pointer is stored in the **HwGetSetControlData** member of MIXER_INFO.

Return Value

None.

Comments

The function zeros the structure before initializing it.

Drivers calling **SoundInitMixerInfo** must include *mixer.h*.

SoundMapPortAddress

PUCHAR
SoundMapPortAddress(
INTERFACE_TYPE *BusType*,
ULONG *BusNumber*,
ULONG *PortBase*,
ULONG *Length*,
PULONG *MemType*
);

The **SoundMapPortAddress** function translates a bus-specific address into the corresponding system-logical address and then, if the logical address is in memory address space, maps the address to a virtual address.

Parameters

BusType
Bus type. INTERFACE_TYPE is defined in *ntddk.h*.

BusNumber
Bus number, as returned from **SoundGetBusNumber**.

PortBase
Port number.

Length
Number of ports.

MemType
Address of a location to receive a value indicating the memory type. A value of zero indicates the return value is a memory address. A value of one indicates the return value is an I/O space address.

Return Value

Returns the base virtual address.

Comments

Use **SoundMapPortAddress** to obtain addresses that are suitable for use with HAL Port I/O routines, such as READ_PORT_UCHAR.

If a value of zero is received in the address pointed to by *memtype*, then the returned virtual address is a mapped address. In this case the driver must call **MmUnmapIoSpace** to unmap the address range, prior to being unloaded.

This function calls **HalTranslateBusAddress** and **MmMapIoSpace**.

SoundMidiDispatch

NTSTATUS
SoundMidiDispatch(
IN OUT PLOCAL_DEVICE_INFO *pLDI*,
IN PIRP *pIrp*,
IN PIO_STACK_LOCATION *IrpStack*
);

The **SoundMidiDispatch** function is the IRP control code dispatcher for MIDI devices.

Parameters

pLDI
Pointer to a LOCAL_DEVICE_INFO structure.

pIrp
Pointer to an IRP.

IrpStack
Pointer to an I/O stack location.

Return Value

Returns STATUS_SUCCESS if the operation succeeds. Otherwise returns an NTSTATUS error code.

Comments

Kernel-mode MIDI device drivers using *soundlib.lib* place the address of this function in the **DispatchRoutine** member of a SOUND_DEVICE_INIT structure. The function is called by *soundlib.lib*'s main dispatcher, **SoundDispatch**.

The function processes the following IRP control codes:

IRP_MJ_CLEANUP
IRP_MJ_CLOSE
IRP_MJ_CREATE
IRP_MJ_DEVICE_CONTROL
IRP_MJ_READ

For IRP_MJ_DEVICE_CONTROL, the function processes the following I/O control codes:

IOCTL_MIDI_GET_CAPABILITIES
IOCTL_MIDI_GET_STATE
IOCTL_MIDI_PLAY
IOCTL_MIDI_SET_STATE

See the *Kernel-Mode Drivers Design Guide* for descriptions of PIRP and PIO_STACK_LOCATION types, along with IRP control codes and I/O control codes.

SoundMidiInDeferred

VOID
SoundMidiInDeferred(
IN PKDPC *pDpc*,
IN PDEVICE_OBJECT *pDeviceObject*,
IN OUT PIRP *pIrpDeferred*,
IN OUT PVOID *Context*
);

The **SoundMidiInDeferred** function is the DPC function that is provided by *soundlib.lib* for MIDI input devices.

Parameters

pDpc
Pointer to a DPC object.

pDeviceObject
Pointer to a device object.

pIrpDeferred
Pointer to an IRP. (Not used by **SoundMidiInDeferred**. See **Comments** section below.)

Context
Pointer to context information. (Not used by **SoundMidiInDeferred**. See **Comments** section below.)

Return Value

None.

Comments

Drivers using *soundlib.lib* should specify this function's address as the **DeferredRoutine** member of each MIDI input device's SOUND_DEVICE_INIT structure. **SoundCreateDevice** passes the address to **IoInitializeDpcRequest**.

A driver causes **SoundMidiInDeferred** to be called by calling **IoRequestDPC** from its ISR. The driver should specify NULL for the *Irp* and *Context* parameters to **IoRequestDPC**, because **SoundMidiInDeferred** does not use them.

SoundMixerChangedItem

VOID
SoundMixerChangedItem(
IN OUT PMIXER_INFO *MixerInfo*,
IN OUT PMIXER_DATA_ITEM *MixerItem*
);

The **SoundMixerChangedItem** function is called when an item on a mixer's **ChangedItems** list changes state.

Parameters

MixerInfo
Pointer to a mixer's MIXER_INFO structure.

MixerItem
Pointer to a MIXER_DATA_ITEM structure.

Return Value

None.

Comments

The **SoundMixerChangedItem** function updates the **LastSet** member of the MIXER_DATA_ITEM structure and moves the structure pointer to the head of the **ChangedItems** list in MIXER_INFO. It then traverses the queue pointed to by **NotifyQueue** in MIXER_INFO, calling **IoCompleteRequest** for each IRP in the queue and thereby delivering notification messages to each requesting client.

Drivers calling **SoundMixerChangedItem** must include *mixer.h*.

SoundMixerDispatch

NTSTATUS
SoundMixerDispatch(
IN OUT PLOCAL_DEVICE_INFO *pLDI*,
IN PIRP *pIrp*,
IN PIO_STACK_LOCATION *IrpStack*
);

The **SoundMixerDispatch** function is the IRP control code dispatcher for mixer devices.

Parameters

pLDI
Pointer to a LOCAL_DEVICE_INFO structure.

pIrp
Pointer to an IRP structure.

IrpStack
Pointer to an IO_STACK_LOCATION structure.

Return Value

Returns STATUS_SUCCESS if the operation succeeds. Otherwise returns an NTSTATUS error code.

Comments

Kernel-mode mixer device drivers using *soundlib.lib* place the address of this function in the **DispatchRoutine** member of a SOUND_DEVICE_INIT structure. The function is called by *soundlib.lib*'s main dispatcher, **SoundDispatch**.

The **SoundMixerDispatch** function processes the following IRP control codes:

IRP_MJ_CLEANUP
IRP_MJ_CLOSE
IRP_MJ_CREATE
IRP_MJ_DEVICE_CONTROL
IRP_MJ_WRITE

For IRP_MJ_DEVICE_CONTROL, the function processes the following I/O control codes:

IOCTL_MIX_GET_CONFIGURATION
IOCTL_MIX_GET_CONTROL_DATA
IOCTL_MIX_GET_LINE_DATA
IOCTL_MIX_REQUEST_NOTIFY

SoundMoveCancellableQueue

```
VOID
    SoundMoveCancellableQueue(
      IN OUT PLIST_ENTRY From,
      IN OUT PLIST_ENTRY To
  );
```

The **SoundMoveCancellableQueue** function moves a queue of cancelable IRPs to another queue.

Parameters

From
Pointer to a queue of IRPs to be moved.

To
Pointer to an empty queue into which IRPs are to be moved.

Return Value

None.

Comments

The **SoundMoveCancellableQueue** function initializes the *To* queue by calling **InitializeListHead**.

Generally, drivers using *soundlib.lib* do not need to manipulate IRP queues because DPCs in *soundlib.lib* handle IRP completion. For drivers that do manipulate IRP queues, the **SoundAddIrpToCancellableQ**, **SoundRemoveFromCancellableQ**, **SoundMoveCancellableQueue**, **SoundFreePendingIrps**, and **SoundFreeQ** functions are provided.

SoundNoVolume

```
VOID
    SoundNoVolume(
```

PLOCAL_DEVICE_INFO *pLDI*
);

The **SoundNoVolume** function should be used as the **HwSetVolume** member of the SOUND_DEVICE_INIT structure associated with a device that either has no volume setting hardware or whose volume is controlled by a mixer device.

Parameters

pLDI
Pointer to the device's LOCAL_DEVICE_INFO structure.

Return Value

None.

Comments

The **SoundNoVolume** function contains only a return statement.

SoundOpenDevicesKey

NTSTATUS
SoundOpenDevicesKey(
IN PWSTR *RegistryPathName*,
OUT PHANDLE *DevicesKey*
);

The **SoundOpenDevicesKey** function opens the **Devices** subkey under the specified registry path. If the **Devices** subkey does not exist, it is created.

Parameters

RegistryPathName
Pointer to a string representing the registry path to the device subkey. Use the path name created by either **SoundEnumSubkeys** or **SoundSaveRegistryPath**.

DevicesKey
Address of a location to receive a handle to the **Devices** subkey.

Return Value

Returns STATUS_SUCCESS if the operation succeeds. Otherwise returns an NTSTATUS error code.

Comments

A driver that calls **SoundSaveDeviceName** to save a device name in the registry must remove the created **\Devices** key, along with all device name subkeys, prior to being unloaded. To remove the **\Devices** key, a driver should:

- Call **SoundOpenDevicesKey** to obtain a handle to the **\Devices** key.
- Call **ZwDeleteKey** to delete the key.
- Call **ZwClose** to close the key handle.

SoundPeakMeter

BOOLEAN
SoundPeakMeter(
IN PWAVE_INFO *WaveInfo*,
OUT PLONG *Amplitudes*
);

The **SoundPeakMeter** function returns the peak amplitude value currently contained in the DMA buffer for the specified waveform device.

Parameters

WaveInfo
Pointer to a WAVE_INFO structure.

Amplitudes
Pointer to a longword. Amplitude values are returned in the longword. The lower word contains the left channel value, and the upper word contains the right channel value. Returned values are between 0x0 and 0xFFFF.

Return Value

Always returns TRUE.

Comments

The **SoundPeakMeter** function is typically called from a mixer driver's **HwGetControlData** function. See MIXER_INFO.

Drivers calling **SoundPeakMeter** must include *wave.h*.

SoundRemoveFromCancellableQ

PIRP
SoundRemoveFromCancellableQ(
PLIST_ENTRY *QueueHead*
);

The **SoundRemoveFromCancellableQ** function removes an IRP from the head of the specified IRP queue, makes the IRP noncancelable, and returns its address.

Parameters

QueueHead
Pointer to the head of a queue of IRPs.

Return Value

Address of an IRP structure containing the removed IRP.

Comments

Generally, drivers using *soundlib.lib* do not need to manipulate IRP queues because DPCs in *soundlib.lib* handle IRP completion. For drivers that do manipulate IRP queues, the **SoundAddIrpToCancellableQ**, **SoundRemoveFromCancellableQ**, **SoundMoveCancellableQueue**, **SoundFreePendingIrps**, and **SoundFreeQ** functions are provided.

SoundReportResourceUsage

NTSTATUS
SoundReportResourceUsage(
IN PDEVICE_OBJECT *DeviceObject*,
IN INTERFACE_TYPE *BusType*,
IN ULONG *BusNumber*,
IN PULONG *InterruptNumber* **OPTIONAL,**
IN KINTERRUPT_MODE *InterruptMode*,
IN BOOLEAN *InterruptShareDisposition*,
IN PULONG *DmaChannel* **OPTIONAL,**
IN PULONG *FirstIoPort* **OPTIONAL,**
IN ULONG *IoPortLength*
);

The **SoundReportResourceUsage** function reserves hardware resources for use by the specified device or driver. This function does not allow reservation of resources already reserved

for another device or driver.

Parameters

DeviceObject
Pointer to either a DEVICE_OBJECT or DRIVER_OBJECT structure. Represents the device or driver for which resources will be reserved.

BusType
Type of bus the device is on. INTERFACE_TYPE is defined in *ntddk.h*.

BusNumber
Number of the bus the device is on.

InterruptNumber
Pointer to the interrupt number, or NULL if no interrupt.

InterruptMode
Interrupt mode (**Latched** or **LevelSensitive**). Ignored if *InterruptNumber* is NULL. KINTERRUPT_MODE is defined in *ntddk.h*.

InterruptShareDisposition
TRUE if interrupt can be shared, FALSE otherwise. Ignored if *InterruptNumber* is NULL.

DmaChannel
Pointer to the device's DMA channel, or NULL if there is no DMA channel.

FirstIoPort
Pointer to the device's first I/O port address, or NULL if there are no I/O ports.

IoPortLength
Number of bytes of I/O space the device uses, starting at *FirstIoPort*. Ignored if *FirstIoPort* is NULL.

Return Value

Returns one of the following values.

Value	Definition
STATUS_SUCCESS	Success.
STATUS_DEVICE_CONFIGURATION_ERROR	Resources already assigned.
STATUS_INSUFFICIENT_RESOURCES	Resources unavailable.

Comments

Before attempting to access device hardware, call **SoundReportResourceUsage** to ensure the resources you intend to use are not already assigned to another device.

You can associate resources with either the driver object or with one of the device objects created by **SoundCreateDevice**. If the driver supports multiple cards, you must associate resources with device objects.

Each time you call **SoundReportResourceUsage** for a particular device object or driver object, you override the resource reservation made with any previous call for the same object.

Typically, a driver acquires resources for each device in turn, and then calls **SoundReportResourceUsage** a final time to declare (to the rest of the system) all the resources used by the card instance.

To free reserved resources, call **SoundFreeDevice**.

SoundSaveDeviceName

```
NTSTATUS
    SoundSaveDeviceName(
      IN PWSTR RegistryPathName,
      IN PLOCAL_DEVICE_INFO pLDI
  );
```

The **SoundSaveDeviceName** function writes a device's name and type into the registry.

Parameters

RegistryPathName
Pointer to a string representing the registry path to the device subkey. Use the path name created by either **SoundEnumSubkeys** or **SoundSaveRegistryPath**.

pLDI
Pointer to the device's LOCAL_DEVICE_INFO structure.

Return Value

Returns one of the following values.

Value	Definition
STATUS_SUCCESS	Success.
STATUS_INSUFFICIENT_RESOURCES	Couldn't create a device name.

Comments

Under the specified registry path, the **SoundSaveDeviceName** function locates the **\Devices** subkey. Under this subkey, the function uses the device's name as a value name and uses the device's type as the value to associate with the name. The function creates a device name from the **PrototypeName** member of the device's SOUND_DEVICE_INIT structure. It obtains the device type from the **DeviceType** member of the passed LOCAL_DEVICE_INFO structure.

After the function is called, the registry should contain the entry **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***\Parameters\Device***Number***\De**
where the value of *DeviceName* is the device type.

Drivers that call **SoundSaveDeviceName** must remove the saved device name from the registry prior to being unloaded. Refer to **SoundOpenDevicesKey**.

SoundSaveDeviceVolume

```
VOID
    SoundSaveDeviceVolume(
      PLOCAL_DEVICE_INFO pLDI,
      PWSTR KeyName
    );
```

The **SoundSaveDeviceVolume** function writes both left and right channel volume settings into the registry.

Parameters

pLDI
Pointer to a LOCAL_DEVICE_INFO structure.

KeyName
The name of the registry subkey under which the volume setting is to be stored. The values are retrieved from the **LeftVolumeName** and **RightVolumeName** members of the structure pointed to by the **DeviceInit** member of *pLDI*.

Return Value

None.

Comments

Left and right volume values are obtained from the **volume** member of the LOCAL_DEVICE_INFO structure. The names associated with these volumes are obtained from the SOUND_DEVICE_INIT structure pointed to by the LOCAL_DEVICE_INFO structure.

Call the **SoundSaveDeviceVolume** function when the system shuts down or when the driver is unloaded. Do not use this function if the volume is controlled by a mixer.

SoundSaveRegistryPath

NTSTATUS
SoundSaveRegistryPath(
IN PUNICODE_STRING *RegistryPathName*,
OUT PWSTR **SavedString*
);

The **SoundSaveRegistryPath** function stores the specified registry path name, appends the **\Parameters** subkey to the name string, and returns a pointer to the stored string.

Parameters

RegistryPathName
Pointer to a registry path name. Typically, this is the path name received by the driver's **DriverEntry** function.

SavedString
Address of a location to receive a pointer to the new string.

Comments

Drivers use the returned path name when calling *soundlib.lib* functions that require a registry path name as input.

Prior to being unloaded, the driver must deallocate the buffer pointed to by *SavedString*.

The **SoundSaveRegistryPath** function is typically used by drivers that support only a single hardware device. Drivers supporting multiple devices call **SoundEnumSubkeys**, which appends **\Parameters\Device***Num* to the path name.

SoundSetErrorCode

NTSTATUS
SoundSetErrorCode(
IN PWSTR *RegistryPath*,
IN ULONG *Value*
);

The **SoundSetErrorCode** function assigns a value to the "Configuration Error" value name in the specified registry path.

Parameters

RegistryPath
Pointer to a full path to a registry key.

Value
Value to set. The value is stored in the registry as a REG_DWORD type.

Return Value

Returns STATUS_SUCCESS if the operation succeeds. Otherwise returns an NTSTATUS error code.

SoundSetLineNotify

VOID
SoundSetLineNotify(
PLOCAL_DEVICE_INFO *pLDI*,
PSOUND_LINE_NOTIFY *LineNotify*
);

The **SoundSetLineNotify** function specifies a function to be called when the status of the specified device changes.

Parameters

pLDI
Pointer to the LOCAL_DEVICE_INFO structure of the device with which the notification function is to be associated.

LineNotify
Pointer to a notification function. Type is PSOUND_LINE_NOTIFY.

Return Value

None.

Comments

The **SoundSetLineNotify** function is called by mixer drivers, so they can receive notification of changes to devices associated with mixer lines.

A separate notification function can be registered for each device created with **SoundCreateDevice**, but code in *soundlib.lib* calls notification functions only for waveform devices and MIDI synthesizers.

If a driver has registered a notification function by calling **SoundSetLineNotify**, then *soundlib.lib* calls the function whenever the status of the device changes (that is, the device starts or stops).

Typically, the driver uses the notification function to send commands to mixer hardware, if necessary, and to call **SoundMixerChangedItem**, which queues the change and notifies clients.

Drivers calling **SoundSetLineNotify** must include *mixer.h*.

SoundSetShareAccess

```
NTSTATUS
    SoundSetShareAccess(
      IN OUT PLOCAL_DEVICE_INFO pLDI,
      IN PIO_STACK_LOCATION IrpStack
      );
```

The **SoundSetShareAccess** function sets a specified device access by calling **IoSetShareAccess**.

Parameters

pLDI
Pointer to a LOCAL_DEVICE_INFO structure.

IrpStack
Pointer to an IO_STACK_LOCATION structure that contains the desired access.

Return Value

Returns STATUS_SUCCESS if the requested access is granted. Otherwise returns STATUS_ACCESS_DENIED or STATUS_DEVICE_BUSY.

Comments

This function can only be called when processing an IRP_MJ_CREATE message, because it references message-specific contents of *IrpStack*. Drivers using the dispatch routines supplied by *soundlib.lib* do not need to call this function.

SoundSetVolumeControlId

```
VOID
    SoundSetVolumeControlId(
      PLOCAL_DEVICE_INFO pLDI,
      UCHAR VolumeControlId
      );
```

The **SoundSetVolumeControlId** function assigns the specified volume control ID to the specified device.

Parameters

pLDI
Pointer to the LOCAL_DEVICE_INFO structure of the device to which the volume control ID is to be assigned.

VolumeControlId
A driver-defined volume control ID value.

Return Value

None.

Comments

The **SoundSetVolumeControlId** function stores the specified value in the **VolumeControlID** member of the LOCAL_DEVICE_INFO structure. The value is driver-defined and enables the driver to associate a device's volume settings with mixer settings. The volume ID is passed as input to functions pointed to by the **HwGetControlData**, **HwGetCombinedControlData**, and **HwSetControlData** members of MIXER_INFO.

SoundTestWaveDevice

int
SoundTestWaveDevice(
IN PDEVICE_OBJECT *pDO*
);

The **SoundTestWaveDevice** function initiates a short DMA transfer to determine if the specified waveform device's interrupt number and DMA channel number are set correctly.

Parameters

pDO
Pointer to the waveform device's device object. To obtain a device object, call **SoundCreateDevice**.

Return Value

Returns one of the following values.

Value	Definition
0	Operation succeeded.
1	Interrupt number is invalid.
2	DMA channel number is invalid.

Comments

Use the **SoundTestWaveDevice** function only if no other means for establishing interrupt and DMA channel numbers are available.

This function only works for devices that perform auto-initialize DMA transfers.

Drivers calling **SoundTestWaveDevice** must include *wave.h*.

SoundVolumeNotify

VOID
SoundVolumeNotify(
IN OUT PLOCAL_DEVICE_INFO *pLDI*
);

The **SoundVolumeNotify** function is used by mixer drivers to notify clients when a device's volume changes.

Parameters

pLDI
Pointer to a LOCAL_DEVICE_INFO structure representing the device whose volume has changed.

Return Value

None.

Comments

The function traverses the IRP queue pointed to by the **VolumeQueue** member of the LOCAL_DEVICE_INFO structure, calling **IoCompleteRequest** for each queue entry.

Mixer drivers only call this function if volume is controlled in software. For an example, see *sndsys.sys*, the kernel-mode driver for the Windows Sound System synthesizer. Sources for this driver are in *\src\mmedia\sndsys\driver*.

SoundWaveDeferred

VOID
SoundWaveDeferred(
PKDPC *pDpc*,
PDEVICE_OBJECT *pDeviceObject*,
PIRP *pIrp*,
PVOID *Context*
);

The **SoundWaveDeferred** function is the DPC function that is provided by *soundlib.lib* for waveform input and output devices.

Parameters

pDpc
Pointer to a DPC object.

pDeviceObject
Pointer to a device object.

pIrpDeferred
Pointer to an IRP. (Not used by **SoundWaveDeferred**. See **Comments** section below.)

Context
Pointer to context information. (Not used by **SoundWaveDeferred**. See **Comments** section below.)

Return Value

None.

Comments

Drivers using *soundlib.lib* should specify this routine's address as the **DeferredRoutine** member of each waveform device's SOUND_DEVICE_INIT structure. **SoundCreateDevice** passes to the address to **IoInitializeDpcRequest**.

A driver causes **SoundWaveDeferred** to be called by calling **IoRequestDPC** from its ISR. The driver should specify NULL for the *Irp* and *Context* parameters to **IoRequestDPC**, because **SoundWaveDeferred** does not use them.

SoundWaveDispatch

NTSTATUS
SoundWaveDispatch(

IN OUT PLOCAL_DEVICE_INFO *pLDI*,
IN PIRP *pIrp*,
IN PIO_STACK_LOCATION *IrpStack*
);

The **SoundWaveDispatch** function is the IRP control code dispatcher for waveform devices.

Parameters

pLDI
Pointer to a LOCAL_DEVICE_INFO structure.

pIrp
Pointer to an IRP structure.

IrpStack
Pointer to an IO_STACK_LOCATION structure.

Return Value

Returns STATUS_SUCCESS if the operation succeeds. Otherwise returns an NTSTATUS error code.

Comments

Kernel-mode waveform device drivers using *soundlib.lib* place the address of the **SoundWaveDispatch** function in the **DispatchRoutine** member of a SOUND_DEVICE_INIT structure. The function is called by *soundlib.lib*'s main dispatcher, **SoundDispatch**.

The function processes the following IRP control codes:

IRP_MJ_CLEANUP
IRP_MJ_CLOSE
IRP_MJ_CREATE
IRP_MJ_DEVICE_CONTROL
IRP_MJ_READ
IRP_MJ_WRITE

For IRP_MJ_DEVICE_CONTROL, the function processes the following I/O control codes:

IOCTL_SOUND_GET_CHANGED_VOLUME
IOCTL_WAVE_GET_CAPABILITIES
IOCTL_WAVE_GET_PITCH
IOCTL_WAVE_GET_PLAYBACK_RATE
IOCTL_WAVE_GET_POSITION
IOCTL_WAVE_GET_POSITION
IOCTL_WAVE_GET_STATE
IOCTL_WAVE_GET_VOLUME
IOCTL_WAVE_QUERY_FORMAT
IOCTL_WAVE_SET_FORMAT
IOCTL_WAVE_SET_LOW_PRIORITY
IOCTL_WAVE_SET_PITCH
IOCTL_WAVE_SET_PLAYBACK_RATE
IOCTL_WAVE_SET_STATE

SoundWriteRegistryDWORD

NTSTATUS
SoundWriteRegistryDWORD(
IN PCWSTR *RegistryPath*,
IN PCWSTR *ValueName*,

IN ULONG *Value*
);

The **SoundWriteRegistryDWORD** function assigns the specified value to the specified value name in the registry, at the specified registry key.

Parameters

RegistryPath
Pointer to full path to a registry key.

ValueName
Pointer to a value name string.

Value
Value to assign to *ValueName*. The value is stored as a REG_DWORD type.

Return Value

Returns STATUS_SUCCESS if the operation succeeds. Otherwise returns an NTSTATUS error code.

SynthCleanup

VOID
SynthCleanup(
IN PGLOBAL_SYNTH_INFO *pGDI*
);

The **SynthCleanup** function deallocates synthesizer resources.

Parameters

pGDI
Pointer to a GLOBAL_SYNTH_INFO structure.

Return Value

None.

Comments

The **SynthCleanup** function calls **MmUnmapIoSpace**.

Drivers calling **SynthCleanup** must include *synthdrv.h*.

SynthInit

NTSTATUS
SynthInit(
IN PDRIVER_OBJECT *pDriverObject*,
IN PWSTR *RegistryPathName*,
IN PGLOBAL_SYNTH_INFO *pGDI*,
IN ULONG *SynthPort*,
IN BOOLEAN *InterruptConnected*,
IN INTERFACE_TYPE *BusType*,
IN ULONG *BusNumber*,
IN PMIXER_DATA_ITEM *MidiOutItem*,
IN UCHAR *VolumeControlId*,
IN BOOLEAN *Multiple*,
IN SOUND_DISPATCH_ROUTINE *DevCapsRoutine*
);

The **SynthInit** function creates a device object for a MIDI synthesizer (AdLib or OPL3) and performs additional initialization tasks.

Parameters

pDriverObject
Address of a location to receive a pointer to a DEVICE_OBJECT structure, if the call succeeds.

RegistryPathName
Pointer to a string representing the registry path to the device subkey. Use the path name created by either **SoundEnumSubkeys** or **SoundSaveRegistryPath**.

pGDI
Pointer to a GLOBAL_SYNTH_INFO structure.

SynthPort
Port address.

InterruptConnected
TRUE if interrupt is connected, FALSE otherwise.

BusType
Bus type. INTERFACE_TYPE is defined in *ntddk.h*.

BusNumber
Bus number.

MidiOutItem
Not used. Should be set to NULL.

VolumeControlId
A driver-defined volume control ID value.

Multiple
TRUE if indexed device names are allowed, FALSE otherwise.

DevCapsRoutine
Pointer to a driver-defined function that returns device capabilities. The function type is SOUND_DISPATCH_ROUTINE. To read about functions that return device capabilities, refer to the description of the **DevCapsRoutine** member of the SOUND_DEVICE_INIT structure.

Return Value

Returns STATUS_SUCCESS if the operation succeeds. Otherwise returns an NTSTATUS error code.

Comments

You do not create a SOUND_DEVICE_INIT structure for a synthesizer device, because *soundlib.lib* creates one internally. It also provides an internal dispatch routine and exclusion routine.

The **SynthInit** function performs the following operations, in order:

1. Calls **SoundCreateDevice** to create a device object and a LOCAL_DEVICE_INFO structure.
2. Sets default volume values for left and right channels in the **Volume** member of the device's LOCAL_DEVICE_INFO structure.
3. Calls **SoundReportResourceUsage** to obtain system resources.
4. Calls **SoundMapPortAddress** to map port addresses.
5. Attempts to reference the synthesizer hardware to verify that it exists at the specified address.
6. Calls **SoundSaveDeviceName** to write the device name in the registry.
7. Stores the specified volume control ID in the device's LOCAL_DEVICE_INFO structure.
8. Determines if the synthesizer chip is OPL3-compatible.
9. Initializes the synthesizer hardware to silence.

Setting *Multiple* to FALSE causes **SoundCreateDevice** to be called with the SOUND_CREATION_NO_NAME_RANGE flag set. Use FALSE if the port address is 0x388.

Synthesizer interrupts are not handled by *soundlib.lib*. If the device supports a hardware interrupt, the driver must provide ISR code to handle it.

Drivers calling **SynthInit** must include *synthdrv.h*.

Structures and Types, *soundlib.lib*

This section describes the structures and types available to kernel-mode audio drivers using *soundlib.lib*. The structures and types are defined in *soundlib.h*, *devices.h*, *wave.h*, *midi.h*, *mixer.h*, and *synthdrv.h*.

GLOBAL_SYNTH_INFO

```
typedef struct _GLOBAL_SYNTH_INFO {
    ULONG           Key;
#define SYNTH_KEY       (*(ULONG *)"Syn ")
    INTERFACE_TYPE  BusType;
    ULONG           BusNumber;
    KMUTEX          MidiMutex;
    ULONG           MemType;
    PDEVICE_OBJECT  DeviceObject;
    PDRIVER_OBJECT  DriverObject;
    SOUND_DISPATCH_ROUTINE  *DevCapsRoutine;
    UCHAR           DeviceInUse;
    volatile BOOLEAN  InterruptFired;     // Interrupt fired?
    BOOLEAN         IsOpl3;               // It's an OPL3
    SYNTH_HARDWARE  Hw;                   // Hardware specific stuff
} GLOBAL_SYNTH_INFO, *PGLOBAL_SYNTH_INFO;
```

The GLOBAL_SYNTH_INFO structure contains context information for a mixer device.

Members

Key
Internal only, for debugging. Should be "Syn ".

BusType
Bus type. INTERFACE_TYPE is defined in *ntddk.h*.

BusNumber
Bus number.

MidiMutex
Internal only. Mutex object used by *soundlib.lib*'s exclusion routine.

MemType
Memory type returned by **SoundMapPortAddress**.

DeviceObject
Pointer to device object returned by **SoundCreateDevice**.

DriverObject
Pointer to driver object received by **DriverEntry**.

DevCapsRoutine
Pointer to a driver-defined function that returns device capabilities. The function type is SOUND_DISPATCH_ROUTINE.

DeviceInUse
Internal only. Indicates device status.

InterruptFired
Set by driver if interrupt fired. Not set by *soundlib.lib*.

IsOpl3
Set if synthesizer is OPL3-compatible.

Hw
Hardware information. Member type is SYNTH_HARDWARE.

Comments

Allocate a GLOBAL_SYNTH_INFO structure from the nonpaged memory pool by calling **ExAllocatePool**, then zero it by calling **RtlZeroMemory**. The structure is initialized by code in **SynthInit**.

LOCAL_DEVICE_INFO

```
typedef struct _LOCAL_DEVICE_INFO {
    ULONG           Key;
#define LDI_WAVE_IN_KEY     (*(ULONG *)"LDWi")
#define LDI_WAVE_OUT_KEY    (*(ULONG *)"LDWo")
#define LDI_MIDI_IN_KEY     (*(ULONG *)"LDMi")
#define LDI_MIDI_OUT_KEY    (*(ULONG *)"LDMo")
#define LDI_AUX_KEY         (*(ULONG *)"LDAx")
#define LDI_MIX_KEY         (*(ULONG *)"LDMx")
    PVOID           pGlobalInfo;
    UCHAR           DeviceType;
    UCHAR           DeviceNumber;
    UCHAR           DeviceIndex;
    UCHAR           CreationFlags;
#define SOUND_CREATION_NO_NAME_RANGE ((UCHAR)0x01)
#define SOUND_CREATION_NO_VOLUME     ((UCHAR)0x02)
    BOOLEAN         PreventVolumeSetting;
    UCHAR           VolumeControlId;
    PSOUND_LINE_NOTIFY   LineNotify;
#ifndef SOUNDLIB_NO_OLD_VOLUME
    WAVE_DD_VOLUME  Volume;
#endif
#ifdef VOLUME_NOTIFY
    LIST_ENTRY      VolumeQueue;
    struct _LOCAL_DEVICE_INFO * MixerDevice;
#endif
#ifdef MASTERVOLUME
    BOOLEAN         MasterVolume;
#endif
    BOOLEAN         VolumeChanged;
    PVOID           DeviceSpecificData;
    PVOID           HwContext;
    ULONG           State;
    PCSOUND_DEVICE_INIT   DeviceInit;
} LOCAL_DEVICE_INFO, *PLOCAL_DEVICE_INFO;
```

Within *soundlib.lib*, the LOCAL_DEVICE_INFO structure is used as the device extension for each device object created by **IoCreateDevice**. Device objects and device extensions are described in the *Kernel-Mode Drivers Design Guide*. The LOCAL_DEVICE_INFO structure is defined in *devices.h*. One LOCAL_DEVICE_INFO structure exists for each device created by **SoundCreateDevice**.

Members

Key
Internal only. Used for debugging. The value is copied from the **Key** member of SOUND_DEVICE_INIT.

pGlobalInfo
Pointer to a driver-defined structure containing device object-specific data. Specified by the *pGDI* parameter of **SoundCreateDevice**.

DeviceType
Copied from the **DeviceType** member of the SOUND_DEVICE_INIT structure.

DeviceNumber
Contains number to append to the end of the device name. Set to 0xFF within **SoundCreateDevice** if the SOUND_CREATION_NO_NAME_RANGE flag is specified.

DeviceIndex
For use by the driver. Contains the *i* parameter specified with **SoundCreateDevice**.

CreationFlags
Contains flags passed as *CreationFlags* parameter of **SoundCreateDevice**.

PreventVolumeSetting
Internal only. Used to prevent volume being shared for devices opened without shared write access. Set by **SoundSetShareAccess**.

VolumeControlId
Set by **SoundSetVolumeControlId**.

LineNotify
Address of a driver-supplied function called when a line status changes. The function type is PSOUND_LINE_NOTIFY. Set by **SoundSetLineNotify**.

Volume
Contains the device's volume setting.

VolumeQueue
IRP queue for SOUND_IOCTL_GET_CHANGED_VOLUME requests not completed.

MixerDevice
If the device object represents a mixer device, this member points to an additional LOCAL_DEVICE_INFO structure for the mixer device. *This member must be set by the driver's mixer initialization code. It is not set by* soundlib.lib.

MasterVolume
Set if this device is the master volume control device.

VolumeChanged
Internal only. If there is no mixer device, *soundlib.lib* sets this after a volume change, to indicate that volume settings need to be updated in the registry before system shutdown.

DeviceSpecificData
Contains the value passed as the *DeviceSpecificData* parameter of **SoundCreateDevice**.

HwContext
Contains the value passed as the *pHw* parameter of **SoundCreateDevice**.

State
Internal only Used by *soundlib.lib* to store the current device state.

DeviceInit
Contains the value passed as the *DeviceInit* parameter to **SoundCreateDevice**.

Comments

Drivers do not allocate LOCAL_DEVICE_INFO structures locally. When a driver calls **SoundCreateDevice**, a pointer to a DEVICE_OBJECT structure is returned. The structure's **DeviceExtension** member is used as the device's LOCAL_DEVICE_INFO structure.

MIDI_INFO

```
typedef struct _MIDI_INFO
{
    ULONG           Key;
#define MIDI_INFO_KEY       (*(ULONG *)"Midi")
        KSPIN_LOCK      DeviceSpinLock;
    #if DBG
        BOOLEAN         LockHeld;
    #endif
        LARGE_INTEGER   RefTime;
        LIST_ENTRY      QueueHead;
```

```
        PVOID           HwContext;
        PMIDI_INTERFACE_ROUTINE HwStartMidiIn, HwStopMidiIn;
        BOOLEAN      (* HwMidiRead)(struct _MIDI_INFO *, PUCHAR);
        VOID         (* HwMidiOut)(struct _MIDI_INFO *, PUCHAR, int);
        BOOLEAN         fMidiInStarted;
        UCHAR           InputPosition;
        UCHAR           InputBytes;
        UCHAR           MidiInputByte[64];
} MIDI_INFO, *PMIDI_INFO;
```

The MIDI_INFO structure contains context information for an external MIDI device.

Members

Key
Internal only, for debugging. Should be "Midi".

DeviceSpinLock
Internal only. Used for DPC synchronization.

LockHeld
Internal only. Used for debugging.

RefTime
Used by *soundlib.lib* to store the start time of an I/O operation, as reference for time stamps.

QueueHead
Internal only. Points to a buffer queue for MIDI input requests.

HwContext
Pointer to a driver-defined structure containing device-specific hardware information. Typically used by functions pointed to by the **HwStartMidiIn**, **HwStopMidiIn**, **HwMidiRead**, and **HwMidiOut** members.

HwStartMidiIn
Pointer to a driver-supplied function that programs the MIDI hardware to start recording. The function type is MIDI_INTERFACE_ROUTINE.

The function is called when **SoundMIDIDispatch** receives a IOCTL_MIDI_SET_STATE command. See *src**mmedia**soundlib**midi.c*.

HwStopMidiIn
Pointer to a driver-supplied function that programs the MIDI hardware to stop recording. The function type is MIDI_INTERFACE_ROUTINE.

The function is called when **SoundMIDIDispatch** receives a IRP_MJ_CLEANUP command. For more information, see *src**mmedia**soundlib**midi.c*.

HwMidiRead
Pointer to a driver-supplied function that reads one input byte. This operation might only consist of fetching the next byte from a buffer that was filled by an ISR. (An example is MPU401 support in *sndblst.sys*.) The function type is:

```
BOOLEAN (* HwMidiRead)(struct _MIDI_INFO *, PUCHAR)
```

The _MIDI_INFO* parameter points to a MIDI_INFO structure and the PUCHAR parameter receives the read byte. The function returns TRUE if a byte was read, and FALSE otherwise.

The function is called when **SoundMIDIDispatch** receives a IRP_MJ_READ command. For more information, see *src**mmedia**soundlib**midi.c*.

This function executes at an IRQL of DISPATCH_LEVEL, so it cannot be pageable and it cannot reference pageable code or data. Also, the only synchronization method it can use is calling **KeStallExecutionProcessor**.

HwMidiOut
Pointer to a driver-supplied function that commands the MIDI hardware to write a string of bytes. Function type is:

```
VOID (* HwMidiOut)(struct _MIDI_INFO *, PUCHAR, int)
```

The first parameter points to a MIDI_INFO structure, the second parameter points to a mapped buffer of bytes, and the third parameter contains the buffer size.

The function is called when **SoundMIDIDispatch** receives a IOCTL_MIDI_PLAY command. For more information, see *\src\mmedia\soundlib\midi.c*.

fMidiInStarted
Internal only. Indicates a MIDI input operation is in progress.

InputPosition
Internal only. Pointer to an internal input buffer.

InputBytes
Internal only. Count of bytes in internal input buffer.

MidiInputByte
Internal only. Pointer to internal input buffer.

Comments

A single MIDI_INFO structure can be used to support simultaneous MIDI input and output. MIDI_INFO is defined in *midi.h*.

Allocate a MIDI_INFO structure from the nonpaged memory pool by calling **ExAllocatePool**, then zero it by calling **RtlZeroMemory**. To initialize a MIDI_INFO structure, call **SoundInitMidiIn**.

To create a MIDI device object, call **SoundCreateDevice** and specify a MIDI_INFO structure pointer for the *DeviceSpecificData* parameter.

MIDI_INTERFACE_ROUTINE

```
typedef BOOLEAN MIDI_INTERFACE_ROUTINE(struct _MIDI_INFO *);
```

MIDI_INTERFACE_ROUTINE is a type definition for functions that send commands to MIDI hardware.

Parameters

_MIDI_INFO*
Pointer to a MIDI_INFO structure.

Comments

Drivers using *soundlib.lib* define functions modeled on this type and place their addresses in the **HwStartMidiIn** and **HwStopMidiIn** members of a WAVE_INFO structure.

MIXER_DATA_ITEM

```
typedef struct _MIXER_DATA_ITEM {
    LIST_ENTRY    Entry;
    LARGE_INTEGER LastSet;
    USHORT        Message;
    USHORT        Id;
} MIXER_DATA_ITEM, *PMIXER_DATA_ITEM;
```

The MIXER_DATA_ITEM structure, defined in *mixer.h*, is used to represent a data item for either a mixer line or a mixer control. The structure is initialized by calling **SoundInitDataItem** and is modified by **SoundMixerChangedItem**.

Members

Entry
Internal only. Contains FLINK and BLINK list pointers. Set by **SoundInitDataItem**.

LastSet
Internal only. Contains relative time of last change. Set by **SoundInitDataItem** and **SoundMixerChangedItem**.

Message

Internal only. Set by **SoundInitDataItem**. Contains type of notification message to send to client. Can be either MM_MIXM_LINE_CHANGE or MM_MIXM_CONTROL_CHANGE.

Id

Internal only. Set by **SoundInitDataItem**. Contains either a line ID or a control ID.

Comments

Drivers allocate one of these structures for each line and control that can change. The structure must be globally allocated, because *soundlib.lib* does not copy it. Instead, *soundlib.lib* modifies the structure's **Entry** element to determine the structure's location within a linked list.

MIXER_INFO

```
typedef struct _MIXER_INFO {
    ULONG    Key;
#define MIX_INFO_KEY       (*(ULONG *)"Mix")
    UCHAR         NumberOfLines;
    UCHAR         NumberOfControls;
    LARGE_INTEGER CurrentLogicalTime;
    LIST_ENTRY    NotifyQueue;
    LIST_ENTRY    ChangedItems;
    PMIXER_DD_GET_SET_DATA HwGetLineData;
    PMIXER_DD_GET_SET_DATA HwGetControlData;
    PMIXER_DD_GET_SET_DATA HwGetCombinedControlData;
    PMIXER_DD_GET_SET_DATA HwSetControlData;
} MIXER_INFO, *PMIXER_INFO;
```

The MIXER_INFO structure contains context information for a mixer device.

Members

Key

Internal only, for debugging. Should be "Mix".

NumberOfLines

Number of mixer lines.

NumberOfControls

Number of mixer controls.

CurrentLogicalTime

Internal only. Incremented each time an item is added to the **ChangedItems** queue. Used as a reference for determining the relative age of changed items.

NotifyQueue

Internal only. Pointer to an IRP queue of change notification targets. An IRP is added each time *soundlib.lib* receives an IOCTL_MIX_REQUEST_NOTIFY message.

ChangedItems

Internal only. Pointer to a list of mixer items of type MIXER_DATA_ITEM. Modified by **SoundMixerChangedItem**.

HwGetLineData

Pointer to a driver-supplied function that **SoundMixerDispatch** calls when it receives an IOCTL_MIX_GET_LINE_DATA message. The function type is PMIXER_DD_GET_SET_DATA, where the *data* parameter is a pointer to the **fdwLine** member of a MIXERLINE structure, and the *length* parameter is the member's size. The function sets one or more of the following flags in the address pointed to by *data*.

Flag	Definition
MIXERLINE_LINEF_ACTIVE	Indicates that the line is active. Used mainly for waveform devices so the application can determine when to poll the peak meter.
MIXERLINE_LINEF_DISCONNECTED	Indicates that the line is permanently

	unavailable. Useful for disabling capabilities available only on some versions of the hardware.
MIXERLINE_LINEF_SOURCE	Indicates this is a source line, not a destination line.

These flags are defined in *mmsystem.h*. The MIXERLINE structure is described in the Win32 SDK.

HwGetControlData
Pointer to a driver-supplied function that **SoundMixerDispatch** calls when it receives an IOCTL_MIX_GET_CONTROL_DATA message. The function type is PMIXER_DD_GET_SET_DATA, where the *length* parameter is the size of the return buffer and the *data* parameter is the return buffer's address. This address is the **paDetails** member of a MIXERCONTROLDETAILS structure. See the description of MIXERCONTROLDETAILS in the Win32 SDK for information on how the buffer should be filled.

HwGetCombinedControlData
Pointer to a driver-supplied function that returns the current values for the volume controls. If the volume is controlled by mixer hardware, then always return 0xFFFF for each control. Otherwise, if the master volume is supported in hardware, return the current value of the control. If the master volume is simulated, return the combined volume and master volume as a volume value**.**

The function type is PMIXER_DD_GET_SET_DATA, where the *data* parameter is a pointer to a WAVE_DD_VOLUME structure and the *length* parameter is the structure's size.

HwSetControlData
Pointer to a driver-supplied function that **SoundMixerDispatch** calls when it receives an IRP_MJ_WRITE message to set control items. The *soundlib.lib* dispatchers for waveform, MIDI synthesizer, and auxiliary audio devices also call this function to set volume levels. Function type is PMIXER_DD_GET_SET_DATA, where the *length* parameter is the size of the data buffer and the *data* parameter is the data buffer's address. This address is the **paDetails** member of a MIXERCONTROLDETAILS structure. Refer to the description of MIXERCONTROLDETAILS in the Win32 SDK to understand the buffer contents. The function should call **SoundMixerChangedItem** if the value of a control changes.

Comments

MIXER_INFO is defined in *mixer.h*.

Allocate a MIXER_INFO structure from the nonpaged memory pool by calling **ExAllocatePool**. To zero and initialize a MIXER_INFO structure, call **SoundInitMixerInfo**.

To create a mixer device object, call **SoundCreateDevice** and specify a MIXER_INFO structure pointer for the *DeviceSpecificData* parameter. Then assign the address of the mixer device's LOCAL_DEVICE_INFO structure to the **MixerDevice** member of every *other* device's LOCAL_DEVICE_INFO structure.

PMIXER_DD_GET_SET_DATA

```
typedef NTSTATUS (* PMIXER_DD_GET_SET_DATA)(struct _MIXER_INFO * MixerInfo, ULONG I
```

PMIXER_DD_GET_SET_DATA is a type definition for functions that set or retrieve information for mixer lines and controls.

Parameters

MixerInfo
Pointer to a MIXER_INFO structure.

ID
A line ID or a control ID. For **HwGetLineData**, the value represents a line ID. For **HwGetControlData**, **HwGetCombinedControlData**, and **HwSetControlData**, it is a control ID.

Length

Length of a data buffer.

Data
Pointer to a data buffer.

Comments

Drivers using *soundlib.lib* define functions modeled on this type and place their addresses in the **HwGetLineData**, **HwGetControlData**, **HwGetCombinedControlData**, and **HwSetControlData** members of a MIXER_INFO structure.

PSOUND_LINE_NOTIFY

```
typedef VOID (* PSOUND_LINE_NOTIFY)(struct _LOCAL_DEVICE_INFO *, UCHAR);
```

PSOUND_LINE_NOTIFY is the type definition for a function that is called when a line status changes.

Parameters

_LOCAL_DEVICE_INFO*
Type for a pointer to a LOCAL_DEVICE_INFO structure.

UCHAR
Type for a line notification code. The following codes are passed to the notification routine by *soundlib.lib*.

Device Type	Notification Code
For waveform devices:	SOUND_LINE_NOTIFY_WAVE SOUND_LINE_NOTIFY_VOICE
For MIDI synthesizers:	Always 0

Comments

Drivers using *soundlib.lib* define a function modeled on this type and call **SoundSetLineNotify** to place its address in the **LineNotify** member of a LOCAL_DEVICE_INFO structure.

SOUND_DEVICE_INIT

```
typedef struct {
          PCWSTR LeftVolumeName, RightVolumeName;
          ULONG DefaultVolume;
          ULONG Type;
          ULONG DeviceType;
          char  Key[4];
          PCWSTR PrototypeName;
          PIO_DPC_ROUTINE DeferredRoutine;
          SOUND_EXCLUDE_ROUTINE *ExclusionRoutine;
          SOUND_DISPATCH_ROUTINE *DispatchRoutine;
          SOUND_DISPATCH_ROUTINE *DevCapsRoutine;
          SOUND_HW_SET_VOLUME_ROUTINE *HwSetVolume;
          ULONG IoMethod;
      } SOUND_DEVICE_INIT;
```

The SOUND_DEVICE_INIT structure associates driver dispatch routines with a driver object. A SOUND_DEVICE_INIT structure must be defined for each logical input or output device. The structure's definition is in *devices.h*.

LeftVolumeName
Registry key value name used when storing the left channel volume in the registry. Used with **SoundSaveDeviceVolume**.

RightVolumeName
Registry key value name used when storing the right channel volume in the registry. Used with

SoundSaveDeviceVolume.

DefaultVolume

Initial volume setting to use during installation. Required for devices with mixers.

Type

Type of device. **SoundCreateDevice** passes this value to **IoCreateDevice**. The following values, defined in *ntddk.h*, should be used.

Value	Definition
FILE_DEVICE_WAVE_IN	For waveform input
FILE_DEVICE_WAVE_OUT	For wave output
FILE_DEVICE_MIDI_IN	For MIDI input
FILE_DEVICE_MIDI_OUT	For MIDI output
FILE_DEVICE_SOUND	For all other audio devices

DeviceType

Type of device, used within *soundlib.lib* and *drvlib.lib*. The following values, defined in *soundcfg.h*, are accepted.

Value	Definition
WAVE_IN	Waveform input device
WAVE_OUT	Waveform output device
MIDI_IN	MIDI input device
MIDI_OUT	MIDI output device
AUX_DEVICE	Auxiliary audio device
MIXER_DEVICE	Mixer device
SYNTH_DEVICE	MIDI Synthesizer device (adlib or opl3)

Key

For debugging purposes. Code in *soundlib.lib* copies this four-character string value into the device's LOCAL_DEVICE_INFO structure.

PrototypeName

Prototype to use for creating a device object name. Unless the SOUND_CREATION_NO_NAME_RANGE flag is specified as a **SoundCreateDevice** parameter, **SoundCreateDevice** appends a sequential number, starting with zero, to this name.

If you are using *mmdrv.dll* as your user-mode driver, then you must use the prototype name that *mmdrv.dll* recognizes for the device. The names recognized by *mmdrv.dll* are predefined and their string IDs can be referenced using the following names.

Name	Where Defined
DD_AUX_DEVICE_NAME_U	*ntddaux.h*
DD_MIDI_IN_DEVICE_NAME_U DD_MIDI_OUT_DEVICE_NAME_U	*ntddmidi.h*
DD_MIX_DEVICE_NAME_U	*ntddmix.h*
DD_WAVE_IN_DEVICE_NAME_U DD_WAVE_OUT_DEVICE_NAME_U	*ntddwave.h*

If you are using a customized user-mode driver, you cannot use the predefined names. For example, in the SOUND_DEVICE_INIT structures for the kernel-mode driver *sndblst.sys*, predefined names are used for MIDI devices but not for waveform, auxiliary, or mixer devices. The result is that *mmdrv.dll* handles user-mode MIDI operations, and *sndblst.dll* handles all others.

DeferredRoutine

Pointer to a deferred procedure call (DPC) routine, which **SoundCreateDevice** passes to **IoInitializeDpcRequest**.

If the device object does not support interrupts, this member must be NULL. For drivers using *soundlib.lib*, specify one of the following DPC routines.

Device Type	DPC Routine
Waveform input and output devices	**SoundWaveDeferred**
MIDI input devices	**SoundMidiInDeferred**
MIDI output, auxiliary audio, and mixers	NULL

ExclusionRoutine
Pointer to a mutual exclusion function, called from within *soundlib.lib* when it is necessary to acquire or release a mutex for the device. To understand under what circumstances this function is called, see the *soundlib.lib* source code, included with this DDK. The function type is SOUND_EXCLUDE_ROUTINE.

DispatchRoutine
Pointer to a function that serves as a dispatcher for IRP function codes received by the driver. Functions supplied by *soundlib.lib* are as follows:

Dispatcher	Purpose
SoundAuxDispatch	Dispatcher for auxiliary audio devices
SoundMidiDispatch	Dispatcher for MIDI input and output devices
SoundMixerDispatch	Dispatcher for mixer devices
SoundWaveDispatch	Dispatcher for waveform input and output devices

The function type is SOUND_DISPATCH_ROUTINE. The specified function is called by **SoundDispatch**.

DevCapsRoutine
Pointer to a driver-defined function that returns device capabilities.

The function type is SOUND_DISPATCH_ROUTINE. The specified function is called by the dispatcher pointed to by the **DispatchRoutine** member, when the dispatcher receives IRP_MJ_DEVICE_CONTROL with an accompanying request for device capabilities. Capabilities for waveform, MIDI, and auxiliary devices are written into the IRP at **Irp->AssociatedIrp.SystemBuffer**, in the form of either a WAVEINCAPS, WAVEOUTCAPS, MIDIINCAPS, MIDIOUTCAPS, or AUXCAPS capabilities structure. (These structures are defined in *mmsystem.h* and described in the Win32 SDK.)

Note: When filling in the **szPname** member of the capabilities structure, remember the following:

- If your user-mode driver is *mmdrv.dll*, you must call **InternalLoadString** to translate string IDs into strings, and return the strings in the **szPname** member.
- If your user-mode driver makes use of *drvlib.lib*, just return the string IDs in the **szPname** member. Code in *drvlib.lib* calls **InternalLoadString**.

For mixer devices only, the following rules apply:

- Capabilities are written into the IRP at **Irp->AssociatedIrp.SystemBuffer**, in the form of a MIXER_DD_CONFIGURATION_DATA structure (defined in *ntddmix.h*). Code in *drvlib.lib* calls the capabilities function only when the device is being initialized. It stores the MIXER_DD_CONFIGURATION_DATA structure contents and returns them to a client, in a MIXERCAPS structure, when requested.
- The capabilities function is called twice (and only twice). The first time, it must only return the size of the capabilities information. The second time it is called, the function returns the capabilities information as a MIXER_DD_CONFIGURATION_DATA structure and a set of associated structures.

As an aid to understanding these special rules, see the mixer capabilities function, **SoundMixerDumpConfiguration**, provided in *sndblst.sys*, in *src\mmedia\sndblst\driver\mixer.c*.

HwSetVolume
Pointer to a driver-supplied function that sets the volume for the device. The function type is SOUND_HW_SET_VOLUME_ROUTINE.

The specified function is called by the dispatcher pointed to by the **DispatchRoutine** member,

when the dispatcher receives IRP_MJ_DEVICE_CONTROL with an accompanying request to set the volume.

For devices without volume setting capabilities, use the **SoundNoVolume** function. Also use **SoundNoVolume** for devices that include mixer hardware, because drivers for these devices include a MIXER_INFO structure, and volume is controlled by a routine pointed to by that structure's **HwSetControlData** member.

IoMethod

Specifies whether the Windows NT I/O Manager should use direct I/O or buffered I/O for data transfers. Audio drivers should specify this value as shown in the following table.

Device Type	I/O Method
Auxiliary	DO_BUFFERED_IO
MIDI input	DO_DIRECT_IO
MIDI output	DO_DIRECT_IO
Mixer	DO_BUFFERED_IO
Wave input	DO_DIRECT_IO
Wave output	DO_DIRECT_IO

For a discussion of direct I/O and buffer I/O methods, refer to the *Kernel-Mode Drivers Design Guide*.

Comments

The SOUND_DEVICE_INIT structure's address is passed to **SoundCreateDevice**. The structure must not be freed and must be nonpaged, because **SoundCreateDevice** does not copy it.

You must initialize all structure members before calling **SoundCreateDevice**. The **LeftVolumeName**, **RightVolumeName**, and **DefaultVolume** members can be initialized to NULL, NULL, and 0, respectively.

SOUND_DISPATCH_ROUTINE

```
typedef NTSTATUS SOUND_DISPATCH_ROUTINE(struct _LOCAL_DEVICE_INFO *, PIRP, PIO_STACK
```

SOUND_DISPATCH_ROUTINE is the type definition of dispatch routines for IRP function codes.

Parameters

_LOCAL_DEVICE_INFO*

Type for a pointer to a LOCAL_DEVICE_INFO structure.

PIRP

Type for a pointer to an IRP structure.

PIO_STACK_LOCATION

Type for a pointer to an IO_STACK_LOCATION structure.

Return Value

Returns STATUS_SUCCESS if the operation succeeds. Otherwise returns an NTSTATUS error code.

Comments

Drivers using *soundlib.lib* place the address of a function modeled on this type in the **DispatchRoutine** and **DevCapsRoutine** members of a SOUND_DEVICE_INIT structure.

SOUND_DMA_BUFFER

```
typedef struct {
    PADAPTER_OBJECT      AdapterObject[2];
    ULONG                BufferSize;
    PVOID                VirtualAddress;
    PHYSICAL_ADDRESS     LogicalAddress;
```

```
    PMDL                Mdl;
} SOUND_DMA_BUFFER, *PSOUND_DMA_BUFFER;
```

The SOUND_DMA_BUFFER structure describes a DMA buffer. It is used as input to the **SoundGetCommonBuffer** and **SoundFreeCommonBuffer** functions.

Members

AdapterObject[2]
Array of pointers to adapter objects. (Only the first array element is used.)

BufferSize
Size of buffer, in bytes.

VirtualAddress
Virtual address of buffer.

LogicalAddress
Logical address of buffer.

Mdl
Pointer to a memory descriptor list (MDL).

SOUND_DOUBLE_BUFFER

```
typedef struct {
    enum {LowerHalf = 0,
          UpperHalf}
                        NextHalf;
    ULONG               BufferSize;
    PUCHAR              Buf;
    ULONG               StartOfData;
    ULONG               nBytes;
    UCHAR               Pad;
} SOUND_DOUBLE_BUFFER, *PSOUND_DOUBLE_BUFFER;
```

The SOUND_DOUBLE_BUFFER structure is an internal structure that describes the usage of a DMA buffer of type SOUND_DMA_BUFFER.

Members

NextHalf
Indicates which half of the buffer is to be used next.

BufferSize
Actual amount of DMA buffer in use. Obtained by calling **SoundGetDMABufferSize**.

Buf
Pointer to a buffer specified by a SOUND_DMA_BUFFER structure.

StartOfData
Start of valid data.

nBytes
Number of bytes in buffer.

Pad
Value to use when padding the buffer.

SOUND_EXCLUDE_CODE

```
typedef enum {
    SoundExcludeOpen,
    SoundExcludeClose,
    SoundExcludeEnter,
    SoundExcludeLeave,
    SoundExcludeQueryOpen
} SOUND_EXCLUDE_CODE;
```

SOUND_EXCLUDE_CODE is an enumerated type that is used with exclusion routines. Exclusion routines are defined using the SOUND_EXCLUDE_ROUTINE type.

Elements

SoundExcludeOpen
The device is being opened.

SoundExcludeClose
The device is being closed.

SoundExcludeEnter
A request for this device is starting.

SoundExcludeLeave
The request is finished.

SoundExcludeQueryOpen
Queries to determine if the device is open.

SOUND_EXCLUDE_ROUTINE

```
typedef BOOLEAN SOUND_EXCLUDE_ROUTINE(struct _LOCAL_DEVICE_INFO *, SOUND_EXCLUDE_COI
```

SOUND_EXCLUDE_ROUTINE is a type definition for routines that handle mutual exclusion operations.

Parameters

_LOCAL_DEVICE_INFO*
Type for a pointer to a LOCAL_DEVICE_INFO structure.

SOUND_EXCLUDE_CODE
Type for a code indicating the type of exclusion operation the routine should handle. See SOUND_EXCLUDE_CODE for the type's definition.

Comments

Drivers using *soundlib.lib* define a function modeled on this type and place its address in the **ExclusionRoutine** member of a SOUND_DEVICE_INIT structure.

The **SoundExcludeOpen** and **SoundExcludeClose** exclusion codes request and release access to the device, to serialize the device's use. **SoundExcludeEnter** and **SoundExcludeLeave** control temporary synchronization to the device. Generally, a driver uses mutex objects to handle these operations.

SOUND_HW_SET_VOLUME_ROUTINE

```
typedef VOID SOUND_HW_SET_VOLUME_ROUTINE(struct _LOCAL_DEVICE_INFO *);
```

SOUND_HW_SET_VOLUME_ROUTINE is the type definition for a routine the sets the volume for a device.

Parameters

_LOCAL_DEVICE_INFO*
Type for a pointer to a LOCAL_DEVICE_INFO structure.

Comments

Drivers using *soundlib.lib* define a function modeled on this type and place its address in the **HwSetVolume** member of a SOUND_DEVICE_INIT structure.

SOUND_QUERY_FORMAT_ROUTINE

```
typedef NTSTATUS SOUND_QUERY_FORMAT_ROUTINE (PLOCAL_DEVICE_INFO, PPCMWAVEFORMAT);
```

SOUND_QUERY_FORMAT_ROUTINE is the type definition for a function used to determine if a specified wave format is supported.

Parameters

PLOCAL_DEVICE_INFO
Type for a pointer to a LOCAL_DEVICE_INFO structure.

PPCMWAVEFORMAT
Type for a wave format. Although PPCMWAVEFORMAT is specified, any PWAVEFORMATEX is valid. Wave formats are defined in *mmsystem.h*.

Comments

Drivers using *soundlib.lib* define a function modeled on this type and place its address in the **QueryFormat** member of a WAVE_INFO structure.

SOUND_REGISTRY_CALLBACK_ROUTINE

```
typedef NTSTATUS SOUND_REGISTRY_CALLBACK_ROUTINE(PWSTR RegistryPathName, PVOID Cont
```

SOUND_REGISTRY_CALLBACK_ROUTINE is the type definition for the callback function that is called by **SoundEnumSubkeys**.

Parameters

RegistryPathName
Pointer to the full registry path to a subkey representing a hardware device. The path format is **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***\Parameters\Device***Number*

Context
Pointer to driver-specified context information. This is the *Context* parameter to **SoundEnumSubkeys**, and is typically the address of a driver-defined structure for storing device-specific information.

Comments

The function should save the path name pointer in device-specific storage, and use it as input to *soundlib.lib* functions requiring a registry path to a hardware device.

Note: Prior to being unloaded, the driver must deallocate the space allocated to the path name.

SYNTH_HARDWARE

```
typedef struct {
    ULONG     Key;                // For debugging
#define SYNTH_HARDWARE_KEY   (*(ULONG *)"Hw  ")
    PUCHAR    SynthBase;          // base port address for synth
} SYNTH_HARDWARE, *PSYNTH_HARDWARE;
```

The SYNTH_HARDWARE structure contains hardware information for synthesizer devices.

Members

Key
Internal only, for debugging. Should be "hw ".

SynthBase
Base of mapped port address space. Returned from **SoundMapPortAddress**.

WAVE_INFO

```
typedef struct _WAVE_INFO {
    ULONG                   Key;
#define WAVE_INFO_KEY       (*(ULONG *)"Wave")
```

```
    PDEVICE_OBJECT          DeviceObject;
    SOUND_DMA_BUFFER        DMABuf;
    SOUND_DOUBLE_BUFFER     DoubleBuffer;
    SOUND_BUFFER_QUEUE      BufferQueue;
    ULONG                   SamplesPerSec;
    UCHAR                   BitsPerSample;
    UCHAR                   Channels;
    BOOLEAN                 FormatChanged;
    PWAVEFORMATEX           WaveFormat;
    BOOLEAN                 LowPrioritySaved;
    PFILE_OBJECT            LowPriorityHandle;
    PLOCAL_DEVICE_INFO      LowPriorityDevice;
    struct {
        SOUND_BUFFER_QUEUE      BufferQueue;
        ULONG                   SamplesPerSec;
        UCHAR                   BitsPerSample;
        UCHAR                   Channels;
        PWAVEFORMATEX           WaveFormat;
        ULONG                   State;
    } LowPriorityModeSave;
    PVOID                   MRB[2];
    KEVENT                  DmaSetupEvent;
    KEVENT                  DpcEvent;
    KEVENT                  TimerDpcEvent;
    KSPIN_LOCK              DeviceSpinLock;
#if DBG
    BOOLEAN                 LockHeld;
#endif
    PKINTERRUPT             Interrupt;
    BOOLEAN                 Direction;
    UCHAR                   DMAType;
    UCHAR                   InterruptHalf;
    volatile BOOLEAN        DMABusy;
    volatile BOOLEAN        DpcQueued;
    ULONG                   Overrun;
    PVOID                   HwContext;
    WORK_QUEUE_ITEM         WaveStopWorkItem;
    KEVENT                  WaveReallyComplete;
    PSOUND_QUERY_FORMAT_ROUTINE QueryFormat;
    PWAVE_INTERFACE_ROUTINE
                            HwSetupDMA,
                            HwStopDMA,
                            HwSetWaveFormat;
    KDPC                    TimerDpc;
    KTIMER                  DeviceCheckTimer;
    BOOLEAN                 GotWaveDpc;
    BOOLEAN                 DeviceBad;
    BOOLEAN                 TimerActive;
    UCHAR                   FailureCount;
} WAVE_INFO, *PWAVE_INFO;
```

The WAVE_INFO structure contains wave device context information.

Members

Key
Internal only, for debugging. Should be "Wave".

DeviceObject
Pointer to a DEVICE_OBJECT structure.

DMABuf
Structure describing a DMA buffer. Structure type is SOUND_DMA_BUFFER. Call **SoundGetCommonBuffer** to fill in this structure.

DoubleBuffer
Internal only. This is an internal structure containing additional information about the DMA buffer. The structure contains a pointer to the buffer described by the **DMABuf** member. The structure type is SOUND_DOUBLE_BUFFER.

BufferQueue
Internal only. For queuing device requests.

SamplesPerSec
Current samples per second.

BitsPerSample
Current bits per sample, per channel.

Channels
Current number of channels.

FormatChanged
Format has changed. Tested by **HwSetWaveFormat** function.

WaveFormat
Format for non-PCM formats.

LowPrioritySaved
LowPriorityHandle
LowPriorityDevice
LowPriorityModeSave
These four members are used by *soundlib.lib* for management of low priority mode, which allows a wave input device to nominate itself as pre-emptible if its dispatch routine receives IOCTL_WAVE_SET_LOW_PRIORITY. Only one user can be in low priority mode at a time.

MRB
Internal only. Contains adapter information.

DmaSetupEvent
Internal only. Stores event to wait on during channel allocation.

DpcEvent
Internal only. Used for synchronization with DPC termination.

TimerDpcEvent
Internal only. Used to track rogue devices.

DeviceSpinLock
Internal only. Used for DPC synchronization.

LockHeld
Internal only. Used for debugging.

Interrupt
Pointer to an interrupt object. Should be obtained from a call to **SoundConnectInterrupt**.

Direction
Set by *soundlib.lib*. Set to TRUE for output, FALSE for input.

DMAType
Type of DMA. One of the following enumerated values:

```
enum {
    SoundNoDMA,
    SoundAutoInitDMA,             // Use auto-initialize
    SoundReprogramOnInterruptDMA, // Reprogram on interrupt
    Sound2ChannelDMA              // Keep 2 channels going
};
```

Sound2ChannelDMA is not currently supported in *soundlib.lib*. It is intended for a device whose hardware uses two DMA channels and alternates between them to achieve continuous sound.

InterruptHalf
Internal only. Reserved for use when the **DMAType** member's value is **SoundReprogramOnInterruptDMA**.

DMABusy
Set by *soundlib.lib* if DMA is in progress.

DpcQueued
Used for detecting an overrun condition. The ISR should test **DpcQueued**. If clear, the ISR can call **IoRequestDpc**. If set, an overrun has occurred and the ISR should set **Overrun** (see below) instead of calling **IoRequestDpc**.

Overrun
Set by the driver's ISR if overrun occurs.

HwContext
Pointer to a driver-defined structure containing device-specific hardware information. Typically used by functions pointed by the **HwSetupDMA**, **HwStopDMA**, and **HwSetWaveFormat** members.

WaveStopWorkItem
Internal only. Structure used for calls to **ExInitializeWorkItem** and **ExQueueWorkItem**, when *soundlib.lib* is using system worker threads.

WaveReallyComplete
Internal only. Set by system worker thread after **HwStopDMA** has returned.

QueryFormat
Pointer to a driver-supplied function called when the driver receives IOCTL_WAVE_SET_FORMAT or IOCTL_WAVE_QUERY_FORMAT message. The function type is SOUND_QUERY_FORMAT_ROUTINE.

HwSetupDMA
Pointer to a driver-supplied function that programs the hardware to start a DMA transfer. Function type is WAVE_INTERFACE_ROUTINE.

The function is called after *soundlib.lib* has set up map registers by calling **IoMapTransfer**. For more information, see *src**mmedia**soundlib**wave.c*.

HwStopDMA
Points to a driver-supplied function that sends commands to the hardware to stop DMA transfers. The function type is WAVE_INTERFACE_ROUTINE.

The function is called just before *soundlib.lib* calls **IoFlushAdapterBuffers**. For more information, see *src**mmedia**soundlib**wave.c*.

Note: **HwStopDMA** must not acquire the device exclusion mutex that *soundlib.lib* uses to synchronize device access. Code in *soundlib.lib* waits for a transfer to complete before starting a new one, and this wait occurs inside a request to the device, when the mutex is owned by the waiting thread. This means that **HwStopDMA** can require extra synchronization code, even though no further hardware calls to the current device can occur until the current transfer is complete.

HwSetWaveFormat
Points to a driver-supplied function that sends commands to the hardware to set the wave format. The function type is WAVE_INTERFACE_ROUTINE.

The function is called just before *soundlib.lib* starts each DMA transfer. For more information, see *src**mmedia**soundlib**wave.c*.

TimerDpc
Internal only. Structure used by *soundlib.lib* for calls to **KeInitializeDpc**.

DeviceCheckTimer
Internal only. Structure used by *soundlib.lib* for calls to **KeInitializeTimer**.

GotWaveDpc
Internal only. Indicates the device is active.

DeviceBad

Internal only. Set if the device doesn't send interrupts.

TimerActive
Internal only. Indicates the device is active.

FailureCount
Internal only. Contains count of failed I/O attempts. If the count reaches 30, **BadDevice** is set.

Comments

One WAVE_INFO structure must be defined for each waveform device (input or output) that can be in operation simultaneously. WAVE_INFO is defined in *wave.h*.

Allocate a WAVE_INFO structure from the nonpaged memory pool by calling **ExAllocatePool**, and then zero it by calling **RtlZeroMemory**. To initialize a WAVE_INFO structure, assign values to the **HwSetupDMA**, **HwStopDMA**, and **HwSetWaveFormat** members and then call **SoundInitializeWaveInfo**.

To create a waveform device object, call **SoundCreateDevice** and specify a WAVE_INFO structure pointer for the *DeviceSpecificData* parameter.

WAVE_INTERFACE_ROUTINE

```
typedef BOOLEAN WAVE_INTERFACE_ROUTINE(struct _WAVE_INFO *);
```

WAVE_INTERFACE_ROUTINE is a type definition for functions that send commands to waveform hardware.

Parameters

_WAVE_INFO*
Type for a pointer to a WAVE_INFO structure.

Comments

Drivers using *soundlib.lib* define functions modeled on this type and place their addresses in the **HwSetupDMA**, **HwStopDMA**, and **HwSetWaveFormat** members of a WAVE_INFO structure.

Audio Compression Manager Drivers

The Audio Compression Manager (ACM) and its associated drivers compress, decompress, convert, and filter waveform file data. The following topics are provided:

- Introduction to the ACM
- Introduction to ACM Drivers
- Designing an ACM Driver
- ACM Driver Reference

Introduction to the ACM

The Audio Compression Manager (ACM) provides a set of API functions that allows a client to perform compressions, decompressions, format conversions, and filtering operations on waveform file data.

The ACM is implemented as a dynamic-link library named *msacm32.dll*. Applications that link with the ACM can call its API functions. The ACM is also used by the Windows NT® wave mapper, *msacm32.drv*, so applications that specify the wave mapper as an input or output device make indirect use of the ACM. To view a diagram that illustrates the relationship of the ACM to other

Windows NT audio software, see Audio Software Components.

Descriptions of the ACM API functions are provided in the Win32 SDK. ACM API functions begin with a prefix of **acm**.

The ACM calls installable, user-mode drivers to perform conversions. Most ACM drivers can be installed with the Control Panel's Multimedia applet and are available to all Win32-based applications. (For more information, see Installing Multimedia Drivers.) Individual applications can also load ACM drivers for their own private use. (For more information, see **acmDriverAdd** in the Win32 SDK, and Designing Local ACM Drivers.)

Introduction to ACM Drivers

This section introduces some general information about ACM drivers. It describes the types of ACM drivers, discusses format tags and filter tags, and introduces the sample ACM drivers that are provided with this DDK.

An ACM driver is a type of user-mode multimedia driver. For a general discussion of multimedia drivers, see Introduction to Multimedia Drivers.

Types of ACM Drivers

There are three types of ACM drivers — codecs, converters, and filters.

Codecs
A *codec* (the term is short for *compressor/decompressor*) converts from one format type to another. Typically, a codec converts between a compressed format, such as MS-ADPCM, and the uncompressed PCM format, which most hardware recognizes.

Converters
A *converter* converts between different formats belonging to the same format type, such as between 44 kHz PCM and 11 kHz PCM formats.

Filters
A *filter* modifies audio data without changing the format. For example, an echo filter might add an echo sound to a 44 kHz PCM waveform file.

For more information on the difference between codecs and converters, see Format Tags and Filter Tags.

ACM drivers provided by Microsoft include codecs that convert the MS-ADPCM, IMA ADPCM, GSM 6.10-compliant, and Truespeech compressed formats to PCM.

The ACM also provides a PCM converter that converts between 8-bit and 16-bit PCM, between mono and stereo, and among various PCM sampling frequencies. This converter is *not* implemented is a separate installable driver. Instead, it is contained within the ACM's DLL file, *msacm32.dll*.

Format Tags and Filter Tags

A format tag represents the name for a format type. A filter tag represents the name for a filter type. Typically, an ACM driver supports one or more types of formats and/or filters. Within Windows NT DDK documentation, ACM data structures, and sample ACM drivers, the term *tag* is used more often than *type*.

Typically, a set of formats or filters is associated with each tag. For example, the sample IMA ADPCM audio codec supports two format tags, namely the WAVE_FORMAT_PCM tag for the PCM format type and the WAVE_FORMAT_IMA_ADPCM tag for the IMA ADPCM format type. For each of these format tags, the driver supports several formats consisting of various sample rates and sample sizes.

To differentiate between codecs and converters, we can say that a codec transforms data from a

format belonging to one format tag into a format belonging to another format tag, while a converter transforms data from one format to another belonging to the same format tag.

Format tags and filter tags are defined in *mmreg.h*. If you are writing a driver for a new format type or filter type, you must register the type by using the Multimedia Developer Registration Kit.

Sample ACM Drivers

The Windows NT DDK includes the following three sample ACM drivers:

IMA ADPCM Audio Codec

The IMA ADPCM Audio Codec, *imaadp32.dll*, converts between PCM and the IMA ADPCM format. This codec is optimized for speed, and illustrates how to design conversion routines to reduce computation time. It also provides a configuration dialog box. The first time the codec is opened, it tries to automatically configure itself by calculating its maximum sample rate.

Source files for *imaadp32.dll* are located in *\ddk\src\mmedia\imaadpcm*.

Microsoft GSM 6.10 Audio Codec

The Microsoft GSM 6.10 Audio Codec, *msgsm32.dll*, implements the GSM 6.10 voice encoding standard, originally developed for digital cellular telephone encoding. The codec converts between GSM 6.10 and PCM formats. Like *imaadp32.dll*, this codec provides a configuration dialog box, and also attempts to automatically configure itself the first time it is opened.

Source files for *msgsm32.dll* are located in *\ddk\src\mmedia\gsm610*.

Microsoft Audio Filter

The Microsoft Audio Filter, *msfltr32.dll*, is a single driver that provides both a volume filter and an echo filter. This driver supports a custom About box and a custom icon, which are explained in Providing a Custom About Box and Providing a Custom Icon.

Source files for *msfltr32.dll* are located in *\ddk\src\mmedia\msfilter*.

You should be able to easily create a new ACM driver by using these samples as models. The source code contains extensive comments that explain how the code works and how it should be modified to implement a new driver. You'll see that, in general, all ACM drivers are very similar to each other, both in their functionality and in their code layout.

Designing an ACM Driver

The best way to design and create a new ACM driver is to modify one of the sample ACM drivers, because all ACM drivers are similar to each other in layout, and because the sample code is well documented to indicate how changes should be made.

This section provides the following topics:

DriverProc in ACM Drivers

Like all other Win32-based user-mode drivers, ACM drivers must export a **DriverProc** entry point which recognizes all of the standard driver messages. The ACM sends messages to its drivers by calling **SendDriverMessage**, which is exported by *winmm.dll* and described in the Win32 SDK.

ACM drivers generally provide support for DRV_OPEN, DRV_CLOSE, DRV_CONFIGURE and DRV_QUERYCONFIGURE messages, as illustrated by the sample ACM drivers. ACM drivers generally do not need to provide much, if any, support for DRV_INSTALL, DRV_LOAD, DRV_ENABLE, DRV_DISABLE, DRV_FREE, or DRV_REMOVE messages.

When an ACM driver receives a DRV_OPEN message from the ACM, it also receives a pointer to an ACMDRVOPENDESC structure. The driver receives the pointer as the *lParam2* argument to its **DriverProc** function.

In addition to supporting the standard messages, the **DriverProc** entry point for ACM drivers must support a set of ACM driver messages.

ACM Driver Messages

The following table lists the messages that the **DriverProc** function in an ACM driver can receive, along with the operation the driver performs when it receives each message. Message definitions are contained in *msacmdrv.h*.

Message	Operation Performed by Driver
ACMDM_DRIVER_ABOUT	Displays an About dialog box.
ACMDM_DRIVER_DETAILS	Returns information about the driver.
ACMDM_DRIVER_NOTIFY	Determines status changes in other drivers.
ACMDM_FILTER_DETAILS	Returns information about a filter.
ACMDM_FILTERTAG_DETAILS	Returns information about a filter tag (type).
ACMDM_FORMAT_DETAILS	Returns information about a format.
ACMDM_FORMAT_SUGGEST	Suggests an input or output format.
ACMDM_FORMATTAG_DETAILS	Returns information about a format tag (type).
ACMDM_HARDWARE_WAVE_CAPS_INPUT	Returns device input capabilities.
ACMDM_HARDWARE_WAVE_CAPS_OUTPUT	Returns device output capabilities.
ACMDM_STREAM_CLOSE	Closes a data stream.
ACMDM_STREAM_CONVERT	Performs conversion operations on supplied data, based on conversion parameters received by ACMDM_STREAM_OPEN.
ACMDM_STREAM_OPEN	Opens a data stream. Includes data conversion parameters.
ACMDM_STREAM_PREPARE	Prepares source and destination data buffers.
ACMDM_STREAM_RESET	Stops an asynchronous conversion operation.
ACMDM_STREAM_SIZE	Estimates the required size of a source or destination buffer.
ACMDM_STREAM_UNPREPARE	Removes preparation performed on source and destination data buffers.

Converting Data Streams

Each ACM driver, whether a codec, converter, or filter, treats data as a stream. A client passes the input stream to the driver in one or more source buffers. The driver performs a conversion operation on the data and returns the converted stream to the client in one or more destination buffers.

ACM drivers can be designed to handle data conversion operations either synchronously or asynchronously. Generally, asynchronous operation only makes sense when a driver can perform hardware-assisted conversions. Drivers providing software-implemented conversions, such as the sample ACM drivers, usually operate synchronously.

A driver opens a stream when a client sends it an ACMDM_STREAM_OPEN message. With this message, the client includes pointers to structures that describe the format and other characteristics of both the source (input) and destination (output) streams. The driver uses this information to determine the types of transformations to perform on the source data.

If the client sends the driver an ACMDM_STREAM_SIZE message, specifying a source (or destination) buffer size, the driver returns the required size of a destination (or source) buffer. The driver uses the source and destination format characteristics, received with the previous ACMDM_STREAM_OPEN message, to determine the necessary source or destination buffer size.

Before the client can pass data buffers to the driver, it must prepare the buffers for use by passing them to the driver with an ACMDM_STREAM_PREPARE message. Each ACMDM_STREAM_PREPARE message includes the address of a stream header structure, defined by ACMDRVSTREAMHEADER, containing pointers to a source buffer and a destination buffer.

When the driver receives an ACMDM_STREAM_CONVERT message from the client, it begins the data transformation. Each ACMDM_STREAM_CONVERT message includes the address of a prepared stream header structure. The algorithm for what happens next depends on whether the driver is designed to operate synchronously or asynchronously.

To convert data synchronously

- The driver applies the appropriate transformation algorithms to the source buffer data and places the results in the destination buffer. It then returns control to the client.
- The client can send additional ACMDM_STREAM_CONVERT messages. Each time the driver receives a message, it converts the data and places it in the destination buffer, then returns control to the client.

To convert data asynchronously

- The driver places the address of the stream header structure in a queue, and returns control to the client. The client is then free to send additional ACMDM_STREAM_CONVERT messages, and the client adds the addresses of the additional header structures to its conversion queue.
- The driver applies the appropriate transformation algorithms to each source buffer and places the results in the associated destination buffer. Each time a buffer is converted, the driver sends the client an MM_ACM_DONE callback message and dequeues the associated stream header structure.
- The driver continues this conversion and notification sequence until all buffers have been converted, or until the client sends an ACMDM_STREAM_RESET message.

When the client has finished the conversion, it sends an ACMDM_STREAM_UNPREPARE message for each stream header structure, and then sends an ACMDM_STREAM_CLOSE message.

Note: If your driver supports asynchronous conversions and a client requests a synchronous conversion (by *not* specifying the ACM_STREAMOPENF_ASYNC flag with the **acmStreamOpen** function, which is described in the Win32 SDK), the ACM manager sets the ACM_STREAMOPENF_ASYNC flag and specifies a local event handle as a callback target. In other words, the driver *always* receives ACM_STREAMOPENF_ASYNC with ACMDM_STREAM_OPEN if it is an asynchronous driver. The ACM receives the callback notification messages sent by the driver, and the conversion appears to operate synchronously from the client's point of view. For more information about callbacks, see Notifying Clients from ACM Drivers.

Real Time Conversions

Generally, ACM drivers should perform conversions in real time. This means the driver should be

able to perform conversion operations fast enough that there are no delays in a recording or playback operation, if a client is requesting the conversion to take place simultaneously with the recording or playback operation. (An example of such a client is the wave mapper.) Consequently, if a driver's conversion algorithm requires relatively large amounts of calculations, it might not be able to run in real time.

When a client sends an ACMDM_STREAM_OPEN message, it can set a flag to indicate that it does not require the conversion to take place in real time. Drivers that cannot provide real time conversions can only operate if the client sets this flag.

Hardware-Assisted Conversions

Some waveform devices support hardware-assisted conversions. These devices accept data in one format, convert it to another format, and return the converted data without playing it. Hardware-assisted conversions are typically faster than conversions that must be entirely implemented in software. Drivers that make use of hardware assistance should probably be written to operate asynchronously.

To access a hardware conversion operation, an ACM driver must call the device's kernel-mode driver, typically by means of the **DeviceIOControl** function described in the Win32 SDK. (For more information about kernel-mode drivers, see Kernel-Mode Multimedia Drivers.)

Notifying Clients from ACM Drivers

Asynchronous ACM drivers are responsible for notifying clients upon the completion of certain driver events. When a client sends an ACMDM_STREAM_OPEN message, it indicates the type of notification, if any, it expects to receive. A client can specify any of the following notification targets:

- A callback function
- A window handle
- An event handle

ACM drivers notify clients by calling the **DriverCallback** function in *winmm.dll*. This function delivers a message to the client's notification target. The **DriverCallback** function also delivers message parameters, if the target type accepts parameters.

Asynchronous ACM drivers must send MM_ACM_OPEN, MM_ACM_CLOSE, and MM_ACM_DONE messages to clients.

Because the sample ACM drivers provided with this DDK operate synchronously, they do not send notification messages.

Installing ACM Drivers

If you are writing an ACM driver that is meant to be available to all applications, the driver must be installed so that the ACM can find it.

On the other hand, if you are writing an ACM driver that is meant to be locally available to a specific application, the driver is not installed. Instead, either the application or the driver calls **acmDriverAdd**, described in the Win32 SDK, to make the driver locally accessible. (For more information about local ACM drivers, see Designing Local ACM Drivers.)

ACM drivers that are meant to be available to all applications are installed by using the Multimedia applet in the Control Panel. You must provide an *oemsetup.inf* file for your driver. Following is a sample *oemsetup.inf* file:

```
[Source Media Descriptions]
    1  = "MSGSM610"  , TAGFILE = disk1

[Installable.Drivers]
msgsm610 = 1:msgsm32.acm, "msacm.msgsm610", "Microsoft GSM 6.10 Audio CODEC" ,,,
```

For a description of *oemsetup.inf* file contents, see Installing Multimedia Drivers. The sample file causes the Multimedia applet to create the following registry entry:

```
msacm.msgsm610 : REG_SZ : msgsm32.acm
```

This entry is placed in the registry path **HKEY_LOCAL_MACHINE \SOFTWARE \Microsoft \Windows NT \CurrentVersion \Drivers32**

The ACM searches this registry path to determine which ACM drivers are installed. For each installed ACM driver, there must be a registry value name of the form **msacm.***alias*, where *alias* is a unique name. This name is generally known as the *driver type*. For ACM drivers, only one driver file can be associated with each driver type.

Notice that the ACM driver's file extension is *.acm*. While not required, this extension is favored over *.dll* for ACM drivers.

Configuring ACM Drivers

It is often necessary for ACM drivers to obtain and store configuration parameters. A driver requiring configuration parameters must display a configuration dialog box when its **DriverProc** function receives a DRV_CONFIGURE message. The dialog box must allow a user to specify configuration parameter values. Optionally, the dialog box can allow the user to select automatic configuration. If your driver provides this automatic configuration option, it must attempt to automatically provide values for all configuration parameters. Two of the sample ACM drivers, the IMA ADPCM Audio Codec and the Microsoft GSM 6.10 Audio Codec, provide automatic configuration.

An ACM driver's **DriverProc** function must respond to DRV_QUERYCONFIGURE messages by indicating whether or not the driver provides a configuration dialog box.

A separate copy of an ACM driver's configuration parameters should be saved for each user. To accomplish this, the driver should store configuration parameters in the registry, under the path **HKEY_CURRENT_USER\Software\Microsoft\Multimedia\msacm.***alias*.

For a description of *alias*, see Installing ACM Drivers. (Use the msacm.*alias* naming scheme even for local drivers, which are not installed.) Since configuration parameters should be stored for each user, the parameters should be obtained and stored when the driver receives a DRV_OPEN message.

It is important for your driver to provide default values for all configuration parameters. In some circumstances, such as the playing of system sounds, there is no user context, so attempts to reference a registry path under HKEY_CURRENT_USER will fail. In such a case your driver must use its default values.

Designing Local ACM Drivers

Local ACM drivers are available only to a specific application. The application must explicitly add the driver to the ACM's list of available drivers with a call to **acmDriverAdd**, which is exported by the ACM and described in the Win32 SDK. If you are writing a local ACM driver, you can locate your driver code in your application or in a separate DLL file.

Including ACM Driver Code in an Application

If you include ACM driver code within an application, the application must define a **DriverProc** function that recognizes ACM driver messages. To access the ACM driver code, your application must:

- Call **acmDriverAdd**, specifying it's own module handle as an input parameter.
- Communicate with the driver code by calling the ACM's API functions.
- Call **acmDriverRemove** when it has finished using the ACM driver code.

Providing a Local ACM Driver as a Separate DLL

If you provide a local ACM driver as a separate DLL, your application must call **LoadLibrary** (see the Win32 SDK) to load the driver. The driver must define a **DriverProc** function that recognizes ACM driver messages. After the driver has been loaded, it must:

- Call **acmDriverAdd**, specifying it's own module handle as an input parameter.
- Respond to ACM driver messages received when the application calls the ACM's API functions.
- Call **acmDriverRemove** when the application signals that it has finished using the driver.

Defining Format Structures and Filter Structures

Associated with each format tag and filter tag is a data structure that is based on the WAVEFORMATEX structure or the WAVEFILTER structure. These structures are described in the Win32 SDK and defined in *mmreg.h*.

If your driver provides support for new format tags or filter tags, you must provide new structure definitions and register them by using the Multimedia Developer Registration Kit.

Providing a Custom About Box

An ACM driver can provide a custom About box. This About box is displayed by the Control Panel's Multimedia applet. If the driver does not provide a custom About box, the Multimedia applet uses a default About box.

When the driver receives an ACMDM_DRIVER_ABOUT message, it should call **DialogBox** (described in the Win32 SDK) to create and display its custom dialog box. If the driver does not provide a custom dialog box, it should return MMMSYSERR_NOTSUPPORTED when it receives an ACMDM_DRIVER_ABOUT message.

To lessen the task involved in writing an ACM driver, and provide user interface consistency, it is better to use the default About box than to provide a custom About box.

Providing a Custom Icon

An ACM driver can provide a custom icon. This icon is displayed by the Control Panel's Multimedia applet inside the driver's Properties and About boxes. If the driver does not provide a custom icon, the Multimedia applet uses a default icon. To provide a custom icon, you should design the icon using a graphics application and define it to be a resource in the driver's resource definition (*.rc*) file. For more information about creating icons, see the Win32 SDK.

When the driver receives an ACMDM_DRIVER_DETAILS message, it should call **LoadIcon** (described in the Win32 SDK) to load the icon, and return the icon's handle in the ACMDRIVERDETAILS structure's **hIcon** member. To use the default icon, the driver should return NULL in **hIcon**.

Providing ACM Support in Device Drivers

Some waveform devices can play and record data that is in a compressed format. For these devices, clients do not have to call ACM functions to convert data streams. However, ACM functions provide one of the means by which clients determine which formats are supported on a system. So even if a particular format is supported by device hardware, the client still might call ACM functions to determine if the format is supported. (The other means by which clients determine if a device supports a compressed format is by specifying the WAVE_FORMAT_DIRECT flag with the WODM_OPEN and WIDM_OPEN audio driver messages.)

If you are designing a user-mode audio driver for a waveform device that supports a format in hardware, your driver's **DriverProc** function must support, at a minimum, the following ACM driver messages:

- ACMDM_DRIVER_DETAILS
- ACMDM_FORMAT_DETAILS
- ACMDM_FORMATTAG_DETAILS
- ACMDM_HARDWARE_WAVE_CAPS_INPUT
- ACMDM_HARDWARE_WAVE_CAPS_OUTPUT

A user-mode audio driver that supports ACM messages is, in reality, both a device driver and an ACM driver. As such, it must be installed twice — once as an audio device driver and once as an ACM driver. (See Installing Multimedia Drivers and Installing ACM Drivers.)

A driver of this type must be capable of determining when it is being opened as an audio driver and when it is being opened as an ACM driver. An easy way to make this determination is to examine the *lParam2* argument to **DriverProc** when a DRV_OPEN message is received. When the driver is being opened by the ACM, this argument is a pointer to an ACMDRVOPENDESC structure. When the driver is being opened by any other client, such as *winmm.dll* or a Control Panel applet, the *lParam2* argument is NULL.

Following is a possible scenario in which the driver is used as both an ACM driver and an audio driver:

1. A client calls **acmDriverDetails** (in the ACM) to send an ACMDM_DRIVER_DETAILS message. The driver sets the specified ACMDRIVERDETAILS structure's ACMDRIVERDETAILS_SUPPORTF_HARDWARE flag.
2. The client detects this flag and calls **acmMetrics** (in the ACM), specifying the ACM_METRIC_HARDWARE_WAVE_OUTPUT flag to obtain a device identifier.
3. The client uses the obtained device identifier as input to **waveOutOpen** (in *winmm.dll*), to open an output stream to a waveform device, and then sends data to the device using **waveOutWrite**.

Writing Portable ACM Drivers

Audio Compression Managers are provided for Windows NT® and for Windows® 95. The interfaces provided by both ACMs are identical. You can write binary-compatible ACM drivers that are portable between Windows NT and Windows 95, by obeying the following rules:

- Do not call API functions that are only available under Windows NT. (Almost all functions described in the Win32 SDK are available under both Windows NT and Windows 95.)
- Do not compile with the UNICODE constant defined.

The second rule requires further discussion. Windows NT provides Unicode versions of all Win32 API functions, but Windows 95 does not (with a few exceptions). However, ACM drivers must *always* pass Unicode strings to the ACM, whether the ACM is running under Windows NT or Windows 95. To get around this conflict, your ACM driver can include a copy of the **LoadStringCodec** function that is defined in the sample ACM drivers.

The **LoadStringCodec** function loads a string resource and converts the string to Unicode, even if the source code was not compiled with the UNICODE constant defined. The converted string can then be passed to the ACM (in an ACMDRIVERDETAILS structure, for example). You can also convert between Unicode and ANSI strings by calling the **MultiByteToWideChar** and **WideCharToMultiByte** functions that are described in the Win32 SDK.

The sample ACM drivers are written to be binary-compatible with Windows NT and Windows 95.

Guidelines for Writing ACM Drivers

Use the following guidelines when writing an ACM driver:

- Allocate driver instance data when the driver receives a DRV_OPEN message, as explained in the description of DRV_OPEN.

- Allocate stream instance data when the driver receives an ACMDM_STREAM_OPEN message, as explained in the description of ACMDM_STREAM_OPEN.
- Do not use global variables. Defining a single DWORD of global data in a DLL allocates 4K of memory in every process that uses the ACM, because your driver is mapped into the address space of each process, and global data space is not shared. Instead of using global data, dynamically allocate local storage space for each driver instance and each stream instance, as needed.
- Do not link to *crtdll.dll*, the dynamic-link version of the C runtime library. This DLL cannot be loaded into all contexts. As a result, your driver will not work correctly for system sounds that are played by means of the **MessageBeep** function. Use Win32 API functions instead of C library functions, or link to a static C runtime library (*libc.lib* or *libcmt.lib*).
- Be careful when calling Win32 functions. If your driver is used in conjunction with playing system sounds, it might get loaded into a context in which these functions fail. This warning pertains to any Win32 function that requires an instance handle, and possibly other functions. Notice, for example, that if the samples call **LoadString** or **LoadIcon**, they do not test for error return values. If you strictly follow the model provided by the sample drivers, you will not have this problem.

ACM Driver Reference

This section describes the messages and structures used by ACM drivers.

Messages, ACM Drivers

This section describes the messages received by ACM drivers. The messages are listed in alphabetic order. They are defined in *msacmdrv.h*.

ACMDM_DRIVER_ABOUT

The ACMDM_DRIVER_ABOUT message requests an ACM driver to display its About dialog box.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_DRIVER_ABOUT

lParam1
Contains a validated window handle, which the driver should use to specify the parent window for the About dialog box. The value can also be -1L (See the **Comments** section below).

lParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*. If the driver does not provide an about box, it should return MMSYSERR_NOTSUPPORTED.

Comments

A client sends the ACMDM_DRIVER_ABOUT message by calling the driver's **DriverProc** entry point, passing the specified parameters.

Typically, this message is sent by the Control Panel's Multimedia applet.

An ACM driver does not have to provide an About dialog box. If it does not, it should always return MMSYSERR_NOTSUPPORTED in response to this message. The ACM provides a default About dialog box, which is displayed if the driver does not provide one.

If the driver does provide an About box, it should display it when it receives this message.

If *lParam1* is -1L, the driver should not display its About dialog box. It should just return MMSYSERR_NOERROR if it provides an About box, and MMSYSERR_NOTSUPPORTED if it does not.

For more information about custom About boxes, see Providing a Custom About Box.

ACMDM_DRIVER_DETAILS

The ACMDM_DRIVER_DETAILS message requests an ACM driver to return detailed information about itself.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_DRIVER_DETAILS

lParam1
Pointer to an ACMDRIVERDETAILS structure. (ACMDRIVERDETAILS is defined in *msacm.h* and described in the Win32 SDK.)

lParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*.

Comments

A client sends the ACMDM_DRIVER_DETAILS message by calling the driver's **DriverProc** entry point, passing the specified parameters. The ACM sends this message when an application calls **acmDriverDetails**, which is described in the Win32 SDK.

Before the driver's **DriverProc** function is called, the ACM verifies that *lParam1* contains a valid pointer and that the ACMDRIVERDETAILS structure's **cbStruct** member contains a size value of at least four.

The driver should fill in the ACMDRIVERDETAILS structure members, up to the number of bytes specified by the **cbStruct** member.

ACM drivers must support this message.

For more information about custom icons, see Providing a Custom Icon.

ACMDM_DRIVER_NOTIFY

The ACMDM_DRIVER_NOTIFY message notifies an ACM driver of changes to other ACM drivers.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the

DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_DRIVER_NOTIFY

lParam1
Not used.

lParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*.

Comments

The ACM sends the ACMDM_DRIVER_NOTIFY message by calling the driver's **DriverProc** entry point, passing the specified parameters, each time a client calls the **acmDriverAdd**, **acmDriverRemove**, or **acmDriverPriority** function. (These functions are described in the Win32 SDK.)

ACM driver support for this message is optional. If the driver supports the message, it can call ACM API functions, such as **acmEnumDrivers** and **acmMetrics**, to determine which drivers have been added, removed, enabled, disabled, or had their priority changed.

ACMDM_FILTER_DETAILS

The ACMDM_FILTER_DETAILS message requests an ACM driver to return information about a filter associated with a specified filter tag.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_FILTER_DETAILS

lParam1
Pointer to an ACMFILTERDETAILS structure. (ACMFILTERDETAILS is defined in *msacm.h* and described in the Win32 SDK.)

lParam2
Contains one of the following flags, specified by the *fdwDetails* parameter of the **acmFilterDetails** function (described in the Win32 SDK):

Flag	Meaning
ACM_FILTERDETAILSF_INDEX	Indicates the **dwFilterIndex** member of the ACMFILTERDETAILS structure contains a filter index. The valid index range is from zero to one less than the **cStandardFilters** member returned in the ACMFILTERTAGDETAILS structure for the filter tag. (See ACMDM_FILTERTAG_DETAILS.)
ACM_FILTERDETAILSF_FILTER	Indicates the client has filled in the WAVEFILTER structure associated with the ACMFILTERDETAILS structure.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*. Possible error codes include:

Error Code	Meaning
MMSYSERR_NOTSUPPORTED	The driver does not support filter operations or the specified query operation.
ACMERR_NOTPOSSIBLE	The input parameter values do not represent a valid filter or filter tag.

Comments

A client sends the ACMDM_FILTER_DETAILS message by calling the driver's **DriverProc** entry point, passing the specified parameters. The ACM sends this message when an application calls **acmFilterDetails**, which is described in the Win32 SDK.

An ACM driver that provides filters must support this message.

The client can do either of the following:

- Specify a filter index, in order to obtain a description of the filter associated with the index.
- Specify a filter description, in order to validate the filter and obtain the filter's string description.

The client specifies the filter tag in the ACMFILTERDETAILS structure's **dwFilterTag** member. The driver returns information for a particular filter belonging to the filter tag, as follows:

- If the ACM_FILTERDETAILSF_INDEX flag is set, the client has specified an index value in the ACMFILTERDETAILS structure's **dwFilterIndex** member. The driver fills in the WAVEFILTER structure for the filter associated with the specified index value. It also fills in the ACMFILTERDETAILS structure's **szFilter**, **fdwSupport**, and **cbStruct** members.
- If the ACM_FILTERDETAILSF_FILTER flag is set, the client has filled in the WAVEFILTER structure. The driver validates the structure contents and, if the contents are valid, fills in the ACMFILTERDETAILS structure's **szFilter**, **fdwSupport**, and **cbStruct** members.

Before calling the driver's **DriverProc** function, the ACM verifies that:

- The ACMFILTERDETAILS structure and its associated WAVEFILTER structure are readable and writeable.
- The size of the ACMFILTERDETAILS structure, contained in its **cbStruct** member, is at least the structure's defined size. (The structure's size can be larger than its defined size, to allow for a longer **szFilter** member or to allow newer, larger structure definitions to be used within drivers under development.)
- The size of the WAVEFILTER structure pointed to by the ACMFILTERDETAILS structure's **pwfltr** member is at least as large as the generic WAVEFILTER structure's defined size. (See **The WAVEFILTER Structure** below.)
- The ACMFILTERDETAILS structure's **fdwSupport** member contains zero.
- The *lParam2* parameter contains a valid flag value.

Before returning, the driver must set the ACMFILTERDETAILS structure's **cbStruct** member to the actual number of bytes returned. The value returned in **cbStruct** must not be greater than the value received from the client.

For more information about filter tags and filter structures, see Format Tags and Filter Tags and Defining Format Structures and Filter Structures.

The WAVEFILTER Structure

The WAVEFILTER structure is a generic structure for describing a filter. Generally, you will extend this structure for your specific filter type, as has been done in the Microsoft Audio Filter. (For examples, see VOLUMEWAVEFILTER and ECHOWAVEFILTER in *mmreg.h*.) When a client sends an ACMDM_FILTER_DETAILS message, it specifies the address of a structure that

you have defined for the specified filter type. This structure is typically larger than the generic WAVEFILTER structure.

ACMDM_FILTERTAG_DETAILS

The ACMDM_FILTERTAG_DETAILS message requests an ACM driver to return information about a filter tag.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_FILTERTAG_DETAILS

lParam1
Pointer to an ACMFILTERTAGDETAILS structure. (ACMFILTERTAGDETAILS is defined in *msacm.h* and described in the Win32 SDK.)

lParam2
Contains one of the following flags specified by the *fdwDetails* parameter of the **acmFilterTagDetails** function (described in the Win32 SDK):

Flag	Meaning
ACM_FILTERTAGDETAILSF_INDEX	Indicates the **dwFilterTagIndex** member of the ACMFILTERTAGDETAILS structure contains a filter tag index. The valid index range is from zero to one less than the **cFilterTags** member returned in the driver's ACMDRIVERDETAILS structure. (See ACMDM_DRIVER_DETAILS.) The driver should return details for the filter tag associated with the index.
ACM_FILTERTAGDETAILSF_FILTERTAG	Indicates the **dwFilterTag** member of the ACMFILTERTAGDETAILS structure contains a filter tag. The driver should return details for the specified filter tag.
ACM_FILTERTAGDETAILSF_LARGESTSIZE	Indicates the driver should return details for the filter tag having the largest filter. The **dwFilterTag** member of ACMFILTERTAGDETAILS can contain a filter tag or WAVE_FILTER_UNKNOWN.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*. Possible error codes include:

Error Code	Meaning
MMSYSERR_NOTSUPPORTED	The driver does not support filter operations or the specified query operation.
ACMERR_NOTPOSSIBLE	The input parameter values don't represent a valid filter or filter tag.

Comments

A client sends the ACMDM_FILTERTAG_DETAILS message by calling the driver's **DriverProc** entry point, passing the specified parameters. The ACM sends this message when an application calls **acmFilterTagDetails**, which is described in the Win32 SDK.

An ACM driver that provides filters must support this message.

The client specifies the filter tag in the ACMFILTERTAGDETAILS structure's **dwFilterTag** member. The driver returns information for a particular tag, as follows:

- If the ACM_FILTERDETAILSF_INDEX flag is set, the client has specified an index value in the ACMFILTERTAGDETAILS structure's **dwFilterTagIndex** member. The driver fills in the ACMFILTERTAGDETAILS structure for the filter tag associated with the specified index value.
- If the ACM_FILTERTAGDETAILSF_FILTERTAG flag is set, the client has specified a filter tag in the ACMFILTERTAGDETAILS structure's **dwFilterTag** member. The driver fills in the ACMFILTERTAGDETAILS structure for the specified filter tag.
- If the ACM_FILTERTAGDETAILSF_LARGESTSIZE flag is set, the driver does one of two things:
  1. If **dwFilterTag** contains WAVE_FILTER_UNKNOWN, the driver fills in the ACMFILTERTAGDETAILS structure for the filter tag having a filter with the largest filter structure size.
  2. If **dwFilterTag** contains a filter tag, the driver fills in the ACMFILTERTAGDETAILS structure for that filter tag, describing the filter with the largest structure size belonging to the specified tag.

Before calling the driver's **DriverProc** function, the ACM verifies that:

- The ACMFILTERTAGDETAILS structure is readable and writeable.
- The size of the ACMFILTERTAGDETAILS structure, contained in its **cbStruct** member, is at least the structure's defined size. (The structure's size can be larger than its defined size, to allow for a longer **szFilterTag** member or to allow newer, larger structure definitions to be used within drivers under development.)
- The ACMFILTERTAGDETAILS structure's **fdwSupport** member contains zero.
- The *lParam2* parameter contains a valid flag value.

Before returning, the driver must set the ACMFILTERTAGDETAILS structure's **cbStruct** member to the actual number of bytes returned. The value returned in **cbStruct** must not be greater than the value received.

For more information about filter tags, see Format Tags and Filter Tags.

ACMDM_FORMAT_DETAILS

The ACMDM_FORMAT_DETAILS message requests an ACM driver to return information about a format associated with a specified format tag.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_FORMAT_DETAILS

lParam1
Pointer to an ACMFORMATDETAILS structure. (ACMFORMATDETAILS is defined in *msacm.h* and described in the Win32 SDK.)

lParam2
Contains one of the following flags, specified by the *fdwDetails* parameter of the

acmFormatDetails function, which is described in the Win32 SDK:

Flag	Meaning
ACM_FORMATDETAILSF_INDEX	Indicates the **dwFormatIndex** member of the ACMFORMATDETAILS structure contains a format index. The valid index range is from zero to one less than the **cStandardFormats** member returned in the ACMFORMATTAGDETAILS structure for the format tag. (See ACMDM_FORMATTAG_DETAILS.)
ACM_FORMATDETAILSF_FORMAT	Indicates the client has filled in the WAVEFORMATEX structure associated with the ACMFORMATDETAILS structure.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*. Possible error codes include:

Error Code	Meaning
MMSYSERR_NOTSUPPORTED	The driver does not support the specified query operation.
ACMERR_NOTPOSSIBLE	The input parameter values do not represent a valid format or format tag.

Comments

A client sends the ACMDM_FORMAT_DETAILS message by calling the driver's **DriverProc** entry point, passing the specified parameters. The ACM sends this message when an application calls the **acmFormatDetails** function, which is described in the Win32 SDK.

All ACM drivers must support this message.

The client can do either of the following:

- Specify a format index, in order to obtain a description of the format associated with the index.
- Specify a format description, in order to validate the format and obtain the format's string description.

The client specifies the format tag in the ACMFORMATDETAILS structure's **dwFormatTag** member. The driver returns information for a particular format belonging to the format tag, as follows:

- If the ACM_FORMATDETAILSF_INDEX flag is set, the client has specified an index value in the ACMFORMATDETAILS structure's **dwFormatIndex** member. The driver fills in the WAVEFORMATEX structure for the format associated with the specified index value. It also fills in the ACMFORMATDETAILS structure's **szFormat**, **fdwSupport**, and **cbStruct** members.
- If the ACM_FORMATDETAILSF_FORMAT flag is set, the client has filled in the WAVEFORMATEX structure. The driver validates the structure contents and, if the contents are valid, fills in the ACMFORMATDETAILS structure's **szFormat**, **fdwSupport**, and **cbStruct** members.

Before calling the driver's **DriverProc** function, the ACM verifies that:

- The ACMFORMATDETAILS structure and its associated WAVEFORMATEX structure are readable and writeable.
- The size of the ACMFORMATDETAILS structure (contained in its **cbStruct** member) is at least the structure's defined size. (The structure's size can be larger than its defined size, to allow for a longer **szFormat** member or to allow newer, larger structure definitions to be used within drivers under development.)

- The size of the WAVEFORMATEX structure associated with the ACMFORMATDETAILS structure's **pwfx** member is at least as large as the generic WAVEFORMATEX structure's defined size. (See **The WAVEFORMATEX Structure** below.)
- The ACMFORMATDETAILS structure's **fdwSupport** member contains zero.
- The *lParam2* parameter contains a valid flag value.

Before returning, the driver must set the ACMFORMATDETAILS structure's **cbStruct** member to the actual number of bytes returned. The value returned in **cbStruct** must not be greater than the value received.

The WAVEFORMATEX Structure

The WAVEFORMATEX structure is a generic structure for describing a waveform format. Generally, you will use this structure as a basis for defining structures for your specific format types, as has been done in the IMA ADPCM Audio Codec. (For an example, see IMAADPCMWAVEFORMAT in *mmreg.h*.) When a client sends an ACMDM_FORMAT_DETAILS message, it specifies the address of a structure that you have defined for the specified format type. This structure is typically larger than the generic WAVEFORMATEX structure.

For more information about format tags and format structures, see Format Tags and Filter Tags and Defining Format Structures and Filter Structures.

Returning a Description String

The WAVEFORMATEX structure's **szFormat** member is used for returning a format description string. If an ACM driver returns a zero-length string in **szFormat**, the ACM creates an internationalized description string for the format. This string includes the format's speed (in Hz), bit depth, and channel setting (mono or stereo), based on the contents of the **nSamplesPerSec**, **wBitsPerSample**, and **nChannels** members of the WAVEFORMATEX structure. If **wBitsPerSample** contains zero, the ACM does not include the bit depth in the description string. You can provide your own description string and return it in **szFormat**, but allowing ACM to generate an internationalized string is preferred.

ACMDM_FORMAT_SUGGEST

The ACMDM_FORMAT_SUGGEST message requests an ACM driver to suggest a destination format for a conversion, given a specified source format.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_FORMAT_SUGGEST

lParam1
Pointer to an ACMDRVFORMATSUGGEST structure. (ACMDRVFORMATSUGGEST is defined in *msacmdrv.h*.)

lParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*. Possible error codes include:

Error Code	Meaning
MMSYSERR_NOTSUPPORTED	The driver does not support format

	suggestion operations.
ACMERR_NOTPOSSIBLE	The driver cannot suggest a destination format, based on the specified source format and restriction flags.

Comments

A client sends the ACMDM_FORMAT_SUGGEST message by calling the driver's **DriverProc** entry point, passing the specified parameters. The ACM sends this message when an application calls the **acmFormatSuggest** function, which is described in the Win32 SDK.

All ACM drivers that provide stream conversions must support this message.

The ACMDRVFORMATSUGGEST structure contains pointers to two WAVEFORMATEX structures. One of these structures describes the client-specified source format. The other structure is used by the driver to return a suggested destination format. The client might specify values for some of the destination structure members, in order to restrict the possible suggestions. For more information, see the description of ACMDRVFORMATSUGGEST.

Given the specified source format and destination restrictions (if any), the driver determines if it can provide a conversion from the specified source format to some destination format. If it can, it returns a description of that format in the destination WAVEFORMATEX structure.

Before calling the driver's **DriverProc** function, the ACM verifies that:

- The flag values contained in the ACMDRVFORMATSUGGEST structure are valid.
- The WAVEFORMATEX structure containing the source format description is readable.
- The WAVEFORMATEX structure specified for receiving the suggested destination format description is writeable.
- The destination WAVEFORMATEX structure's size, contained in the ACMDRVFORMATSUGGEST structure's **cbwfxDst** member, is large enough to receive a format structure for an appropriate destination format.

ACMDM_FORMATTAG_DETAILS

The ACMDM_FORMATTAG_DETAILS message requests an ACM driver to return information about a format tag.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_FORMATTAG_DETAILS

lParam1
Pointer to an ACMFORMATTAGDETAILS structure. (ACMFORMATTAGDETAILS is defined in *msacm.h* and described in the Win32 SDK.)

lParam2
Contains one of the following flags, specified by the *fdwDetails* parameter of the **acmFormatTagDetails** function, which is described in the Win32 SDK:

Flag	Meaning
ACM_FORMATTAGDETAILSF_INDEX	Indicates the **dwFormatTagIndex** member of the ACMFORMATTAGDETAILS structure contains a format tag index. The valid index range is from zero to one less than the **cFormatTags** member returned in the

	driver's ACMDRIVERDETAILS structure. (See ACMDM_DRIVER_DETAILS.) The driver should return details for the format tag associated with the index.
ACM_FORMATTAGDETAILSF_FORMATTAG	Indicates the **dwFormatTag** member of the ACMFORMATTAGDETAILS structure contains a format tag. The driver should return details for the specified format tag.
ACM_FORMATTAGDETAILSF_LARGESTSIZE	Indicates the driver should return details for the format tag having the largest format. The **dwFormatTag** member of ACMFORMATTAGDETAILS can contain a format tag or WAVE_FORMAT_UNKNOWN.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*. Possible error codes include:

Error Code	Meaning
MMSYSERR_NOTSUPPORTED	The driver does not support the specified query operation.
ACMERR_NOTPOSSIBLE	The input parameter values do not represent a valid format or format tag.

Comments

A client sends the ACMDM_FORMATTAG_DETAILS message by calling the driver's **DriverProc** entry point, passing the specified parameters. The ACM sends this message when an application calls the **acmFormatTagDetails** function, which is described the Win32 SDK.

All ACM drivers must support this message.

The client specifies the format tag in the ACMFORMATTAGDETAILS structure's **dwFormatTag** member. The driver returns information for a particular tag, as follows:

- If the ACM_FORMATDETAILSF_INDEX flag is set, the client has specified an index value in the ACMFORMATTAGDETAILS structure's **dwFormatTagIndex** member. The driver fills in the ACMFORMATTAGDETAILS structure for the format tag associated with the specified index value.
- If the ACM_FORMATTAGDETAILSF_FORMATTAG flag is set, the client has specified a format tag in the ACMFORMATTAGDETAILS structure's **dwFormatTag** member. The driver fills in the ACMFORMATTAGDETAILS structure for the specified format tag.
- If the ACM_FORMATTAGDETAILSF_LARGESTSIZE flag is set, the driver does one of two things:
  1. If **dwFormatTag** contains WAVE_FORMAT_UNKNOWN, the driver fills in the ACMFORMATTAGDETAILS structure for the format tag that has a format with the largest format structure size.
  2. If **dwFormatTag** contains a format tag, the driver fills in the ACMFORMATTAGDETAILS structure for that format tag, describing the format with the largest structure size belonging to the specified tag.

Before calling the driver's **DriverProc** function, the ACM verifies that:

- The ACMFORMATTAGDETAILS structure is readable and writeable.
- The size of the ACMFORMATTAGDETAILS structure (contained in its **cbStruct** member) is at least the structure's defined size. (The structure's size can be larger than its defined size, to

allow for a longer **szFormatTag** member or to allow newer, larger structure definitions to be used within drivers under development.)

- The ACMFORMATTAGDETAILS structure's **fdwSupport** member contains zero.
- The *lParam2* parameter contains a valid flag value.

If the format tag is WAVE_FORMAT_PCM, then the driver should return a zero-length string in **szFormatTag**. The ACM provides a description string for this format.

Before returning, the driver must set the ACMFORMATTAGDETAILS structure's **cbStruct** member to the actual number of bytes returned. The value returned in **cbStruct** must not be greater than the value received.

For more information about format tags, see Format Tags and Filter Tags.

ACMDM_HARDWARE_WAVE_CAPS_INPUT

The ACMDM_HARDWARE_WAVE_CAPS_INPUT message requests an ACM driver to return hardware capabilities for a waveform input device.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_HARDWARE_WAVE_CAPS_INPUT

lParam1
Pointer to a WAVEINCAPS structure. (WAVEINCAPS is defined in *mmsystem.h* and described in the Win32 SDK.)

lParam2
Size of the WAVEINCAPS structure pointed to by *lParam1*.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*. If the driver does not support waveform hardware, it should return MMSYSERR_NOTSUPPORTED.

Comments

The ACM sends the ACMDM_HARDWARE_WAVE_CAPS_INPUT message by calling the driver's **DriverProc** entry point, passing the specified parameters, each time a client calls the **acmMetrics** function with the ACM_METRIC_HARDWARE_WAVE_INPUT index argument. (The **acmMetrics** function is described in the Win32 SDK.)

The driver receives the address of a WAVEINCAPS structure. The driver must fill in the structure.

Only ACM drivers that provide access to waveform input hardware need to support this message. If your driver supports the message, it must return ACMDRIVERDETAILS_SUPPORTF_HARDWARE in the ACMDRIVERDETAILS structure provided with the ACMDM_DRIVER_DETAILS message. For more information, see Providing ACM Support in Device Drivers.

ACMDM_HARDWARE_WAVE_CAPS_OUTPUT

The ACMDM_HARDWARE_WAVE_CAPS_OUTPUT message requests an ACM driver to return hardware capabilities for a waveform output device.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_HARDWARE_WAVE_CAPS_OUTPUT

lParam1
Pointer to a WAVEOUTCAPS structure. (WAVEOUTCAPS is defined in *mmsystem.h* and described in the Win32 SDK.)

lParam2
Size of the WAVEOUTCAPS structure pointed to by *lParam1*.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*. If the driver does not support waveform hardware, it should return MMSYSERR_NOTSUPPORTED.

Comments

The ACM sends the ACMDM_HARDWARE_WAVE_CAPS_OUTPUT message by calling the driver's **DriverProc** entry point, passing the specified parameters, each time a client calls the **acmMetrics** function with the ACM_METRIC_HARDWARE_WAVE_OUTPUT index argument. (The **acmMetrics** function is described in the Win32 SDK.)

The driver receives the address of a WAVEOUTCAPS structure. The driver must fill in the structure.

Only ACM drivers that provide access to waveform output hardware need to support this message. If your driver supports the message, it must return ACMDRIVERDETAILS_SUPPORTF_HARDWARE in the ACMDRIVERDETAILS structure provided with the ACMDM_DRIVER_DETAILS message. For more information, see Providing ACM Support in Device Drivers.

ACMDM_STREAM_CLOSE

The ACMDM_STREAM_CLOSE message requests an ACM driver to close a conversion stream that was opened with an ACMDM_STREAM_OPEN message.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_STREAM_CLOSE

lParam1
Pointer to an ACMDRVSTREAMINSTANCE structure.

lParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error

codes defined in *msacm.h*. An asynchronous driver might have to return ACMERR_BUSY if a conversion operation has not completed.

Comments

A client sends the ACMDM_STREAM_CLOSE message by calling the driver's **DriverProc** entry point, passing the specified parameters. The ACM sends this message when an application calls the **acmStreamClose** function, which is described in the Win32 SDK.

All ACM drivers that provide stream conversions must support this message. For more information about stream conversions, see Converting Data Streams.

If the driver supports asynchronous operations, and if the client has specified the ACM_STREAMOPENF_ASYNC flag (contained in the ACMDRVSTREAMINSTANCE structure's **fdwOpen** member), then the driver should send the client an MM_ACM_CLOSE callback message, by calling the **DriverCallback** function, after the operation completes.

ACMDM_STREAM_CONVERT

The ACMDM_STREAM_CONVERT message requests a ACM driver to perform a conversion operation on a specified conversion stream.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_STREAM_CONVERT

lParam1
Pointer to an ACMDRVSTREAMINSTANCE structure.

lParam2
Pointer to an ACMDRVSTREAMHEADER structure.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*.

Comments

A client sends the ACMDM_STREAM_CONVERT message by calling the driver's **DriverProc** entry point, passing the specified parameters. The ACM sends this message when an application calls the **acmStreamConvert** function, which is described in the Win32 SDK.

All ACM drivers that provide stream conversions must support this message. For more information about stream conversions, see Converting Data Streams.

The ACMDRVSTREAMINSTANCE structure received with this message is the same structure that was received with a previous ACMDM_STREAM_OPEN message. The driver does not need to validate the structure's contents again.

The ACMDRVSTREAMHEADER structure identifies the source and destination data buffers. The source buffer contains the data to be converted. The driver places converted data into the destination buffer.

The driver must check the flags in the ACMDRVSTREAMHEADER structure's **fdwConvert** member. These flags indicate how converted data should be returned.

Because stream conversions are time-critical operations, ACMDM_STREAM_CONVERT messages must be processed efficiently. The driver should perform as much processing as

possible in response to the ACMDM_STREAM_OPEN message.

If the driver supports asynchronous operations, and if the client has specified the ACM_STREAMOPENF_ASYNC flag (contained in the ACMDRVSTREAMINSTANCE structure's **fdwOpen** member), then the driver must do the following when it has finished converting the data in the source buffer:

- Set the ACMDRVSTREAMHEADER structure's ACMSTREAMHEADER_STATUSF_DONE flag.
- Send the client an MM_ACM_DONE callback message, by calling the **DriverCallback** function.

Asynchronous drivers can make use of the ACMDRVSTREAMHEADER structure's ACMSTREAMHEADER_STATUSF_INQUEUE flag, along with the structure's **padshNext** member, to maintain a conversion queue of stream header structures.

Before calling the driver's **DriverProc** function, the ACM verifies that:

- The ACMDRVSTREAMHEADER structure is readable and writeable, and of the proper size.
- The ACMDRVSTREAMHEADER structure's **fdwConvert** member contains valid flag values.
- The ACMDRVSTREAMHEADER structure's buffers have been prepared (see ACMDM_STREAM_PREPARE), and the specified buffer sizes are not larger than their prepared sizes.
- The ACMDRVSTREAMHEADER structure is not currently in an asynchronous driver's conversion queue. (That is, the structure's ACMSTREAMHEADER_STATUSF_INQUEUE flag is not set.)

ACMDM_STREAM_OPEN

The ACMDM_STREAM_OPEN message requests an ACM driver to either open a conversion stream or indicate whether the specified conversion is supported.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_STREAM_OPEN

lParam1
Pointer to an ACMDRVSTREAMINSTANCE structure.

lParam2
Not used.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*. If a specified conversion is not supported, the driver should return ACMERR_NOTPOSSIBLE.

Comments

A client sends the ACMDM_STREAM_OPEN message by calling the driver's **DriverProc** entry point, passing the specified parameters. The ACM sends this message when an application calls the **acmStreamOpen** function, which is described in the Win32 SDK.

All ACM drivers that provide stream conversions must support this message. For more information about stream conversions, see Converting Data Streams.

If the client has specified the ACM_STREAMOPENF_QUERY flag (contained in the

ACMDRVSTREAMINSTANCE structure's **fdwOpen** member), then the driver should not open a stream instance. It should return MMSYSERR_NOERROR if the conversion is possible, and MMSYSERR_NOTPOSSIBLE otherwise.

When a driver receives an ACMDM_STREAM_OPEN message, it should first determine if it can perform the specified conversion. If it can, then it should perform instance initialization operations for the stream, such as determining which conversion routines to use and allocating instance-specific resources.

Store stream instance data in a local, dynamically allocated structure. Store a pointer to the structure in the ACMDRVSTREAMINSTANCE structure's **dwDriver** member.

If the driver supports asynchronous operations, and if the client has specified the ACM_STREAMOPENF_ASYNC flag (contained in the ACMDRVSTREAMINSTANCE structure's **fdwOpen** member), then the driver should send the client an MM_ACM_OPEN callback message, by calling the **DriverCallback** function, after the operation completes.

Before calling the driver's **DriverProc** function, the ACM verifies that:

- The ACMDRVSTREAMINSTANCE structure's **fdwOpen** member contains valid flag values.
- The source and destination WAVEFORMATEX structures are readable.
- If a WAVEFILTER structure is specified, it is readable and the source and destination format structures contain identical information.
- If the client has specified different source and destination format tags, then the driver has declared itself to be a codec by setting ACMDRIVERDETAILS_SUPPORTF_CODEC in the ACMDRIVERDETAILS structure's **fdwSupport** member. (See ACMDM_DRIVER_DETAILS.)
- If the client has specified source and destination formats associated with a single format tag, then the driver has declared itself to be a converter by setting ACMDRIVERDETAILS_SUPPORTF_CONVERTER in the ACMDRIVERDETAILS structure's **fdwSupport** member. (See ACMDM_DRIVER_DETAILS.)
- If the client has specified a filter operation, then the driver has declared itself to be a filter by setting ACMDRIVERDETAILS_SUPPORTF_FILTER in the ACMDRIVERDETAILS structure's **fdwSupport** member. (See ACMDM_DRIVER_DETAILS.)

If the client has specified the ACM_STREAMOPENF_NONREALTIME flag (contained in the ACMDRVSTREAMINSTANCE structure's **fdwOpen** member), then the driver can perform the conversion without time constraints. However, if this flag is not specified, and the driver cannot perform the conversion in real time, then it should return ACMERR_NOTPOSSIBLE. For more information, see Real Time Conversions.

ACMDM_STREAM_PREPARE

The ACMDM_STREAM_PREPARE message requests an ACM driver to prepare the buffers associated with an ACMDRVSTREAMHEADER structure for use.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_STREAM_PREPARE

lParam1
Pointer to an ACMDRVSTREAMINSTANCE structure.

lParam2
Pointer to an ACMDRVSTREAMHEADER structure.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*.

Comments

A client sends the ACMDM_STREAM_PREPARE message by calling the driver's **DriverProc** entry point, passing the specified parameters. The ACM sends this message when an application calls the **acmStreamPrepareHeader** function, which is described in the Win32 SDK.

Support for this message is optional. If a driver supports ACMDM_STREAM_PREPARE, it must support ACMDM_STREAM_UNPREPARE.

If the driver returns MMSYSERR_NOTSUPPORTED, the ACM prepares the buffers for use. For most drivers, this behavior is sufficient. If the driver does perform buffer preparation, it should return MMSYSERR_NOERROR. In either case, the ACM sets ACMSTREAMHEADER_STATUSF_PREPARED in the ACMDRVSTREAMHEADER structure's **fdwStatus** member. The driver never modifies this flag. (If you want both your driver *and* the ACM to perform buffer preparation operations, the driver should return MMSYSERR_NOTSUPPORTED after performing its preparation activity. The ACM can then also perform its preparation operation.)

Before calling the driver's **DriverProc** function, the ACM verifies that:

- The ACMDRVSTREAMHEADER structure is readable and writeable.
- The ACMDRVSTREAMHEADER structure's **cbStruct** member contains a size value that is at least as large as the structure's defined size.
- The specified buffers have not already been prepared.

For more information about the use of ACMDM_STREAM_PREPARE, see Converting Data Streams.

ACMDM_STREAM_RESET

The ACMDM_STREAM_RESET message requests an ACM driver to stop conversion operations for the specified stream.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_STREAM_RESET

lParam1
Pointer to an ACMDRVSTREAMINSTANCE structure.

lParam2
Contains the *fdwReset* argument to the **acmStreamReset** function.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*.

Comments

A client sends the ACMDM_STREAM_RESET message by calling the driver's **DriverProc** entry point, passing the specified parameters. The ACM sends this message when an application calls

the **acmStreamReset** function, which is described in the Win32 SDK.

Only asynchronous drivers receive this message. If a client calls **acmStreamReset** for a synchronous driver, the ACM returns MMSYSERR_NOERROR without calling the driver.

When an asynchronous driver receives this message, it should set the ACMSTREAMHEADER_STATUSF_DONE flag, and clear the ACMSTREAMHEADER_STATUSF_INQUEUE flag, in every ACMDRVSTREAMHEADER structure contained in its conversion queue.

For more information about stream conversions, see Converting Data Streams.

ACMDM_STREAM_SIZE

The ACMDM_STREAM_SIZE message requests an ACM driver to return the size required for a source (or destination) buffer, given a specified destination (or source) buffer size along with source and destination format descriptions.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_STREAM_SIZE

lParam1
Pointer to an ACMDRVSTREAMINSTANCE structure.

lParam2
Pointer to an ACMDRVSTREAMSIZE structure.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*.

Comments

A client sends the ACMDM_STREAM_SIZE message by calling the driver's **DriverProc** entry point, passing the specified parameters. The ACM sends this message when an application calls the **acmStreamSize** function, which is described in the Win32 SDK.

All ACM drivers that provide stream conversions must support this message. For more information about stream conversions, see Converting Data Streams.

Based on the input arguments, the driver must answer one of the following questions:

- Given a specified source buffer size, how large does a destination buffer need to be in order to hold all of the converted data?
- Given a specified destination buffer size, what is the largest amount of source data that can specified without overflowing the destination buffer?

The ACMDRVSTREAMINSTANCE structure received with this message is the same structure that was received with a previous ACMDM_STREAM_OPEN message. The driver does not need to validate the structure's contents again.

The driver examines the ACMDRVSTREAMSIZE structure to determine which buffer (source or destination) the client has supplied. The ACMDRVSTREAMINSTANCE structure contains structures that describe the source and destination formats, and, possibly, a filter specification. The driver uses this information to determine the size of the requested buffer.

If the driver returns a buffer length of zero, the ACM provides an error return code of

ACMERR_NOTPOSSIBLE to **acmStreamSize**.

ACMDM_STREAM_UNPREPARE

The ACMDM_STREAM_UNPREPARE message requests an ACM driver to clear the preparation of the buffers associated with an ACMDRVSTREAMHEADER structure.

Parameters

dwDriverID
Driver instance identifier. This is the value returned by the driver in response to the DRV_OPEN message.

hDriver
Driver handle.

uMsg
ACMDM_STREAM_UNPREPARE

lParam1
Pointer to an ACMDRVSTREAMINSTANCE structure.

lParam2
Pointer to an ACMDRVSTREAMHEADER structure.

Return Value

The driver should return MMSYSERR_NOERROR if the operation succeeds. Otherwise it should return one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*.

Comments

A client sends the ACMDM_STREAM_UNPREPARE message by calling the driver's **DriverProc** entry point, passing the specified parameters. The ACM sends this message when an application calls the **acmStreamUnprepareHeader** function, which is described in the Win32 SDK.

Support for this message is optional. If a driver supports ACMDM_STREAM_PREPARE, it must support ACMDM_STREAM_UNPREPARE.

If the driver returns MMSYSERR_NOTSUPPORTED, the ACM clears the buffer preparation. For most drivers, this behavior is sufficient. If the driver does clear buffer preparation, it should return MMSYSERR_NOERROR. In either case, the ACM clears ACMSTREAMHEADER_STATUSF_PREPARED in the ACMDRVSTREAMHEADER structure's **fdwStatus** member. The driver never modifies this flag. (If you want both your driver *and* the ACM to clear buffer preparations, the driver should return MMSYSERR_NOTSUPPORTED after clearing its preparation. The ACM can then also clear its preparation.)

Before calling the driver's **DriverProc** function, the ACM verifies that:

- The ACMDRVSTREAMHEADER structure is readable and writeable.
- The ACMDRVSTREAMHEADER structure's **cbStruct** member contains a size value that is at least as large as the structure's defined size.
- The buffers have been previously prepared.
- The buffer addresses and sizes match those specified when these buffers were prepared.
- The buffers are not currently in use by an asynchronous driver, based on the ACMDRVSTREAMHEADER structure's ACMSTREAMHEADER_STATUSF_INQUEUE flag value.

For more information about the use of ACMDM_STREAM_UNPREPARE, see Converting Data Streams.

MM_ACM_CLOSE

The MM_ACM_CLOSE callback message notifies a client that an asynchronous ACM driver has

finished processing an ACMDM_STREAM_CLOSE message.

Parameters

dwMsg
MM_ACM_CLOSE

dwParam1
NULL

dwParam2
NULL

Comments

An asynchronous ACM driver sends an MM_ACM_CLOSE message to its client, by means of a callback, when the driver finishes processing an ACMDM_STREAM_CLOSE message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

For more information about the use of MM_ACM_CLOSE, see Notifying Clients from ACM Drivers and Converting Data Streams.

MM_ACM_DONE

The MM_ACM_DONE callback message notifies a client that an asynchronous ACM driver has finished processing an ACMDM_STREAM_CONVERT message.

Parameters

dwMsg
MM_ACM_DONE

dwParam1
Address of the ACMDRVSTREAMHEADER structure that was received with the ACMDM_STREAM_CONVERT message.

dwParam2
NULL

Comments

An asynchronous ACM driver sends an MM_ACM_DONE message to its client, by means of a callback, when the driver finishes processing an ACMDM_STREAM_CONVERT message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

For more information about the use of MM_ACM_DONE, see Notifying Clients from ACM Drivers and Converting Data Streams.

MM_ACM_OPEN

The MM_ACM_OPEN callback message notifies a client that an asynchronous ACM driver has finished processing an ACMDM_STREAM_OPEN message.

Parameters

dwMsg
MM_ACM_OPEN

dwParam1
NULL

dwParam2
NULL

Comments

An asynchronous ACM driver sends an MM_ACM_OPEN message to its client, by means of a callback, when the driver finishes processing an ACMDM_STREAM_OPEN message. The driver

sends the message to the client by calling **DriverCallback**, passing the specified parameters.

For more information about the use of MM_ACM_OPEN, see Notifying Clients from ACM Drivers and Converting Data Streams.

Structures, ACM Drivers

This section describes the structures used by ACM drivers. The structures are listed in alphabetical order.

ACMDRVFORMATSUGGEST

```
typedef struct {
    DWORD cbStruct;
    DWORD fdwSuggest;
    LPWAVEFORMATEX pwfxSrc;
    DWORD cbwfxSrc;
    LPWAVEFORMATEX pwfxDst;
    DWORD cbwfxDst;
} ACMDRVFORMATSUGGEST;
```

The ACMDRVFORMATSUGGEST structure contains client-specified input arguments to the **acmFormatSuggest** function. The ACM fills in this structure with the client's input arguments and passes it to an ACM driver with an ACMDM_FORMAT_SUGGEST message. ACMDRVFORMATSUGGEST is defined in *msacmdrv.h*.

Members

cbStruct
Size, in bytes, of the ACMDRVFORMATSUGGEST structure.

fdwSuggest
Contains restriction flags that limit the possible destination formats. Can contain any combination of the following flags.

Flag	Meaning
ACM_FORMATSUGGESTF_WFORMATTAG	The **wFormatTag** member of the WAVEFORMATEX structure pointed to by **pwfxDst** contains a format tag. The driver can only suggest a destination format that is associated with the specified format tag.
ACM_FORMATSUGGESTF_NCHANNELS	The **nChannels** member of the WAVEFORMATEX structure pointed to by **pwfxDst** contains a channel value. The driver can only suggest a destination format whose channel value matches the specified value.
ACM_FORMATSUGGESTF_NSAMPLESPERSEC	The **nSamplesPerSec** member of the WAVEFORMATEX structure pointed to by **pwfxDst** contains a sample rate. The driver can only suggest a destination format whose sample rate matches the specified rate.
ACM_FORMATSUGGESTF_WBITSPERSAMPLE	The **nBitsPerSample** member of the WAVEFORMATEX structure pointed to by **pwfxDst** contains a sample size. The driver can only suggest a destination format whose sample size matches the specified size.

pwfxSrc
Pointer to a WAVEFORMATEX structure describing the source format.

cbwfxSrc
Size, in bytes, of the WAVEFORMATEX structure pointed to by **pwfxSrc**.

pwfxDst
Pointer to a WAVEFORMATEX structure to receive the suggested destination format

description. If flags in the **fdwSuggest** member are set, corresponding members of this structure contain client-specified values. The driver should fill in all empty structure members.

cbwfxDst
Size, in bytes, of the WAVEFORMATEX structure pointed to by **pwfxDst**. The driver cannot return an extended WAVEFORMATEX structure (see the **Comments** section) that is larger than this size.

Comments

The WAVEFORMATEX structures specified for source or destination formats might be extended structures defined for particular formats. (For example, see IMAADPCMWAVEFORMAT in *mmreg.h*.) Check the structure's **wFormatTag** member to determine the format type and hence the specific structure being passed.

For more information about format structures, see Defining Format Structures and Filter Structures.

ACMDRVOPENDESC

```
typedef struct {
    DWORD cbStruct;
    FOURCC fccType;
    FOURCC fccComp;
    DWORD dwVersion;
    DWORD dwFlags;
    DWORD dwError;
    LPCSTR pszSectionName;
    LPCSTR pszAliasName;
    DWORD dnDevNode;
} ACMDRVOPENDESC;
```

The ACMDRVOPENDESC structure is used by the ACM for passing information to an ACM driver, when the ACM sends the driver a DRV_OPEN message. ACMDRVOPENDESC is defined in *msacmdrv.h*.

Members

cbStruct
Size, in bytes, of the ACMDRVOPENDESC structure.

fccType
Contains a four-character code identifying the driver type. The driver must compare this value with ACMDRIVERDETAILS_FCCTYPE_AUDIOCODEC, which is defined in *msacm.h* to equal the string "audc". If the member contents does not match this string, the driver must fail the open request by specifying a **DriverProc** return value of zero.

fccComp
Not used. Defined to contain a four-character code identifying the driver sub-type.

dwVersion
Contains the ACM's version number. The version number's format is 0x*AABBCCCC*, where *AA* is the major version number, *BB* is the minor version number, and *CCCC* is the build number. This value is also returned by the ACM's **acmGetVersion** function, described in the Win32 SDK.

dwFlags
Contains flags. This member is identical to the *fdwOpen* argument passed to **acmDriverOpen**. No flags are currently defined.

dwError
Used by drivers to supply an error code. User-mode drivers are restricted to specifying a **DriverProc** return value of zero for all error types. To provide better error resolution, ACM drivers can specify an error code in this member, if they set the **DriverProc** function's return value to zero. The error code can be one of the MMSYSERR error codes defined in *mmsystem.h*, or one of the ACMERR error codes defined in *msacm.h*.

pszSectionName
Contains the registry key under which the driver's alias is stored. For more information, see Installing ACM Drivers.

pszAliasName
Contains the driver's alias. This is the driver's "msacm.*alias*" entry in the registry. For more information, see Installing ACM Drivers.

dnDevNode
Device node ID.

Comments

When the ACM calls an ACM driver's **DriverProc** entry point and specifies a DRV_OPEN message, it includes an ACMDRVOPENDESC structure as the *lParam2* parameter to **DriverProc**. The ACM sends a DRV_OPEN message when an application calls the **acmDriverOpen** function, which is described in the Win32 SDK. For additional information, see **DriverProc** in ACM Drivers.

ACM drivers do *not* always receive this structure when they receive a DRV_OPEN message. They only receive the structure if they are called by the ACM. Circumstances in which a driver is not called by the ACM are as follows:

- The driver might be called by a Control Panel applet for configuration purposes.
- The driver might be designed to be both an ACM driver and an audio device driver. When such a driver is called by *winmm.dll* for device operations, it does not receive the structure. (For more information, see Providing ACM Support in Device Drivers.)

ACMDRVSTREAMHEADER

```
typedef struct {
    DWORD cbStruct;
    DWORD fdwStatus;
    DWORD dwUser;
    LPBYTE pbSrc;
    DWORD cbSrcLength;
    DWORD cbSrcLengthUsed;
    DWORD dwSrcUser;
    LPBYTE pbDst;
    DWORD cbDstLength;
    DWORD cbDstLengthUsed;
    DWORD dwDstUser;
    DWORD fdwConvert;
    LPACMDRVSTREAMHEADER padshNext;
    DWORD fdwDriver;
    DWORD dwDriver;
    DWORD fdwPrepared;
    DWORD dwPrepared;
    LPBYTE pbPreparedSrc;
    DWORD cbPreparedSrcLength;
    LPBYTE pbPreparedDst;
    DWORD cbPreparedDstLength;
} ACMDRVSTREAMHEADER;
```

The ACMDRVSTREAMHEADER structure describes a source buffer and a destination buffer associated with a conversion stream. The structure is used with the ACMDM_STREAM_PREPARE, ACMDM_STREAM_UNPREPARE, and ACMDM_STREAM_CONVERT messages. ACMDRVSTREAMHEADER is defined in *msacmdrv.h*.

Members

cbStruct
Contains the size, in bytes, of the ACMDRVSTREAMHEADER structure.

fdwStatus
Contains status flags. The defined flags are as follows:

Flag	Meaning
ACMSTREAMHEADER_STATUSF_DONE	Indicates that a conversion is complete. For synchronous conversion, the ACM sets this flag when the driver returns from an ACMDM_STREAM_CONVERT message. For asynchronous drivers, the driver sets this flag after the data has been converted. The ACM clears the flag before sending each ACMDM_STREAM_CONVERT message.
ACMSTREAMHEADER_STATUSF_PREPARED	Indicates that the data buffers have been prepared. This flag is set by the ACM, regardless of whether the driver or the ACM prepared the buffers. See ACMDM_STREAM_PREPARE and ACMDM_STREAM_UNPREPARE.
ACMSTREAMHEADER_STATUSF_INQUEUE	Used by the driver, during asynchronous conversions, to indicate the structure has been queued for conversion. The driver is responsible for setting and clearing this flag. See ACMDM_STREAM_CONVERT.

dwUser
Contains information supplied by a client for its own use.

pbSrc
Pointer to a source buffer. For an ACMDM_STREAM_CONVERT message, this buffer contains the data to be converted.

cbSrcLength
Length, in bytes, of the source buffer pointed to by **pbSrc**. For the ACMDM_STREAM_PREPARE and ACMDM_STREAM_UNPREPARE messages, this value represents the maximum source buffer size. For ACMDM_STREAM_CONVERT, this value represents the length of the data in the buffer.

cbSrcLengthUsed
Length, in bytes, of source data that has been converted. This value is set by the driver to indicate the number of bytes in the source buffer that the driver actually converted. The value cannot be greater than the value in **cbSrcLength**.

dwSrcUser
Contains information supplied by a client for its own use.

pbDst
Pointer to a destination buffer. For an ACMDM_STREAM_CONVERT message, the driver fills this buffer with converted data.

cbDstLength
Length, in bytes, of the destination buffer pointed to by **pbDst**.

cbDstLengthUsed
Length, in bytes, of destination data that has been converted. This value is set by the driver to indicate the number of converted bytes that it has placed in the destination buffer. The value cannot be greater than the value in **cbDstLength**. If the conversion fails, the driver must set this value to zero.

dwDstUser
Contains information supplied by a client for its own use.

fdwConvert
Contains one of the following values:

- For the ACMDM_STREAM_PREPARE message, the value specified as the **acmStreamPrepareHeader** function's *fdwPrepare* argument. (Not used.)
- For the ACMDM_STREAM_UNPREPARE message, the value specified as the **acmStreamUnprepareHeader** function's *fdwUnprepare* argument. (Not used.)

- For the ACMDM_STREAM_CONVERT message, the value specified as the **acmStreamConvert** function's *fdwConvert* argument.

For ACMDM_STREAM_CONVERT, the following flags are defined.

Flags	Meaning
ACM_STREAMCONVERTF_BLOCKALIGN	Indicates that only whole blocks of source data should be converted. The size of a block is obtained from the source format's WAVEFORMATEX structure (see ACMDRVSTREAMINSTANCE). If the flag is set, the driver should not convert extra bytes that do not make up a whole block. Generally, clients set this flag for all buffers in a conversion stream except the last one.
ACM_STREAMCONVERTF_START	Indicates that the driver should re-initialize stream instance data, such as predictor coefficients or scale factors, to default starting values. This flag can be specified with the ACM_STREAMCONVERTF_END flag.
ACM_STREAMCONVERTF_END	Indicates that the driver should return end-of-stream data, such as tail end echo data for an echo filter, in the destination buffer, along with data converted from the source buffer. This flag can be specified with the ACM_STREAMCONVERTF_START flag.

padshNext
Pointer to another ACMDRVSTREAMHEADER structure. An asynchronous driver can use this member for creating a queue of pending conversion requests. The ACM clears the member prior to sending an ACMDM_STREAM_PREPARE or ACMDM_STREAM_CONVERT message.

fdwDriver
Contains stream instance information supplied by the driver for its own use. This member is intended for storing driver-defined flags, but you can use it for any purpose you wish. The ACM clears this member prior to sending an ACMDM_STREAM_PREPARE message. Otherwise its value is preserved from one message to the next.

dwDriver
Contains stream instance information supplied by the driver for its own use. You can use this member for any purpose you wish. The ACM clears this member prior to sending an ACMDM_STREAM_PREPARE message. Otherwise its value is preserved from one message to the next.

fdwPrepared
Used by ACM only. Contains the *fdwPrepared* argument to the **acmStreamPrepareHeader** function.

dwPrepared
Used by ACM only. Contains the *has* argument to the **acmStreamPrepareHeader** function.

pbPreparedSrc
Used by ACM only. Contains the address of source the buffer supplied with the **acmStreamPrepareHeader** function.

cbPreparedSrcLength
Used by ACM only. Contains the length of the source buffer supplied with the **acmStreamPrepareHeader** function.

pbPreparedDst
Used by ACM only. Contains the address of the destination buffer supplied with the **acmStreamPrepareHeader** function.

cbPreparedDstLength
Used by ACM only. Contains the length of the destination buffer supplied with the **acmStreamPrepareHeader** function.

ACMDRVSTREAMINSTANCE

```
typedef struct {
    DWORD cbStruct;
    LPWAVEFORMATEX pwfxSrc;
    LPWAVEFORMATEX pwfxDst;
    LPWAVEFILTER pwfltr;
    DWORD dwCallback;
    DWORD dwInstance;
    DWORD fdwOpen;
    DWORD fdwDriver;
    DWORD dwDriver;
    HACMSTREAM has;
} ACMDRVSTREAMINSTANCE;
```

The ACMDRVSTREAMINSTANCE structure describes an instance of a conversion stream. ACMDRVSTREAMINSTANCE is defined in *msacmdrv.h*.

Members

cbStruct
Contains the size, in bytes, of the ACMDRVSTREAMINSTANCE structure.

pwfxSrc
Pointer to a WAVEFORMATEX structure that defines the source format for a conversion stream. (WAVEFORMATEX is described in the Win32 SDK.)

pwfxDst
Pointer to a WAVEFORMATEX structure that defines the destination format for a conversion stream.

pwfltr
Pointer to an optional WAVEFILTER structure that defines a filter to be used on a conversion stream. This member is NULL if the application has not specified a filter. (WAVEFILTER is described in the Win32 SDK.)

dwCallback
Contains the application-specified *dwCallback* argument to the **acmStreamOpen** function.

dwInstance
Contains the application-specified *dwInstance* argument to the **acmStreamOpen** function.

fdwOpen
Contains the application-specified *dwOpen* argument to the **acmStreamOpen** function, which consists of a set of flags. (For flag descriptions, refer to **acmStreamOpen** in the Win32 SDK.)

fdwDriver
Contains driver-defined stream instance information. While intended for storing flag values, an ACM driver can use this member to store any instance-specific DWORD value. Because the same ACMDRVSTREAMINSTANCE structure is passed with all stream messages associated with a particular stream instance, the stored value can be read or modified each time a stream message is received, and the last saved value will be available the next time a stream message is received.

dwDriver
Contains driver-defined stream instance information. An ACM driver can use this member for storing any instance-specific DWORD value, such as a pointer to a local, dynamically allocated structure. Because the same ACMDRVSTREAMINSTANCE structure is passed with all stream messages associated with a particular stream instance, the stored value can be read or modified each time a stream message is received, and the last saved value will be available the next time a stream message is received.

has
Contains the ACM-defined client handle to the open conversion stream.

Comments

The ACM allocates an ACMDRVSTREAMINSTANCE structure each time an application calls

acmStreamOpen. This ACMDRVSTREAMINSTANCE structure is then passed to the driver with all stream messages associated with a particular stream instance. Information in the structure does not change, so if a driver validates information within the structure when it receives an ACMDM_STREAM_OPEN message, it does not have to validate the information again when it receives subsequent messages for the same stream instance.

ACMDRVSTREAMSIZE

```
typedef struct {
    DWORD cbStruct;
    DWORD fdwSize;
    DWORD cbSrcLength;
    DWORD cbDstLength;
} ACMDRVSTREAMSIZE;
```

The ACMDRVSTREAMSIZE structure contains information needed by an ACM driver to respond to an ACMDM_STREAM_SIZE message. ACMDRVSTREAMSIZE is defined in *msacmdrv.h*.

Members

cbStruct
Size, in bytes, of the ACMDRVSTREAMSIZE structure.

fdwSize
Contains one of the following flags, indicating the query type.

Flag	Meaning
ACM_STREAMSIZEF_SOURCE	Indicates the client has specified the size, in bytes, of a source buffer in the **cbSrcLength** member. The driver should return the required destination buffer length in **cbDstLength**.
ACM_STREAMSIZEF_DESTINATION	Indicates the client has specified the size, in bytes, of a destination buffer in the **cbDstLength** member. The driver should return the required source buffer length in **cbSrcLength**.

cbSrcLength
Size, in bytes, of the source buffer. The flag value specified in **fdwSize** indicates whether this value is supplied by the client or by the driver.

cbDstLength
Size, in bytes, of the destination buffer. The flag value specified in **fdwSize** indicates whether this value is supplied by the client or by the driver.

Video Capture Device Drivers

The following topics explain how to write video capture drivers for Windows NT®:

- Introduction to Video Capture Drivers
- Designing a User-Mode Video Capture Driver
- Designing a Kernel-Mode Video Capture Driver
- Video Capture Driver Reference

For a general discussion of multimedia device drivers, refer to Introduction to Multimedia Drivers.

Introduction to Video Capture Drivers

The following topics provide an introduction to video capture drivers:

Capabilities of Video Capture Drivers

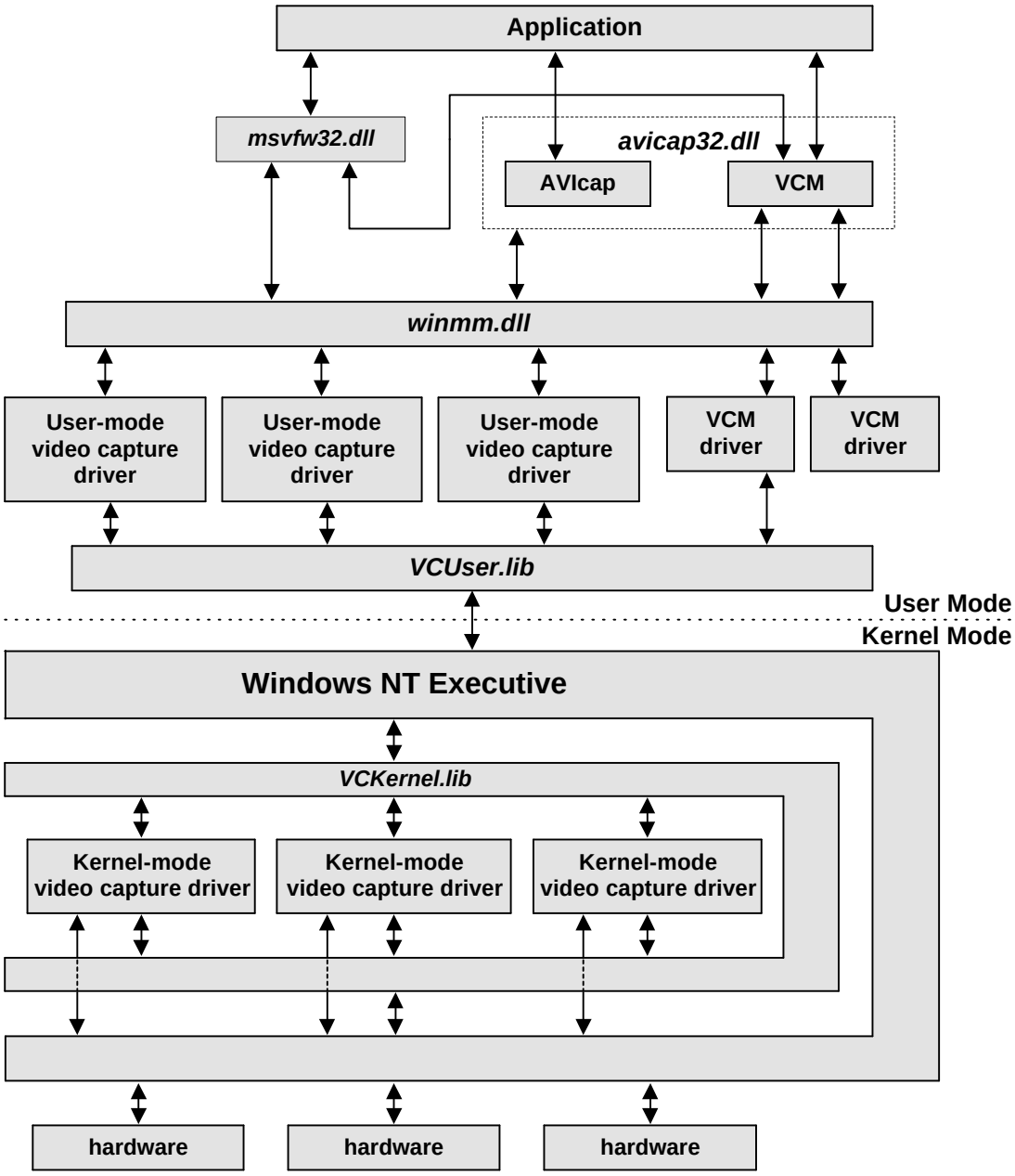
Video capture device drivers can capture video input images that device hardware has stored in a frame buffer. They can pass these images to client applications as device-independent bitmaps (DIBs). If the hardware allows, they can display captured images as an overlay on an output display device.

APIs provided by the Microsoft Video for Windows Development Kit and the AVI capture (AVIcap) window class allow applications to capture either single video images or streams of images, and to view images on an overlay display as they are captured. The AVIcap window class is described in the Win32 SDK.

The Video Compression Manager allows applications to play back previously recorded bitmap images. For more information about the Video Compression Manager and its drivers, see Video Compression Manager Drivers.

Video Capture Software Components

The following diagram illustrates the relationship of the major Windows NT video software components.



The components in the diagram include:

Application
Any user-mode, Win32-based application that creates an AVIcap window (described in the Win32 SDK), calls the video API functions (described in the Video for Windows Development Kit), or calls the Video Compression Manager functions (also described in the Win32 SDK).

msvfw32.dll
Microsoft Video for Windows dynamic-link library. Exports the video API functions described in the Video for Windows Development Kit. Also exports the MCIWnd window class and DRAWDIB functions, described in the Win32 SDK. Additionally, this library includes the Video Compression Manager. For more information about the Video Compression Manager, see Video Compression Manager Drivers.

avicap32.dll
Dynamic-link library supporting the AVI capture window class.

winmm.dll
Dynamic-link library that exports **SendDriverMessage**, which calls a user-mode driver's **DriverProc** function. See *winmm.dll*.

User-mode video capture drivers
Dynamic-link libraries that communicate with kernel-mode drivers.

VCM drivers
Dynamic-link libraries that compress or decompress video data and either return it to the caller or send it to a kernel-mode driver.

VCUser.lib
Library used as a basis for user-mode video capture drivers. For details, see Using *VCUser.lib*.

VCKernel.lib
Library used as a basis for kernel-mode video capture drivers. For details, see Using *VCKernel.lib*.

Kernel-mode video capture drivers
Kernel-mode code that communicates with the Windows NT Executive in order to access device hardware.

Sample Video Capture Drivers

The Windows NT DDK includes the source code for the following video capture drivers and dynamic-link libraries:

Drivers or Libraries	Location of Source Files
Truevision Bravado video capture drivers	*ddk**src**mmedia**vidcap**bravado*
Video Spigot video capture drivers	*ddk**src**mmedia**vidcap**spigot*
Microsoft YUV compressor/decompressor	*ddk**src**mmedia**vidcap**msyuv*
User-mode video capture driver library (For details, see Using *VCUser.lib*.)	*ddk**src**mmedia**vidcap**vcuser*
Kernel-mode video capture driver library (For details, see Using *VCKernel.lib*.)	*ddk**src**mmedia**vidcap**vckernel*

For the Bravado and Spigot driver samples, code for both the user-mode and the kernel-mode driver is provided. Under the listed driver directory, a *dll* subdirectory contains the user-mode driver sources, and a *driver* subdirectory contains the kernel-mode driver sources.

The Truevision Bravado video capture drivers (*bravado.dll* and *bravado.sys*) and hardware support image capture, scaling, and overlay. They do not support clipping. The hardware accepts compressed YUV-formatted data for output.

The Video Spigot video capture drivers (*spigot.dll* and *spigot.sys*) and hardware support image capture and scaling. They do not support overlay or clipping.

The Microsoft YUV compressor/decompressor (codec) is a user-mode driver (*msyuv.dll*) that can draw compressed YUV-formatted video data by sending it to *bravado.sys*, the kernel-mode driver for the Truevision Bravado hardware. For more information about video codecs, see Video Compression Manager Drivers.

All of the sample user-mode drivers, including the YUV codec (*msyuv.dll*), are built by using *VCUser.lib*.

All of the sample kernel-mode drivers are built by using *VCKernel.lib*.

Designing a User-Mode Video Capture Driver

User-mode video capture drivers are implemented as dynamic-link libraries. This section contains the following topics to assist you in designing a user-mode video capture driver:

- **DriverProc** in User-Mode Video Capture Drivers
- User-Mode Video Capture Driver Messages
- Introduction to Video Channels

DriverProc in User-Mode Video Capture Drivers

Like all other Win32-based user-mode drivers, user-mode video capture drivers must export a **DriverProc** entry point that recognizes the standard driver messages. Video capture driver clients, such as *msvfw32.dll* and *avicap32.dll*, send messages to video capture drivers by calling **SendDriverMessage**, which is exported by *winmm.dll* and described in the Win32 SDK.

When a user-mode video capture driver receives a DRV_OPEN message from *avicap32.dll* or *msvfw32.dll*, it also receives a pointer to a VIDEO_OPEN_PARMS structure. For more information about the use of this structure, see Opening Video Channels.

In addition to supporting the standard messages, the **DriverProc** entry point for user-mode video capture drivers must support a set of user-mode video capture driver messages.

User-Mode Video Capture Driver Messages

The following table lists the messages that the **DriverProc** function in a user-mode video capture driver can receive, along with the operation the driver performs when it receives each message. Message definitions are contained in *msviddrv.h* and *msvideo.h*.

Message	Operation Performed by Driver
DVM_CONFIGURESTORAGE	Saves or restores configuration information.
DVM_DIALOG	Displays a dialog box to obtain configuration information.
DVM_DST_RECT	Sets or retrieves destination rectangle parameters.
DVM_FORMAT	Sets or retrieves video-capture format parameters.
DVM_FRAME	Transfers data from the frame buffer.
DVM_GET_CHANNEL_CAPS	Returns channel capabilities.
DVM_GETERRORTEXT	Returns the text string associated with an error number.
DVM_GETVIDEOAPIVER	Returns the video API version used by the driver.
DVM_PALETTE	Sets or retrieves a logical palette.
DVM_PALETTERGB555	Sets a logical palette and an RGB555 translation table.
DVM_SRC_RECT	Sets or retrieves source rectangle parameters.
DVM_STREAM_ADDBUFFER	Adds a data buffer to a capture stream.
DVM_STREAM_FINI	Terminates a capture stream.
DVM_STREAM_GETERROR	Returns a stream's error status.
DVM_STREAM_GETPOSITION	Returns the current position within a

	capture stream.
DVM_STREAM_INIT	Initializes a capture stream.
DVM_STREAM_PREPAREHEADER	Prepares a data buffer.
DVM_STREAM_RESET	Stops and resets a capture stream.
DVM_STREAM_START	Starts a capture stream.
DVM_STREAM_STOP	Stops a capture stream.
DVM_STREAM_UNPREPAREHEADER	Removes preparation from a data buffer.
DVM_UPDATE	Updates a screen overlay area.

Introduction to Video Channels

Video capture device drivers define the four logical video channels shown in the following table. Each channel represents a portion of the data path between video hardware and system memory.

Channel	Constant	Definition
External In	VIDEO_EXTERNALIN	Data path from an external video source, such as a TV, VCR, or camera, into a frame buffer.
Video In	VIDEO_IN	Data path from the frame buffer to system memory.
Video Out	VIDEO_OUT	Data path from system memory to the frame buffer.
External Out	VIDEO_EXTERNALOUT	Data path from the frame buffer to an output display, such as an overlay window on a user's screen.

User-mode video capture drivers implement the concept of logical video channels, and clients can open the channels they need to create a complete input and/or output data path. Logical channels do *not* exist within kernel-mode drivers.

The sample video capture drivers provided with the Windows NT DDK support the VIDEO_EXTERNALIN, VIDEO_IN, and VIDEO_EXTERNALOUT channels. Support for these channels means that users employing the AVIcap window class or the Video For Windows API can capture video images from an external source, store the images in memory or a file, and view the captured images as they are received. The sample user-mode video capture drivers do not support the VIDEO_OUT channel, so users cannot use them to play back recorded images. Instead, user applications call the Video Compression Manager's API to send drawing requests to Video Compression Manager drivers.

For more information about video channels, see Opening Video Channels and Configuring Video Channels.

Opening Video Channels

A client must independently open each video channel that it intends to use. Therefore, a user-mode video capture driver receives a separate DRV_OPEN message for each channel. When a driver receives a DRV_OPEN message, it must check the **dwFlags** member of the accompanying VIDEO_OPEN_PARMS structure to determine which video channel to open. The member's value can be one of the following constants, which are defined in *msvideo.h*:

- VIDEO_EXTERNALIN
- VIDEO_IN
- VIDEO_OUT
- VIDEO_EXTERNALOUT

Besides testing for these constants, the driver must also return the detected constant as the return

value for **DriverProc**, as illustrated in the **vidOpen** function within each sample user-mode driver. This constant then becomes the value passed to the driver as the *dwDriverID* parameter, when a client makes subsequent calls to **DriverProc** to send the driver user-mode video capture driver messages.

Configuring Video Channels

User-mode video capture drivers can allow users to specify configuration parameters for each video channel. After a client opens a channel, it can send a DVM_DIALOG message for the opened channel. In response to the message, the driver displays a dialog box that allows the user to specify parameters for the channel type. Typically, the driver provides a different dialog box for each channel type. If the driver does not support a channel type, or the driver does not allow user modification to the channel's parameters, it can return the DV_ERR_NOTSUPPORTED error code.

The sample video capture driver, *bravado.dll*, uses dialog boxes to obtain the following channel configuration information.

Channel	Information Obtained
VIDEO_EXTERNALIN	Hue, video source standard, connector number, cable format.
VIDEO_IN	Data format, destination image size. (Also see Setting the Video Data Format.)
VIDEO_OUT	Not supported.
VIDEO_EXTERNALOUT	Saturation, brightness, contrast, red, green, blue.

User-mode drivers should store channel configuration parameters for each user by writing them into the registry, under the path **HKEY_CURRENT_USER \Software \Microsoft \Multimedia \Video Capture ***DriverName,* where *DriverName* represents the name of your driver. For more information about obtaining and storing channel configuration parameters, see Configuring Video Channels, Using *VCUser.lib*.

The driver should provide default values for all channel configuration parameters, and it should use the default values if it has not stored values for the current user in the previously listed registry path.

Changing the format can alter the dimensions of the active portion of the frame buffer, and can also affect bit depth and color space representation. Additionally, changing between NTSC and PAL video standards can affect image dimensions. Therefore, clients typically send a DVM_FORMAT message to request the current format after they have sent a DVM_DIALOG message for the VIDEO_EXTERNALIN channel.

Clients can also request a driver to store a channel's configuration parameters in a file, or to restore the parameters from a previously created file, by sending a DVM_CONFIGURESTORAGE message.

Setting the Video Data Format

Developers of video capture clients expect video capture data to be stored as device independent bitmaps (DIBs). DIB contents are described by BITMAPINFO and BITMAPINFOHEADER structures, which are discussed in the Win32 SDK. A client can specify a data format, or query the driver for the current format, by sending the driver a DVM_FORMAT message and including the address of a BITMAPINFOHEADER structure.

You can also allow users to modify the format. Typically, you include format options in the dialog boxes associated with VIDEO_IN and VIDEO_OUT channels, and your driver displays these dialog boxes when it receives DVM_DIALOG messages. For more information, see Configuring Video Channels.

The sample video capture drivers do not support DVM_FORMAT for the VIDEO_OUT channel,

because the VIDEO_OUT channel is handled by Video Compression Manager drivers.

Your driver should provide default values for all format variables. These default values should represent a format that can be handled most efficiently by the video capture hardware. The driver should use the default values if client-specified (via DVM_FORMAT) or user-specified (via DVM_DIALOG) values are not available.

Following is a typical scenario in which a client modifies the format because of limited system storage space:

1. Prior to opening an input stream, a client sends a DVM_FORMAT message, with the VIDEO_CONFIGURE_GET flag set, to determine the current format.
2. The client calculates the amount of storage needed to store a captured video stream if the current format is used.
3. The client attempts to allocate enough storage to save the stream. The allocation fails because there is not enough storage space available.
4. The client sends a DVM_FORMAT message to change the format to one that requires less storage space.
5. The client again attempts to allocate enough storage to save the stream. This time the allocation succeeds.
6. The client opens the stream and saves the input data. (For details, see Transferring Video Capture Data.)

Setting Source and Destination Rectangles

Video capture drivers can support the use of source and destination rectangles. A rectangle's typical use is dependent on the channel for which it is defined, as illustrated by the following table.

Channel	Source Rectangle	Destination Rectangle
VIDEO_EXTERNALIN	Specifies the portion of the analog image to be digitized.	Specifies the portion of the frame buffer to receive digitized input data.
VIDEO_IN	Specifies the portion of the frame buffer to be copied to the client.	Not applicable.
VIDEO_OUT	Not applicable.	Specifies the portion of the frame buffer to copy client data into.
VIDEO_EXTERNALOUT	Specifies the portion of the frame buffer to display inside the overlay area.	Specifies the portion of the display device to use as the overlay area.

A driver does not have to support all rectangles for all channels. Two messages, DVM_SRC_RECT and DVM_DST_RECT, are provided to allow clients to set and query the size of rectangles. Drivers that define rectangles on some or all channels must support these two messages. Even if your driver does not allow client modification of rectangle sizes, it should allow the client to send these messages to determine a rectangle's predefined size. Rectangles are always specified using screen coordinates, which are described in the Win32 SDK.

After a client changes a VIDEO_EXTERNALOUT rectangle's coordinates, it typically sends a DVM_UPDATE message to request the driver to update the display.

Clients send the DVM_GET_CHANNEL_CAPS message to determine the rectangular capabilities that a driver provides for a particular channel. In response to this message, your driver indicates its overlay, clipping, and scaling capabilities, along with allowable re-sizing parameters.

Setting Palettes

Video capture drivers can allow clients to specify a logical color palette and, optionally, an RGB555-format translation table. Two messages, DVM_PALETTE and DVM_PALETTERGB555, are defined.

The DVM_PALETTE message allows a client to specify a logical palette or to obtain the current palette from the driver. When a client specifies a logical palette, the driver typically creates a translation table by using the **CreatePalette** and **GetNearestPaletteIndex** functions described in the Win32 SDK, as illustrated in the sample video capture drivers. By sending a DVM_PALETTERGB555 message, a client can specify both a palette and an RGB555 translation table.

Drivers use the translation tables when converting frame buffer data into DIBs. (The palette location corresponding to an RGB color is found by using an RRRRRGGGGGBBBBB binary value as an index into the translation table, where the five most significant bits of each color component are used to create the index).

Drivers typically create a default palette when processing the DRV_OPEN message.

For more information about color palettes, see the Win32 SDK.

Transferring Video Capture Data

The APIs provided for capturing video images enable clients to receive single captured frames and to receive streams of frames that are captured at a client-specified rate. The next section discusses transferring single frames, and the following section explains the algorithm for transferring streams of captured data.

Transferring Single Frames

Application developers sometimes need to transfer single frames of video information. For instance, a developer might want to record animated sequences that are generated one frame at a time, or to display a single image, such as a photograph.

Clients can transfer single frames of video information by sending the DVM_FRAME message, along with the address of a VIDEOHDR structure. Clients specify the VIDEO_IN channel to obtain a frame from the device's frame buffer, or the VIDEO_OUT channel to send a frame to the frame buffer.

If the channel is VIDEO_IN, the client includes the address of an empty data buffer. By calling the kernel-mode driver, the user-mode driver transfers the contents of the device's frame buffer into the data buffer. If the channel is VIDEO_OUT, the client includes the address of a filled data buffer. The user-mode driver sends the data buffer's contents to the kernel-mode driver, which in turn transfers the contents to the device's frame buffer. The sample video capture drivers support single frame transfers only on the VIDEO_IN channel.

When handling single-frame transfers, drivers must take into account current settings for video format, source or destination rectangles, and color palettes.

Transferring Streams of Captured Data

Video capture drivers must provide the capability of handling video capture input streams. Video capture drivers do *not* support output streams. Output streams are handled by Video Compression Manager drivers.

To transfer a stream of captured video input data, a client must:

1. Establish settings for video format, source or destination rectangles, and color palettes.
2. Initialize the appropriate video channels.
3. Allocate data buffers, prepare them for use, and pass them to the driver to be filled.
4. Request the driver to start filling and returning the data buffers.
5. Copy data from the buffers and pass them back to the driver for re-use.

6. Request the driver to end the transfer operation.

The first streaming message a user-mode video capture driver should receive a is DVM_STREAM_INIT, along with a channel specification. The following table lists the operations that a driver should perform, based on the specified channel.

Channel	Operation to Perform for DMV_STREAM_INIT Message
VIDEO_EXTERNALIN	Set up hardware to allow capture operations.
VIDEO_IN	Save client-specified capture rate and event notification information.
VIDEO_OUT	Not used.
VIDEO_EXTERNALOUT	Set up hardware to allow overlay operations.

Clients dynamically allocate buffers to receive captured data. Before a client can use a data buffer in a streaming operation, it must pass the buffer's address to the user-mode driver by calling DVM_STREAM_PREPAREHEADER, so the driver can prepare the buffer for use. The client then sends a DVM_STREAM_ADDBUFFER message for each prepared buffer, which requests the driver to add the buffer's address to a queue of buffers waiting to be filled.

To start the input operation, the clients sends a DVM_STREAM_START message. The user-mode driver passes the request to the kernel-mode driver, which causes input operations to begin, typically by enabling device interrupts so that an interrupt occurs each time the frame buffer has been filled. Each time the frame buffer is filled, its contents are copied into the next available client buffer. The user-mode driver then sends the client an MM_DRVM_DATA callback message. The client copies the data from the buffer and re-adds the buffer to the user-mode driver's buffer queue by sending another DVM_STREAM_ADDBUFFER message. The client can send a DVM_STREAM_GETERROR message to determine if the driver has dropped any captured frames because of a lack of available buffers.

When the client is ready to stop the input stream, it sends a DVM_STREAM_STOP message. It can also send DVM_STREAM_RESET, which indicates to the user-mode driver that it should not fill any remaining data buffers. The client can then send a DVM_STREAM_UNPREPAREHEADER message for each buffer and de-allocate the buffers. Finally, the client sends a DVM_STREAM_FINI message to close the streaming session.

For all stream messages except DVM_STREAM_INIT and DVM_STREAM_FINI, the client-specified channel should always be VIDEO_IN.

Notifying Clients from Video Capture Drivers

User-mode video capture drivers are responsible for notifying clients upon the completion of certain driver events. When a client sends a DVM_STREAM_INIT message, it indicates the type of notification, if any, it expects to receive. A client can specify any of the following notification targets:

- A callback function
- A window handle

User-mode drivers notify clients by calling the **DriverCallback** function in *winmm.dll*. This function delivers a message to the client's notification target. The function also delivers message parameters, if the target type accepts parameters.

Video capture drivers must send MM_DRVM_OPEN, MM_DRVM_CLOSE, MM_DRVM_DATA, and MM_DRVM_ERROR messages to clients.

If you are using *VCUser.lib* to develop your driver, you do not have to call **DriverCallback** to send callback messages. Code within *VCUser.lib* handles delivery of callback messages for you, if you call **VC_StreamInit** when processing the DVM_STREAM_INIT message.

Using *VCUser.lib*

This section contains the following topics that explain how to use the user-mode video capture library, *VCUser.lib*:

Introduction to *VCUser.lib*

The *VCUser.lib* library exports a set of functions that provide a convenient interface between a user-mode video capture driver and its kernel-mode counterpart. If you link your user-mode driver with *VCUser.lib*, your driver can easily communicate with a kernel-mode driver that is using *VCKernel.lib*, the companion library for kernel-mode video capture drivers.

The *VCUser.lib* library is typically used by:

- User-mode video capture drivers that support the user-mode video capture driver messages.
- User-mode video codecs that support output operations by making use of decompressing hardware. An example driver is the sample *msyuv.dll* driver provided with the Windows NT DDK. For more information about video codecs, see Video Compression Manager Drivers.

Typically, *VCUser.lib* library functions communicate with a kernel-mode driver by calling the **DeviceIoControl** function, described in the Win32 SDK, to send I/O control codes to the driver. Your driver typically calls *VCUser.lib* functions in response to the receipt of standard driver messages or user-mode video capture driver messages.

To use *VCUser.lib* with your user-mode driver

1. Include *vcuser.h* and *vcstruct.h* in your driver.
2. Link your driver with *VCUser.lib*.

Installing and Configuring a Kernel-Mode Driver, Using *VCUser.lib*

User-mode drivers are responsible for installing their associated kernel-mode drivers and for providing kernel-mode drivers with configuration parameter values. The first *VCUser.lib* function that a user-mode driver should call is **VC_OpenProfileAccess**. This function, which specifies the kernel-mode driver's name and sets up storage for access to the Service Control Manager and the registry, should be called when the driver receives a DRV_LOAD message.

Next, the driver should call **VC_ConfigAccess**. If the client has Administrators privilege and is thus allowed to install drivers, this function establishes a connection to the Service Control Manager. The driver should call the function when it receives a DRV_QUERYCONFIGURE message.

When the driver receives a DRV_CONFIGURE message, it should call **VC_InstallDriver** to install the kernel-mode driver. This function allows you to specify a callback function, from which your driver can make calls to **VC_WriteProfile** to store driver configuration parameters in the registry. If your kernel-mode driver encounters installation errors, it should write an error code into the registry. Your user-mode driver can call **VC_ReadProfile** to check the error code or to obtain the stored configuration information.

Upon receipt of a DRV_REMOVE message, your driver should call **VC_RemoveDriver** to remove

the kernel-mode driver's installation. The last message your driver can receive is DRV_FREE, at which time it should call **VC_CloseProfileAccess** to remove its connection to the Service Control Manager.

Opening and Closing a Device, Using *VCUser.lib*

When your user-mode driver receives a DRV_OPEN message it should call **VC_OpenDevice** to open a device. The following two restrictions apply to opening a video capture device:

1. A user-mode video capture driver can only allow one instance of a device to be open at time.
2. A kernel-mode video capture driver can only support one hardware device.

There are two ways to specify a device when calling **VC_OpenDevice** — by device name or by index number. Because a kernel-mode driver can only support one hardware device, you specify a device's name by specifying the kernel-mode driver's name, such as "bravado", without appending a device number.

Video capture device objects are assigned names of vidcap0, vidcap1, vidcap2, and so on. By specifying an index number instead of a device name, you can open any video capture device without knowing its name. For example, specifying an index of "1" opens the device "vidcap1". The sample video capture driver *msyuv.dll* uses this index number in order to search for a device that supports a specific data format. For more information about device objects, see the *Kernel-Mode Drivers Design Guide*.

Your driver receives a DRV_OPEN message for each video channel (VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, VIDEO_EXTERNALOUT) that the client will be using, but **VC_OpenDevice** should only be called once.

If your hardware provides overlay capabilities, then prior to calling **VC_OpenDevice** your driver must determine characteristics of the user's display device, such as the display's horizontal and vertical resolutions, and bits per pixel. The kernel-mode driver needs this information when constructing and positioning an overlay image. The user-mode driver can obtain characteristics of the user's display device by calling **GetDeviceCaps**, which is described in the Win32 SDK. Your user-mode driver should call **VC_WriteProfile** to store the characteristics in the registry, where your kernel-mode driver can access them when needed.

Upon receipt of a DRV_CLOSE message, your driver should call **VC_CloseDevice**. Your driver will receive a DRV_CLOSE message for each open video channel, but **VC_CloseDevice** should only be called once.

Configuring Video Channels, Using *VCUser.lib*

Video channel configuration parameters should be stored in the registry, individually for each user. To store and retrieve a user's channel configuration parameters, a user-mode driver can call **VC_WriteProfileUser** and **VC_ReadProfileUser**. The driver generally obtains channel configuration parameters, and stores them in the registry, when it receives DVM_DIALOG messages.

The *VCUser.lib* library provides three functions for passing channel configuration parameters to the kernel-mode driver. The **VC_ConfigDisplay** function is used for passing parameters associated with the output display. The **VC_ConfigFormat** function is used for passing data format parameters. The **VC_ConfigSource** function is used for passing parameters associated with the input source. For more information about channel configuration parameters, see Configuring Video Channels.

Supporting Overlay Capabilities, Using *VCUser.lib*

To determine the hardware's overlay capabilities, a user-mode driver can call **VC_GetOverlayMode**. This function indicates whether or not overlay capabilities are provided, and also indicates the type of support provided for color keying and rectangle specifications. Based on the returned information, the driver can call **VC_SetKeyColourPalIdx**, **VC_SetKeyColourRGB**, or **VC_GetKeyColour** to set or retrieve the overlay's key color. The

driver should call **VC_SetOverlayRect** to specify the area to be overlaid and **VC_SetOverlayOffset** to specify which portion of the frame buffer to display in the overlay area. To enable the hardware's overlay capabilities, the driver calls **VC_Overlay**.

Capturing Video Data, Using *VCUser.lib*

To enable the hardware's video capture capability, your driver should call **VC_Capture**. To obtain a single frame of capture data, your driver can call **VC_Frame**.

To continuously obtain captured data as a stream, *VCUser.lib* provides a set of input stream functions, which your driver can call when it receives user-mode video capture driver messages that specify stream operations. The input stream functions are:

VC_StreamInit
Initializes *VCUser.lib*'s capture streaming capabilities.

VC_StreamAddBuffer
Adds empty buffers to a buffer queue.

VC_StreamStart
Starts capturing frames and copying them into queued buffers, and notifying the client each time a buffer is filled.

VC_GetStreamPos
Returns the amount of time that has passed since the stream started.

VC_GetStreamError
Returns the number of times a frame was dropped because a buffer was not available.

VC_StreamStop
Stops capturing frames.

VC_StreamReset
Returns unused buffers to the client.

VC_StreamFini
Disables *VCUser.lib*'s capture streaming capabilities.

For more information about capturing data, see Transferring Video Capture Data.

Playing Back Captured Video Data, Using *VCUser.lib*

The *VCUser.lib* library provides a single function, **VC_DrawFrame**, for playing back capture data. If the **VC_GetOverlayMode** function indicates that the hardware provides overlay capabilities and supports the desired data format, a driver can call **VC_DrawFrame** to send a bitmap to the hardware's frame buffer for display in the overlay area.

Notifying Clients, Using *VCUser.lib*

User-mode drivers using *VCUser.lib* do not need to send callback messages to clients. When the driver calls **VC_StreamInit**, client-specified callback information is passed to *VCUser.lib* so that it can send callback messages at appropriate times.

For more information about callback messages, see Notifying Clients From Video Capture Drivers.

Designing a Kernel-Mode Video Capture Driver

Kernel-mode video capture drivers are responsible for accessing video capture hardware. They are implemented as services under the control of the Windows NT Service Control Manager. This section provides topics that explain implementing **DriverEntry** in a video capture driver and using *VCKernel.lib*.

DriverEntry in a Video Capture Driver

Like all kernel-mode multimedia drivers, kernel-mode video capture drivers must provide a

DriverEntry function for handling initialization and configuration operations. This **DriverEntry** for multimedia drivers is the first function called after a kernel-mode driver is loaded.

For more information about the contents of a video capture driver's **DriverEntry** function, see Initializing and Configuring a Driver, Using *VCKernel.lib*. The **DriverEntry** function for the sample video capture driver, *bravado.sys*, is located in *\src\mmedia\vidcap\bravado\driver\init.c*.

Using *VCKernel.lib*

This section contains the following topics that explain how to use the kernel-mode video capture library, *VCKernel.lib*:

- Introduction to *VCKernel.lib*
- Initializing and Configuring a Driver, Using *VCKernel.lib*
- Opening and Closing a Device, Using *VCKernel.lib*
- Configuring Video Channels, Using *VCKernel.lib*
- Accessing Video Capture Hardware, Using *VCKernel.lib*
- Handling Interrupts, Using *VCKernel.lib*
- Synchronizing Driver Activities, Using *VCKernel.lib*
- Supporting Overlay Capabilities, Using *VCKernel.lib*
- Capturing Video Data, Using *VCKernel.lib*
- Playing Back Captured Video Data, Using *VCKernel.lib*

Introduction to *VCKernel.lib*

The *VCKernel.lib* library exports a set of functions that provide a convenient interface between a kernel-mode video capture driver and its user-mode counterpart. If you link your kernel-mode driver with *VCKernel.lib*, your driver can easily respond to I/O request packets (IRPs) sent to it by a user-mode driver that is using *VCUser.lib*, the companion library for user-mode video capture drivers. (For more information about IRPs, see the *Kernel-Mode Drivers Design Guide*.)

The *VCKernel.lib* library provides a dispatch function that recognizes the I/O request codes and control codes that a user-mode driver can place inside the IRPs. The driver provides a set of driver functions used with *VCKernel.lib*, which the library calls in response to the receipt of the control codes.

To use *VCKernel.lib* with your kernel-mode driver

1. Include *vckernel.h* and *vcstruct.h* in your driver.
2. Link your driver with *VCKernel.lib*.

Initializing and Configuring a Driver, Using *VCKernel.lib*

Initialization and configuration operations take place in a kernel-mode driver's **DriverEntry** function. Video capture drivers using *VCKernel.lib*, such as *bravado.sys*, must call **VC_Init** from within **DriverEntry**, before calling any other *VCKernel.lib* function. The **VC_Init** function creates a device object named "vidcap*x*", where *x* is incremented from 0 for each new video capture device object, and creates an entry for the device in the registry.

After calling **VC_Init**, *bravado.sys* calls **VC_ReadProfile** to obtain hardware configuration parameters that were stored by the user-mode driver. These parameters are used as input to **VC_GetResources**, which reserves system resources for the device, and maps the device's I/O address space and frame buffer into system address space.

Kernel-mode drivers using *VCKernel.lib* must provide a set of driver functions used with *VCKernel.lib*, so *bravado.sys* next calls **VC_GetCallbackTable** to get the address of *VCKernel.lib*'s callback table, and fills in the table with the addresses of driver-supplied functions.

After the callback table has been filled, device hardware can be initialized. The *bravado.sys* driver

places initial values in device registers and then calls **VC_ConnectInterrupt** to connect the driver's interrupt service routine with the device's interrupt number. Then it confirms that interrupts can be received.

If *bravado.sys* detects an error during execution of its **DriverEntry** function, it calls **VC_WriteProfile** to write an error code into the registry. The user-mode driver can read the registry entry to determine if the kernel-mode driver initialized properly.

Before a kernel-mode driver is unloaded, *VCKernel.lib* calls **VC_Cleanup**, which in turn calls the driver-supplied **CleanupFunc** function. The **VC_Cleanup** function releases resources allocated by **VC_Init** and **VC_GetResources**. The driver's **CleanupFunc** function might disable hardware and free any driver memory allocations. Besides being called by *VCKernel.lib* before the driver is unloaded, **VC_Cleanup** is typically called by the driver itself if the driver detects an error within **DriverEntry** any time after calling **VC_Init**.

Opening and Closing a Device, Using *VCKernel.lib*

Kernel-mode video capture drivers can provide functions for opening and closing devices. If the driver includes a **DeviceOpenFunc** function, then *VCKernel.lib* calls the function when a device is being opened. The function might enable hardware or allocate memory needed for I/O operations. The sample driver, *bravado.sys*, uses its **DeviceOpenFunc** function to obtain characteristics of the user's display device, which are placed in the registry by the user-mode driver (for details, see Opening and Closing a Device, Using *VCUser.lib*).

If the driver includes a **DeviceCloseFunc** function, *VCKernel.lib* calls the function when the device is being closed.

Configuring Video Channels, Using *VCKernel.lib*

Kernel-mode video capture drivers can provide three functions for setting device characteristics that are based on the channel configuration options defined by the user-mode driver. These functions are:

ConfigDisplayFunc
Sets characteristics of the overlay display.

ConfigFormatFunc
Sets video data format characteristics.

ConfigSourceFunc
Sets characteristics of the video source.

Code in *VCKernel.lib* calls these functions in response to requests from the user-mode driver, which are explained in Configuring Video Channels, Using *VCUser.lib*.

Whenever video data format characteristics are changed, the kernel-mode driver should call **VC_SetImageSize** to indicate the maximum number of bytes needed to store an image, using the current format.

Accessing Video Capture Hardware, Using *VCKernel.lib*

After a kernel-mode video capture driver's **DriverEntry** function has called **VC_GetResources**, which maps the device's I/O address space and frame buffer into system address space, the driver can use *VCKernel.lib* functions and macros to access device hardware. The following functions read or write the device's port address space:

VC_In
VC_Out

The following macros are useful for reading or writing the device's frame buffer:

VC_ReadIOMemoryBlock
VC_ReadIOMemoryBYTE

VC_ReadIOMemoryULONG
VC_ReadIOMemoryUSHORT
VC_WriteIOMemoryBlock
VC_WriteIOMemoryBYTE
VC_WriteIOMemoryULONG
VC_WriteIOMemoryUSHORT

Handling Interrupts, Using *VCKernel.lib*

All kernel-mode video capture drivers using *VCKernel.lib* must provide both an interrupt service routine (ISR) and a deferred procedure call (DPC) function. The ISR is referred to as the driver's **InterruptAcknowledge** function, and the DPC function is referred to as its **CaptureService** function.

Synchronizing Driver Activities, Using *VCKernel.lib*

Code that references the same objects that a driver's interrupt service routine (ISR) references — typically structures and device registers — must be synchronized to avoid simultaneous attempts at referencing the same object. Likewise, code that references the same objects that the driver's deferred procedure call (DPC) function references — typically the frame buffer — must be synchronized.

Kernel-mode drivers normally run at an IRQL of PASSIVE_LEVEL. When a device interrupt occurs, the IRQL increases to the device's IRQL while the ISR executes. The ISR acknowledges the interrupt and schedules the execution of a DPC function. The DPC function finishes servicing the interrupt, typically by copying frame buffer data. It executes at an IRQL of DISPATCH_LEVEL, which is between PASSIVE_LEVEL and the device's level.

Video capture drivers using *VCKernel.lib* should use the **VC_SynchronizeExecution** function to synchronize references to objects that the device's ISR uses. The function uses a spin lock and the device's IRQL to maintain exclusive use of the referenced object.

Drivers should use the **VC_SynchronizeDPC** function to synchronize references to objects that the device's DPC function uses. The **VC_SynchronizeDPC** function uses a spin lock and the DISPATCH_LEVEL IRQL to maintain exclusive use of the referenced object.

Use of **VC_SynchronizeExecution** and **VC_SynchronizeDPC** ensures that multiple processors cannot simultaneously reference the same object, and that lower-priority code executing on the current processor cannot obtain access to the object.

For more information about spin locks, and IRQLs, see the *Kernel-Mode Drivers Design Guide*.

Supporting Overlay Capabilities, Using *VCKernel.lib*

Kernel-mode video capture drivers support a device's overlay capabilities by providing a set of functions that *VCKernel.lib* can call in response to requests from a user-mode driver. If the hardware supports overlay capabilities, the driver must provide the following set of functions:

OverlayFunc
Enables overlay.

GetOverlayModeFunc
Returns overlay capabilities.

SetOverlayRectsFunc
Sets overlay display rectangles.

SetOverlayOffsetFunc
Sets the overlay offset rectangle that is used for panning.

GetKeyColourFunc
Returns the overlay area's current key color.

SetKeyPalIdxFunc

Sets the overlay area's key color to a palette index number.

SetKeyRGBFunc
Sets the overlay area's key color to an RGB value.

Capturing Video Data, Using *VCKernel.lib*

From a kernel-mode video capture driver's viewpoint, all captured data is presented to the user-mode driver as a stream. If the user-mode driver requests only a single frame, it opens a stream and provides only a single buffer to receive frame contents. To support video capture streams, your driver must provide the following functions, which *VCKernel.lib* calls in response to requests from a user-mode driver:

StreamInitFunc
Initializes a stream.

StreamStartFunc
Starts recording frames.

StreamGetPositionFunc
Returns the time since recording started.

StreamStopFunc
Stops recording frames.

StreamFiniFunc
Closes the stream.

Code in *VCKernel.lib* maintains a queue of buffers received from the user-mode driver, passes the address of the next available buffer to the driver's **CaptureService** function each time it is called, and returns unused buffers to the user-mode driver when the user-mode driver resets the stream. (For more information about capture streams and user-mode drivers see Capturing Video Data, Using *VCUser.lib*.)

Your driver must also provide a **CaptureFunc** function, which enables and disables the hardware's ability to capture frames.

Playing Back Captured Video Data, Using *VCKernel.lib*

If a device supports video playback, its kernel-mode video capture driver must provide a **DrawFrameFunc** function, which *VCKernel.lib* can call in response to a request from a user-mode driver. This function copies bitmap data in the device's frame buffer, while performing data conversion operations, if necessary. If the device supports playback, the driver's **GetOverlayModeFunc** function must provide a return value that indicates the video formats the device can accept.

Video Capture Driver Reference

The following sections describe the messages, functions, structures, and macros used by video capture drivers.

Sections relating to user-mode drivers include:

- Messages, User-Mode Video Capture Drivers
- Structures, User-Mode Video Capture Drivers
- Functions, *VCUser.lib*
- Structures, *VCUser.lib*

Sections relating to kernel-mode drivers include:

- Functions, *VCKernel.lib*
- Structures, *VCKernel.lib*
- Driver Functions Used with *VCKernel.lib*

- Macros, Kernel-Mode Video Capture Drivers

Messages, User-Mode Video Capture Drivers

This section describes the messages received by user-mode video capture drivers. The messages are listed in alphabetical order. They are defined in *msviddrv.h* and *msvideo.h*.

DVM_CONFIGURESTORAGE

The DVM_CONFIGURESTORAGE message requests a user-mode video capture driver to save or restore a video channel's configuration parameters.

Parameters

dwDriverID
Video channel identifier. One of VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, VIDEO_EXTERNALOUT. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_CONFIGURESTORAGE

lParam1
Pointer to a client-specified unique string.

lParam2
Contains flags. The following flags are defined:

Flag	Definition
VIDEO_CONFIGURE_GET	The channel's configuration parameters should be restored from a file.
VIDEO_CONFIGURE_SET	The channel's configuration parameters should be saved in a file.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_CONFIGURESTORAGE message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoConfigureStorage** function, which is described in the Video for Windows Development Kit.

If the value contained in *lParam2* is VIDEO_CONFIGURE_SET, the driver should store the specified channel's current configuration parameters in a file. The file name should consist of, or be based on, the string pointed to by *lParam1*.

If the value contained in *lParam2* is VIDEO_CONFIGURE_GET, the driver should read the specified channel's saved configuration parameters from the saved file, and use them as the channel's current settings. You can assume the client will specify the same *lParam1* value for saving and restoring the parameters.

DVM_DIALOG

The DVM_DIALOG message requests a user-mode video capture driver to display a dialog box that obtains user-specified parameters for a video channel.

Parameters

dwDriverID
Video channel identifier. One of VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, VIDEO_EXTERNALOUT. (See Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_DIALOG

lParam1
Contains the handle to the parent window.

lParam2
Contains flags. The following flags are defined.

Flag	Definition
VIDEO_DLG_QUERY	See the following **Comments** section.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_DIALOG message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoDialog** function, which is described in the Video for Windows Development Kit.

If the driver provides a dialog box for the specified channel, it should obtain user settings and store them in the registry, as explained in Configuring Video Channels.

If a client sends a DVM_DIALOG message and specifies a channel for which the driver does not provide a dialog box, the driver should return DV_ERR_NOTSUPPORTED.

If the VIDEO_DLG_QUERY flag is set in *dwParam2* and the driver supports a dialog box for the specified channel, it should return DV_ERR_OK. If the flag is set and the driver does not provide a dialog box for the channel, it should return DV_ERR_NOTSUPPORTED.

DVM_DST_RECT

The DVM_DST_RECT message requests a user-mode video capture driver to set or return a video channel's destination rectangle.

Parameters

dwDriverID
Video channel identifier. One of VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, VIDEO_EXTERNALOUT. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_DST_RECT

lParam1
Pointer to a RECT structure. For more information on the RECT structure, see the Win32 SDK.

lParam2
Contains flags. The following flags are defined.

Flag	Definition
VIDEO_CONFIGURE_CURRENT	The driver sets or returns the current destination rectangle description.
VIDEO_CONFIGURE_GET	The driver returns the current rectangle description to the client.

VIDEO_CONFIGURE_MAX	The driver returns the maximum allowable destination rectangle size. (Used only with VIDEO_CONFIGURE_GET.)
VIDEO_CONFIGURE_MIN	The driver returns the minimum allowable destination rectangle size. (Used only with VIDEO_CONFIGURE_GET.)
VIDEO_CONFIGURE_QUERY	The driver indicates whether or not it supports the specified request.
VIDEO_CONFIGURE_SET	The driver receives a client-specified rectangle description.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_DST_RECT message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoMessage** function, which is described in the Video for Windows Development Kit.

The driver should test the VIDEO_CONFIGURE_SET and VIDEO_CONFIGURE_GET flags to determine whether to set or to return a rectangle description. If neither of these flags is set, the driver should return an error code.

The meaning of a destination rectangle is dependent on the specified video channel, as explained in Setting Source and Destination Rectangles.

User-mode video capture drivers using *VCUser.lib* can call **VC_SetOverlayRect** when processing the DVM_DST_RECT message for the VIDEO_EXTERNALOUT channel.

DVM_FORMAT

The DVM_FORMAT message requests a user-mode video capture driver to set a specified data format, or to return the current format, for the specified channel.

Parameters

dwDriverID
Video channel identifier. One of VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, VIDEO_EXTERNALOUT. (For details, see Opening Video Channels.) Typically, drivers only allow VIDEO_IN for this message.

hDriver
Driver handle.

uMsg
DVM_FORMAT

lParam1
Contains flags. The following flags are defined.

Flag	Definition
VIDEO_CONFIGURE_CURRENT	The driver sets or returns the current format.
VIDEO_CONFIGURE_GET	The driver returns the requested information to the client.
VIDEO_CONFIGURE_MAX	The driver returns the maximum-sized format. (Used only with VIDEO_CONFIGURE_GET.)
VIDEO_CONFIGURE_MIN	The driver returns the minimum-sized format. (Used only with

	VIDEO_CONFIGURE_GET.)
VIDEO_CONFIGURE_NOMINAL	The driver returns the nominal format. (Used only with VIDEO_CONFIGURE_GET.)
VIDEO_CONFIGURE_QUERY	The driver indicates whether or not it supports the specified request.
VIDEO_CONFIGURE_QUERYSIZE	The driver returns the size, in bytes, of the format description.
VIDEO_CONFIGURE_SET	The driver sets a client-specified format description.

lParam2

Pointer to a VIDEOCONFIGPARMS structure. For the DVM_FORMAT message, the structure members are used as follows:

lpdwReturn
Pointer to a DWORD in which the driver returns the size, in bytes, of the BITMAPINFOHEADER structure, if the VIDEO_CONFIGURE_QUERYSIZE flag is set.

lpData1
Pointer to a client-supplied BITMAPINFOHEADER structure. (For more information about the BITMAPINFOHEADER structure, see the Win32 SDK.)

dwSize1
Size, in bytes, of the BITMAPINFOHEADER structure.

lpData2
Not used.

dwSize2
Not used.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_FORMAT message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoConfigure** function, which is described in the Video for Windows Development Kit.

The driver should test the VIDEO_CONFIGURE_SET and VIDEO_CONFIGURE_GET flags to determine whether to set or to return a format description. If neither of these flags is set, the driver should return an error code.

When a client sends this message, it must also include the address of a BITMAPINFOHEADER structure (unless either the VIDEO_CONFIGURE_QUERY or VIDEO_CONFIGURE_QUERYSIZE flag is set). Depending on whether the client has set VIDEO_CONFIGURE_SET or VIDEO_CONFIGURE_GET, the driver either reads or writes the contents of this structure.

If the VIDEO_CONFIGURE_QUERYSIZE flag is set, the driver just returns the size of the BITMAPINFOHEADER structure.

For more information on how drivers should allow applications and users to set the video data format, see Setting the Video Data Format.

User-mode video capture drivers using *VCUser.lib* can call **VC_ConfigFormat** when processing the DVM_FORMAT message.

DVM_FRAME

The DVM_FRAME message requests a user-mode video capture driver to transfer a single frame to or from the video device.

Parameters

dwDriverID
Video channel identifier. One of VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, VIDEO_EXTERNALOUT. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_FRAME

lParam1
Pointer to a VIDEOHDR structure.

lParam2
Size of the VIDEOHDR structure.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_FRAME message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoFrame** function, which is described in the Video for Windows Development Kit.

The client uses the VIDEOHDR structure to specify a buffer. The driver uses this buffer as either a destination or source for DIB-formatted data, depending on whether the specified channel is VIDEO_IN or VIDEO_OUT. For the VIDEO_IN channel, the driver transfers data from the device's frame buffer into the specified buffer. For the VIDEO_OUT channel, the driver transfers data from the specified buffer into the device's frame buffer.

The driver should process this message synchronously, not returning to the client until the transfer is complete.

The sample video capture drivers only support the DVM_FRAME message for the VIDEO_IN channel.

For more information about handling the DVM_FRAME message, see Transferring Single Frames.

User-mode video capture drivers using *VCUser.lib* can call **VC_Frame** when processing the DVM_FRAME message for the VIDEO_IN channel.

DVM_GET_CHANNEL_CAPS

The DVM_GET_CHANNEL_CAPS message requests a user-mode video capture driver to return the capabilities of a video channel.

Parameters

dwDriverID
Video channel identifier. One of VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, VIDEO_EXTERNALOUT. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_GET_CHANNEL_CAPS

lParam1
Pointer to a CHANNEL_CAPS structure.

lParam2
Size of the CHANNEL_CAPS structure.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_GET_CHANNEL_CAPS message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoGetChannelCaps** function, which is described in the Video for Windows Development Kit.

The driver fills in the client-supplied CHANNEL_CAPS structure with channel information for the specified channel.

DVM_GETERRORTEXT

The DVM_GETERRORTEXT message requests a user-mode video capture driver to return a text string associated with an error code.

Parameters

dwDriverID
Video channel identifier. One of VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, VIDEO_EXTERNALOUT. (For details, see Opening Video Channels.) Can be NULL for this message.

hDriver
Driver handle.

uMsg
DVM_GETERRORTEXT

lParam1
Pointer to a VIDEO_GETERRORTEXT_PARMS structure.

lParam2
Not used.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*.

Comments

A client sends the DVM_GETERRORTEXT message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoGetErrorText** function, which is described in the Video for Windows Development Kit.

You can define customized error codes for a video capture driver. If you do, you need to also define a text string for each error code and place the strings in a resource (*.rc*) file. Clients use the DVM_GETERRORTEXT message to retrieve the text string for a specified custom error code.

Drivers should call the **LoadString** function, described in the Win32 SDK, to retrieve a string. This function truncates the string if the supplied buffer is too small.

DVM_GETVIDEOAPIVER

The DVM_GETVIDEOAPIVER message requests a user-mode video capture driver to return the version of the video capture API used by the driver.

Parameters

dwDriverID
Video channel identifier. One of VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, VIDEO_EXTERNALOUT. (For details, see Opening Video Channels.) Can be NULL for this message.

hDriver

Driver handle.

uMsg
DVM_GETVIDEOAPIVER

lParam1
Pointer to a DWORD to receive the version.

lParam2
Not used.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_GETVIDEOAPIVER message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoMessage** function, which is described in the Video for Windows Development Kit.

The driver should respond to this message by returning the VIDEOAPIVERSION constant in the address pointed to by the *lParam1* parameter. The VIDEOAPIVERSION constant is defined in *msviddrv.h*.

DVM_PALETTE

The DVM_PALETTE message requests a user-mode video capture driver to store or return a logical palette.

Parameters

dwDriverID
Video channel identifier. One of VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, VIDEO_EXTERNALOUT. (For details, see Opening Video Channels.) Typically, drivers only allow VIDEO_IN for this message.

hDriver
Driver handle.

uMsg
DVM_PALETTE

lParam1
Contains flags. The following flags are defined.

Flag	Definition
VIDEO_CONFIGURE_CURRENT	The driver sets or returns the current palette.
VIDEO_CONFIGURE_GET	The driver returns the requested information to the client.
VIDEO_CONFIGURE_MAX	The driver returns the maximum-sized palette. (Used only with VIDEO_CONFIGURE_GET.)
VIDEO_CONFIGURE_MIN	The driver returns the minimum-sized palette. (Used only with VIDEO_CONFIGURE_GET.)
VIDEO_CONFIGURE_NOMINAL	The driver returns the nominal palette. (Used only with VIDEO_CONFIGURE_GET.)
VIDEO_CONFIGURE_QUERY	The driver indicates whether or not it supports the specified request.
VIDEO_CONFIGURE_QUERYSIZE	The driver returns the size, in bytes, of the current palette.

VIDEO_CONFIGURE_SET	The driver sets a client-specified palette description.

lParam2

Pointer to a VIDEOCONFIGPARMS structure. For the DVM_PALETTE message, the structure members are used as follows:

lpdwReturn
Pointer to a DWORD in which the driver returns the size, in bytes, of the currently stored palette, if the VIDEO_CONFIGURE_QUERYSIZE flag is set.

lpData1
Pointer to a client-supplied LOGPALETTE structure. (For more information about the LOGPALETTE structure, see the Win32 SDK.)

dwSize1
Size, in bytes, of the LOGPALETTE structure.

lpData2
Not used.

dwSize2
Not used.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_PALETTE message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoConfigure** function, which is described in the Video for Windows Development Kit.

The driver should test the VIDEO_CONFIGURE_SET and VIDEO_CONFIGURE_GET flags to determine whether to set or to return a format description. If neither of these flags is set, the driver should return an error code.

When a client sends this message, it must also include the address of a LOGPALETTE structure (unless either the VIDEO_CONFIGURE_QUERY or VIDEO_CONFIGURE_QUERYSIZE flag is set). Depending on whether the client has set VIDEO_CONFIGURE_SET or VIDEO_CONFIGURE_GET, the driver either writes or reads the contents of this structure.

If the VIDEO_CONFIGURE_QUERYSIZE flag is set, the driver just returns the size of the currently stored palette, including the palette entry array.

For more information on how drivers can handle palettes and color translation tables, see Setting Palettes.

User-mode video capture drivers using *VCUser.lib* can call **VC_ConfigFormat** when processing the DVM_PALETTE message.

See Also

DVM_PALETTERGB555

DVM_PALETTERGB555

The DVM_PALETTERGB555 message requests a user-mode video capture driver to store a logical palette and an RGB555 translation table.

Parameters

dwDriverID

Video channel identifier. One of VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, VIDEO_EXTERNALOUT. (For details, see Opening Video Channels.) Typically, drivers only allow VIDEO_IN for this message.

hDriver
Driver handle.

uMsg
DVM_PALETTERGB555

lParam1
Contains flags. The following flags are defined.

Flag	Definition
VIDEO_CONFIGURE_SET	The driver sets a client-specified palette description and translation table.
VIDEO_CONFIGURE_QUERY	The driver indicates whether or not it supports the DVM_PALETTERGB555 message.

lParam2
Pointer to a VIDEOCONFIGPARMS structure. For the DVM_PALETTERGB555 message, the structure members are used as follows:

lpdwReturn
Pointer to a DWORD in which the driver returns the size, in bytes, of the currently stored palette, if the VIDEO_CONFIGURE_QUERYSIZE flag is set.

lpData1
Pointer to a client-supplied LOGPALETTE structure. (For more information about the LOGPALETTE structure, see the Win32 SDK.)

dwSize1
Size, in bytes, of the LOGPALETTE structure.

lpData2
Pointer to a 32-kilobyte RGB555 translation table.

dwSize2
Size of the translation table. Must be 32768.

Return Value
The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments
A client sends the DVM_PALETTERGB555 message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoConfigure** function, which is described in the Video for Windows Development Kit.

This message does not support the VIDEO_CONFIGURE_GET flag. Clients can send DVM_PALETTE to obtain the current palette.

For more information on how drivers can handle palettes and color translation tables, see Setting Palettes.

User-mode video capture drivers using *VCUser.lib* can call **VC_ConfigFormat** when processing the DVM_PALETTERGB555 message.

DVM_SRC_RECT

The DVM_SRC_RECT message requests a user-mode video capture driver to set or return a video channel's source rectangle.

Parameters
dwDriverID
Video channel identifier. One of VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, VIDEO_EXTERNALOUT. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_SRC_RECT

lParam1
Pointer to a RECT structure. For more information on the RECT structure, see the Win32 SDK.

lParam2
Contains flags. The following flags are defined.

Flag	Definition
VIDEO_CONFIGURE_CURRENT	The driver sets or returns the current source rectangle description.
VIDEO_CONFIGURE_GET	The driver returns the current rectangle description to the client.
VIDEO_CONFIGURE_MAX	The driver returns the maximum allowable source rectangle size. (Used only with VIDEO_CONFIGURE_GET.)
VIDEO_CONFIGURE_MIN	The driver returns the minimum allowable source rectangle size. (Used only with VIDEO_CONFIGURE_GET.)
VIDEO_CONFIGURE_QUERY	The driver indicates whether or not it supports the specified request.
VIDEO_CONFIGURE_SET	The driver receives a client-specified rectangle description.

Return Value
The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments
A client sends the DVM_SRC_RECT message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoMessage** function, which is described in the Video for Windows Development Kit.

The driver should test the VIDEO_CONFIGURE_SET and VIDEO_CONFIGURE_GET flags to determine whether to set or to return a rectangle description. If neither of these flags is set, the driver should return an error code.

The meaning of a source rectangle is dependent on the specified video channel, as explained in Setting Source and Destination Rectangles.

User-mode video capture drivers using *VCUser.lib* can call **VC_SetOverlayOffset** when processing the DVM_SRC_RECT message on the VIDEO_EXTERNALIN channel.

DVM_STREAM_ADDBUFFER

The DVM_STREAM_ADDBUFFER message requests a user-mode video capture driver to add an empty input buffer to its input buffer queue.

Parameters
dwDriverID
Video channel identifier. For this message, the driver should only accept VIDEO_IN. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_STREAM_ADDBUFFER

lParam1

Pointer to a VIDEOHDR structure.

lParam2

Size of the VIDEOHDR structure.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_STREAM_ADDBUFFER message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoStreamAddBuffer** function, which is described in the Video for Windows Development Kit.

The VIDEOHDR structure describes the buffer. Before a buffer can be added to the driver's queue, the client must prepare it by sending a DVM_STREAM_PREPAREHEADER message. Drivers can confirm that the buffer has been prepared by testing the VHDR_PREPARED flag in the VIDEOHDR structure's **dwFlags** member. Code in *msvfw32.dll* and *avicap32.dll* checks this flag before sending a DVM_STREAM_ADDBUFFER message.

Drivers assume that the VIDEOHDR structure and the data buffer pointed to by its **lpData** member have been allocated with **GlobalAlloc**, using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with **GlobalLock**. (For further information about **GlobalAlloc** and **GlobalLock**, see the Win32 SDK.)

For information about using data buffers with video capture streams, see Transferring streams of Captured Data.

User-mode video capture drivers using *VCUser.lib* can call **VC_StreamAddBuffer** when processing the DVM_STREAM_ADDBUFFER message.

DVM_STREAM_ALLOCBUFFER

The DVM_STREAM_ALLOCBUFFER message requests a user-mode video capture driver to allocate a buffer from a device's local, non-system memory.

Parameters

dwDriverID

Video channel identifier. One of VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, VIDEO_EXTERNALOUT. (For details, see Opening Video Channels.)

hDriver

Driver handle.

uMsg

DVM_STREAM_ALLOCBUFFER

lParam1

Pointer to a location to receive the address of the returned buffer.

lParam2

Requested buffer size, in bytes.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_STREAM_ALLOCBUFFER message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoMessage** function, which is described in the Video for Windows Development Kit.

If the hardware provides on-board memory that can be used as buffer space to receive captured video data, the driver should allocate a buffer using the size specified by *lParam2*, return the buffer's address in the location pointed to by *lParam1*, and provide a return value of DV_ERR_OK. If no buffer space is available, the driver should return NULL in the location pointed to by *lParam1*. If the hardware does not provide on-board memory, the driver should return DV_ERR_NOTSUPPORTED.

Note: The video capture driver libraries, *VCUser.lib* and *VCKernel.lib*, do not support this message. Therefore, the sample video capture drivers do not support the message either.

See Also

DVM_STREAM_ADDBUFFER

DVM_STREAM_FREEBUFFER

DVM_STREAM_FINI

The DVM_STREAM_FINI message requests a user-mode video capture driver to close a capture stream on a video channel.

Parameters

dwDriverID
Video channel identifier. For this message, the driver should only accept VIDEO_EXTERNALIN, VIDEO_IN, and VIDEO_EXTERNALOUT. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_STREAM_FINI

lParam1
Not used.

lParam2
Not used.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_STREAM_FINI message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoStreamFini** function, which is described in the Video for Windows Development Kit.

If all the input buffers sent with DVM_STREAM_ADDBUFFER haven't been returned to the client, the driver should fail the message. If all buffers have been returned, the driver requests the kernel-mode driver to disable the hardware's data capture and overlay operations.

After the driver has finished closing the VIDEO_IN channel, it must send an MM_DRVM_CLOSE callback message.

For more information about video capture streams, see Transferring streams of Captured Data.

User-mode video capture drivers using *VCUser.lib* can call the following functions to terminate a stream, based on the specified channel.

Channel	*VCUser.lib* Function
VIDEO_EXTERNALIN	**VC_Capture**
VIDEO_IN	**VC_StreamFini**
VIDEO_EXTERNALOUT	**VC_Overlay**

See Also

DVM_STREAM_INIT

DVM_STREAM_FREEBUFFER

The DVM_STREAM_FREEBUFFER message requests a user-mode video capture driver to free a buffer that was allocated in response to a DVM_STREAM_ALLOCBUFFER message.

Parameters

dwDriverID
Video channel identifier. One of VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, VIDEO_EXTERNALOUT. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_STREAM_FREEBUFFER

lParam1
Buffer address returned in response to a previous DVM_STREAM_ALLOCBUFFER message.

lParam2
Not used.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_STREAM_FREEBUFFER message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoMessage** function, which is described in the Video for Windows Development Kit.

If the hardware provides on-board memory that can be used as buffer space to receive captured video data, the driver can support the DVM_STREAM_ALLOCBUFFER and DVM_STREAM_FREEBUFFER messages. The DVM_STREAM_FREEBUFFER message is used to deallocate buffer space that was allocated in response to DVM_STREAM_ALLOCBUFFER. If the hardware does not provide on-board memory, the driver should return DVM_ERR_NOTUSUPPORTED.

Note: The video capture driver libraries, *VCUser.lib* and *VCKernel.lib*, do not support this message. Therefore, the sample video capture drivers do not support the message either.

DVM_STREAM_GETERROR

The DVM_STREAM_GETERROR message requests a user-mode video capture driver to return a capture stream's current error status.

Parameters

dwDriverID
Video channel identifier. For this message, the driver should only accept VIDEO_IN. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_STREAM_GETERROR

lParam1
Pointer to a DWORD to receive an error code.

lParam2
Pointer to a DWORD to receive the number of frames dropped.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_STREAM_GETERROR message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoStreamGetError** function, which is described in the Video for Windows Development Kit.

When the driver receives this message it should return an error code (defined in *msvideo.h*) indicating the current status of the capture stream.

If the driver has had to drop captured frames because the client has not sent enough buffers to receive them, the driver should return DV_ERR_NO_BUFFERS in the address pointed to by *lParam1*, and should return the number of dropped frames in the address pointed to by *lParam2*. The driver should then reset its count of dropped frames to zero.

For more information about video capture streams, see Transferring streams of Captured Data.

User-mode video capture drivers using *VCUser.lib* can call **VC_GetStreamError** when processing the DVM_STREAM_GETERROR message.

DVM_STREAM_GETPOSITION

The DVM_STREAM_GETPOSITION message requests a user-mode video capture driver to return the current stream position.

Parameters

dwDriverID
Video channel identifier. For this message, the driver should only accept VIDEO_IN. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_STREAM_GETPOSITION

lParam1
Pointer to an MMTIME structure. (The MMTIME structure is defined in *mmsystem.h* and described in the Win32 SDK.)

lParam2
Size, in bytes, of the MMTIME structure.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_STREAM_GETPOSITION message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoStreamGetPosition** function, which is described in the Video for Windows Development Kit.

The client specifies a requested time format in the MMTIME structure's **wType** member. If the driver cannot support the requested format, it uses a format it can support, and places the format type in **wType**. Video-capture drivers generally use the millisecond time format.

The driver should initialize the stream position when it receives a DVM_STREAM_START or a

DVM_STREAM_RESET message.

For more information about video capture streams, see Transferring streams of Captured Data.

User-mode video capture drivers using *VCUser.lib* can call **VC_GetStreamPos** when processing the DVM_STREAM_GETPOSITION message.

DVM_STREAM_INIT

The DVM_STREAM_INIT message requests a user-mode video capture driver to initialize a video channel for streaming.

Parameters

dwDriverID
Video channel identifier. For this message, the driver should only accept VIDEO_EXTERNALIN, VIDEO_IN, and VIDEO_EXTERNALOUT. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_STREAM_INIT

lParam1
Pointer to a VIDEO_STREAM_INIT_PARMS structure.

lParam2
Size, in bytes, of the VIDEO_STREAM_INIT_PARMS structure.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_STREAM_INIT message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoStreamInit** function, which is described in the Video for Windows Development Kit.

Typical stream initialization operations include requesting the kernel-mode driver to enable the hardware's data capture and overlay operations.

The VIDEO_STREAM_INIT_PARMS structure contains the client-specified capture frequency. It also contains callback information for use when notifying clients of events. For more information about notifying clients, see Notifying Clients from Video Capture Drivers.

After the driver finishes initializing the VIDEO_IN channel, it must send an MM_DRVM_OPEN callback message.

For more information about video capture streams, see Transferring streams of Captured Data.

User-mode video capture drivers using *VCUser.lib* can call the following functions to initialize a stream, based on the specified channel.

Channel	*VCUser.lib* Function
VIDEO_EXTERNALIN	**VC_Capture**
VIDEO_IN	**VC_StreamInit**
VIDEO_EXTERNALOUT	**VC_Overlay**

DVM_STREAM_PREPAREHEADER

The DVM_STREAM_PREPAREHEADER message requests a user-mode video capture driver to prepare a data buffer for use.

Parameters

dwDriverID
Video channel identifier. For this message, the driver should only accept VIDEO_IN. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_STREAM_PREPAREHEADER

lParam1
Pointer to a VIDEOHDR structure.

lParam2
Size of the VIDEOHDR structure.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_STREAM_PREPAREHEADER message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoStreamPrepareHeader** function, which is described in the Video for Windows Development Kit.

Use of this message is meant to ensure that the specified buffer is accessible by the kernel-mode driver. If the driver returns DV_ERR_NOTSUPPORTED, then *msvfw32.dll* or *avicap32.dll* will prepare the buffer. For most drivers, this behavior is sufficient. If the driver does perform buffer preparation, it should return DV_ERR_OK, which causes *msvfw32.dll* or *avicap32.dll* to set the VHDR_PREPARED flag in the VIDEOHDR structure's **dwFlags** member.

Drivers assume that the VIDEOHDR structure and the data buffer pointed to by its **lpData** member have been allocated with **GlobalAlloc**, using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with **GlobalLock**. (For further information about **GlobalAlloc** and **GlobalLock**, see the Win32 SDK.)

For information about using data buffers with video capture streams, see Transferring Streams of Captured Data.

DVM_STREAM_RESET

The DVM_STREAM_RESET message requests a user-mode video capture driver to stop input of a capture stream, return all buffers to the client, and set the current position to zero.

Parameters

dwDriverID
Video channel identifier. For this message, the driver should only accept VIDEO_IN. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_STREAM_RESET

lParam1
Not used.

lParam2
Not used.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one

of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_STREAM_RESET message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoStreamReset** function, which is described in the Video for Windows Development Kit.

When a driver receives DVM_STREAM_RESET, it should restore the state that it established in response to a DVM_STREAM_INIT message. For each unused buffer that was received with a DVM_STREAM_ADDBUFFER message, the driver must set VHDR_DONE in the **dwFlags** member of the buffer's VIDEOHDR structure and then send an MM_DRVM_DATA callback message.

After a client sends DVM_STREAM_RESET, you should expect it to send either a DVM_STREAM_START message to restart the stream or a DVM_STREAM_FINI message to terminate the stream.

For more information about video capture streams, see Transferring streams of Captured Data.

User-mode video capture drivers using *VCUser.lib* can call **VC_StreamReset** when processing the DVM_STREAM_RESET message.

DVM_STREAM_START

The DVM_STREAM_START message requests a user-mode video capture driver to start a video stream.

Parameters

dwDriverID
Video channel identifier. For this message, the driver should only accept VIDEO_IN. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_STREAM_START

lParam1
Not used.

lParam2
Not used.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_STREAM_START message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoStreamStart** function, which is described in the Video for Windows Development Kit.

When a driver receives this message, it should begin transferring frame buffer contents to queued data buffers. Typically, a user-mode driver creates a separate thread to handle communication with the kernel-mode driver. Data buffers should be filled and returned to the client in the order they are received from the client.

Each time a buffer has been filled, the user-mode driver should set VHDR_DONE in the **dwFlags** member of the buffer's VIDEOHDR structure, and then send an MM_DRVM_DATA callback message to the client.

For more information about video capture streams, see Transferring Streams of Captured Data.

User-mode video capture drivers using *VCUser.lib* can call **VC_StreamStart** when processing the DVM_STREAM_START message.

DVM_STREAM_STOP

The DVM_STREAM_STOP message requests a user-mode video capture driver to stop a video stream.

Parameters

dwDriverID
Video channel identifier. For this message, the driver should only accept VIDEO_IN. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_STREAM_STOP

lParam1
Not used.

lParam2
Not used.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_STREAM_STOP message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoStreamStop** function, which is described in the Video for Windows Development Kit.

When a driver receives this message it stops the input stream, typically by requesting the kernel-mode driver to disable capture interrupts. The driver retains its queue of empty buffers. If a buffer has been partially filled, the driver marks it as done and places the actual length of the data in the **dwBytesUsed** member of the buffer's VIDEOHDR structure.

If the client has not previously sent a DVM_STREAM_START message, the DVM_STREAM_STOP message has no effect and the driver should return DV_ERR_OK.

For more information about video capture streams, see Transferring Streams of Captured Data.

User-mode video capture drivers using *VCUser.lib* can call **VC_StreamStop** when processing the DVM_STREAM_STOP message.

DVM_STREAM_UNPREPAREHEADER

The DVM_STREAM_UNPREPAREHEADER message requests a user-mode video capture driver to remove the preparation from a data buffer.

Parameters

dwDriverID
Video channel identifier. For this message, the driver should only accept VIDEO_IN. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_STREAM_UNPREPAREHEADER

lParam1
Pointer to a VIDEOHDR structure.

lParam2
Size of the VIDEOHDR structure.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_STREAM_UNPREPAREHEADER message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoStreamUnprepareHeader** function, which is described in the Video for Windows Development Kit.

Use of this message is meant to remove the preparation that was performed by means of a DVM_STREAM_PREPAREHEADER message. If the driver returns DV_ERR_NOTSUPPORTED, then *msvfw32.dll* or *avicap32.dll* will remove the preparation. For most drivers, this behavior is sufficient. If the driver does remove buffer preparation, it should return DV_ERR_OK, which causes *msvfw32.dll* or *avicap32.dll* to clear the VHDR_PREPARED flag in the VIDEOHDR structure's **dwFlags** member.

If the driver receives a DVM_STREAM_UNPREPAREHEADER message for a buffer that has not been prepared, the driver should just return DV_ERR_OK.

For information about using data buffers with video capture streams, see Transferring streams of Captured Data.

DVM_UPDATE

The DVM_UPDATE message requests a user-mode video capture driver to update the overlay display.

Parameters

dwDriverID
Video channel identifier. For this message, the driver should only accept VIDEO_EXTERNALOUT. (For details, see Opening Video Channels.)

hDriver
Driver handle.

uMsg
DVM_UPDATE

lParam1t
Handle to a client window.

lParam2
Handle to a display device context.

Return Value

The driver should return DV_ERR_OK if the operation succeeds. Otherwise, it should return one of the DV_ERR error codes defined in *msvideo.h*. Custom error codes are also allowed (see DVM_GETERRORTEXT).

Comments

A client sends the DVM_UPDATE message by calling the driver's **DriverProc** entry point, passing the specified parameter values. Applications can send this message by calling the **videoUpdate** function, which is described in the Video for Windows Development Kit.

If the hardware supports color keying, the driver should repaint the current key color onto the overlay area. If the key color has not yet been specified, the driver should specify it. The driver should create a brush using the key color, convert the screen coordinates describing the overlay area into client coordinates, and repaint the overlay area, as illustrated in the sample drivers.

Typically, a client sends this message whenever its window receives a WM_PAINT, WM_MOVE, WM_POSITIONCHANGED, or WM_SIZE message. It also typically sends the message after sending a DVM_SRC_RECT or DVM_DST_RECT message.

User-mode video capture drivers using *VCUser.lib* can call the following functions to set or to determine the key color:

VC_GetKeyColour
VC_SetKeyColourPalIdx
VC_SetKeyColourRGB

MM_DRVM_CLOSE

The MM_DRVM_CLOSE callback message notifies a client that a user-mode video capture driver has finished processing a DVM_STREAM_FINI message.

Parameters

dwMsg
MM_DRVM_CLOSE

dwParam1
Not used.

dwParam2
Not used.

Comments

A user-mode video capture driver sends an MM_DRVM_CLOSE message to its client, by means of a callback, when the driver finishes processing a DVM_STREAM_FINI message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the MM_DRVM_CLOSE message only if the client has previously specified a notification target with a DVM_STREAM_INIT message.

Drivers using *VCUser.lib* do not have to call **DriverCallback** to send callback messages. Code within *VCUser.lib* handles delivery of callback messages for the driver.

For more information about notifying clients, see Notifying Clients from Video Capture Drivers.

MM_DRVM_DATA

The MM_DRVM_DATA callback message notifies a client that a user-mode video capture driver has filled a buffer with capture data.

Parameters

dwMsg
MM_DRVM_DATA

dwParam1
Pointer to a VIDEOHDR structure describing the data buffer that has been filled.

dwParam2
Not used.

Comments

A user-mode video capture driver sends an MM_DRVM_DATA message to its client, by means of a callback, when the driver has filled a buffer with capture data. The driver uses the *dwParam1* argument to indicate which buffer has been filled. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the MM_DRVM_DATA message only if the client has previously specified a notification target with a DVM_STREAM_INIT message. The VIDEOHDR structure was received along with a DVM_STREAM_ADDBUFFER message.

Drivers using *VCUser.lib* do not have to call **DriverCallback** to send callback messages. Code within *VCUser.lib* handles delivery of callback messages for the driver.

For more information about notifying clients, see Notifying Clients from Video Capture Drivers.

MM_DRVM_ERROR

The MM_DRVM_ERROR video capture message is sent to a client when a user-mode video capture driver detects an error.

Parameters

dwMsg
MM_DRVM_ERROR

dwParam1
Number of frames dropped.

dwParam2
Not used.

Comments

A user-mode video capture driver sends an MM_DRVM_ERROR message to its client, by means of a callback, when the driver detects an error. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

A driver can send this message for any reason, but the message is typically used to report dropped frames. The driver drops frames if the client has not provided enough buffers to receive them. Drivers use the *dwParam1* argument to indicate the number of dropped frames.

The driver sends the MM_DRVM_ERROR message only if the client has previously specified a notification target with a DVM_STREAM_INIT message.

Drivers using *VCUser.lib* do not have to call **DriverCallback** to send callback messages. Code within *VCUser.lib* handles delivery of callback messages for the driver.

For more information about notifying clients, see Notifying Clients from Video Capture Drivers.

MM_DRVM_OPEN

The MM_DRVM_OPEN callback message notifies a client that a user-mode video capture driver has finished processing a DVM_STREAM_INIT message.

Parameters

dwMsg
MM_DRVM_OPEN

dwParam1
Not used.

dwParam2
Not used.

Comments

A user-mode video capture driver sends an MM_DRVM_OPEN message to its client, by means of a callback, when the driver finishes processing a DVM_STREAM_INIT message. The driver sends the message to the client by calling **DriverCallback**, passing the specified parameters.

The driver sends the MM_DRVM_OPEN message only if the client has specified a notification target with the DVM_STREAM_INIT message.

Drivers using *VCUser.lib* do not have to call **DriverCallback** to send callback messages. Code within *VCUser.lib* handles delivery of callback messages for the driver.

For more information about notifying clients, see Notifying Clients from Video Capture Drivers.

Structures, User-Mode Video Capture Drivers

This section describes the structures used by Win32-based video capture drivers. The structures are listed in alphabetic order.

CHANNEL_CAPS

```
typedef struct {
DWORD dwFlags;
DWORD dwSrcRectXMod;
DWORD dwSrcRectYMod;
DWORD dwSrcRectWidthMod;
DWORD dwSrcRectHeightMod;
DWORD dwDstRectXMod;
DWORD dwDstRectYMod;
DWORD dwDstRectWidthMod;
DWORD dwDstRectHeightMod;
} CHANNEL_CAPS;
```

The CHANNEL_CAPS structure is used by user-mode video capture drivers to return the capabilities of a video channel. Clients request channel capabilities by sending a DVM_GET_CHANNEL_CAPS message. The structure is defined in *msvideo.h*.

Members

dwFlags

Returns flags providing information about the channel. The following flags are defined.

Flag	Definition
VCAPS_OVERLAY	Indicates the channel is capable of overlay. This flag is used only for VIDEO_EXTERNALOUT channels.
VCAPS_SRC_CAN_CLIP	Indicates the source rectangle can be set smaller than the maximum dimensions.
VCAPS_DST_CAN_CLIP	Indicates the destination rectangle can be set smaller than the maximum dimensions.
VCAPS_CAN_SCALE	Indicates the source rectangle can be a different size than the destination rectangle.

dwSrcRectXMod

Returns the granularity allowed when positioning the source rectangle in the horizontal direction.

dwSrcRectYMod

Returns the granularity allowed when positioning the source rectangle in the vertical direction.

dwSrcRectWidthMod

Returns the granularity allowed when setting the width of the source rectangle.

dwSrcRectHeightMod

Returns the granularity allowed when setting the height of the source rectangle.

dwDstRectXMod

Returns the granularity allowed when positioning the destination rectangle in the horizontal direction.

dwDstRectYMod

Returns the granularity allowed when positioning the destination rectangle in the vertical direction.

dwDstRectWidthMod

Returns the granularity allowed when setting the width of the destination rectangle.

dwDstRectHeightMod

Returns the granularity allowed when setting the height of the source rectangle.

Comments

For the video channel specified with the DVM_GET_CHANNEL_CAPS message, the driver fills in the CHANNEL_CAPS structure with the granularity, in number of pixels, of the positioning points, heights, and widths of source and destination rectangles. If, for example, the device only allows source rectangle positioning on 8-bit x-coordinate boundaries, the value returned for **dwSrcRectXMod** should be eight. If the device allows arbitrarily positioned rectangles, with arbitrary sizes, the structure members should all be set to one.

Rectangle dimensions indicated by modulus operators are considered advisory. When an application tries to set a rectangle size with a DVM_SRC_RECT or DVM_DST_RECT message, you can assume it will check the return value to ensure that the driver accepted the request. For example, if **dwDstRectWidthMod** is set to 64, the application might try to set destination rectangles with pixel widths of 64, 128, 192, 256, and so on. The driver might support only a subset of these sizes. If the application tries to specify an unsupported size, the driver should return DV_ERR_NOTSUPPORTED.

VIDEO_GETERRORTEXT_PARMS

```
typedef struct {
DWORD dwError;
LPWSTR lpText;
DWORD dwLength;
} VIDEO_GETERRORTEXT_PARMS;
```

The VIDEO_GETERRORTEXT_PARMS structure is used by user-mode video capture drivers to return error message text in response to a DVM_GETERRORTEXT message. The structure is defined in *msviddrv.h*.

Members

dwError
Contains the error number.

lpText
Pointer to a buffer into which the driver places the error text string.

dwLength
Length of the buffer.

Comments

The client specifies the error number, along with the address and length of the string buffer. The driver fills in the string buffer with the error text.

VIDEO_OPEN_PARMS

```
typedef struct {
DWORD dwSize;
FOURCC fccType;
FOURCC fccComp;
DWORD dwVersion;
DWORD dwFlags;
DWORD dwError;
LPVOID pV1Reserved;
LPVOID pV2Reserved;
DWORD dnDevNode;
} VIDEO_OPEN_PARMS;
```

The VIDEO_OPEN_PARMS structure defines the type of channel to open on a video-capture device. User-mode video capture drivers receive this structure along with a DRV_OPEN message. The structure is defined in *msviddrv.h*.

Members

dwSize
Contains the size of the VIDEO_OPEN_PARMS structure.

fccType
Contains a four-character code identifying the type of channel being opened. Must be "vcap".

fccComp
Not used.

dwVersion
Contains the current version of the video capture API, which is defined by VIDEOAPIVERSION in *msviddrv.h*.

dwFlags
Specifies flags used to indicate the type of channel. The following flags are defined.

Flag	Definition
VIDEO_EXTERNALIN	Data path from an external video source, such as a TV, VCR, or camera, into a frame buffer.
VIDEO_IN	Data path from the frame buffer to system memory.
VIDEO_OUT	Data path from system memory to the frame buffer.
VIDEO_EXTERNALOUT	Data path from the frame buffer to an output display, such as an overlay window on a user's screen.

dwError
Error value supplied by the driver if the open operation fails.

pV1Reserved
Reserved.

pV2Reserved
Reserved.

dnDevNode
Devnode for PnP devices.

Comments

The VIDEO_OPEN_PARMS structure is identical to the ICOPEN structure used by installable compressors, allowing a single driver to handle both video capture and compression messages.

For more information about the VIDEO_OPEN_PARMS structure, see **DriverProc** in User-Mode Video Capture Drivers and Opening Video Channels.

VIDEO_STREAM_INIT_PARMS

```
typedef struct {
DWORD dwMicroSecPerFrame;
DWORD dwCallback;
DWORD dwCallbackInst;
DWORD dwFlags;
DWORD hVideo;
} VIDEO_STREAM_INIT_PARMS;
```

The VIDEO_STREAM_INIT_PARMS structure contains information needed to initialize a video capture stream. User-mode video-capture drivers receive this structure along with a DVM_STREAM_INIT message. The structure is defined in *msviddrv.h*.

Members

dwMicroSecPerFrame
Contains the client-specified capture frequency, in microseconds. Represents how often the driver should capture a frame.

dwCallback
Contains either the address of a callback function, a window handle, or NULL, based on flags set in the **dwFlags** member.

dwCallbackInst
Contains client-specified instance data passed to the callback function, if CALLBACK_FUNCTION is set in **dwFlags**.

dwFlags
Contains flags. Can contain one (or none) of the following flags.

Flag	Definition
CALLBACK_WINDOW	Indicates **dwCallback** contains a window handle.
CALLBACK_FUNCTION	Indicates **dwCallback** contains a callback function address.

hVideo
Contains a handle to a video channel.

Comments

The **dwCallback**, **dwCallbackInst**, and **dwFlags** members contain information needed by the driver to deliver callback messages. For more information, see Notifying Clients from Video Capture Drivers.

VIDEOCONFIGPARMS

```
typedef struct {
LPDWORD lpdwReturn;
LPVOID lpData1;
DWORD dwSize1;
LPVOID lpData2;
DWORD dwSize2;
} VIDEOCONFIGPARMS;
```

The VIDEOCONFIGPARMS structure is used to send or return message-specific configuration parameters. User-mode video capture drivers receive this structure along with DVM_FORMAT, DVM_PALETTE, and DVM_PALETTERGB555 messages. The structure is defined in *msviddrv.h*.

Members

lpdwReturn
Pointer to a DWORD to be filled with a message-specific return value.

lpData1
Pointer to message-specific data.

dwSize1
Size, in bytes, of data passed in **lpData1**.

lpData2
Pointer to message-specific data.

dwSize2
Size, in bytes, of data passed in **lpData2**.

Comments

For message-specific information about the contents of the **lpdwReturn**, **lpData1**, and **pData2** members, see the descriptions of DVM_FORMAT, DVM_PALETTE, and DVM_PALETTERGB555.

VIDEOHDR

```
typedef struct {
LPBYTE lpData;
DWORD dwBufferLength;
DWORD dwBytesUsed;
DWORD dwTimeCaptured;
DWORD dwUser;
DWORD dwFlags;
DWORD dwReserved[4];
} VIDEOHDR;
```

The VIDEOHDR structure describes a client-supplied buffer for receiving video capture data. Drivers receive this structure along with DVM_STREAM_ADDBUFFER,

DVM_STREAM_PREPAREHEADER, and DVM_STREAM_UNPREPAREHEADER messages. The structure is defined in *msvideo.h*.

Members

lpData
Pointer to a buffer.

dwBufferLength
Length of the buffer.

dwBytesUsed
Number of bytes used in the buffer.

dwTimeCaptured
Capture time for the frame, in milliseconds, relative to the capture time of the stream's first frame.

dwUser
Contains 32 bits of client-specified data.

dwFlags
Contains flags. The following flags are defined.

Flag	Definition
VHDR_DONE	Set by the device driver to indicate it is finished with the data buffer and is returning the buffer to the client.
VHDR_PREPARED	Indicates whether or not the buffer has been prepared for use. See DVM_STREAM_PREPAREHEADER.
VHDR_INQUEUE	Set by the driver to indicate the buffer is in the driver's buffer queue.
VHDR_KEYFRAME	Set by the device driver to indicate a key frame.

dwReserved[4]
Reserved for use by the device driver. Typically, drivers use this area to maintain a buffer queue.

Comments

The client supplies values for the **lpData**, **dwBufferLength**, and **dwUser** members. The driver fills in the **dwBytesUsed**, **dwTimeCaptured**, and **dwFlags** members. Drivers using *VCUser.lib* do not need to maintain the **dwFlags** member, because code within *VCUser.lib* handles the flags.

Functions, *VCUser.lib*

This section describes the functions provided to user-mode video capture drivers that are using *VCUser.lib*. The functions are listed in alphabetic order.

VC_Capture

BOOL
VC_Capture(
VCUSER_HANDLE *vh*,
BOOL *bAcquire*
);

The **VC_Capture** function requests the kernel-mode driver to enable or disable acquisition of video capture data into the frame buffer.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

bAcquire
Set to TRUE to enable capture, or FALSE to disable capture.

Return Value

Returns TRUE if successful. Otherwise returns FALSE.

Comments

User-mode video capture drivers should call **VC_Capture** with *bAcquire* set to TRUE when processing a DVM_STREAM_INIT command for the VIDEO_EXTERNALIN channel. They should call the function with *bAcquire* set to FALSE when processing a DVM_STREAM_FINI command for the VIDEO_EXTERNALIN channel.

Disabling capture when overlay is enabled has the effect of freezing the captured video.

The **VC_Capture** function calls **DeviceIoControl** (described in the Win32 SDK) to send either an IOCTL_VIDC_CAPTURE_ON or an IOCTL_VIDC_CAPTURE_OFF control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives one of these control codes, its **CaptureFunc** function is called.

VC_CloseDevice

VOID
VC_CloseDevice(
VCUSER_HANDLE *vh*
);

The **VC_CloseDevice** function closes a video capture device that was previously opened with **VC_OpenDevice**.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

Return Value

None.

Comments

The **VC_CloseDevice** function closes the kernel-mode driver handle that was obtained with **VC_OpenDevice**. It also removes the worker thread and event handles that *VCUser.lib* creates for internal use when the driver calls **VC_OpenDevice**.

A user-mode driver should call **VC_CloseDevice** when it receives a DRV_CLOSE message. Typically, the driver receives multiple DRV_CLOSE messages, because DRV_CLOSE is sent each time the client closes one of the video channels (VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, or VIDEO_EXTERNALOUT). The driver only needs to call **VC_CloseDevice** for the client's last DRV_CLOSE message, as illustrated by the sample video capture drivers.

VC_CloseProfileAccess

VOID
VC_CloseProfileAccess(
PVC_PROFILE_INFO *pProfile*
);

The **VC_CloseProfileAccess** function removes registry access information obtained from **VC_OpenProfileAccess**.

Parameters

pProfile
Address of the VC_PROFILE_INFO structure returned by **VC_OpenProfileAccess**.

Return Value

None.

Comments

The **VC_CloseProfileAccess** function closes the connection to the Service Control Manager that was created by a previous call to **VC_ConfigAccess**.

After calling **VC_CloseProfileAccess**, a user-mode driver cannot access its kernel-mode driver or the registry.

A user-mode driver should call **VC_CloseProfileAccess** when its **DriverProc** function receives a DRV_FREE message.

See Also

VC_OpenProfileAccess

VC_ConfigAccess

BOOL
VC_ConfigAccess(
PVC_PROFILE_INFO *pProfile*
);

The **VC_ConfigAccess** function creates a connection to the Service Control Manager, if the client has Administrator privilege.

Parameters

pProfile
Address of the VC_PROFILE_INFO structure returned by **VC_OpenProfileAccess**.

Return Value

Returns TRUE if the client has Administrators privilege. Otherwise returns FALSE.

Comments

A user-mode driver must call **VC_ConfigAccess** when it receives a DRV_QUERYCONFIGURE message. Because a user must have Administrator privilege in order to perform driver configuration operations, the driver should provide a failure return value for **DriverProc** in response to DRV_QUERYCONFIGURE, if **VC_ConfigAccess** returns FALSE. This notifies the caller that configuration operations cannot be performed.

VC_ConfigDisplay

BOOL
VC_ConfigDisplay(
VCUSER_HANDLE *vh*,
PCONFIG_INFO *pConfig*
);

The **VC_ConfigDisplay** function sends display-configuration information to a kernel-mode video capture driver.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

pConfig
Pointer to a CONFIG_INFO structure.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

User-mode video capture drivers use the **VC_ConfigDisplay** function to send channel configuration parameters to the kernel-mode driver. Typically, this function is used to send video

overlay parameters associated with the VIDEO_EXTERNALOUT channel.

This **VC_ConfigDisplay** function calls **DeviceIoControl** (described in the Win32 SDK) to send an IOCTL_VIDC_CONFIG_DISPLAY control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives this control code, its **ConfigDisplayFunc** function is called.

For more information about channel configuration, see Configuring Video Channels.

VC_ConfigFormat

BOOL
VC_ConfigFormat(
VCUSER_HANDLE *vh*,
PCONFIG_INFO *pConfig*
);

The **VC_ConfigFormat** function sends format-configuration information to a kernel-mode video capture driver.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

pConfig
Pointer to a CONFIG_INFO structure.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

User-mode video capture drivers use the **VC_ConfigFormat** function to send data format parameters to the kernel-mode driver. Typically, a driver calls **VC_ConfigFormat** when its **DriverProc** function receives a DVM_FORMAT message, or after it displays a dialog box to allow the user to change the format.

The **VC_ConfigFormat** function calls **DeviceIoControl** (described in the Win32 SDK) to send an IOCTL_VIDC_CONFIG_FORMAT control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives this control code, its **ConfigFormatFunc** function is called.

For more information about changing the video format, see Configuring Video Channels and Setting the Video Data Format.

VC_ConfigSource

BOOL
VC_ConfigSource(
VCUSER_HANDLE *vh*,
PCONFIG_INFO *pConfig*
);

The **VC_ConfigSource** function sends source-configuration information to a kernel-mode video capture driver.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

pConfig
Pointer to a CONFIG_INFO structure.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

User-mode video capture drivers use the **VC_ConfigSource** function to send channel configuration parameters to the kernel-mode driver. Typically, this function is used to send video source parameters associated with the VIDEO_EXTERNALIN channel.

The **VC_ConfigSource** function calls **DeviceIoControl** (described in the Win32 SDK) to send an IOCTL_VIDC_CONFIG_SOURCE control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives this control code, its **ConfigSourceFunc** function is called.

For more information about channel configuration, see Configuring Video Channels.

VC_DrawFrame

BOOL
VC_DrawFrame(
VCUSER_HANDLE *vh*,
PDRAWBUFFER *pDraw*
);

The **VC_DrawFrame** function requests a kernel-mode video capture driver to place video data into the frame buffer so that it can be displayed.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

pDraw
Pointer to a DRAWBUFFER structure.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

The driver supplies a DRAWBUFFER structure describing the frame to be drawn.

Drivers can call **VC_GetOverlayMode** to determine if the hardware supports playback of compressed data formats.

The **VC_DrawFrame** function calls **DeviceIoControl** (described in the Win32 SDK) to send an IOCTL_VIDC_DRAW_FRAME control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives this control code, its **DrawFrameFunc** function is called.

VC_Frame

BOOL
VC_Frame(
VCUSER_HANDLE *vh*,
LPVIDEOHDR *lpvh*
);

The **VC_Frame** function requests a kernel-mode video capture driver to capture and return a single frame from the VIDEO_IN channel.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

lpvh

Pointer to a VIDEOHDR structure.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

The **VC_Frame** function returns a single frame by opening a capture stream and supplying only a single data buffer, thereby stopping the stream after one transfer. The stream is then closed. The client is not aware of this stream and, from the client's point of view, the operation appears to behave synchronously.

The driver supplies a VIDEOHDR structure describing an empty buffer. The kernel-mode driver fills the buffer with the contents of the frame buffer.

For more information about video capture data transfers, see Transferring Video Capture Data.

VC_GetKeyColour

DWORD
VC_GetKeyColour(
VCUSER_HANDLE *vh*
);

The **VC_GetKeyColour** function requests a kernel-mode video capture driver to return the device's current key color.

Parameters

vh

Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

Return Value

If the hardware supports a key color, the function returns the key color. Otherwise the function returns zero. See the following **Comments** section.

Comments

Before calling **VC_GetKeyColour**, the driver should call **VC_GetOverlayMode** and test the VCO_KEYCOLOUR_RGB flag to determine if, based on the current data format, the kernel-mode driver has stored the key color as an RGB color or as a palette index number. The **VC_GetKeyColour** function's return value is either an RGBQUAD type or a palette index number, based on the VCO_KEYCOLOUR_RGB flag.

The **VC_GetKeyColour** function calls **DeviceIoControl** (described in the Win32 SDK) to send an IOCTL_VIDC_GET_KEY_COLOUR control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives this control code, its **GetKeyColourFunc** function is called.

See Also

VC_SetKeyColourPalIdx, **VC_SetKeyColourRGB**

VC_GetOverlayMode

ULONG
VC_GetOverlayMode(
VCUSER_HANDLE *vh*
);

The **VC_GetOverlayMode** function requests a kernel-mode video capture driver to return the device's overlay, color keying, and decompression capabilities.

Parameters

vh

Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

Return Value

Returns an unsigned long value containing flags that indicate the types of overlay capabilities supported by the hardware. Zero indicates that overlay is not supported. A nonzero return value can contain the following flags.

Flag	Definition
VCO_KEYCOLOUR	Indicates the device supports a key color.
VCO_KEYCOLOUR_FIXED	Indicates the key color cannot be modified.
VCO_KEYCOLOUR_RGB	If set, indicates the key color must be specified as an RGB color. If clear, indicates the key color must be specified as a palette index number.
VCO_SIMPLE_RECT	Indicates the device supports a single rectangular overlay region.
VCO_COMPLEX_REGION	Indicates the device supports a complex (multi-rectangle) overlay region.
VCO_CAN_DRAW_Y411	Indicates the device can display bitmaps that contain YUV 4:1:1-formatted data.
VCO_CAN_DRAW_S422	Indicates the device can display bitmaps that contain YUV 4:2:2-formatted data.

Comments

The VCO_CAN_DRAW_Y411 and VCO_CAN_DRAW_S422 flags indicate the types of compressed formats the hardware supports for playback by means of the **VC_DrawFrame** function.

The **VC_GetOverlayMode** function calls **DeviceIoControl** (described in the Win32 SDK) to send an IOCTL_VIDC_OVERLAY_MODE control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives this control code, its **GetOverlayModeFunc** function is called.

VC_GetStreamError

ULONG
VC_GetStreamError(
VCUSER_HANDLE *vh*
);

The **VC_GetStreamError** function returns the count of frames that have been dropped from the currently active capture stream since the last time **VC_GetStreamError** was called.

Parameters

vh

Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

Return Value

Returns the number of dropped frames.

Comments

When transferring streams of captured data, a kernel-mode driver drops a frame if a hardware interrupt indicates the frame buffer is full, but the queue of client-supplied buffers is empty. Code within *VCUser.lib* keeps track of dropped frames. You do not have to provide any code to handle dropped frames.

VC_GetStreamPos

BOOL

VC_GetStreamPos(
VCUSER_HANDLE *vh*,
LPMMTIME *pTime*
);

The **VC_GetStreamPos** function requests a kernel-mode video capture driver to return the current position within the capture stream.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

pTime
Pointer to an MMTIME structure. (The MMTIME structure is described in the Win32 SDK.)

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

The **VC_GetStreamPos** function returns the time, in milliseconds, since **VC_StreamStart** was called, by filling in the MMTIME structure.

The **VC_GetStreamPos** function calls **DeviceIoControl** (described in the Win32 SDK) to send an IOCTL_VIDC_GET_POSITION control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives this control code, its **StreamGetPositionFunc** function is called.

VC_InstallDriver

LRESULT
VC_InstallDriver(
PVC_PROFILE_INFO *pProfile*,
PPROFILE_CALLBACK *pCallback*,
PVOID *pContext*
);

The **VC_InstallDriver** function installs a kernel-mode driver and allows modification of configuration parameters.

Parameters

pProfile
Address of a VC_PROFILE_INFO structure returned by **VC_OpenProfileAccess**, containing the kernel-mode driver's name.

pCallback
Pointer to a driver-supplied function that is called before the kernel-mode driver is reloaded. Can be NULL.

The function should return TRUE if the installation should continue, or FALSE if the installation should be terminated. It uses the following prototype definition:

```
BOOL pCallback (PVOID pContext);
```

pContext
Pointer to a driver-defined structure that is passed as input to the function pointed to by *pCallback*. Can be NULL.

Return Value

Returns one the following values.

Value	Definition
DRVCNF_OK	The driver is correctly loaded and started. System restart is not required.
DRVCNF_CANCEL	An error occurred.

Comments

Under Windows NT, kernel-mode drivers are considered to be services under the control of the Service Control Manager. The **VC_InstallDriver** function establishes a kernel-mode driver as a service by performing the following operations, in order:

1. Calling **OpenSCManager** to create a connection to the local Service Control Manager. **OpenSCManager** is called with a desired access type of SC_MANAGER_ALL_ACCESS, which requires Administrators privilege. (**OpenSCManager** is described in the Win32 SDK.)
2. Calling **CreateService** to create a kernel-mode driver service and obtain a service handle. The function sets the service's start type to SERVICE_DEMAND_START, so it will not automatically reload when the system is restarted. (**CreateService** is described in the Win32 SDK.)
3. Unloading the kernel-mode driver, if it is loaded.
4. Calling the driver-supplied function specified by the *pCallback* parameter. Generally, drivers use this function to modify configuration parameters in the registry, using **VC_WriteProfile**.
5. Reloading and restarting the kernel-mode driver, setting its start type to SERVICE_SYSTEM_START, so it will automatically reload and restart when the system is restarted.

A result of calling **CreateService** is the creation of a driver subkey under the **\Services** registry key. The path to the subkey is **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName*, where *DriverName* is the driver name specified as input to **VC_OpenProfileAccess**.

Typically, a user-mode video capture driver calls **VC_InstallDriver** when its **DriverProc** function receives a DRV_CONFIGURE message.

VC_OpenDevice

```
VCUSER_HANDLE
  VC_OpenDevice(
    PWCHAR pDeviceName,
    int DeviceIndex
    );
```

The **VC_OpenDevice** function opens a specified video capture device.

Parameters

pDeviceName

Pointer to a UNICODE string containing the name of the device to open. If NULL, *DeviceIndex* specifies the device.

DeviceIndex

Index number used to identify the device, if *pDriverName* is NULL. If *pDriverName* points to a string, *DeviceIndex* must be zero (see the following **Comments** section).

Return Value

Returns a handle to the kernel-mode driver, if the operation succeeds. Otherwise returns NULL.

Comments

A user-mode driver should call **VC_OpenDevice** when it receives a DRV_OPEN message. Typically, the driver receives multiple DRV_OPEN messages, because DRV_OPEN is sent each time the client opens one of the video channels (VIDEO_EXTERNALIN, VIDEO_IN, VIDEO_OUT, or VIDEO_EXTERNALOUT). The driver only needs to call **VC_OpenDevice** for the client's first DRV_OPEN message, as illustrated by the sample video capture drivers.

If *pDeviceName* points to a string, that string is used as the device name. Because the current implementation does not allow a single kernel-mode video capture driver to support more than one device, a device number is *not* appended to this string. The *DeviceIndex* value must be zero.

The *pDriverName* value must be a pointer to the same string that was specified as input to **VC_OpenProfileAccess**.

If *pDeviceName* is NULL, then the *DeviceIndex* parameter specifies an index value. This value is appended to the device name "vidcap", which is defined by DD_VIDCAP_DEVICE_NAME_U in *ntddvidc.h*. For kernel-mode drivers using *VCKernel.lib*, this name is used by **VC_Init** when creating a device object. Thus, specifying a *DeviceIndex* value allows a user-mode driver to attempt to open a video capture device without specifying a particular device name, such as "bravado". See the sample *msyuv.dll* driver for an example of using the *DeviceIndex* parameter. (For more information about device objects, see the *Kernel-Mode Drivers Design Guide*.)

The **VC_OpenDevice** function calls **CreateFile**, described in the Win32 SDK, to open the device. As a result of this call, the kernel-mode driver receives an IRP_MJ_CREATE function code. When a kernel-mode driver using *VCKernel.lib* receives this function code, its **DeviceOpenFunc** function is called.

VC_OpenProfileAccess

PVC_PROFILE_INFO
VC_OpenProfileAccess(
PWCHAR *DriverName*
);

The **VC_OpenProfileAccess** function creates a VC_PROFILE structure that is used for storing information needed to access the Service Control Manager and the registry.

Parameters

DriverName
Pointer to a UNICODE driver name string.

Return Value

Returns a pointer to a VC_PROFILE_INFO structure, if the operation succeeds. Otherwise returns NULL. (This structure is private to *VCUser.lib*.)

Comments

A user-mode driver should call **VC_OpenProfileAccess** when its **DriverProc** function receives a DRV_LOAD message.

The string specified for *DriverName* should be the name of the kernel-mode driver file, without an extension. For example, the string used for the sample Bravado driver is "bravado". This name is written into the registry by **VC_InstallDriver**.

See Also

VC_CloseProfileAccess

VC_Overlay

BOOL
VC_Overlay(
VCUSER_HANDLE *vh*,
BOOL *bOverlay*
);

The **VC_Overlay** function requests a kernel-mode video capture driver to enable or disable the hardware's overlay capabilities, using the current rectangle and key color settings.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

bOverlay

Set TRUE to enable overlay, or FALSE to disable overlay.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

User-mode video capture drivers using *VCUser.lib* should call the **VC_Overlay** function with *bOverlay* set to TRUE when processing a DVM_STREAM_INIT command for the VIDEO_EXTERNALOUT channel. They should call the function with *bOverlay* set to FALSE when processing a DVM_STREAM_FINI command for the VIDEO_EXTERNALOUT channel.

The **VC_Overlay** function calls **DeviceIoControl** (described in the Win32 SDK) to send either an IOCTL_VIDC_OVERLAY_ON or an IOCTL_VIDC_OVERLAY_OFF control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives one of these control codes, its **OverlayFunc** function is called.

See Also

VC_SetOverlayRect

VC_ReadProfile

```
DWORD
  VC_ReadProfile(
    PVC_PROFILE_INFO pProfile,
    PWCHAR ValueName,
    DWORD dwDefault
    );
```

The **VC_ReadProfile** function reads the DWORD value associated with the specified value name, under the driver's **\Parameters** registry key.

Parameters

pProfile
Address of the VC_PROFILE_INFO structure returned by **VC_OpenProfileAccess**.

ValueName
Pointer to a UNICODE string identifying the name of a registry value.

dwDefault
Specifies a default value that is returned if an error occurs locating or reading the requested value.

Return Value

Returns the value assigned to the *ValueName* string, if the operation succeeds. If the *ValueName* string does not exist in the registry, cannot be accessed, or is not a REG_DWORD type, the function returns the value specified by *dwDefault*.

Comments

The value name and value are read from the registry path **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***\Parameters,** where *DriverName* is the driver name specified as input to **VC_OpenProfileAccess**.

To store values under a driver's **\Parameters** registry key, call **VC_WriteProfile**.

Note: A function named **VC_ReadProfile** is also provided by *VCKernel.lib* for kernel-mode video capture drivers. To see that function's description, click here.

See Also

VC_ReadProfileString, **VC_ReadProfileUser**, **VC_WriteProfile**, **VC_WriteProfileUser**

```
BOOL
  VC_ReadProfileString(
    PVC_PROFILE_INFO pProfile,
    PWCHAR ValueName,
    PWCHAR ValueString,
    DWORD ValueLength
    );
```

The **VC_ReadProfileString** function reads the string value associated with the specified value name, under the driver's **\Parameters** registry key.

Parameters

pProfile
Address of the VC_PROFILE_INFO structure returned by **VC_OpenProfileAccess**.

ValueName
Pointer to a UNICODE string identifying the name of a registry value.

ValueString
Pointer to a buffer to receive the requested string value.

ValueLength
Length, in bytes, of the buffer pointed to by *ValueString*.

Return Value

Returns TRUE if the operation succeeds. If the *ValueName* string does not exist in the registry, cannot be accessed, is not a REG_SZ (string) type, or if the specified buffer is not large enough to hold the returned string, the function returns FALSE.

Comments

If the operation succeeds, the **VC_ReadProfileString** function copies the string value into the specified buffer.

The value name and value are read from the registry path **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***\Parameters**, where *DriverName* is the driver name specified as input to **VC_OpenProfileAccess**.

See Also

VC_ReadProfile, **VC_ReadProfileUser**, **VC_WriteProfile**, **VC_WriteProfileUser**

VC_ReadProfileUser

```
DWORD
  VC_ReadProfileUser(
    PVC_PROFILE_INFO pProfile,
    PWCHAR ValueName,
    DWORD dwDefault
    );
```

The **VC_ReadProfileUser** function reads the DWORD value associated with the specified value name, under the user's profile information in the registry.

Parameters

pProfile
Address of the VC_PROFILE_INFO structure returned by **VC_OpenProfileAccess**.

ValueName
Pointer to a UNICODE string identifying the name of a registry value.

dwDefault
Specifies a default value that is returned if an error occurs locating or reading the requested value.

Return Value

Returns the value assigned to the *ValueName* string, if the operation succeeds. If the *ValueName* string does not exist in the registry, cannot be accessed, or is not a REG_DWORD type, the function returns the value specified by *dwDefault*.

Comments

The value name and value are read from the registry path **HKEY_CURRENT_USER \Software \Microsoft \Multimedia \Video Capture ***DriverName***,** where *DriverName* is the driver name specified as input to **VC_OpenProfileAccess**.

To store values under this registry key, call **VC_WriteProfileUser**.

See Also

VC_ReadProfile, **VC_ReadProfileString**, **VC_WriteProfile**, **VC_WriteProfileUser**

VC_RemoveDriver

LRESULT
VC_RemoveDriver(
PVC_PROFILE_INFO *pProfile*
);

The **VC_RemoveDriver** function unloads a kernel-mode video capture driver and marks the kernel-mode driver service for deletion.

Parameters

pProfile
Address of the VC_PROFILE_INFO structure returned by **VC_OpenProfileAccess**.

Return Value

Returns one the following error values.

Value	Definition
DRVCNF_OK	Indicates the driver service has been marked for deletion.
DRVCNF_CANCEL	Indicates an error occurred.

Comments

The **VC_RemoveDriver** function performs the following operations, in order:

1. Unloads the kernel-mode driver, if it is loaded.
2. Sets the kernel-mode driver service's start type to SERVICE_DEMAND_START, so it will not automatically reload when the system is restarted. (For details, see **ChangeServiceConfig** in the Win32 SDK.)
3. Calls **DeleteService**, described in the Win32 SDK, to mark the kernel-mode driver service for deletion.

A user-mode driver should call **VC_RemoveDriver** when its **DriverProc** function receives a DRV_REMOVE message. Before calling **VC_RemoveDriver**, the driver should call **VC_ConfigAccess** to determine if the client has Administrators privilege.

The profile information handle specified by *pProfile* remains open after this function removes the registry entry. Use **VC_CloseProfileAccess** to close the profile information handle after calling **VC_RemoveDriver**.

VC_SetKeyColourPalIdx

BOOL
VC_SetKeyColourPalIdx(
VCUSER_HANDLE *vh***,**

WORD *PaletteIndex*
);

The **VC_SetKeyColourPalIdx** function requests a kernel-mode video capture driver to set the overlay destination image's key color to a specified palette index.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

PaletteIndex
Index into the current color palette. This value should be one that can be passed to the PALETTEINDEX macro to obtain a COLORREF value. The PALETTEINDEX macro and the COLORREF type are described in the Win32 SDK.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

Before setting the key color, the driver should call **VC_GetOverlayMode** and test the VCO_KEYCOLOUR_FIXED flag to determine if the key color can be set. If the color can be set, the driver should test the VCO_KEYCOLOUR_RGB flag to determine if, based on the current data format, the kernel-mode driver stores the key color as an RGB color or as a palette index number. If the key color is stored as an RGB color, the driver should call **VC_SetKeyColourRGB** to set it. If the key color is stored as a palette index, the driver should call **VC_SetKeyColourPalIdx** to set it.

The **VC_SetKeyColourPalIdx** function calls **DeviceIoControl** (described in the Win32 SDK) to send an IOCTL_VIDC_SET_KEY_PALIDX control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives this control code, its **SetKeyPalIdxFunc** function is called.

See Also

VC_GetKeyColour, **VC_SetKeyColourRGB**

VC_SetKeyColourRGB

BOOL
VC_SetKeyColourRGB(
VCUSER_HANDLE *vh*,
PRGBQUAD *pRGB*
);

The **VC_SetKeyColourRGB** function requests a kernel-mode video capture driver to set the overlay destination image's key color to a specified RGB color.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

pRGB
Pointer to an RGBQUAD structure containing an RGB color specification. (The RGBQUAD structure is described in the Win32 SDK.)

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

Before setting the key color, the driver should call **VC_GetOverlayMode** and test the VCO_KEYCOLOUR_FIXED flag to determine if the key color can be set. If the color can be set, the driver should test the VCO_KEYCOLOUR_RGB flag to determine if, based on the current

data format, the kernel-mode driver stores the key color as an RGB color or as a palette index number. If the key color is stored as an RGB color, the driver should call **VC_SetKeyColourRGB** to set it. If the key color is stored as a palette index, the driver should call **VC_SetKeyColourPalIdx** to set it.

The **VC_SetKeyColourRGB** function calls **DeviceIoControl** (described in the Win32 SDK) to send an IOCTL_VIDC_SET_KEY_RGB control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives this control code, its **SetKeyRGBFunc** function is called.

See Also

VC_GetKeyColour, **VC_SetKeyColourPalIdx**

VC_SetOverlayOffset

```
BOOL
  VC_SetOverlayOffset(
    VCUSER_HANDLE vh,
    PRECT prc
    );
```

The **VC_SetOverlayOffset** function requests a kernel-mode video capture driver to set the overlay offset rectangle. This rectangle defines which portion of the frame buffer appears in the overlay window.

Parameters

vh

Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

prc

Pointer to a RECT structure containing screen coordinates that describe the offset rectangle. (The RECT structure is described in the Win32 SDK.)

Return Value

Returns TRUE if the operation succeeds. Otherwise, returns FALSE.

Comments

User-mode drivers using *VCUser.lib* typically call **VC_SetOverlayOffset** when a client sends the DVM_SRC_RECT message for the VIDEO_EXTERNALOUT channel. If the source image is larger than the overlay window, the client typically provides scroll bars to allow the user to pan across the source image, and sends DVM_SRC_RECT messages when the user scrolls the view.

The *prc* parameter specifies the portion of the frame buffer to send to the overlay window. The kernel-mode driver should display the top left corner of the specified offset rectangle in the top left corner of the overlay window.

The **VC_SetOverlayOffset** function calls **DeviceIoControl** (described in the Win32 SDK) to send an IOCTL_VIDC_OVERLAY_OFFSET control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives this control code, its **SetOverlayOffsetFunc** function is called.

VC_SetOverlayRect

```
BOOL
  VC_SetOverlayRect(
    VCUSER_HANDLE vh,
    POVERLAY_RECTS pOR
    );
```

The **VC_SetOverlayRect** function requests a kernel-mode video capture driver to set the overlay region of an output display.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

pOR
Pointer to an OVERLAY_RECTS structure describing one or more overlay rectangles, using screen coordinates.

Return Value

Returns TRUE if the operation succeeds. Otherwise, returns FALSE.

Comments

User-mode drivers using *VCUser.lib* typically call the **VC_SetOverlayRect** function when a client sends the DVM_DST_RECT message for the VIDEO_EXTERNALOUT channel. The function specifies the portion of the display device to use as an overlay area.

If the OVERLAY_RECTS structure contains a single rectangle, the rectangle defines the overlay region. If the structure contains more than one rectangle, the first rectangle defines the bounding rectangle for the overlay region, and additional rectangles represent parts of the overlay region. All rectangles are specified using screen coordinates.

The driver should specify complex (multiple) rectangles only if the kernel-mode driver returns the VCO_COMPLEX_RECT flag when the **VC_GetOverlayMode** function is called. The sample video capture drivers do not accept complex rectangle specifications.

The **VC_SetOverlayRect** function calls **DeviceIoControl** (described in the Win32 SDK) to send an IOCTL_VIDC_OVERLAY_RECTS control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives this control code, its **SetOverlayRectsFunc** function is called.

VC_StreamAddBuffer

```
BOOL
  VC_StreamAddBuffer(
    VCUSER_HANDLE vh,
    LPVIDEOHDR lpvh
    );
```

The **VC_StreamAddBuffer** function adds a buffer to a video capture driver's queue of buffers used to receive video capture input frames.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

lpvh
Pointer to a VIDEOHDR structure defining the buffer to be queued.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

A user-mode video capture driver using *VCUser.lib* should call **VC_StreamAddBuffer** when its **DriverProc** function receives a DVM_STREAM_ADDBUFFER message.

If the driver has not called **VC_StreamStart** to start reading frames, the specified buffer is added to the user-mode driver's buffer queue. If **VC_StreamStart** has been called, the **VC_StreamAddBuffer** function tries to send the buffer to the kernel-mode driver. To limit the amount of memory the kernel-mode driver must lock, a maximum of two buffers are queued to the kernel-mode driver at one time. Any additional buffers are queued in the user-mode driver until the kernel-mode driver's queue contains less than two buffers. The **VC_StreamAddBuffer** function always returns after the buffer is queued, without waiting for captured data to be placed

in the buffer.

To send a buffer to the kernel-mode driver, the **VC_StreamAddBuffer** function calls **DeviceIoControl** (described in the Win32 SDK), specifying an IOCTL_VIDC_ADD_BUFFER control code. If the kernel-mode driver is using *VCKernel.lib*, and if the specified buffer length is smaller than the image size value specified with **VC_SetImageSize**, the buffer is not queued. If the buffer size is acceptable, *VCKernel.lib* places the buffer's I/O request packet (IRP) in a queue, where it stays until needed to receive a captured frame. (For more information about IRPs, see the *Kernel-Mode Drivers Design Guide*.)

If the kernel-mode driver fails to lock the buffer in the client's address space (because the buffer is too large for the system's memory resources), then *VCUser.lib* sends IOCTL_VIDC_CAP_TO_SYSBUF, IOCTL_VIDC_PARTIAL_CAPTURE, and IOCTL_VIDC_FREE_SYSBUF, control codes to request the kernel-mode driver to copy frame buffer data into a system-allocated buffer. The data is then copied to the client's buffers as smaller, partial frame sections.

VC_StreamFini

BOOL
VC_StreamFini(
VCUSER_HANDLE *vh*
);

The **VC_StreamFini** ends a streaming operation that was initiated by calling **VC_StreamInit**.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

A user-mode video capture driver using *VCUser.lib* should call **VC_StreamFini** when its **DriverProc** function receives a DVM_STREAM_FINI message.

A driver should call the **VC_StreamFini** function only after it has called **VC_StreamStop**. The function removes the thread that was created by **VC_StreamInit**.

The **VC_StreamFini** function calls **DeviceIoControl** (described in the Win32 SDK) to send an IOCTL_VIDC_STREAM_RESET control code, followed by an IOCTL_VIDC_STREAM_FINI control code, to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives the VIDC_STREAM_FINI control code, its **StreamFiniFunc** function is called.

VC_StreamInit

BOOL
VC_StreamInit(
VCUSER_HANDLE *vh*,
PVCCALLBACK *pCallback*,
ULONG *FrameRate*
);

The **VC_StreamInit** function initializes a video capture stream.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

pCallback
Pointer to a client notification target (callback function or window handle) received as input with

a DVM_STREAM_INIT message.

FrameRate
Capture rate (microseconds per frame) received as input with a DVM_STREAM_INIT message.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

A user-mode video capture driver using *VCUser.lib* should call **VC_StreamInit** when its **DriverProc** function receives a DVM_STREAM_INIT message.

This function performs the following operations:

- Stores the *pCallback* value for use when notifying the client. For more information about notifying clients, see Notifying Clients from Video Capture Drivers.
- Creates a separate user-mode worker thread to handle stream operations.

The **VC_StreamInit** function calls **DeviceIoControl** (described in the Win32 SDK) to send an IOCTL_VIDC_STREAM_INIT control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives this control code, its **StreamInitFunc** function is called. The *FrameRate* value is passed to the kernel-mode driver along with the IOCTL_VIDC_STREAM_INIT message.

When the initialization operation is complete, the worker thread sends the client an MM_DRVM_OPEN callback message.

VC_StreamReset

BOOL
VC_StreamReset(
VCUSER_HANDLE *vh*
);

The **VC_StreamReset** function stops the capture stream, if **VC_StreamStop** has not been called, and then cancels all queued buffers.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

A user-mode video capture driver using *VCUser.lib* should call **VC_StreamReset** when its **DriverProc** function receives a DVM_STREAM_RESET message.

The **VC_StreamReset** function calls **DeviceIoControl** (described in the Win32 SDK) to send an IOCTL_VIDC_STREAM_RESET control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives this control code, its **StreamStopFunc** function is called if the capture stream has not been previously stopped. Then *VCKernel.lib* returns unused buffers to the user-mode driver. Code in *VCUser.lib* marks all queued buffers as done by setting VHDR_DONE in the **dwFlags** member of each buffer's VIDEOHDR structure, and sends an MM_DRVM_DATA callback message to the client for each buffer.

See Also

VC_StreamStop

VC_StreamStart

```
BOOL
  VC_StreamStart(
    VCUSER_HANDLE vh
    );
```

The **VC_StreamStart** function starts a capture stream.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

A user-mode video capture driver using *VCUser.lib* should call **VC_StreamStart** when its **DriverProc** function receives a DVM_STREAM_START message.

A driver cannot call **VC_StreamStart** unless it has previously called **VC_StreamInit**.

If the driver has previously called **VC_StreamAddBuffer** to add buffers to its local queue, the **VC_StreamStart** function calls **DeviceIoControl** (described in the Win32 SDK), specifying an IOCTL_VIDC_ADD_BUFFER control code, to send up to two buffers to the kernel-mode driver.

The **VC_StreamStart** function then calls **DeviceIoControl** to send an IOCTL_VIDC_STREAM_START control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives this control code, its **StreamStartFunc** function is called to start the capturing of input data.

Each time the kernel-mode driver fills a buffer and returns it to the user-mode driver, code in *VCUser.lib* sets VHDR_DONE in the **dwFlags** member of the buffer's VIDEOHDR structure, and sends an MM_DRVM_DATA callback message to the client.

VC_StreamStop

```
BOOL
  VC_StreamStop(
    VCUSER_HANDLE vh
    );
```

The **VC_StreamStop** function stops a capture stream.

Parameters

vh
Handle to the kernel-mode driver, obtained from **VC_OpenDevice**.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

A user-mode video capture driver using *VCUser.lib* should call **VC_StreamStop** when its **DriverProc** function receives a DVM_STREAM_STOP message.

The **VC_StreamStop** function then calls **DeviceIoControl** to send an IOCTL_VIDC_STREAM_STOP control code to the specified kernel-mode driver. When a kernel-mode driver using *VCKernel.lib* receives this control code, its **StreamStopFunc** function is called.

After your driver calls **VC_StreamStop**, kernel-mode driver will fill and return the buffer it is currently using, but it will not dequeue any more buffers.

VC_WriteProfile

BOOL
VC_WriteProfile(
PVC_PROFILE_INFO *pProfile*,
PWCHAR *ValueName*,
DWORD *Value*
);

The **VC_WriteProfile** function assigns the specified DWORD value to the specified value name, under the driver's **\Parameters** registry key.

Parameters

pProfile
Address of the VC_PROFILE_INFO structure returned by **VC_OpenProfileAccess**.

ValueName
Pointer to a UNICODE string identifying the name of a registry value.

Value
Value to be assigned to the *ValueName* in the registry.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

The value name and value are written to the registry path **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***\Parameters**, where *DriverName* is the driver name specified as input to **VC_OpenProfileAccess**.

If the **\Parameters** subkey does not exist, it is created. If the specified value name does not exist under the **\Parameters** subkey, it is created.

To read values from a driver's **\Parameters** registry key, call **VC_ReadProfile**.

Note: A function named **VC_WriteProfile** is also provided by *VCKernel.lib* for kernel-mode video capture drivers. To see that function's description, click here.

See Also

VC_ReadProfile, **VC_ReadProfileString**, **VC_ReadProfileUser**, **VC_WriteProfileUser**

VC_WriteProfileUser

BOOL
VC_WriteProfileUser(
PVC_PROFILE_INFO *pProfile*,
PWCHAR *ValueName*,
DWORD *Value*
);

The **VC_WriteProfileUser** function assigns the specified DWORD value to the specified value name, under the user's profile information in the registry.

Parameters

pProfile
Address of the VC_PROFILE_INFO structure returned by **VC_OpenProfileAccess**.

ValueName
Pointer to a UNICODE string identifying the name of a registry value.

Value
Value to be assigned to *ValueName* in the registry.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

The value name and value are written to the registry path **HKEY_CURRENT_USER \Software \Microsoft \Multimedia \Video Capture ***DriverName***,** where *DriverName* is the driver name specified as input to **VC_OpenProfileAccess**. If the specified value name does not exist, it is created.

To read values from this registry key, call **VC_ReadProfileUser**.

See Also

VC_ReadProfile, **VC_ReadProfileString**, **VC_ReadProfileUser**, **VC_WriteProfile**

Structures, *VCUser.lib*

This section describes the structures available to user-mode video capture drivers using *VCUser.lib*.

CONFIG_INFO

```
typedef struct _CONFIG_INFO {
ULONG ulSize;
BYTE ulData[1];
} CONFIG_INFO, *PCONFIG_INFO;
```

The CONFIG_INFO structure is used to describe a customized structure of configuration information. It is defined in *vcstruct.h*.

Members

ulSize
Size of the customized structure.

ulData[1]
First data member of the customized structure.

Comments

The CONFIG_INFO structure is a generic structure that allows you to define customized structures for storing configuration information, and to pass pointers to those structures from your user-mode driver to your kernel-mode driver with calls to **VC_ConfigDisplay**, **VC_ConfigFormat**, and **VC_ConfigSource**.

To pass configuration information to your kernel-mode driver, define customized structures, fill them, and then cast them to the PCONFIG_INFO type when passing them to **VC_ConfigDisplay**, **VC_ConfigFormat**, or **VC_ConfigSource**. See the sample video capture drivers for examples.

Because the structures are passed from user mode to kernel mode, you cannot include pointers as structure members.

DRAWBUFFER

```
typedef struct _DRAWBUFFER {
PUCHAR  lpData;
ULONG   ulWidth;
ULONG   ulHeight;
ULONG   Format;
RECT    rcSource;
} DRAWBUFFER, *PDRAWBUFFER;
```

The DRAWBUFFER structure describes the data, frame size, and format for drawing a video frame. It is defined in *vcstruct.h*.

Members

lpData
Pointer to frame data.

ulWidth
Frame width in pixels.

ulHeight
Frame height in pixels.

Format
Video data format.

rcSource
Rectangle describing portion of the frame to be drawn.

Comments

User-mode video capture drivers use the DRAWBUFFER structure when calling **VC_DrawFrame**. The **ulWidth** and **ulHeight** members describe the entire frame. The **rcSource** member specifies the portion of the frame to be drawn.

OVERLAY_RECTS

```
typedef struct _OVERLAY_RECTS {
    ULONG ulCount;      // total number of rects in array
    RECT rcRects[1];    // ulCount rectangles
} OVERLAY_RECTS, *POVERLAY_RECTS;
```

The OVERLAY_RECTS structure describes the rectangles constituting the active area of an overlay display. The structure is defined in *vcstruct.h*.

Members

ulCount
Number of rectangles specified by **rcRects**.

rcRects[1]
Array of RECT structures containing rectangle descriptions. Rectangles are specified in screen coordinates. The RECT structure is described in the Win32 SDK.

Comments

User-mode video capture drivers use the OVERLAY_RECTS structure when calling **VC_SetOverlayRect**.

VCCALLBACK

```
typedef struct _VCCALLBACK {
    DWORD dwCallback;
    DWORD dwFlags;
    HDRVR hDevice;
    DWORD dwUser;
} VCCALLBACK, *PVCCALLBACK;
```

The VCCALLBACK structure contains callback information needed by the **VC_StreamInit** function. The structure is defined in *vcuser.h*.

Members

dwCallback
Contains either the address of a callback function, a window handle, or NULL, based on flags set in the **dwFlags** member. The driver should copy the value of the VIDEO_STREAM_INIT_PARMS structure's **dwCallback** member into this member.

dwFlags
Contains flags. Can contain one (or none) of the following flags.

Flag	Definition
CALLBACK_WINDOW	Indicates **dwCallback** contains a window handle.
CALLBACK_FUNCTION	Indicates **dwCallback** contains a callback function address.

The driver should copy the value of the VIDEO_STREAM_INIT_PARMS structure's **dwFlags** member into this member.

hDevice
Contains a handle to a video channel. The driver should copy the value of the VIDEO_STREAM_INIT_PARMS structure's **hVideo** member into this member.

dwUser
Contains client-specified instance data passed to the callback function, if CALLBACK_FUNCTION is set in **dwFlags**. The driver should copy the value of the VIDEO_STREAM_INIT_PARMS structure's **dwCallbackInst** member into this member.

Comments

As indicated by the preceding member descriptions above, the VCCALLBACK structure is used to pass VIDEO_STREAM_INIT_PARMS structure members to **VC_StreamInit**.

Functions, *VCKernel.lib*

This section describes the functions available to kernel-mode video capture drivers using *VCKernel.lib*. The functions are listed in alphabetical order.

VC_AccessData

BOOLEAN
VC_AccessData(
PDEVICE_INFO *pDevInfo*,
PUCHAR *pData*,
ULONG *Length*,
PACCESS_ROUTINE *pAccessFunc*,
PVOID *pContext*
);

The **VC_AccessData** function provides a means by which kernel-mode video capture drivers can be protected from encountering access violations when referencing user-mode data.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

pData
Pointer to user-mode data.

Length
Length of data specified by *pData*.

pAccessFunc
Pointer to a callback function that accesses the specified data. The callback function must use the following prototype format:

```
BOOLEAN AccessFunc (PDEVICE_INFO pDevInfo, PUCHAR pData, ULONG Length, PVOID pCon
```

pContext
Pointer to driver-supplied context information that is passed to the callback function.

Return Value

If the data specified by *pData* can be accessed, the function returns the value returned by the callback function. Otherwise the function returns FALSE.

Comments

The kernel-mode driver supplies the address of data to be accessed, along with the address of a callback function that performs an operation on the data, such as copying a bitmap into the frame buffer. The **VC_AccessData** function wraps the callback function in an exception handler so that if an access violation occurs, the kernel-mode driver can continue to execute. If an access violation occurs, the exception handler returns FALSE.

The function can only be called within the context of the user's thread. It must not be called from the **InterruptAcknowledge** or **CaptureService** functions, because they execute within a system context.

VC_AllocMem

PVOID
VC_AllocMem(
PDEVICE_INFO *pDevInfo*,
ULONG *Length*
);

The **VC_AllocMem** function allocates a specified amount of nonpaged memory for use by a kernel-mode video capture driver.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

Length
Size, in bytes, of memory to be allocated.

Return Value

Returns a pointer to the allocated memory, if the operation succeeds. Otherwise returns NULL.

Comments

The **VC_AllocMem** function calls **ExAllocatePool**, with a pool type of **NonPagedPool**. The allocated memory can be referenced by the driver's **InterruptAcknowledge** and **CaptureService** function.

VC_Cleanup

VOID
VC_Cleanup(
PDRIVER_OBJECT *pDriverObject*
);

The **VC_Cleanup** function deallocates the system resources that were allocated by a previous call to **VC_Init**.

Parameters

pDriverObject
The driver object pointer received as input to the driver's **DriverEntry** function.

Return Value

None.

Comments

A kernel-mode video capture driver should call **VC_Cleanup** if a failure occurs within its **DriverEntry** function after **VC_Init** has been called.

Additionally, the *VCKernel.lib* library specifies **VC_Cleanup** as the function to be called when the kernel-mode driver is unloaded (by assigning the function's address to the driver object's **DriverUnload** member).

The **VC_Cleanup** function performs the following operations, in the order listed:

1. Cancels outstanding IRPs.
2. Calls the kernel-mode driver's **CleanupFunc** function.
3. Unmaps mapped I/O memory space.
4. Removes the device's interrupt objects.
5. Releases system resources (interrupt number, DMA channel, and so on) reserved for the device.
6. Removes the device object associated with the device.

For more information about driver objects, device objects and interrupt objects, see the *Kernel-Mode Drivers Design Guide*.

VC_ConnectInterrupt

BOOLEAN
VC_ConnectInterrupt(
PDEVICE_INFO *pDevInfo*,
ULONG *Interrupt*,
BOOLEAN *bLatched*
);

The **VC_ConnectInterrupt** function creates a connection between the specified interrupt number and a kernel-mode video capture driver's interrupt service routine.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

Interrupt
Interrupt number to use.

bLatched
Set to TRUE if the interrupt is latched or FALSE if the interrupt is level-sensitive.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

The **VC_ConnectInterrupt** function calls **HalGetInterruptVector** to obtain a system interrupt vector and **IoConnectInterrupt** to connect *VCKernel.lib*'s generic ISR to the interrupt vector. (When an interrupt occurs, *VCKernel.lib*'s generic ISR calls the driver's **InterruptAcknowledge** function.)

Before calling **VC_ConnectInterrupt**, a kernel-mode driver must call **VC_GetResources**. It must also call **VC_GetCallbackTable** and fill in the returned callback table.

VC_Delay

VOID
VC_Delay(
int *nMillisecs*
);

The **VC_Delay** function delays execution of a kernel-mode video capture driver's current thread.

Parameters

nMillisecs
Delay interval, in milliseconds.

Return Value

None.

Comments

The **VC_Delay** function puts the calling kernel-mode thread into a non-alertable wait state, in kernel mode, for at least the specified number of milliseconds.

To call **VC_Delay**, your driver's IRQL must be less than DISPATCH_LEVEL.

See Also

VC_Stall

VC_FreeMem

VOID
VC_FreeMem(
PDEVICE_INFO *pDevInfo*,
PVOID *pData*,
ULONG *Length*
);

The **VC_FreeMem** function frees memory space that was allocated by calling **VC_AllocMem**.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

pData
Pointer that was returned by **VC_AllocMem**.

Length
Size of memory allocation that was requested using **VC_AllocMem.**

Return Value

None.

Comments

The **VC_FreeMem** function calls **ExFreePool** to deallocate the specified memory space.

VC_GetCallbackTable

PVC_CALLBACK
VC_GetCallbackTable(
PDEVICE_INFO *pDevInfo*
);

The **VC_GetCallbackTable** function returns the address of *VCKernel.lib*'s callback table.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

Return Value

Pointer to a VC_CALLBACK structure.

Comments

The **VC_GetCallbackTable** function returns the address of a VC_CALLBACK structure. Kernel-mode video capture drivers using *VCKernel.lib* must call this function and then fill in the VC_CALLBACK structure with the addresses of the driver's callback functions. Code in *VCKernel.lib* uses the structure as a dispatch table when calling the driver's callback functions.

The driver must call **VC_GetCallbackTable** and fill in the table from within its **DriverEntry**

function. Because the table contains the addresses of interrupt handlers, it should be filled in before the driver attempts to initialize hardware, in case the initialization process generates interrupts.

VC_GetFrameBuffer

PUCHAR
VC_GetFrameBuffer(
PDEVICE_INFO *pDevInfo*
);

The **VC_GetFrameBuffer** function returns a pointer to the system address space corresponding to the device's frame buffer.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

Return Value

Pointer to the system address space to which the frame buffer is mapped.

Comments

A kernel-mode driver must call the **VC_GetFrameBuffer** function to obtain the mapped address of the device's frame buffer. The frame buffer's bus-relative physical address range is mapped to nonpaged system space by the **VC_GetResources** function.

VC_GetHWInfo

PVOID
VC_GetHWInfo(
PDEVICE_INFO *pDevInfo*
);

The **VC_GetHWInfo** function returns a pointer to the driver-specified structure that was allocated by a previous call to **VC_Init**.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

Return Value

Pointer to the driver-specified structure.

Comments

When a kernel-mode video capture driver calls **VC_Init**, it specifies the size of a structure for which nonpaged space is allocated within the device object's device extension area. The driver can call **VC_GetHWInfo** to obtain the address of the area that was allocated for this structure. Drivers can use this structure to store device-specific context information.

For more information about device objects and device extensions, see the *Kernel-Mode Drivers Reference*.

VC_GetResources

BOOLEAN
VC_GetResources(
PDEVICE_INFO *pDevInfo*,
PDRIVER_OBJECT *pDriverObject*,
PUCHAR *pPortBase*,
ULONG *NrOfPorts*,

ULONG *Interrupt*,
BOOLEAN *bLatched*,
PUCHAR *pFrameBuffer*,
ULONG *FrameLength*,
);

The **VC_GetResources** function reserves system resources for a device, and maps the device's I/O address space and frame buffer into system address space.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

pDriverObject
The driver object pointer received as input to the driver's **DriverEntry** function.

pPortBase
Bus-relative physical base address of the device's I/O port address space.

NrOfPorts
Number of port addresses.

Interrupt
Interrupt number.

bLatched
Set to TRUE if the interrupt is latched or FALSE if the interrupt is level-sensitive.

pFrameBuffer
Bus-relative physical base address of the device's frame buffer.

FrameLength
Length of the frame buffer.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE.

Comments

A kernel-mode video capture driver using *VCKernel.lib* must call **VC_GetResources** from within its **DriverEntry** function. The **VC_GetResources** function performs the following operations, in the order listed:

1. Determines the bus type.
2. Maps the device's I/O port space to system space.
3. Maps the device's frame buffer to system space.
4. Reserves the interrupt number, mapped I/O port space, and mapped frame buffer space for use by the device.

The **VC_GetResources** function calls **HalTranslateBusAddress**, **MmMapIoSpace**, and **IoReportResourceUsage**.

VC_In

BYTE
VC_In(
PDEVICE_INFO *pDevInfo*,
BYTE *bOffset*
);

The **VC_In** function reads one byte from a device's mapped I/O port address space.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

bOffset
Offset from the mapped I/O port's base address.

Return Value
Byte contents of the specified port address offset.

Comments
A kernel-mode video capture driver specifies the base address of a device's I/O port address space when calling **VC_GetResources**. The *bOffset* parameter to **VC_In** specifies an offset from that base address.

See Also
VC_Out

VC_Init

PDEVICE_INFO
VC_Init(
PDRIVER_OBJECT *pDriverObject*,
PUNICODE_STRING *szRegistryPathName*,
ULONG *HWInfoSize*
);

The **VC_Init** function creates a device object and stores the device name in the registry.

Parameters
pDriverObject
The driver object pointer that was received as input to the driver's **DriverEntry** function.

szRegistryPathName
Pointer to a string containing the registry path to the driver's subkey. Use the path name that was received as the *RegistryPathName* argument to **DriverEntry**.

HWInfoSize
Size, in bytes, of a driver-defined structure used for storing device-specific context information. Can be zero.

Return Value
Returns a pointer to a DEVICE_INFO structure.

Comments
A kernel-mode video capture driver using *VCKernel.lib* must call **VC_Init** from within its **DriverEntry** function, before calling any other *VCKernel.lib* functions.

The DEVICE_INFO structure pointer returned by **VC_Init** is used as input to subsequent calls to other *VCKernel.lib* functions. Contents of the DEVICE_INFO structure are not available to the kernel-mode driver.

The driver uses the *HWInfoSize* parameter to specify the size of a driver-defined structure. The **VC_Init** function allocates nonpaged space for this structure within the device object's *device extension* area. The driver can call **VC_GetHWInfo** to obtain the address of the area that was allocated for the structure. Drivers can use this structure to store device-specific context information. (For more information about device objects and device extensions, see the *Kernel-Mode Drivers Design Guide*.)

The **VC_Init** function calls **IoCreateDevice** to create a device object. The device's name is "vidcap", which is defined by DD_VIDCAP_DEVICE_NAME_U in *ntddvidc.h*, with an appended number (0, 1, 2, and so on). The function writes this device name into the registry, under the registry path **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***\Parameters**, where *DriverName* is the driver name that the user-mode driver specified as input to **VC_OpenProfileAccess** when it installed the kernel-mode driver.

VC_Out

VOID
VC_Out(
PDEVICE_INFO *pDevInfo*,
BYTE *bOffset*,
BYTE *bData*
);

The **VC_Out** function writes one byte into a device's mapped I/O port address space.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

bOffset
Offset from the mapped I/O port's base address.

bData
Data byte to be written.

Return Value

None.

Comments

A kernel-mode video capture driver specifies the base address of a device's I/O port address space when calling **VC_GetResources**. The *bOffset* parameter to **VC_Out** specifies an offset from that base address.

See Also

VC_In

VC_ReadProfile

DWORD
VC_ReadProfile(
PDEVICE_INFO *pDevInfo*,
PWCHAR *szValueName*,
DWORD *dwDefault*
);

The **VC_ReadProfile** function reads the DWORD value associated with the specified value name, under the driver's **\Parameters** registry key.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

szValueName
Pointer to a UNICODE string identifying the name of a registry value.

dwDefault
Specifies a default value that is returned if an error occurs locating or reading the requested value.

Return Value

Returns the value assigned to the *ValueName* string, if the operation succeeds. Otherwise the function returns the value specified by *dwDefault*.

Comments

The value name and value are read from the registry path

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services*DriverName***\Parameters**, where *DriverName* is the driver name specified as input to **VC_OpenProfileAccess**.

To store values under a driver's **\Parameters** registry key, call **VC_WriteProfile**.

Note: A function named **VC_ReadProfile** is also provided by *VCUser.lib* for user-mode video capture drivers. To see that function's description, click here.

VC_SetImageSize

VOID
VC_SetImageSize(
PDEVICE_INFO *pDevInfo***,**
int *ImageSize*
);

The **VC_SetImageSize** function specifies the maximum number of bytes needed to store an image using the current format.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

ImageSize
Maximum number of bytes needed to store an image.

Comments

A kernel-mode video capture driver should call **VC_SetImageSize** whenever a new data format is selected. The number specified for *ImageSize* should represent the largest number of bytes that an image can be, using the current format. In other words, the value should represent the smallest size that a buffer can be in order to store an entire capture frame, using the current format.

Code within *VCKernel.lib* uses the specified *ImageSize* value to determine if each buffer received with an IOCTL_VIDC_ADD_BUFFER control code is large enough to receive the frame buffer contents. Additionally, if the system's memory resources are too low to allow locking of client-specified buffers into the client's address space, *VCKernel.lib* allocates a system buffer of *ImageSize* size to receive the frame buffer contents, which are then copied into the client's buffers in smaller segments.

See Also

VC_StreamAddBuffer

VC_Stall

VOID
VC_Stall(
int *nMicrosecs*
);

The **VC_Stall** function stalls the current processor for the specified number of microseconds.

Parameters

nMicrosecs
Number of microseconds to stall.

Return Value

None.

Comments

The **VC_Stall** function causes the current processor to execute a processor-specific wait loop until

the specified time has passed. The function is useful for pausing between device access operations, if there is a potential for fast processors to send instructions to the device at a speed that is too high for the device.

Delays longer than 25 microseconds are not recommended.

Your driver can call **VC_Install** when executing at any IRQL.

See Also

VC_Delay

VC_SynchronizeDPC

BOOLEAN
VC_SynchronizeDPC(
PDEVICE_INFO *pDevInfo*,
PSYNC_ROUTINE *pSync*,
PVOID *pContext*
);

The **VC_SynchronizeDPC** function synchronizes access to objects that are referenced by a kernel-mode video capture driver's **CaptureService** function.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

pSync
Pointer to a callback function. The callback function must use the following prototype format:

```
BOOLEAN pSync(PVOID pContext);
```

pContext
Pointer to context information to be passed to the callback function.

Return Value

Returns the callback function's return value.

Comments

A driver's **CaptureService** function executes at an IRQL of DISPATCH_LEVEL. If other code within the driver must access the same objects (frame buffers, data structures, and so on) that the **CaptureService** function references, then all code that references the objects, *including* the code within **CaptureService**, must be synchronized by using **VC_SynchronizeDPC**.

To use **VC_SynchronizeDPC**, place each piece of code that references the objects to be protected into a callback function. Specify each callback function as a *pSync* parameter to a **VC_SynchronizeDPC** call. Typically, you use the *pContext* parameter to indicate the objects to be referenced.

The **VC_SynchronizeDPC** function acquires a spin lock and executes the callback function at an IRQL of DISPATCH_LEVEL. If you access an object only within code that is included in **VC_SynchronizeDPC** callbacks, then other processors cannot simultaneously access the object, and lower-priority code on the current processor cannot obtain access.

Code that calls **VC_SynchronizeDPC** must be executing at an IRQL of DISPATCH_LEVEL or lower.

See Also

VC_SynchronizeExecution

VC_SynchronizeExecution

BOOLEAN

```
VC_SynchronizeExecution(
    PDEVICE_INFO pDevInfo,
    PSYNC_ROUTINE pSync,
    PVOID pContext
);
```

The **VC_SynchronizeExecution** function synchronizes access to objects that are referenced by a kernel-mode video capture driver's **InterruptAcknowledge** function.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

pSync
Pointer to a callback function. The callback function must use the following prototype format:

```
BOOLEAN pSync(PVOID pContext);
```

pContext
Pointer to context information to be passed to the callback function.

Return Value

Returns the callback function's return value.

Comments

A driver's **InterruptAcknowledge** function executes at a device IRQL (DIRQL). If other code within the driver must access the same objects (generally, device registers) that the **InterruptAcknowledge** function references, then all code that references the objects, *except* the code within **InterruptAcknowledge**, must be synchronized by using **VC_SynchronizeExecution**.

To use **VC_SynchronizeExecution**, place each piece of code that references the objects to be protected into a callback function. Specify each callback function as a *pSync* parameter to a **VC_SynchronizeExecution** call. Typically, you use the *pContext* parameter to indicate the objects to be referenced.

The **VC_SynchronizeExecution** function calls the system's **KeSynchronizeExecution** function to acquire a spin lock and execute the callback function at the same DIRQL that the **InterruptAcknowledge** function uses. If you access an object only within **InterruptAcknowledge** or within code that is included in **VC_SynchronizeExecution** callbacks, then other processors cannot simultaneously access the object, and lower-priority code on the current processor cannot obtain access.

Code that calls **VC_SynchronizeExecution** must be executing at the device's DIRQL or lower. A driver's **CaptureService** function, which executes at an IRQL of DISPATCH_LEVEL, can call **VC_SynchronizeExecution**.

The driver's **InterruptAcknowledge** function should not call **VC_SynchronizeExecution**, because the Windows NT I/O Manager handles its synchronization.

See Also

VC_SynchronizeDPC

VC_WriteProfile

```
BOOL
VC_WriteProfile(
    PDEVICE_INFO pDevInfo,
    PWCHAR szValueName,
    DWORD ValueData
    );
```

The **VC_WriteProfile** function assigns the specified DWORD value to the specified value name, under the driver's **\Parameters** registry key.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

szValueName
Pointer to a UNICODE string identifying the name of a registry value.

ValueData
Value to be assigned to *ValueName* in the registry.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns false.

Comments

The value name and value are written to the registry path **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services***DriverName***\Parameters**, where *DriverName* is the driver name specified as input to **VC_OpenProfileAccess**.

To read values from a driver's **\Parameters** registry key, call **VC_ReadProfile**.

Note: A function named **VC_WriteProfile** is also provided by *VCUser.lib* for user-mode video capture drivers. To see that function's description, click here.

Structures, *VCKernel.lib*

This section describes the structures available to kernel-mode video capture drivers using *VCKernel.lib*.

VC_CALLBACK

```
typedef struct _VC_CALLBACK {
    BOOLEAN (*DeviceOpenFunc)(PDEVICE_INFO);
    BOOLEAN (*DeviceCloseFunc)(PDEVICE_INFO);
    BOOLEAN (*ConfigFormatFunc)(PDEVICE_INFO, PCONFIG_INFO);
    BOOLEAN (*ConfigDisplayFunc)(PDEVICE_INFO, PCONFIG_INFO);
    BOOLEAN (*ConfigSourceFunc)(PDEVICE_INFO, PCONFIG_INFO);
    DWORD   (*GetOverlayModeFunc) (PDEVICE_INFO);
    BOOLEAN (*SetKeyRGBFunc)(PDEVICE_INFO, PRGBQUAD);
    BOOLEAN (*SetKeyPalIdxFunc)(PDEVICE_INFO, ULONG);
    BOOLEAN (*SetOverlayRectsFunc)(PDEVICE_INFO, POVERLAY_RECTS);
    BOOLEAN (*SetOverlayOffsetFunc)(PDEVICE_INFO, PRECT);
    ULONG   (*GetKeyColourFunc)(PDEVICE_INFO);
    BOOLEAN (*CaptureFunc)(PDEVICE_INFO, BOOL);
    BOOLEAN (*OverlayFunc)(PDEVICE_INFO, BOOL);
    BOOLEAN (*StreamInitFunc)(PDEVICE_INFO, ULONG);
    BOOLEAN (*StreamFiniFunc)(PDEVICE_INFO);
    BOOLEAN (*StreamStartFunc)(PDEVICE_INFO);
    BOOLEAN (*StreamStopFunc)(PDEVICE_INFO);
    ULONG   (*StreamGetPositionFunc)(PDEVICE_INFO);
    BOOLEAN (*InterruptAcknowledge)(PDEVICE_INFO);
    ULONG (*CaptureService)(PDEVICE_INFO, PUCHAR, PULONG, ULONG);
    BOOLEAN (*DrawFrameFunc)(PDEVICE_INFO, PDRAWBUFFER);
    BOOLEAN (*CleanupFunc)(PDEVICE_INFO);
} VC_CALLBACK, * PVC_CALLBACK;
```

The VC_CALLBACK structure is a dispatch table used by *VCKernel.lib* to call functions provided by kernel-mode video capture drivers. The structure is defined in *vckernel.h*.

Members

DeviceOpenFunc
Pointer to a kernel-mode video capture driver's **DeviceOpenFunc** function.

DeviceCloseFunc
Pointer to a kernel-mode video capture driver's **DeviceCloseFunc** function.

ConfigFormatFunc
Pointer to a kernel-mode video capture driver's **ConfigFormatFunc** function.

ConfigDisplayFunc
Pointer to a kernel-mode video capture driver's **ConfigDisplayFunc** function.

ConfigSourceFunc
Pointer to a kernel-mode video capture driver's **ConfigSourceFunc** function.

GetOverlayModeFunc
Pointer to a kernel-mode video capture driver's **GetOverlayModeFunc** function.

SetKeyRGBFunc
Pointer to a kernel-mode video capture driver's **SetKeyRGBFunc** function.

SetKeyPalIdxFunc
Pointer to a kernel-mode video capture driver's **SetKeyPalIdxFunc** function.

SetOverlayRectsFunc
Pointer to a kernel-mode video capture driver's **SetOverlayRectsFunc** function.

SetOverlayOffsetFunc
Pointer to a kernel-mode video capture driver's **SetOverlayOffsetFunc** function.

GetKeyColourFunc
Pointer to a kernel-mode video capture driver's **GetKeyColourFunc** function.

CaptureFunc
Pointer to a kernel-mode video capture driver's **CaptureFunc** function.

OverlayFunc
Pointer to a kernel-mode video capture driver's **OverlayFunc** function.

StreamInitFunc
Pointer to a kernel-mode video capture driver's **StreamInitFunc** function.

StreamFiniFunc
Pointer to a kernel-mode video capture driver's **StreamFiniFunc** function.

StreamStartFunc
Pointer to a kernel-mode video capture driver's **StreamStartFunc** function.

StreamStopFunc
Pointer to a kernel-mode video capture driver's **StreamStopFunc** function.

StreamGetPositionFunc
Pointer to a kernel-mode video capture driver's **StreamGetPositionFunc** function.

InterruptAcknowledge
Pointer to a kernel-mode video capture driver's **InterruptAcknowledge** function.

CaptureService
Pointer to a kernel-mode video capture driver's **CaptureService** function.

DrawFrameFunc
Pointer to a kernel-mode video capture driver's **DrawFrameFunc** function.

CleanupFunc
Pointer to a kernel-mode video capture driver's **CleanupFunc** function.

Comments

A kernel-mode video capture driver using *VCKernel.lib* is responsible for filling in *VCKernel.lib*'s VC_CALLBACK structure. The driver obtains the structure's address by calling **VC_GetCallbackTable**. The driver should obtain the structure's address and fill in the table from within its **DriverEntry** function, before hardware initialization is attempted.

Driver Functions Used with *VCKernel.lib*

This section describes, in alphabetic order, the driver-supplied functions that kernel-mode video capture drivers must provide, if they are using *VCKernel.lib*.

CaptureFunc

BOOLEAN
*CaptureFunc***(**
PDEVICE_INFO *pDevInfo***,**
BOOL *bCapture*
);

The **CaptureFunc** function is provided by kernel-mode video capture drivers to enable and disable capturing video data. The **CaptureFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

bCapture
Set to TRUE if data capture is to be enabled, and FALSE if it is to be disabled.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DEVICE_CONFIGURATION_ERROR.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **CaptureFunc** function when the driver receives an IOCTL_VIDC_CAPTURE_ON or IOCTL_VIDC_CAPTURE_OFF control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_Capture**.

The driver might also call the **CaptureFunc** function itself, while transferring streams of captured data. The driver typically disables data acquisition between the time the frame buffer has been filled and the time the driver has finished copying its contents, and then re-enables it.

Support for a **CaptureFunc** function is required. The driver must place the address of its **CaptureFunc** function in the VC_CALLBACK structure supplied by *VCKernel.lib*. If the driver does not support the function, *VCKernel.lib* sets the Win32 error code value to STATUS_INVALID_DEVICE_REQUEST if the driver receives an IOCTL_VIDC_CAPTURE_ON or IOCTL_VIDC_CAPTURE_OFF control code.

If possible, the function should disable data acquisition in a manner that freezes the current overlay display.

CaptureService

ULONG
*CaptureService***(**
PDEVICE_INFO *pDevInfo***,**
PUCHAR *pBuffer***,**
PULONG *pTimeStamp***,**
ULONG *BufferLength*
);

The **CaptureService** function is a kernel-mode video capture driver's deferred procedure call (DPC) function, used for copying captured video data from the frame buffer to client-supplied buffers. The function is driver-defined, and the **CaptureService** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo

Pointer to the DEVICE_INFO structure returned by **VC_Init**.

pBuffer
Pointer to a page-locked buffer. Can be NULL (see the following **Comments** section).

pTimeStamp
Address of a ULONG location to return the time, in milliseconds, at which the frame was captured

BufferLength
Size, in bytes, of the buffer specified by *pBuffer*.

Return Value

If the driver has finished using the buffer pointed to by *pBuffer*, it should return the number of bytes written to the buffer. If the driver returns zero and *pBuffer* is not NULL, *VCKernel.lib* will return the same buffer the next time it calls **CaptureService**.

Comments

The *VCKernel.lib* library provides a generic DPC function that is called each time the driver's **InterruptAcknowledge** function returns TRUE. This generic DPC function calls the driver's **CaptureService** function, which is responsible for copying the current frame buffer contents into a client buffer. The function executes at an IRQL of DISPATCH_LEVEL.

The **CaptureService** function receives, in *pBuffer*, a pointer to the next available buffer, which has been page-locked. (This buffer usually is client-supplied, but if the system has limited memory resources, the buffer might be one that *VCKernel.lib* allocated from system space. This situation is irrelevant to the driver.)

If *pBuffer* is NULL, no buffers are available. In this case, the driver should drop the frame contents (typically by just returning zero), and *VCKernel.lib* will increment its count of dropped frames.

If *pBuffer* is not NULL, and if the driver does not completely fill the specified buffer with a single frame, it can return zero to indicate that *VCKernel.lib* should specify the same buffer the next time it calls **CaptureService**.

The driver returns a time stamp that is relative to the beginning of the stream and is reset by the driver's **StreamStartFunc** function. The time stamp should be recorded by the driver's **InterruptAcknowledge** function — not its **CaptureService** function — because there can be a time delay before the latter function is called.

If other code within the kernel-mode driver references the same objects that the **CaptureService** function references, the driver must use **VC_SynchronizeDPC** to synchronize access to the objects.

Support for a **CaptureService** function is required. The driver must place the address of its **CaptureService** function in the VC_CALLBACK structure supplied by *VCKernel.lib*.

For more information about deferred procedure calls (DPCs) see the *Kernel Mode Drivers Design Guide*.

CleanupFunc

BOOLEAN
*CleanupFunc***(**
PDEVICE_INFO *pDevInfo*
);

The **CleanupFunc** function performs operations that must be completed before a kernel-mode video capture driver is unloaded. The function is provided by the driver, and the **CleanupFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo

Pointer to the DEVICE_INFO structure returned by **VC_Init**.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. (The *VCKernel.lib* library does not currently test the return value.)

Comments

A kernel-mode driver's **CleanupFunc** function is called by the *VCKernel.lib* library's **VC_Cleanup** function. The **CleanupFunc** function might disable hardware and deallocate memory space that was allocated by the driver's **DriverEntry** function.

Support for a **CleanupFunc** function is optional. If a driver does support the function, it must place its address in the VC_CALLBACK structure supplied by *VCKernel.lib*.

See Also

DeviceCloseFunc

ConfigDisplayFunc

BOOLEAN
*ConfigDisplayFunc***(**
PDEVICE_INFO *pDevInfo***,**
PCONFIG_INFO *pConfig*
);

The **ConfigDisplayFunc** function sets characteristics of the overlay display. The function is provided by kernel-mode video capture drivers, and the **ConfigDisplayFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

pConfig
Pointer to a CONFIG_INFO structure.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DEVICE_CONFIGURATION_ERROR.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **ConfigDisplayFunc** function when the driver receives an IOCTL_VIDC_CONFIG_DISPLAY control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_ConfigDisplay**.

A kernel-mode driver typically uses its **ConfigDisplayFunc** function to set display hardware parameters, based on information that the user-mode driver has provided in the CONFIG_INFO structure.

Support for a **ConfigDisplayFunc** function is required, if the device provides overlay capabilities. The driver must place the address of its **ConfigDisplayFunc** function in the VC_CALLBACK structure supplied by *VCKernel.lib*. If a driver that does not provide a **ConfigDisplayFunc** function receives an IOCTL_VIDC_CONFIG_DISPLAY control code, *VCKernel.lib* sets the Win32 error code value to STATUS_INVALID_DEVICE_REQUEST.

ConfigFormatFunc

BOOLEAN
*ConfigFormatFunc***(**
PDEVICE_INFO *pDevInfo***,**
PCONFIG_INFO *pConfig*

);

The **ConfigFormatFunc** function sets video data format characteristics within a kernel-mode video capture driver. The function is provided by the driver, and the **ConfigFormatFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

pConfig
Pointer to a CONFIG_INFO structure.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DEVICE_CONFIGURATION_ERROR.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **ConfigFormatFunc** function when the driver receives an IOCTL_VIDC_CONFIG_FORMAT control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_ConfigFormat**.

A kernel-mode driver typically uses its **ConfigFormatFunc** function to set format-based information, such as color translation table contents, scaling parameters, and image size, based on information that the user-mode driver has provided in the CONFIG_INFO structure. The function should also call **VC_SetImageSize** to notify *VCKkernel.lib* of the maximum image size.

Support for a **ConfigFormatFunc** function is required. The driver must place the address of its **ConfigFormatFunc** function in the VC_CALLBACK structure supplied by *VCKernel.lib*. If a driver that does not provide a **ConfigFormatFunc** function receives an IOCTL_VIDC_CONFIG_FORMAT control code, *VCKernel.lib* sets the Win32 error code value to STATUS_INVALID_DEVICE_REQUEST.

ConfigSourceFunc

BOOLEAN
*ConfigSourceFunc***(**
PDEVICE_INFO *pDevInfo***,**
PCONFIG_INFO *pConfig*
);

The **ConfigSourceFunc** function sets characteristics of the video source. The function is provided by kernel-mode video capture drivers, and the **ConfigSourceFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

pConfig
Pointer to a CONFIG_INFO structure.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DEVICE_CONFIGURATION_ERROR.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **ConfigSourceFunc** function when the driver receives an IOCTL_VIDC_CONFIG_SOURCE control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_ConfigSource**.

A kernel-mode driver typically uses its **ConfigSourceFunc** function to set video source hardware

parameters, based on information that the user-mode driver has provided in the CONFIG_INFO structure.

Support for a **ConfigSourceFunc** function is required. The driver must place the address of its **ConfigSourceFunc** function in the VC_CALLBACK structure supplied by *VCKernel.lib*. If a driver that does not provide a **ConfigSourceFunc** function receives an IOCTL_VIDC_CONFIG_SOURCE control code, *VCKernel.lib* sets the Win32 error code value to STATUS_INVALID_DEVICE_REQUEST.

DeviceCloseFunc

BOOLEAN
*DeviceCloseFunc***(**
PDEVICE_INFO *pDevInfo*
);

The **DeviceCloseFunc** function performs operations that must be completed when a video capture device is closed. The function is provided by the kernel-mode driver, and the **DeviceCloseFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DRIVER_INTERNAL_ERROR.

Coments

The *VCKernel.lib* library calls the kernel-mode driver's **DeviceCloseFunc** function when it receives an IRP_MJ_CLOSE function code, which indicates the client is closing the device. The **DeviceCloseFunc** function might disable hardware and deallocate memory space that the driver allocated when the device was opened.

Support for a **DeviceCloseFunc** function is optional. If a driver does support the function, it must place its address in the VC_CALLBACK structure supplied by *VCKernel.lib*.

See Also

CleanupFunc

DeviceOpenFunc

BOOLEAN
*DeviceOpenFunc***(**
PDEVICE_INFO *pDevInfo*
);

The **DeviceOpenFunc** function performs operations that must be completed when a video capture device is opened. The function is provided by the kernel-mode driver, and the **DeviceOpenFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DRIVER_INTERNAL_ERROR.

Coments

The *VCKernel.lib* library calls the kernel-mode driver's **DeviceOpenFunc** function when it receives an IRP_MJ_CREATE function code, which indicates a client is opening the device. User-mode drivers using *VCUser.lib* send this function code by calling **VC_OpenDevice**. The **DeviceOpenFunc** function might enable hardware or allocate memory space. The sample driver, *bravado.sys*, uses its **DeviceOpenFunc** function to obtain characteristics of the user's display device, which are placed in the registry by the user-mode driver (see Opening and Closing a Device, Using *VCUser.lib*).

Support for a **DeviceOpenFunc** function is optional. If a driver does support the function, it must place its address in the VC_CALLBACK structure supplied by *VCKernel.lib*.

See Also

DeviceCloseFunc

DrawFrameFunc

```
BOOLEAN
  DrawFrameFunc(
    PDEVICE_INFO pDevInfo,
    PDRAWBUFFER pDraw
    );
```

The **DrawFrameFunc** function copies bitmap data into the frame buffer. The function is provided by the kernel-mode driver, and the **DrawFrameFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

pDraw
Pointer to a DRAWBUFFER structure.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DRIVER_INTERNAL_ERROR.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **DrawFrameFunc** function when the driver receives an IOCTL_VIDC_DRAW_FRAME control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_DrawFrame**.

The *pDraw* parameter points to a DRAWBUFFER structure describing the bitmap data to be copied into the frame buffer. The driver should read the structure's **Format** member to determine if the specified format is one that either the device accepts for playback, or that the driver can convert into one the device accepts. (You should use the driver's **GetOverlayModeFunc** function to indicate the supported formats.)

To protect the driver from access violations, you should place code that references the supplied bitmap in a routine that can be called by using **VC_AccessData**. To obtain the frame buffer's address, the driver should call **VC_GetFrameBuffer**.

The driver should read data from the bitmap, convert it if necessary, and place it in the frame buffer. Current settings for key color and overlay rectangle description should not be changed.

Support for a **DrawFrameFunc** function is required, if the device supports playback. If a driver does support the function, it must place its address in the VC_CALLBACK structure supplied by *VCKernel.lib*. If a driver that does not provide a **DrawFrameFunc** function receives an IOCTL_VIDC_DRAW_FRAME control code, *VCKernel.lib* sets the Win32 error code value to STATUS_INVALID_DEVICE_REQUEST.

The sample kernel-mode driver, *bravado.sys*, supports video playback, but its companion

user-mode driver, *bravado.dll*, does not. Instead, the *msyuv.dll* codec calls *bravado.sys* to play back YUV-formatted compressed data.

GetKeyColourFunc

ULONG
GetKeyColourFunc(
PDEVICE_INFO *pDevInfo*
);

The **GetKeyColourFunc** function returns the current key color. The function is provided by the kernel-mode driver, and the **GetKeyColourFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

Return Value

Returns the current key color. See the following **Comments** section.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **GetKeyColourFunc** function when the driver receives an IOCTL_VIDC_GET_KEY_COLOUR control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_GetKeyColour**.

The driver returns the key color as either an RGBQUAD type or as a palette index number. The return type must be consistent with the value of the VCO_KEYCOLOUR_RGB flag returned by the driver's **GetOverlayModeFunc** function. That is, if the driver sets the flag, it must return an RGBQUAD-typed key color.

Support for a **GetKeyColourFunc** function is required, if the device supports a key color. If a driver does support the function, it must place its address in the VC_CALLBACK structure supplied by *VCKernel.lib*. If a driver that does not provide a **GetKeyColourFunc** function receives an IOCTL_VIDC_GET_KEY_COLOUR control code, *VCKernel.lib* sets the Win32 error code value to STATUS_INVALID_DEVICE_REQUEST.

GetOverlayModeFunc

DWORD
GetOverlayModeFunc(
PDEVICE_INFO *pDevInfo*
);

The **GetOverlayModeFunc** function returns the device's overlay capabilities. The function is provided by the kernel-mode driver, and the **GetOverlayModeFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

Return Value

Returns a DWORD value containing flags. The following flags are defined.

Flag	Definition
VCO_KEYCOLOUR	Indicates the device supports a key color.
VCO_KEYCOLOUR_FIXED	Indicates the key color cannot be modified.
VCO_KEYCOLOUR_RGB	If set, indicates the key color must be specified as an RGB color. If clear, indicates the key color must be

	specified as a palette index number.
VCO_SIMPLE_RECT	Indicates the device supports a single rectangular overlay region.
VCO_COMPLEX_REGION	Indicates the device supports a complex (multi-rectangle) overlay region.
VCO_CAN_DRAW_Y411	Indicates the device can display bitmaps that contain YUV 4:1:1-formatted data.
VCO_CAN_DRAW_S422	Indicates the device can display bitmaps that contain YUV 4:2:2-formatted data.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **GetOverlayModeFunc** function when the driver receives an IOCTL_VIDC_OVERLAY_MODE control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_GetOverlayMode**.

The driver sets the appropriate flags in the return value, based on the hardware's capabilities.

Support for a **GetOverlayModeFunc** function is required. The driver must place the function's address in the VC_CALLBACK structure supplied by *VCKernel.lib*.

InterruptAcknowledge

BOOLEAN
*InterruptAcknowledge***(**
PDEVICE_INFO *pDevInfo*
);

The **InterruptAcknowledge** function is a kernel-mode video capture driver's interrupt service routine (ISR), used to acknowledge a device interrupt. The function is driver-defined, and the **InterruptAcknowledge** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

Return Value

Returns TRUE if a frame is available and it is time to capture another frame. Otherwise returns FALSE. (See the following **Comments** section.)

Comments

The *VCKernel.lib* library provides a generic ISR that is called each time a device interrupt occurs on the interrupt number that the driver passed to **VC_ConnectInterrupt**. This generic ISR calls the driver's **InterruptAcknowledge** function.

Typically the **InterruptAcknowledge** function re-enables the device interrupt, if a capture stream operation is in progress.

The function is responsible for determining if it is time to capture a frame. It should compare the time since the last frame was captured to the client-specified time between frames. (The driver receives the client-specified time between frames as input to its **StreamInitFunc** function.)

If it is now time to capture a frame, and if a full frame is available, the function must return TRUE. Returning TRUE causes the generic ISR to call **IoRequestDPC**, which schedules the driver's **CaptureService** (DPC) function, which in turn is responsible for capturing the frame.

Like all device ISRs under Windows NT, the driver's **InterruptAcknowledge** function executes at the device's IRQL and should be written to execute as quickly as possible. You should place data transfer operations in the driver's **CaptureService** function. The **InterruptAcknowledge** function should record the frame's time stamp.

If other code within the kernel-mode driver references the same objects that the

InterruptAcknowledge function references, the driver must use **VC_SynchronizeExecution** to synchronize access to the objects.

Support for an **InterruptAcknowledge** function is required. The driver must place the address of its **InterruptAcknowledge** function in the VC_CALLBACK structure supplied by *VCKernel.lib*.

For more information about interrupt service routines (ISRs) and deferred procedure calls (DPC's) see the *Kernel Mode Drivers Design Guide*.

OverlayFunc

BOOLEAN
*OverlayFunc***(**
PDEVICE_INFO *pDevInfo***,**
BOOL *bOverlay*
);

The **OverlayFunc** function is provided by kernel-mode video capture drivers to enable and disable the overlay display. The **OverlayFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

bOverlay
Set to TRUE if data capture is to be enabled, and FALSE if it is to be disabled.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DEVICE_CONFIGURATION_ERROR.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **OverlayFunc** function when the driver receives an IOCTL_VIDC_OVERLAY_ON or IOCTL_VIDC_OVERLAY_OFF control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_Overlay**.

Support for an **OverlayFunc** function is required, if the device provides overlay capabilities. The driver must place the address of its **OverlayFunc** function in the VC_CALLBACK structure supplied by *VCKernel.lib*. If a driver that does not support the **OverlayFunc** function receives an IOCTL_VIDC_OVERLAY_ON or IOCTL_VIDC_OVERLAY_OFF control code, *VCKernel.lib* sets the Win32 error code value to STATUS_INVALID_DEVICE_REQUEST.

SetKeyPalIdxFunc

BOOLEAN
*SetKeyPalIdxFunc***(**
PDEVICE_INFO *pDevInfo***,**
ULONG *palidx*
);

The **SetKeyPalIdxFunc** function sets the overlay destination image's key color to the specified palette index number. The function is provided by the kernel-mode driver, and the **SetKeyPalIdxFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

palidx
Palette index number.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DEVICE_CONFIGURATION_ERROR.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **SetKeyPalIdxFunc** function when the driver receives an IOCTL_VIDC_SET_KEY_PALIDX control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_SetKeyColourPalIdx**.

You can assume that the driver's **SetKeyPalIdxFunc** function will be called only if a previous call to its **GetOverlayModeFunc** function has returned with the VCO_KEYCOLOUR flag set and the VCO_KEYCOLOUR_RGB flag cleared.

Support for a **SetKeyPalIdxFunc** function is required, if the device supports a modifiable key color. If a driver does support the function, it must place its address in the VC_CALLBACK structure supplied by *VCKernel.lib*. If a driver that does not provide a **SetKeyPalIdxFunc** function receives an IOCTL_VIDC_SET_KEY_PALIDX control code, *VCKernel.lib* sets the Win32 error code value to STATUS_INVALID_DEVICE_REQUEST.

For more information about palette index numbers, see **VC_SetKeyColourPalIdx**.

See Also

GetKeyColourFunc, **SetKeyRGBFunc**

SetKeyRGBFunc

BOOLEAN
*SetKeyRGBFunc***(**
PDEVICE_INFO *pDevInfo***,**
PRGBQUAD *pRGB*
);

The **SetKeyRGBFunc** function sets the overlay destination image's key color to the specified RGB color. The function is provided by the kernel-mode driver, and the **SetKeyRGBFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

pRGB
Pointer to an RGBQUAD structure containing an RGB color specification. (The RGBQUAD structure is described in the Win32 SDK.)

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DEVICE_CONFIGURATION_ERROR.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **SetKeyRGBFunc** function when the driver receives an IOCTL_VIDC_SET_KEY_PALIDX control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_SetKeyColourRGB**.

You can assume that the drivers **SetKeyRGBFunc** function will be called only if a previous call to its **GetOverlayModeFunc** function has returned with both the VCO_KEYCOLOUR and VCO_KEYCOLOUR_RGB flags set.

Support for a **SetKeyRGBFunc** function is required, if the device supports a modifiable key color. If a driver does support the function, it must place its address in the VC_CALLBACK structure supplied by *VCKernel.lib*. If a driver that does not provide a **SetKeyRGBFunc** function receives an IOCTL_VIDC_SET_KEY_RGB control code, *VCKernel.lib* sets the Win32 error code value to

STATUS_INVALID_DEVICE_REQUEST.

See Also

GetKeyColourFunc, **SetKeyPalIdxFunc**

SetOverlayOffsetFunc

BOOLEAN
*SetOverlayOffsetFunc***(**
PDEVICE_INFO *pDevInfo***,**
PRECT *prc*
);

The **SetOverlayOffsetFunc** function sets the overlay offset rectangle. This rectangle defines which portion of the frame buffer appears in the overlay window. The function is provided by the kernel-mode driver, and the **SetOverlayOffsetFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

prc
Pointer to a RECT structure containing screen coordinates that describe the offset rectangle. (The RECT structure is described in the Win32 SDK.)

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DEVICE_CONFIGURATION_ERROR.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **SetOverlayOffsetFunc** function when the driver receives an IOCTL_VIDC_OVERLAY_OFFSET control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_SetOverlayOffset**.

For more information about implementing the offset rectangle, see **VC_SetOverlayOffset**.

Support for a **SetOverlayOffsetFunc** function is required, if the device supports overlay and panning of the source image. If a driver does support the function, it must place its address in the VC_CALLBACK structure supplied by *VCKernel.lib*. If a driver that does not support a **SetOverlayOffsetFunc** function receives an IOCTL_VIDC_OVERLAY_OFFSET control code, *VCKernel.lib* sets the Win32 error code value to STATUS_INVALID_DEVICE_REQUEST.

See Also

SetOverlayRectsFunc

SetOverlayRectsFunc

BOOLEAN
*SetOverlayRectsFunc***(**
PDEVICE_INFO *pDevInfo***,**
POVERLAY_RECTS *pOR*
);

The **SetOverlayRectsFunc** function sets the overlay region. The function is provided by the kernel-mode driver, and the **SetOverlayRectsFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

pOR
Pointer to an OVERLAY_RECTS structure describing one or more overlay rectangles, using screen coordinates.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DEVICE_CONFIGURATION_ERROR.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **SetOverlayRectsFunc** function when the driver receives an IOCTL_VIDC_OVERLAY_RECTS control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_SetOverlayRect**.

After the user-mode driver enables overlay by calling **VC_Overlay**, the kernel-mode driver should display captured video source images inside the overlay area of the output display. (If the device supports color keying, then the user-mode driver typically repaints the overlay area with the key color when it receives a DVM_UPDATE message.)

Drivers for devices that support scaling generally do not scale the video source image to the overlay area. Instead, scaling of the captured image is typically based on the capture format and controlled by the **ConfigFormatFunc** function. Thus, the image might not fit into the overlay rectangle. If the driver's **SetOverlayOffsetFunc** function has not been called to set an offset rectangle, the driver should align the top left corner of the video source with the top left corner of the overlay region. If the source rectangle is larger than the overlay rectangle, the driver should crop the source as necessary at the bottom and right sides.

If a format has not been specified, the sample driver, *bravado.sys*, does scale the captured image to the overlay rectangle, in order to support applications that only overlay the source image without capturing frames.

Support for a **SetOverlayRectsFunc** function is required, if the device supports overlay. If a driver does support the function, it must place its address in the VC_CALLBACK structure supplied by *VCKernel.lib*. If a driver that does not support a **SetOverlayRectsFunc** function receives an IOCTL_VIDC_OVERLAY_RECTS control code, *VCKernel.lib* sets the Win32 error code value to STATUS_INVALID_DEVICE_REQUEST.

See Also

SetOverlayOffsetFunc

StreamFiniFunc

BOOLEAN
*StreamFiniFunc***(**
PDEVICE_INFO *pDevInfo*
);

The **StreamFiniFunc** function ends a streaming operation that was initiated by the **StreamInitFunc** function. The function is provided by the kernel-mode driver, and the **StreamFiniFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DRIVER_INTERNAL_ERROR.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **StreamFiniFunc** function when the driver

receives an IOCTL_VIDC_STREAM_FINI control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_StreamFini**.

You can assume that your driver's **StreamStopFunc** is called, and all pending buffers are returned to the client, before the driver's **StreamFiniFunc** function is called. The **StreamFiniFunc** function should be used to deallocate resources that were allocated by the **StreamInitFunc** function. If the **StreamFiniFunc** function does not need to perform any operations, it can just return TRUE.

Support for a **StreamFiniFunc** function is required, if the device and driver support capture streams. If a driver does support the function, it must place its address in the VC_CALLBACK structure supplied by *VCKernel.lib*. If a driver that does not support a **StreamFiniFunc** function receives an IOCTL_VIDC_STREAM_FINI control code, *VCKernel.lib* sets the Win32 error code value to STATUS_INVALID_DEVICE_REQUEST.

StreamGetPositionFunc

ULONG
*StreamGetPositionFunc***(**
PDEVICE_INFO *pDevInfo*
);

The **StreamGetPositionFunc** function returns the current position within the capture stream. The function is provided by the kernel-mode driver, and the **StreamGetPositionFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

Return Value

Returns the current stream position, in milliseconds. This is the time that has passed since **StreamStartFunc** was called.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **StreamGetPositionFunc** function when the driver receives an IOCTL_VIDC_GET_POSITION control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_GetStreamPos**.

Support for a **StreamGetPositionFunc** function is required, if the device and driver support capture streams. If a driver does support the function, it must place its address in the VC_CALLBACK structure supplied by *VCKernel.lib*. If a driver that does not support a **StreamGetPositionFunc** function receives an IOCTL_VIDC_GET_POSITION control code, *VCKernel.lib* sets the Win32 error code value to STATUS_INVALID_DEVICE_REQUEST.

StreamInitFunc

BOOLEAN
*StreamInitFunc***(**
PDEVICE_INFO *pDevInfo***,**
ULONG *FrameRate*
);

The **StreamInitFunc** function initializes a video capture stream. The function is provided by the kernel-mode driver, and the **StreamInitFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

FrameRate
Capture rate (microseconds per frame).

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DRIVER_INTERNAL_ERROR.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **StreamInitFunc** function when the driver receives an IOCTL_VIDC_STREAM_INIT control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_StreamInit**. You can assume *VCKernel.lib* will not call the **StreamInitFunc** function if a stream has already been started.

The driver should store the frame rate so that it is accessible to the **InterruptAcknowledge** function. It might also verify that it has received a format selection. In deciding which operations to place in a driver's **StreamInitFunc** function and which to place in its **StreamStartFunc** function, remember that **StreamInitFunc** is called once for each stream, which **StreamStartFunc** can be called multiple times.

Support for a **StreamInitFunc** function is required, if the device and driver support capture streams. If a driver does support the function, it must place its address in the VC_CALLBACK structure supplied by *VCKernel.lib*. If a driver that does not support a **StreamInitFunc** function receives an IOCTL_VIDC_STREAM_INIT control code, *VCKernel.lib* sets the Win32 error code value to STATUS_INVALID_DEVICE_REQUEST.

StreamStartFunc

BOOLEAN
*StreamStartFunc***(**
PDEVICE_INFO *pDevInfo*
);

The **StreamStartFunc** function starts capturing frames for a capture stream. The function is provided by the kernel-mode driver, and the **StreamStartFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DRIVER_INTERNAL_ERROR.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **StreamStartFunc** function when the driver receives an IOCTL_VIDC_STREAM_START control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_StreamStart**. You can assume *VCKernel.lib* will not call the **StreamStartFunc** function unless the driver's **StreamInitFunc** function has been previously called.

The driver's **StreamStartFunc** function must reset the stream position to zero milliseconds. The function must also enable interrupts. Once interrupts are enabled, the following events occur:

- When the device interrupts, *VCKernel.lib*'s generic ISR is called, which in turn calls the driver's **InterruptAcknowledge** function. If, based on the client-specified frame rate, it is time to capture a frame, then *VCKernel.lib*'s generic DPC function is queued.
- The I/O manager calls the generic DPC function, which dequeues one of the IRPs that was queued when the user-mode driver called **VC_StreamAddBuffer**. Then the generic DPC function calls the driver's **CaptureService** function, passing the buffer address.

- After the **CaptureService** function returns, *VCKernel.lib*'s generic DPC function calls **IoCompleteRequest** to return the buffer's IRP to the user-mode driver.

Support for a **StreamStartFunc** function is required, if the device and driver support capture streams. If a driver does support the function, it must place its address in the VC_CALLBACK structure supplied by *VCKernel.lib*. If a driver that does not provide a **StreamStartFunc** function receives an IOCTL_VIDC_STREAM_START control code, *VCKernel.lib* sets the Win32 error code value to STATUS_INVALID_DEVICE_REQUEST.

StreamStopFunc

BOOLEAN
*StreamStopFunc***(**
PDEVICE_INFO *pDevInfo*
);

The **StreamStopFunc** function stops capturing frames for a capture stream. The function is provided by the kernel-mode driver, and the **StreamStopFunc** name is a placeholder for a driver-specified function name.

Parameters

pDevInfo
Pointer to the DEVICE_INFO structure returned by **VC_Init**.

Return Value

Returns TRUE if the operation succeeds. Otherwise returns FALSE. If FALSE, *VCKernel.lib* sets the Win32 error code value to STATUS_DRIVER_INTERNAL_ERROR.

Comments

The *VCKernel.lib* library calls a kernel-mode driver's **StreamStopFunc** function when the driver receives an IOCTL_VIDC_STREAM_STOP control code. User-mode drivers using *VCUser.lib* send this control code by calling **VC_StreamStop**. You can assume *VCKernel.lib* will not call the **StreamStopFunc** function unless the driver's **StreamInitFunc** function has been previously called.

The driver's **StreamStopFunc** function must stop the stream, typically by disabling device interrupts.

Support for a **StreamStopFunc** function is required, if the device and driver support capture streams. If a driver does support the function, it must place its address in the VC_CALLBACK structure supplied by *VCKernel.lib*. If a driver that does not provide a **StreamStopFunc** function receives an IOCTL_VIDC_STREAM_STOP control code, *VCKernel.lib* sets the Win32 error code value to STATUS_INVALID_DEVICE_REQUEST.

Macros, Kernel-Mode Video Capture Drivers

This section describes the macros that are available to kernel-mode video capture drivers. They are listed in alphabetic order and are defined in *vckernel.h*.

VC_ReadIOMemoryBlock

VC_ReadIOMemoryBlock(*dst***,** *src***,** *cnt***)**

The **VC_ReadIOMemoryBlock** macro reads a specified number of bytes from a device's mapped I/O memory space into system memory space.

Parameters

dst
Pointer to a buffer to receive data copied from mapped I/O space.

src

Address of the first mapped I/O memory location from which data is to be copied.

cnt

Number of bytes to be copied from I/O space.

Return Value

None.

Comments

Kernel-mode video capture drivers typically use the **VC_ReadIOMemoryBlock** macro to read a block of memory within the frame buffer. Drivers can obtain the frame buffer's base address by calling **VC_GetFrameBuffer**.

Using the **VC_ReadIOMemoryBlock** macro to read I/O space helps ensure driver portability across system platforms.

VC_ReadIOMemoryBYTE

VC_ReadIOMemoryBYTE(*p*)

The **VC_ReadIOMemoryBYTE** macro reads a single byte from a device's mapped I/O memory space.

Parameters

p

Address of the mapped I/O memory location from which data is to be copied.

Return Value

Returns the byte value read from the specified memory address.

Comments

Kernel-mode video capture drivers typically use the **VC_ReadIOMemoryBYTE** macro to read a location within the frame buffer. Drivers can obtain the frame buffer's base address by calling **VC_GetFrameBuffer**.

Using the **VC_ReadIOMemoryBYTE** macro to read I/O space helps ensure driver portability across system platforms.

VC_ReadIOMemoryULONG

VC_ReadIOMemoryULONG(*p*)

The **VC_ReadIOMemoryULONG** macro reads a single unsigned longword value from a device's mapped I/O memory space.

Parameters

p

Address of the mapped I/O memory location from which data is to be copied.

Return Value

Returns the unsigned longword value read from the specified memory address.

Comments

Kernel-mode video capture drivers typically use the **VC_ReadIOMemoryULONG** macro to read a location within the frame buffer. Drivers can obtain the frame buffer's base address by calling **VC_GetFrameBuffer**.

Using the **VC_ReadIOMemoryULONG** macro to read I/O space helps ensure driver portability across system platforms.

VC_ReadIOMemoryUSHORT

VC_ReadIOMemoryUSHORT(*p*)

The **VC_ReadIOMemoryUSHORT** macro reads a single unsigned word value from a device's mapped I/O memory space.

Parameters

p
Address of the mapped I/O memory location from which data is to be copied.

Return Value

Returns the unsigned word value read from the specified memory address.

Comments

Kernel-mode video capture drivers typically use the **VC_ReadIOMemoryUSHORT** macro to read a location within the frame buffer. Drivers can obtain the frame buffer's base address by calling **VC_GetFrameBuffer**.

Using the **VC_ReadIOMemoryUSHORT** macro to read I/O space helps ensure driver portability across system platforms.

VC_WriteIOMemoryBlock

VC_WriteIOMemoryBlock(*dst*, *src*, *cnt*)

The **VC_WriteIOMemoryBlock** macro writes a specified number of bytes from system memory space into a device's mapped I/O memory space.

Parameters

dst
Address of the first mapped I/O memory location into which data is to be copied.

src
Pointer to a buffer from which data is to be copied.

cnt
Number of bytes to be copied into I/O space.

Return Value

None.

Comments

Kernel-mode video capture drivers typically use the **VC_WriteIOMemoryBlock** macro to write a block of memory within the frame buffer. Drivers can obtain the frame buffer's base address by calling **VC_GetFrameBuffer**.

Using the **VC_WriteIOMemoryBlock** macro to write into I/O space helps ensure driver portability across system platforms.

VC_WriteIOMemoryBYTE

VC_WriteIOMemoryBYTE(*p, b*)

The **VC_WriteIOMemoryBYTE** macro writes a single byte into a device's mapped I/O memory space.

Parameters

p
Address of the mapped I/O memory location into which data is to be copied.

b
Byte value to be written.

Return Value

None.

Comments

Kernel-mode video capture drivers typically use the **VC_WriteIOMemoryBYTE** macro to write into a location within the frame buffer. Drivers can obtain the frame buffer's base address by calling **VC_GetFrameBuffer**.

Using the **VC_WriteIOMemoryBYTE** macro to write into I/O space helps ensure driver portability across system platforms.

VC_WriteIOMemoryULONG

VC_WriteIOMemoryULONG(*p, l***)**

The **VC_WriteIOMemoryULONG** macro writes an unsigned longword into a device's mapped I/O memory space.

Parameters

p
Address of the mapped I/O memory location into which data is to be copied.

l
Unsigned longword value to be written.

Return Value

None.

Comments

Kernel-mode video capture drivers typically use the **VC_WriteIOMemoryULONG** macro to write into a location within the frame buffer. Drivers can obtain the frame buffer's base address by calling **VC_GetFrameBuffer**.

Using the **VC_WriteIOMemoryULONG** macro to write into I/O space helps ensure driver portability across system platforms.

VC_WriteIOMemoryUSHORT

VC_WriteIOMemoryUSHORT(*p, w***)**

The **VC_WriteIOMemoryUSHORT** macro writes an unsigned word into a device's mapped I/O memory space.

Parameters

p
Address of the mapped I/O memory location into which data is to be copied.

w
Unsigned word value to be written.

Return Value

None.

Comments

Kernel-mode video capture drivers typically use the **VC_WriteIOMemoryUSHORT** macro to write into a location within the frame buffer. Drivers can obtain the frame buffer's base address by calling **VC_GetFrameBuffer**.

Using the **VC_WriteIOMemoryUSHORT** macro to write into I/O space helps ensure driver portability across system platforms.

Video Compression Manager Drivers

Video compression drivers provide low-level compression, decompression, and rendering services. The algorithms used by these drivers can be hardware- or software-based. These drivers are used for the following services:

- Compressing and decompressing data
- Rendering compressed data to the display
- Handling text and custom data.

The following topics describe the interface for these drivers:

- Writing video compression, decompression, and rendering drivers
- How a driver handles the system messages for the installable-driver interface
- How a driver handles messages specific to compressing, decompressing, and rendering data
- Alphabetic reference to the messages and data structures used to write compression, decompression, and rendering drivers.

Compression Driver Architecture

Compression drivers are DLLs that compress or decompress video or other types of data in response to requests from the system or applications. Applications never access the compression driver directly. Instead, an application calls a corresponding Win32 compression function that directs the system to send the request to the driver in the form of a message.

The typical compression driver combines both compression and decompression functions. When a compression driver receives a request, it usually receives video or other data that needs to be compressed or decompressed. The video data can be either still bitmaps or motion-video frames. The driver typically receives compressed data from an application that has opened an AVI file; the driver is expected to return the uncompressed data to the application. The driver typically receives uncompressed data from a video source, such as a disk file or a video device.

Some compression drivers also write directly to the display or display driver. Such drivers, called rendering drivers, can replace the display driver or take over some of the responsibilities of the driver. These drivers handle a set of messages, the ICM_DRAW messages, in addition to the decompression messages defined for the services that return the decompressed video to the client application. Rendering drivers can reside in the same DLLs as compressors and decompressors, or they can reside in a separate DLL.

The DriverProc Function, Compression Drivers

Compression drivers provide a **DriverProc** entry point to process messages related to requests for video compression and decompression. The **DriverProc** function processes messages sent by the system to the driver as the result of an application call to a video compression and decompression function. For example, when an application opens a video compression and decompression driver, the system sends the specified driver a DRV_OPEN message. The driver's **DriverProc** function receives and processes this message. Your **DriverProc** function should return ICERR_UNSUPPORTED for any messages that it does not handle.

Handling DRV_OPEN and DRV_CLOSE, Compression Drivers

Like other installable drivers, client applications must open a video compression and decompression driver before using it, and close it when finished so the driver will be available to other applications. When a driver receives an open request, it returns a value that the system will use for *dwDriverID* sent with subsequent messages. When your driver receives other messages, it can use this value to identify instance data needed for operation. Your drivers can use this data to

maintain information related to the client that opened the driver.

Compression and decompression drivers should support more than one client simultaneously. If you support more than one client simultaneously, though, remember to check the *dwDriverID* parameter to determine which client is being accessed.

If the driver is opened for configuration by the Drivers option of the Control Panel, *lParam2* contains zero. When opened this way, your driver should respond to the DRV_CONFIGURE and DRV_QUERYCONFIGURE messages.

If opened for compression or decompression services, *lParam2* contains a pointer to an ICOPEN data structure.

Compressor Configuration

Video compression and decompression drivers can receive a series of configuration messages. System configuration messages are typically sent by the Drivers option of the Control Panel to configure the hardware. Video compression- and decompression-specific configuration messages are typically initiated by the client application or from dialog boxes displayed by your driver. Your driver should use these messages to configure the driver.

Configuration Messages Sent by the System

Installable drivers can supply a configuration dialog box for users to access through the Drivers option in the Control Panel. If your driver supports different options, it should allow user configuration. Any hardware-related settings should be stored in the system registry in a section having the same name as the driver.

Like other installable drivers, your driver will receive DRV_CONFIGURE and DRV_QUERYCONFIGURE messages from the Drivers option of the Control Panel. If your driver controls hardware that needs to be configured, it should return a nonzero value for the DRV_QUERYCONFIGURE system message and display a hardware configuration dialog box for the DRV_CONFIGURE system message.

Messages for Configuring the Driver State

The video compression-specific and decompression-specific configuration messages are typically initiated by the client application or from dialog boxes displayed by your driver. Your driver should use these messages to configure the driver.

If your driver is configurable, it should support the ICM_CONFIGURE message for driver configuration. It should also use this message to set parameters for compression or decompression. Any options the user selects in the dialog box displayed for ICM_CONFIGURE should be saved as part of the state information referenced by the ICM_GETSTATE and ICM_SETSTATE messages.

The ICM_GETSTATE and ICM_SETSTATE messages query and set the internal state of your compression or decompression driver. State information is device dependent, and your driver must define its own data structure for it. While the client application reserves a memory block for the information, it will obtain the size needed for the memory block from your driver. If your driver receives ICM_GETSTATE with a NULL pointer for *dwParam1*, the client application is requesting that your driver return the size of its state information. Conversely, if your driver receives ICM_GETSTATE with *dwParam1* pointing to a block of memory, and *dwParam2* specifying the size of the memory block, the client application is requesting that your driver transfer the state information to the memory block.

When your driver receives ICM_SETSTATE, with *dwParam1* pointing to a block of memory and *dwParam2* specifying the size of the memory block, the client application is requesting that your driver restore its configuration from the state information contained in the memory block. Before setting the state, your driver should verify that the state information applies to your driver. One technique for verifying the data is to reserve the first DWORD in the state data structure for the four-character code used to identify your driver. If you set this DWORD for data returned for

ICM_GETSTATE, you can use it to verify the data supplied with ICM_SETSTATE. If ICM_SETSTATE has a NULL pointer for *dwParam1*, it indicates that your driver should return to its default state.

State information should not contain any data that is absolutely required for data decompression—any such data should be part of the format you return for the ICM_DECOMPRESS_GET_FORMAT message.

Messages Used to Interrogate the Driver

The client application sends the ICM_ABOUT message to display your driver's About dialog box. The client application sets *dwParam1* to –1; the application is querying whether your driver supports display of an About box. Your driver returns ICERR_OK if it does, and it returns ICERR_UNSUPPORTED if it does not. Your driver should only display an About box if the client application specifies a window handle in *dwParam1*. The window handle indicates the parent of the dialog box.

The client application uses the ICM_GETINFO message to obtain a description of your driver. Your driver should respond to this message by filling in the ICINFO structure it receives with the message. The flags your driver sets in the structure tell the client application which capabilities the driver supports. Your driver will not typically use the **szDriver[128]** member. This member is used to specify the module that contains the driver. Set the flags corresponding to the capabilities of your driver in the low-ordered word of the **dwFlags** member. You can use the high-ordered word for driver-specific flags.

Configuration Messages for Compression Quality

For the video compression and decompression interface, quality is indicated by an integer in the range of 0 to 10,000. A quality level of 7500 typically indicates an acceptable image quality. A quality level of 0 typically indicates a very low quality level (possibly even a totally black image). As the quality level moves from an acceptable level to low quality, the image might have a loss of color as the colors in the color table are merged, or as the color resolution of each pixel decreases. If your driver supports temporal compression (it needs information from the previous frame to decompress the current frame), low and high quality might imply how much this type of compression can degrade image quality. For example, your driver might limit the compression of a high-quality image to preserve sharp detail and color fidelity. Conversely, your driver might sacrifice these qualities to obtain very compressed output files.

If your driver supports quality values, it maps the values to its internal definitions used by the compression algorithms. Thus, the definition of image quality will vary from driver to driver, and, quite possibly, from compression algorithm to compression algorithm. Even though the values are not definitive, your driver should support as many individual values as possible.

The client application obtains the capabilities for compression quality with the ICM_GETDEFAULTQUALITY and ICM_GETQUALITY messages. If your driver supports quality levels, it should respond to the ICM_GETDEFAULTQUALITY message by returning a value between 0 and 10000 that corresponds to a good default quality level for your compressor. Your driver should return the current quality level for the ICM_GETQUALITY message.

The client application sends the ICM_SETQUALITY message to set the quality level of your driver. Your driver should pass the quality value directly to the compression routine.

If your driver supports quality levels, it should set the VIDCF_QUALITY flag when it responds to the ICM_GETINFO message.

Configuration Messages for Key-Frame Rate and Buffer Queue

The client application uses the ICM_GETDEFAULTKEYFRAMERATE message to obtain the driver's recommendation for the key-frame spacing for compressing data. (A key frame is a frame in a video sequence that does not require information from a previous frame for decompression.) If the client application does not specify another value, this value determines how frequently the client application sends an uncompressed image to your driver with the

ICCOMPRESS_KEYFRAME flag set. If your driver supports this option, it should specify the key-frame rate in the DWORD pointed to by *dwParam1* and return ICERR_OK. If it does not support this option, return ICERR_UNSUPPORTED.

The client application uses ICM_GETBUFFERSWANTED to determine if your driver will maintain a queue of buffers. Your driver might maintain a queue of buffers if it renders the decompressed data and is designed to keep its hardware pipelines full. If your driver supports this option, it should specify the number of buffers in the DWORD pointed to by *dwParam1* and return ICERR_OK. If it does not support this option, return ICERR_UNSUPPORTED.

Decompressing Video Data

The client application sends a series of messages to your driver to coordinate decompressing video data. The coordination involves the following activities:

- Setting the driver state
- Specifying the input format and determining the decompression format
- Preparing to decompress video
- Decompressing the video
- Ending decompression

The following messages are used by video compression and decompression drivers for these decompression activities.

Message	Description
ICM_DECOMPRESSEX	Decompresses a frame of data into a buffer provided by the client application.
ICM_DECOMPRESSEX_BEGIN	Prepares a driver for decompressing data.
ICM_DECOMPRESSEX_END	Cleans up after decompressing.
ICM_DECOMPRESS_GET_FORMAT	Obtains a suggestion for a good format for the decompressed data.
ICM_DECOMPRESSEX_QUERY	Determines if a driver can decompress a specific input format.
ICM_DECOMPRESS_GET_PALETTE	Returns the color table of the output data structure.

The video decompressed with these messages is returned to the client application, which handles the display of data. If you want your driver to control the video timing or directly update the display, use the ICM_DRAW messages. If you return the decompressed video to the client application, your driver can decompress data using either software or hardware with the ICM_DECOMPRESS messages.

Restoring the Driver State

The client application restores the driver state by sending ICM_SETSTATE. The client application recalls the state information from the 'strd' data chunk of the AVI file. (The information was originally obtained with the ICM_GETSTATE message.) The client application does not validate any data in the state information. It simply transfers the state information to your driver.

The client application sends the information to your driver in a buffer pointed to by *dwParam1*. The size of the buffer is specified in *dwParam2*. The organization of the data in the buffer is driver dependent. If *dwParam1* is NULL, your driver should return to its default state.

Note All information required for decompressing the image data should be part of the format data. Only optional compression parameters can be included with the state information.

Specifying the Input Format and Determining the Decompression Format

Depending on how the client application will use the decompressed data, it will send either ICM_DECOMPRESS_GET_FORMAT or ICM_DECOMPRESSEX_QUERY to specify the input format and determine the decompression format. The client application sends ICM_DECOMPRESS_GET_FORMAT to have your driver suggest the decompressed format. The client application sends ICM_DECOMPRESSEX_QUERY to determine if your driver supports a format it is suggesting.

The ICM_DECOMPRESS_GET_FORMAT message sends a pointer to a BITMAPINFO data structure in *dwParam1*. This structure specifies the format of the incoming compressed data. The input format was obtained by the client application from the 'strf' chunk in the AVI file. While the output format is specified by *dwParam2*, your driver must use the message to determine how the parameter is defined.

If your driver receives ICM_DECOMPRESS_GET_FORMAT, both *dwParam1* and *dwParam2* point to BITMAPINFO data structures. The input format data is contained in the *dwParam1* structure. Your driver should fill in the *dwParam2* BITMAPINFO data structure with information about the format it will use to decompress the data. If your driver can handle the format, return the number of bytes used for the *dwParam2* structure as the return value. If your driver cannot handle the input format, or the input format from the 'strf' chunk is incorrect, your driver should return ICERR_BADFORMAT to fail the message.

If you have format information in addition to that specified in the BITMAPINFOHEADER structure, you can add it immediately after this structure. If you do this, update the **biSize** member to specify the number of bytes used by the structure and your additional information. If a color table is part of the BITMAPINFO information, it follows immediately after your additional information. Return ICERR_OK when your driver has finished updating the data format.

If your driver receives ICM_DECOMPRESSEX_QUERY, *dwParam1* points to an ICDECOMPRESSEX data structure containing the input format data. If **lpbiDst** is NULL, your decompression driver can use any output format. In this case, the client application is querying whether your driver can decompress the input format, and the output format doesn't matter. If **lpbiDst** points to a BITMAPINFO structure, the suggested format will be the native or best format for the decompressed data. For example, if playback is on an 8-bit device, the client application will suggest an 8-bit DIB.

If your driver supports the specified input and output format (which might also include stretching the image according to the values specified for the **xDst**, **yDst**, **dxDst**, **dyDst**, **xSrc**, **ySrc**, **dxSrc**, and **dySrc** members), or if it supports the specified input with NULL specified for **lpbiDst**, return ICERR_OK to indicate the driver accepts the formats.

Your driver does not have to accept the formats suggested. If you fail the message by returning ICERR_BADFORMAT, the client application suggests alternate formats until your driver accepts one. If your driver exhausts the list of formats usually used, the client application requests a format with ICM_DECOMPRESS_GET_FORMAT.

If you are decompressing to 8-bit data, your driver also receives the ICM_DECOMPRESS_GET_PALETTE message. Your driver should add a color table to the BITMAPINFO data structure and specify the number of palette entries in the **biClrUsed** member. The space reserved for the color table is always 256 colors.

Preparing to Decompress Video

When the client application is ready, it sends the ICM_DECOMPRESSEX_BEGIN message to the driver. The client application uses *dwParam1* to point to an ICDECOMPRESSEX structure and sets *dwParam2* to its size. The **lpbiSrc** and **lpbiDest** members of the ICDECOMPRESSEX data structure describe the input and output formats. If either of the formats is incorrect, your driver should return ICERR_BADFORMAT. Your driver should create any tables and allocate any memory that it needs to decompress data efficiently. When done, return ICERR_OK.

Decompressing the Video

The client application sends ICM_DECOMPRESSEX each time it has an image to decompress.

The client application uses the flags in the file index to ensure the initial frame in a decompression sequence is a key frame.

The ICDECOMPRESSEX data structure specified in *dwParam1* contains the decompression parameters. The value specified in *dwParam2* specifies the size of the structure.

The format of the input data is specified in a BITMAPINFOHEADER structure pointed to by **lpbiInput**. The input data is in a buffer specified by **lpInput**. The **lpbiOutput** and **lpOutput** members contain pointers to the format data and buffer used for the output data.

The client application sets the ICDECOMPRESS_HURRYUP flag in the **dwFlags** member to direct your driver to decompress the data at a faster rate. The ICDECOMPRESS_PREROLL flag indicates the client application is sending frames in advance of a frame that will actually be displayed. The client application will not display any data decompressed with these flags. This might let your driver avoid decompressing a frame or data, or let it minimally decompress when it needs information from this frame to prepare for decompressing a following frame.

The ICDECOMPRESS_UPDATE flag indicates the client application has specified that the screen be updated.

The ICDECOMPRESS_NULLFRAME flag indicates the frame does not have any data.

The ICDECOMPRESS_NOTKEYFRAME flag indicates the frame is not a key frame.

Ending Decompression

Your driver receives ICM_DECOMPRESSEX_END when the client application no longer needs data decompressed. For this message, your driver should free the resources it allocated for the ICM_DECOMPRESSEX_BEGIN message.

Other Messages Received During Decompression

Decompression drivers also receive the ICM_DRAW_START and ICM_DRAW_STOP messages. These messages tell the driver when the client application starts and stops drawing the images. Most decompression drivers can ignore these messages.

Supporting the ICM_DECOMPRESS and ICM_DECOMPRESSEX Messages

The ICM_DECOMPRESSEX messages replace the ICM_DECOMPRESS messages and add extended decompression capabilities. These messages let drivers decompress images described with source and destination rectangles.

The ICM_DECOMPRESSEX messages serve the same purpose as the ICM_DECOMPRESS, ICM_DECOMPRESS_BEGIN, ICM_DECOMPRESS_END, and ICM_DECOMPRESS_QUERY messages, except that the ICM_DECOMPRESSEX messages use the ICDECOMPRESSEX structure. This structure contains the image input format and data, the output format and data, the source rectangle, and the destination rectangle. For compatibility, your driver should support the ICM_DECOMPRESS messages as well as the ICM_DECOMPRESSEX messages.

The ICM_DECOMPRESSEX messages also let applications decompress to a frame buffer under the following conditions:

- The decompressor supports a frame buffer.
- MSVideo can access the frame buffer.
- The decompressor supports decompression to the format of the frame buffer (for example, it supports the correct bit depth and it can handle upside-down formats).
- The palette must be an identity palette if the device uses 8-bit color depth.
- The decompressor must support 48-bit pointers if the linear frame buffer is simulated with a banked display card. MSVideo sets the **biCompression** member of the BITMAPINFOHEADER structure to 1632 when the decompressor needs to use 48-bit pointers to access the image.

Compressing Video Data

When used to compress video data, your driver receives a series of messages, similar to decompressing data. The client application sends messages to your driver to coordinate the following activities:

- Obtaining the driver state
- Specifying the input format and determining the compression format
- Preparing to compress video
- Compressing the video
- Ending compression

The following messages are used by video compression drivers.

Message	Description
ICM_COMPRESS	Compresses a frame of data into the buffer provided by the client application.
ICM_COMPRESS_BEGIN	Prepares for compressing data.
ICM_COMPRESS_END	Cleans up after compressing.
ICM_COMPRESS_GET_FORMAT	Obtains the driver's suggested output format of the compressed data.
ICM_COMPRESS_GET_SIZE	Obtains the maximum size of one frame of data when it is compressed in the output format.
ICM_COMPRESS_QUERY	Determines if a driver can compress a specific input format.

The video compressed with these messages is returned to the client application. When compressing data, your driver can use either software or hardware to do the compression.

Note When MSVideo recompresses a file, each frame is decompressed to a full frame before it is passed to the compressor.

Obtaining the Driver State

The client application obtains the driver state by sending ICM_GETSTATE. The client application determines the size of the buffer needed for the state information by sending this message with *dwParam1* set to NULL. Your driver should respond to the message by returning the size of the buffer it needs for state information.

After it determines the buffer size, the client application resends the message with *dwParam1* pointing to a block of memory it allocated. The *dwParam2* parameter specifies the size of the memory block. Your driver should respond by filling the memory with its state information. If your driver uses state information, include only optional decompression parameters with the state information. State information typically includes the setup specified by the user in the dialog box resulting from an ICM_CONFIGURE message. Any information required for decompressing the image data must be included with the format data. When done, your driver should return the size of the state information.

The client application does not validate any data in the state information. It simply stores the state information in the 'strd' data chunk of the AVI file.

Specifying the Input Format and Determining the Compression Format

The client application uses the ICM_COMPRESS_GET_FORMAT or ICM_COMPRESS_QUERY message to specify the input format and determine the compression (output) format. The client application sends ICM_COMPRESS_GET_FORMAT to have your driver suggest the compressed format. The client application sends ICM_COMPRESS_QUERY to determine if your driver supports a format it is suggesting.

Both messages have a pointer to a BITMAPINFO data structure in *dwParam1*. This structure specifies the format of the incoming uncompressed data. The contents of *dwParam2* depend on the message.

To have your driver suggest the format, the client application determines the size of the buffer needed for the compressed data format by sending ICM_COMPRESS_GET_FORMAT. When requesting the buffer size, the client application uses *dwParam1* to point to a BITMAPINFO structure and sets *dwParam2* to NULL. Based on the input format, your driver should return the number of bytes needed for the format buffer. Return a buffer size at least large enough to hold a BITMAPINFOHEADER data structure and a color table.

The client application gets the output format by sending ICM_COMPRESS_GET_FORMAT with valid pointers to BITMAPINFO structures in both *dwParam1* and *dwParam2*. Your driver should return the output format in the buffer pointed to by *dwParam2*. If your driver can produce multiple formats, the format selected by your driver should be the one that preserves the greatest amount of information rather than one that compresses to the most compact size. This will preserve image quality if the video data is later edited and recompressed.

The output format data becomes the 'strf' chunk in the AVI RIFF file. The data must start out like a BITMAPINFOHEADER data structure. You can include any additional information required to decompress the file after the BITMAPINFOHEADER data structure. A color table (if used) follows this information.

If you have format data following the BITMAPINFOHEADER structure, update the **biSize** member to specify the number of bytes used by the structure and your additional data. If a color table is part of the BITMAPINFO information, it follows immediately after your additional information.

If your driver cannot handle the input format, it returns ICMERR_BADFORMAT to fail the message.

If your driver receives ICM_COMPRESS_QUERY, the *dwParam1* parameter points to a BITMAPINFO data structure containing the input format data. The *dwParam2* parameter will either be NULL or contain a pointer to a BITMAPINFO structure describing the compressed format specified by the client application. If *dwParam2* is NULL, your compression driver can use any output format. (The client application is just querying whether your driver can handle the input.) If *dwParam2* points to a BITMAPINFO structure, the client application is suggesting the output format.

If your driver supports the specified input and output format, or it supports the specified input with NULL specified for *dwParam2*, return ICERR_OK to indicate the driver accepts the formats. Your driver does not have to accept the suggested format. If you fail the message by returning ICERR_BADFORMAT, the client application suggests alternate formats until your driver accepts one. If your driver exhausts the list of formats typically used, the client application requests a format with ICM_COMPRESS_GET_FORMAT.

Initialization for the Compression Sequence

When the client application is ready to start compressing data, it sends the ICM_COMPRESS_BEGIN message. The client application uses *dwParam1* to point to the format of the data being compressed, and uses *dwParam2* to point to the format for the compressed data. If your driver cannot handle the formats, or if they are incorrect, your driver should return ICERR_BADFORMAT to fail the message.

Before the client application starts compressing data, it sends ICM_COMPRESS_GET_SIZE. For this message, the client application uses *dwParam1* to point to the input format and uses *dwParam2* to point to the output format. Your driver should return the worst-case size (in bytes) that it expects a compressed frame to occupy. The client application uses this size value when it allocates buffers for the compressed video frame.

Client applications might send the ICM_COMPRESS_FRAMES_INFO message to set the compression parameters. The *dwParam1* parameter specifies a pointer to an

ICCOMPRESSFRAMES structure. For this message, the **GetData** and **PutData** members are not used. The *dwParam2* parameter specifies the size of the structure. A compressor can use this message to make decisions about the amount of space allocated for each frame while compressing.

Compressing the Video

The client application sends ICM_COMPRESS for each frame it wants compressed. It uses *dwParam1* to point to an ICCOMPRESS structure containing the parameters used for compression. Your driver uses the buffers pointed to by the members of ICCOMPRESS for returning information about the compressed data.

Your driver returns the actual size of the compressed data in the **biSizeImage** member in the BITMAPINFOHEADER data structure pointed to by the **lpbiOutput** member of ICCOMPRESS.

The format of the input data is specified in a BITMAPINFOHEADER structure pointed to by **lpbiInput**. The input data is in a buffer specified by **lpInput**. The **lpbiOutput** and **lpOutput** members contain pointers to the format data and buffer used for the output data. Your driver must indicate the size of the compressed video data in the **biSizeImage** member in the BITMAPINFO structure specified for **lpbiOutput**.

The **dwFlags** member specifies flags used for compression. The client application sets the ICCOMPRESS_KEYFRAME flag if the input data should be treated as a key frame. (A key frame is one that does not require data from a previous frame for decompression.) When this flag is set, your driver should treat the image as the initial image in a sequence.

The **lpckid** member specifies a pointer to a buffer used to return the chunk ID for data in the AVI file. Your driver should assign a two-character code for the chunk ID only if it uses a custom chunk ID.

The **lpdwFlags** member specifies a pointer to a buffer used to return flags for the AVI index. The client application will add the returned flags to the file index for this chunk. If the compressed frame is a key frame (a frame that does not require a previous frame for decompression), your driver should set the AVIIF_KEYFRAME flag in this member. Your driver can define its own flags, but they must be set in the high word only.

The **lFrameNum** member specifies the frame number of the frame to compress. If your driver is performing fast temporal compression, check this member to see if frames are being sent out of order or if the client application is having a frame recompressed.

The **dwFrameSize** member indicates the maximum size (in bytes) for the compressed frame. If it specifies zero, your driver determines the size of the compressed image. If it is nonzero, your driver should try to compress the frame to within the specified size. This might require your driver to sacrifice image quality (or make some other trade-off) to obtain the size goal. Your driver should support this if it sets the VIDCF_CRUNCH flag when it responds to the ICM_GETINFO message.

The **dwQuality** member specifies the compression quality. Your driver should support this if it sets the VIDCF_QUALITY flag when it responds to the ICM_GETINFO message.

The format of the previous data is specified in a BITMAPINFOHEADER structure pointed to by **lpbiPrev**. The input data is in a buffer specified by **lpPrev**. Your driver will use this information if it performs temporal compression (that is, it needs the previous frame to compress the current frame). If your driver supports temporal compression, it should set the VIDCF_TEMPORAL flag when it responds to the ICM_GETINFO message. If your driver supports temporal compression and does not need the information in the **lpbiPrev** and **lpPrev** members, it should set the VIDCF_FASTTEMPORALC flag when it responds to the ICM_GETINFO message. The VIDCF_FASTEMPORALC flag can decrease the processing time because your driver does not need to access data specified in **lpbiPrev** and **lpPrev**.

When your driver has finished compressing the data, it returns ICERR_OK.

Ending Compression

Your driver receives ICM_COMPRESS_END when the client application no longer needs data compressed, or when the client application is changing the format or palette. After sending ICM_COMPRESS_END, the client application must send ICM_COMPRESS_BEGIN to continue compressing data. The client application does not need to send an ICM_COMPRESS_END message for each ICM_COMPRESS_BEGIN message. It can send ICM_COMPRESS_BEGIN to restart compression without sending ICM_COMPRESS_END.

When the driver is no longer needed, the system closes it by sending DRV_CLOSE.

Rendering Directly to Video Hardware

Drivers that can render video directly to hardware should support the ICM_DRAW messages in addition to the ICM_DECOMPRESS messages. The ICM_DRAW messages render data directly to hardware rather than into a data buffer returned to the client application by the decompression driver.

Your driver receives a series of messages from the client application to coordinate the following activities to render a video sequence:

- Setting the driver state
- Specifying the input format
- Preparing to decompress video
- Decompressing the video
- Ending decompression

The following ICM_DRAW messages are used by renderers for these decompression activities.

Message	Description
ICM_DRAW	Decompresses a frame of data and draws it.
ICM_DRAW_BEGIN	Prepares to draw data.
ICM_DRAW_END	Cleans up after decompressing an image to the screen.
ICM_DRAW_REALIZE	Realizes a palette.
ICM_DRAW_QUERY	Determines if the driver can render data in a specific format.
ICM_DRAW_SUGGESTFORMAT	Has the driver suggest an output format.

The video decompressed with the ICM_DRAW messages is retained by your driver, which handles the display of data. These messages control only the decompression process. The messages used to control the drawing are described separately. Your driver will receive the ICM_DRAW messages only if it sets the VIDCF_DRAW flag when it responds to the ICM_GETINFO message.

Setting the Driver State

The client application restores the driver state by sending ICM_SETSTATE. The process for handling this message is the same as for the ICM_DECOMPRESS messages.

Specifying the Input and Output Formats

Because your driver handles the drawing of video, the client application does not need to determine the output format. The client application must only determine if your driver can handle the input format. It sends ICM_DRAW_QUERY to determine if your driver supports the input format. The input format is specified with a pointer to a BITMAPINFO data structure in *dwParam1*. The *dwParam2* parameter is not used.

If your driver supports the specified input format, return ICERR_OK to indicate the driver accepts the format. If your driver does not support the format, return ICERR_BADFORMAT.

Suggesting the Output Format

Your driver might also receive the ICM_DRAW_SUGGESTFORMAT message. Typically, this message is sent by the decompression portion of a driver to the drawing portion of a driver to obtain the best decompressed format for the data when the drawing portion can handle several formats. The *dwParam1* parameter of ICM_DRAW_SUGGESTFORMAT points to an ICDRAWSUGGEST structure and the *dwParam2* parameter specifies its size.

The **lpbiIn** member specifies a pointer to the structure containing the compressed input format. The **lpbiSuggest** member specifies a pointer to a buffer used to return the suggested output format.

The **dxSrc**, **dySrc**, **dxDst**, and **dyDst** members specify the width and height of the source and destination rectangles.

If returning format information, return ICERR_OK. If the **lpbiSuggest** member is NULL, return the amount of memory required for the suggested output format structure.

Preparing to Draw Data

When the client application is ready, it sends the ICM_DRAW_BEGIN message to the driver to prepare the driver for decompressing the stream. Your driver should create any tables and allocate any memory that it needs to decompress data efficiently.

The client application sets *dwParam1* to the ICDRAWBEGIN data structure. The size of this structure is contained in *dwParam2*. If the ICDRAW_QUERY flag is set in the **dwFlags** member, the client application is interrogating your driver to determine if can decompress the data with the parameters specified in the ICDRAWBEGIN data structure. Your driver should return ICM_ERR_OK if it can accept the parameters. It should return ICM_ERR_NOTSUPPORTED if it does not accept them.

When the ICDRAW_QUERY flag is set, ICM_DRAW_BEGIN will not be paired with ICM_DRAW_END. Your driver will receive another ICM_DRAW_BEGIN message without this flag to start the actual decompression sequence.

Your driver can ignore the palette handle specified in the **hpal** member.

The **hwnd** and **hdc** members specify the handle of the window and DC used for drawing. These members are valid only if the ICDRAW_HDC flag is set in the **dwFlags** member.

The **xDst** and **yDst** members specify the x- and y-position of the upper-right corner of the destination rectangle. (This is relative to the current window or display context.) The **dxDst** and **dyDst** members specify the width and height of the destination rectangle. These members are valid only if the ICDRAW_HDC flag is set. The ICDRAW_FULLSCREEN flag indicates the entire screen should be used for display and overrides any values specified for these members.

The **xSrc**, **ySrc**, **dxSrc**, and **dySrc** members specify a source rectangle used to clip the frames of the video sequence. The source rectangle is stretched to fill the destination rectangle. The **xSrc** and **ySrc** members specify the x- and y-position of the upper-right corner of the source rectangle. (This is relative to a full-frame image of the video.) The **dxSrc** and **dySrc** members specify the width and height of the source rectangle.

Your driver should stretch the image from the source rectangle to fit the destination rectangle. If the client application changes the size of the source and destination rectangles, it will send the ICM_DRAW_END message and specify new rectangles with a new ICM_DRAW_BEGIN message. For more information about handling the source and destination rectangles, see the **StretchDIBits** function.

The **lpbi** member specifies a pointer to a BITMAPINFOHEADER data structure containing the input format.

The **dwRate** member specifies the decompression rate in an integer format. To obtain the rate in frames-per-second, divide this value by the value in *dwScale*. Your driver uses these values when it handles the ICM_DRAW_START message.

If your driver can decompress the data with the parameters specified in the ICDRAWBEGIN data structure, your driver should return ICERR_OK and allocate any resources it needs to efficiently decompress the data. If your driver cannot decompress the data with the parameters specified, your driver should fail the message by returning ICERR_NOTSUPPORTED. When this message fails, your driver does not receive an ICM_DRAW_END message, so it should not prepare its resources for other ICM_DRAW messages.

Drawing the Data

The client application sends ICM_DRAW each time it has data to decompress. (Your driver should use this message to decompress data. It should wait for the ICM_DRAW_START message before it begins to render the data.) The client application uses the flags in the file index to ensure the first frame in a series of decompressed frames starts with a key-frame boundary. Your driver must allocate the memory it needs for the decompressed data.

The ICDRAW data structure specified in *dwParam1* contains the decompression parameters. The value specified in *dwParam2* specifies the size of the structure. The format of the input data is specified in a BITMAPINFOHEADER structure pointed to by **lpFormat**. The input data is in a buffer specified by **lpData**. The number of bytes in the input buffer is specified by **cbData**.

The client application sets the ICDRAW_HURRYUP flag in the **dwFlags** member to direct your driver to decompress data at a faster rate. For example, the client application might use this flag when the video is starting to lag behind the audio. If your driver cannot speed up its decompression and rendering performance, it might be necessary to avoid rendering a frame of data. The client application sets the ICDRAW_UPDATE flag and sets **lpData** to NULL to have your driver update the screen based on data previously received.

The client application sets the ICDRAW_PREROLL flag if it is sending data in advance of the data to be rendered. For example, if the client application will display frame 10 of a compressed sequence, it sets this flag for the first 9 frames. This sends the key frame and other intermediate frames to the driver so it can decompress the tenth frame.

The client application sets the ICDRAW_UPDATE flag to have the driver refresh the screen.

The ICDRAW_NOTKEYFRAME flag indicates the data is not a key frame.

The ICDRAW_NULLFRAME flag indicates the previous frame should be repeated.

When your driver has finished decompressing the data, it returns ICERR_OK. After the driver returns from this message, the client application deallocates or reuses the memory containing the format and image data. If your driver needs the format or image data for future use, it should copy the data it needs before it returns from the message.

Ending Drawing

Your driver receives ICM_DRAW_END when the client application no longer needs data decompressed or rendered. For this message, your driver should free the resources it allocated for the ICM_DRAW_BEGIN message. Your driver should also leave the display in the full-screen mode.

After sending ICM_DRAW_END, the client application must send ICM_DRAW_BEGIN to continue decompressing data. The client application does not have to send an ICM_DRAW_END message for each ICM_DRAW_BEGIN message. The client application can use ICM_DRAW_BEGIN to restart decompression without sending ICM_DRAW_END.

Rendering the Data

The client application sends the following messages to control the driver's internal clock for rendering the decompressed data.

Message	Description
ICM_DRAW_GETTIME	Obtains the value of the driver's internal clock if it is handling the timing of drawing frames.

ICM_DRAW_SETTIME	Sets the driver's internal clock if it is handling the timing of drawing frames.
ICM_DRAW_START	Starts the internal clock of a driver if it handles the timing of drawing frames.
ICM_DRAW_STOP	Stops the internal clock of a driver if it handles the timing of drawing frames.
ICM_DRAW_WINDOW	Informs the driver the display window has been moved, hidden, or displayed.
ICM_DRAW_FLUSH	Flushes any frames that are waiting to be drawn.
ICM_DRAW_RENDERBUFFER	Draws a frame waiting to be drawn.
ICM_DRAW_CHANGEPALETTE	Changes the palette.

The client application sends ICM_DRAW_START to have your driver start (or continue) rendering data at the rate specified by the ICM_DRAW_BEGIN message. The ICM_DRAW_STOP message pauses the internal clock. Neither of these messages use *dwParam1*, *dwParam2*, or a return value.

The client application uses ICM_DRAW_GETTIME to obtain the value of the internal clock. Your driver returns the current-time value (this is usually frame numbers for video) in the DWORD pointed to by *dwParam1*. The current time is relative to the start of drawing.

The client application uses ICM_DRAW_SETTIME to set the value of the internal clock. Typically, the client application uses this message to synchronize the driver's clock to an external clock. Your driver should set its clock to the value (this is usually frame numbers for video) specified in the DWORD pointed to by *dwParam1*.

The client application sends ICM_DRAW_FLUSH to have your driver discard any frames that have not been drawn.

The client application sends ICM_DRAW_RENDERBUFFER to have your driver draw the image currently buffered.

The client application sends ICM_DRAW_CHANGEPALETTE when the palette changes in the movie.

Notifying Applications of Compression and Decompression Status

A client application sends the ICM_SET_STATUS_PROC message to receive notification messages about the progress of compression or decompression. When received, drivers should periodically send notification messages to the callback function specified with the message during compression or decompression. Support of this function is optional but highly recommended if compression or decompression takes longer than approximately one tenth of one second.

The *dwParam1* parameter of the message specifies a pointer to an ICSETSTATUSPROC structure and *dwParam2* specifies the size of the ICSETSTATUSPROC data structure. The **lParam** member specifies a constant passed to the status procedure when it is called. The **fpfnStatus** member specifies a pointer to the status function. This is NULL if status messages should not be sent. The status function has the following prototype:

```
LONG MyStatusProc(LPARAM lParam, UINT message)
```

The *lParam* parameter specifies the constant specified in the **lParam** member of the ICSETSTATUSPROC structure. The *message* parameter specifies one of the following messages the driver sends.

Message	Description
ICSTATUS_START	Indicates the operation is starting.
ICSTATUS_STATUS	Indicates the operation is proceeding, and is *l* percent done.

ICSTATUS_END	Indicates the operation is finishing.
ICSTATUS_YIELD	Indicates a length operation is proceeding.

Using Installable Compressors for Nonvideo Data

Installable compressors are not necessarily limited to video data. By using a different value than 'vidc' in the **fccType** member, you can specify that your installable driver expects to handle a type of data that is not video. Four-character codes for nonvideo data should also be registered.

While VidEdit does not support data that is not audio or video, MCIAVI does provide limited support for other data types, using installable renderers. If you create a stream with a four-character code type that does not represent audio or video, its type and handler information are used to search for a driver capable of handling the data. The search follows the same procedure used for installable compressor drivers.

Writing a driver to render nonvideo data is very similar to rendering video, with the following differences:

- The format used is not a BITMAPINFO structure. The format is defined by the class of decompressor.
- The ICM_DECOMPRESS messages are not used. All data is rendered using the ICM_DRAW messages because there is no defined decompressed form for arbitrary data.

Testing Video Compression and Decompression Drivers

You can exercise both the compression and decompression capabilities of a driver with the VidEdit editing tool. You can also exercise the decompression capabilities of a driver with MCIAVI. (One way to test the decompression capabilities is to preview an unedited file in VidEdit. In this case, VidEdit uses MCIAVI to decompress the file.)

Messages, Compression Drivers

The following topics describe‹This section describes› the messages used by compression drivers.

ICM_ABOUT

The ICM_ABOUT message is sent to a video compression driver to display its About dialog box.

Parameters

dwParam1

Specifies a handle to a window (**HWND**) that should correspond to the parent of the displayed dialog box.

If *dwParam1* is –1, the driver returns ICERR_OK if it has an About dialog box but it should not display the dialog box. The driver returns ICERR_UNSUPPORTED if it does not display a dialog box.

dwParam2

Not used.

Return Value

Returns ICERR_OK if the driver supports this message. Otherwise, returns ICERR_UNSUPPORTED.

ICM_COMPRESS

The ICM_COMPRESS message is sent to a video compression driver to compress a frame of data into an application-supplied buffer.

Parameters

dwParam1

Specifies a pointer to an ICCOMPRESS data structure. The following members of ICCOMPRESS specify the compression parameters:

The **lpbiInput** member contains the format of the uncompressed data; the data itself is in a buffer pointed to by **lpInput**.

The **lpbiOutput** member contains a pointer to the output (compressed) format, and **lpOutput** contains a pointer to a buffer used for the compressed data.

The **lpbiPrev** member contains a pointer to the format of the previous frame, and **lpPrev** contains a pointer to a buffer used for the previous data. These members are used by drivers that do temporal compression.

The driver should use the **biSizeImage** member of the BITMAPINFOHEADER structure associated with **lpbiOutput** to return the size of the compressed frame.

The **lpckid** member points to a DWORD. If the pointer is not NULL, the driver should specify a two-character code for the chunk ID in the DWORD. The chunk ID should correspond to the chunk ID used in the AVI file.

The **lpdwFlags** member points to a DWORD. The driver should fill the DWORD with the flags that should go in the AVI index. In particular, if the returned frame is a key frame, your driver should set the AVIIF_KEYFRAME flag.

The **dwFrameSize** member contains the size into which the compressor should try to make the frame fit. The size value is used for compression methods that can make tradeoffs between compressed image size and image quality.

The **dwQuality** member contains the specific quality the compressor should use, if it supports quality.

dwParam2

Specifies the size of the ICCOMPRESS structure.

Return Value

Returns ICERR_OK if successful. Otherwise, returns an error number.

ICM_COMPRESS_BEGIN

The ICM_COMPRESS_BEGIN message is sent to a video compression driver to prepare it for compressing data.

Parameters

dwParam1

Specifies a pointer to a BITMAPINFO data structure indicating the input format.

dwParam2

Specifies a pointer to a BITMAPINFO data structure indicating the output format.

Return Value

Returns ICERR_OK if the specified compression is supported. Otherwise, returns ICERR_BADFORMAT if the input or output format is not supported.

Comments

The driver should set up any tables or memory that it needs to compress the data formats efficiently when it receives the ICM_COMPRESS message.

The ICM_COMPRESS_BEGIN and ICM_COMPRESS_END messages do not nest. If your driver receives an ICM_COMPRESS_BEGIN message before compression is stopped with ICM_COMPRESS_END, it should restart compression with new parameters.

ICM_COMPRESS_END

The ICM_COMPRESS_END message is sent to a video compression driver to end compression.

The driver should clean up after compressing and release any memory allocated for ICM_COMPRESS_BEGIN.

Parameters

dwParam1
Not used.

dwParam2
Not used.

Return Value

Returns ICERR_OK if successful. Otherwise, returns an error number.

Comments

The ICM_COMPRESS_BEGIN and ICM_COMPRESS_END messages do not nest. If your driver receives an ICM_COMPRESS_BEGIN message before compression is stopped with ICM_COMPRESS_END, it should restart compression with new parameters.

ICM_COMPRESS_FRAMES_INFO

The ICM_COMPRESS_FRAMES_INFO message is sent to a compression driver to set the parameters for the pending compression.

Parameters

dwParam1
Specifies a pointer to an ICCOMPRESSFRAMES structure. For this message, the **GetData** and **PutData** members are not used.

dwParam2
Specifies the size of the structure pointed to by *dwParam1*.

Return Value

Returns ICERR_OK if successful.

Comments

A compressor can use this message to make decisions about the amount of space allocated for each frame while compressing.

ICM_COMPRESS_GET_FORMAT

The ICM_COMPRESS_GET_FORMAT message is sent to a video compression driver to obtain the output format of the compressed data.

Parameters

dwParam1
Specifies a pointer to a BITMAPINFO data structure indicating the input format.

dwParam2
Specifies zero or a pointer to a BITMAPINFO data structure used to return the output format.

Return Value

If *dwParam2* is zero, the driver returns the size of the output format. If *dwParam2* is not zero, the driver returns a status value (either ICERROR_OK or an error status).

Comments

If *dwParam2* is zero, the driver should only return the size of the output format.

If *dwParam2* is nonzero, the driver should fill the BITMAPINFO data structure with the default output format corresponding to the input format specified for *dwParam1*. If the compressor can produce several formats, the default format should be the one that preserves the greatest amount of information.

For example, the Microsoft Video Compressor can compress 16-bit data into either an 8-bit palettized compressed form or a 16-bit true-color compressed form. The 16-bit format more accurately represents the original data, and thus is returned for this message.

ICM_COMPRESS_GET_SIZE

The ICM_COMPRESS_GET_SIZE message is sent to a video compression driver to obtain the maximum size of one frame of data when it is compressed in the output format.

Parameters

dwParam1

Specifies a pointer to a BITMAPINFO data structure indicating the input format.

dwParam2

Specifies a pointer to a BITMAPINFO data structure indicating the output format.

Return Value

Returns the maximum number of bytes a single compressed frame can occupy.

Comments

Typically, applications send this message to determine how large a buffer to allocate for the compressed frame.

The driver should calculate the size of the largest possible frame based on the input and target formats.

ICM_COMPRESS_QUERY

The ICM_COMPRESS_QUERY message is sent to a video compression driver to determine if it can compress a specific input format, or if it can compress the input format to a specific output format.

Parameters

dwParam1

Specifies a pointer to a BITMAPINFO data structure describing the input format.

dwParam2

Specifies a pointer to a BITMAPINFO data structure describing the output format, or zero. Zero indicates any output format is acceptable.

Return Value

Returns ICERR_OK if the specified compression is supported. Otherwise, returns ICERR_BADFORMAT, indicating that the input or output format is not supported.

Comments

On receiving this message, the driver should examine the BITMAPINFO structure associated with *dwParam1* to see if it can compress the input format. The driver should return ICERR_OK only if it can compress the input format to the output format specified for *dwParam2*. (If any output format is acceptable, *dwParam2* is zero.)

ICM_CONFIGURE

The ICM_CONFIGURE message is sent to a video compression driver to display its configuration dialog box.

Parameters

dwParam1

Specifies a handle to a window (**HWND**) that should correspond to parent of the displayed dialog box.

not display the dialog box. The driver returns ICERR_UNSUPPORTED if it does not display a dialog box.

dwParam2
Not used.

Return Value

Returns ICERR_OK if the driver supports this message. Otherwise, returns ICERR_UNSUPPORTED.

Comments

This message is distinct from the DRV_CONFIGURE message used for hardware configuration. The dialog box for this message should let the user configure the internal state referenced by ICM_GETSTATE and ICM_SETSTATE. For example, this dialog box might let the user change parameters affecting the quality level and other similar compression options.

ICM_DECOMPRESS

The ICM_DECOMPRESS message is sent to a video compression driver to decompress a frame of data into an application-supplied buffer.

Parameters

dwParam1
Specifies a pointer to an ICDECOMPRESS structure.

dwParam2
Specifies the size of the ICDECOMPRESS structure.

Return Value

Returns ICERR_OK if successful. Otherwise, returns an error number.

Comments

If the driver is to decompress data directly to the screen instead of a buffer, it receives the ICM_DRAW message.

The driver should return an error if this message is received before the ICM_DECOMPRESS_BEGIN message.

ICM_DECOMPRESS_BEGIN

The ICM_DECOMPRESS_BEGIN message is sent to a video compression driver for decompressing data. When the driver receives this message, it should allocate buffers and do any time-consuming operations so it can process ICM_DECOMPRESS messages efficiently.

Parameters

dwParam1
Specifies a pointer to a BITMAPINFO data structure describing the input format.

dwParam2
Specifies a pointer to a BITMAPINFO data structure describing the output format.

Return Value

Returns ICERR_OK if the specified decompression is supported. Otherwise, returns ICERR_BADFORMAT indicating that the input or output format is not supported.

Comments

To have the driver decompress data directly to the screen, an application sends the ICM_DRAW_BEGIN message.

ICM_DECOMPRESS_BEGIN and ICM_DECOMPRESS_END do not nest. If your driver receives an ICM_DECOMPRESS_BEGIN message before decompression is stopped with ICM_DECOMPRESS_END, it should restart decompression with new parameters.

ICM_DECOMPRESS_END

The ICM_DECOMPRESS_END message is sent to a video decompression driver to have it clean up after decompressing.

Parameters

dwParam1
Not used.

dwParam2
Not used.

Return Value

Returns ICERR_OK if successful. Otherwise, returns an error number.

Comments

The driver should free any resources allocated for ICM_DECOMPRESS_BEGIN.

ICM_DECOMPRESS_BEGIN and ICM_DECOMPRESS_END do not nest. If your driver receives ICM_DECOMPRESS_BEGIN before decompression is stopped with ICM_DECOMPRESS_END, it should restart decompression with new parameters.

ICM_DECOMPRESS_GET_FORMAT

The ICM_DECOMPRESS_GET_FORMAT message is sent to a video decompression driver to obtain the format of the decompressed data.

Parameters

dwParam1
Specifies a pointer to a BITMAPINFO data structure describing the input format.

dwParam2
Specifies zero or a pointer to a BITMAPINFO data structure used to return the output format.

Return Value

If *dwParam2* is zero, the driver returns the size of the output format. If *dwParam2* is not zero, the driver returns a status value (either ICERROR_OK or an error status).

Comments

If *dwParam2* is zero, the driver returns only the size of the output format. An application sets *dwParam2* to zero to determine the size of the buffer it needs to allocate.

If *dwParam2* is nonzero, the driver should fill the BITMAPINFO data structure with the default output format corresponding to the input format specified for *dwParam1*. If the compressor can produce several different formats, the default format should be the one that preserves the greatest amount of information.

For example, if a driver can produce either 24-bit full-color images or 8-bit gray-scale images, the default should be 24-bit images. This ensures the highest possible image quality if the video data must be edited and recompressed.

ICM_DECOMPRESS_GET_PALETTE

The ICM_DECOMPRESS_GET_PALETTE message is sent to a video decompression driver to obtain the color table of the output BITMAPINFOHEADER structure.

Parameters

dwParam1
Specifies a pointer to a BITMAPINFOHEADER data structure indicating the input format.

dwParam2
Specifies zero or a pointer to a BITMAPINFOHEADER data structure used to return the color

table. The space reserved for the color table is always at least 256 colors.

Return Value

Returns the size of the output format or an error code.

Comments

If *dwParam2* is zero, the driver returns only the size of the output format. Applications specify zero to determine the size of the output format.

If *dwParam2* is nonzero, the driver sets the **biClrUsed** member of the BITMAPINFOHEADER data structure to the number of colors in the color table. The driver fills the **bmiColors** members of the BITMAPINFO data structure with the actual colors.

The driver should support this message only if it uses a palette other than the one specified in the input format.

ICM_DECOMPRESS_QUERY

The ICM_DECOMPRESS_QUERY message is sent to a video compression driver to determine if the driver can decompress a specific input format, or if it can decompress the input format to a specific output format.

Parameters

dwParam1

Specifies a pointer to a BITMAPINFO structure describing the input format.

dwParam2

Specifies zero or a pointer to a BITMAPINFO structure used to return the output format. Zero indicates that any output format is acceptable.

Return Value

Returns ICERR_OK if the specified decompression is supported. Otherwise returns ICERR_BADFORMAT, indicating that the input or output format is not supported.

ICM_DECOMPRESS_SET_PALETTE

The ICM_DECOMPRESS_SET_PALETTE message specifies a palette for a video decompression driver to use if it is decompressing to a format that uses a palette.

Parameters

DwParam1

Address of a BITMAPINFOHEADER structure whose color table contains the colors that should be used if possible. If zero, use the default set of output colors.

DwParam2

Not used.

Return Values

Returns ICERR_OK if the decompression driver can precisely decompress images to the suggested palette using the set of colors as they are arranged in the palette. Returns ICERR_UNSUPPORTED otherwise.

Comments

This message should not affect decompression already in progress; rather, colors passed using this message should be returned in response to future ICM_DECOMPRESS_GET_FORMAT and ICM_DECOMPRESS_GET_PALETTE messages. Colors are sent back to the decompression driver in a future ICM_DECOMPRESS_BEGIN message.

This message is used primarily when a driver decompresses images to the screen and another application that uses a palette is in the foreground, forcing the decompression driver to adapt to a foreign set of colors.

The ICM_DECOMPRESSEX message is sent to a video compression driver to decompress a frame of data directly to the screen, decompress to an upside-down DIB, or decompress images described with source and destination rectangles. This message is similar to ICM_DECOMPRESS, except that it uses the ICDECOMPRESSEX structure to decompression information.

Parameters

dwParam1
Specifies a pointer to an ICDECOMPRESSEX structure.

dwParam2
Specifies the size of the ICDECOMPRESSEX structure.

Return Value

Returns ICERR_OK if successful. Otherwise, returns an error number.

Comments

If the driver is to decompress data directly to the screen instead of a buffer, it receives the ICM_DRAW message.

The driver should return an error if this message is received before the ICM_DECOMPRESSEX_BEGIN message.

ICM_DECOMPRESSEX_BEGIN

The ICM_DECOMPRESSEX_BEGIN message is sent to a video compression driver for decompressing data. When the driver receives this message, it should allocate buffers and do any time-consuming operations so that it can process ICM_DECOMPRESSEX messages efficiently.

Parameters

dwParam1
Specifies a pointer to an ICDECOMPRESSEX data structure describing the input and output formats.

dwParam2
Specifies the size of an ICDECOMPRESSEX data structure.

Return Value

Returns ICERR_OK if the specified decompression is supported. Otherwise returns ICERR_BADFORMAT, indicating that the input or output format is not supported.

Comments

To have the driver decompress data directly to the screen, an application sends the ICM_DRAW_BEGIN message.

The ICM_DECOMPRESSEX_BEGIN and ICM_DECOMPRESSEX_END messages do not nest. If your driver receives an ICM_DECOMPRESSEX_BEGIN message before decompression is stopped with ICM_DECOMPRESSEX_END, it should restart decompression with new parameters.

ICM_DECOMPRESSEX_END

The ICM_DECOMPRESSEX_END message is sent to a video decompression driver to have it clean up after decompressing.

Parameters

dwParam1
Not used.

dwParam2

Not used.

Return Value

Returns ICERR_OK if successful. Otherwise, returns an error number.

Comments

The driver should free any resources allocated for ICM_DECOMPRESSEX_BEGIN.

The ICM_DECOMPRESSEX_BEGIN and ICM_DECOMPRESSEX_END messages do not nest. If your driver receives ICM_DECOMPRESSEX_BEGIN before decompression is stopped with ICM_DECOMPRESSEX_END, it should restart decompression with new parameters.

ICM_DECOMPRESSEX_QUERY

The ICM_DECOMPRESSEX_QUERY message is sent to a video compression driver to determine if the driver can decompress a specific input format, or if it can decompress the input format to a specific output format.

Parameters

dwParam1

Specifies a pointer to an ICDECOMPRESSEX structure describing the input format.

dwParam2

Specifies zero or a pointer to an ICDECOMPRESSEX structure used to return the output format. Zero indicates that any output format is acceptable.

Return Value

Returns ICERR_OK if the specified decompression is supported. Otherwise, returns ICERR_BADFORMAT indicating that the input or output format is not supported.

ICM_DRAW

The ICM_DRAW message is sent to a rendering driver to decompress a frame of data and draw it to the screen.

Parameters

dwParam1

Specifies a pointer to an ICDRAW structure.

dwParam2

Specifies the size of the ICDRAW structure.

Return Value

Returns ICERR_OK if successful. Otherwise, returns an error number.

Comments

If the ICDRAW_UPDATE flag is set in the **dwFlags** member of the ICDRAW data structure, the area of the screen used for drawing is invalid and needs to be updated. This depends on the contents of **dwFlags**.

If the ICDRAW_UPDATE flag is set and **dwFlags** is NULL, the driver should update the entire destination rectangle with the current image. If the driver does not keep an offscreen image, it can fail this message.

If the ICDRAW_UPDATE flag is set and **dwFlags** is nonNULL, the driver should draw the data and make sure the entire destination is updated. If the driver does not keep an offscreen image, it can fail this message.

If the ICDRAW_HURRYUP flag is set in the **dwFlags** member, the calling application is requesting the driver to proceed as quickly as possible, possibly not even updating the screen.

If the ICDRAW_PREROLL flag is set in the **dwFlags** member, this video frame is merely preliminary information and should not be displayed, if possible. For instance, if play is to start

from frame 10, and frame 0 is the nearest previous key frame, frames 0 through 9 will have the ICDRAW_PREROLL flag set.

If the driver is to decompress data into a buffer instead of drawing directly to the screen, ICM_DECOMPRESS is sent instead.

ICM_DRAW_BEGIN

The ICM_DRAW_BEGIN message is sent to a rendering driver to prepare it for drawing data.

Parameters

dwParam1

Specifies a pointer to an ICDRAWBEGIN data structure describing the input format.

dwParam2

Specifies the size of the ICDRAWBEGIN data structure describing the input format.

Return Value

Returns ICERR_OK if the driver supports drawing the data to the screen in the manner and format specified. Otherwise, returns ICERR_BADFORMAT if the input or output format is not supported, or ICERR_NOTSUPPORTED if the message is not supported.

Comments

If the driver is to decompress data into a buffer instead of drawing directly to the screen, ICM_DECOMPRESS_BEGIN is sent rather than this one.

If the driver does not support drawing directly to the screen, it should return ICERR_NOTSUPPORTED.

The ICM_DRAW_BEGIN and ICM_DRAW_END messages do not nest. If your driver receives ICM_DRAW_BEGIN before decompression is stopped with ICM_DRAW_END, it should restart decompression with new parameters.

ICM_DRAW_CHANGEPALETTE

The ICM_DRAW_CHANGEPALETTE message is sent to a rendering driver if the palette of the movie is changing.

Parameters

dwParam1

Points to a BITMAPINFO structure that contains the new format and optional color table.

dwParam2

Not used. Set to zero.

Return Value

Returns ICERR_OK if successful.

Comments

This message should be supported by installable rendering handlers if they draw palettized DIBs.

ICM_DRAW_END

The ICM_DRAW_END message is sent to rendering drivers to clean up after decompressing an image to the screen.

Parameters

dwParam1

Not used.

dwParam2

Not used.

Return Value

Returns ICERR_OK if successful. Otherwise, returns an error number.

Comments

The ICM_DRAW_BEGIN and ICM_DRAW_END messages do not nest. If your driver receives ICM_DRAW_BEGIN before decompression is stopped with ICM_DRAW_END, it should restart decompression with new parameters.

ICM_DRAW_FLUSH

The ICM_DRAW_FLUSH message is sent to a rendering driver to flush any frames that are waiting to be drawn.

Parameters

dwParam1
Not used.

dwParam2
Not used.

Return Value

None.

Comments

This message is used only by hardware that does its own asynchronous decompression, timing, and drawing.

ICM_DRAW_GET_PALETTE

The ICM_DRAW_GET_PALETTE message is sent to a rendering driver to obtain a palette.

Parameters

dwParam1
Not used.

dwParam2
Not used.

Return Value

The driver should return a palette handle. It should return NULL or IC_UNSUPPORTED if it doesn't have a handle to return.

ICM_DRAW_GETTIME

The ICM_DRAW_GETTIME message is sent to a rendering driver to obtain the current value of its internal clock if it is handling the timing of drawing frames.

Parameters

dwParam1
Specifies a pointer to a LONG data type used to return the current time. The return value should be specified in samples. This corresponds to frames for video.

dwParam2
Not used.

Return Value

Returns ICERR_OK if successful.

Comments

This message is generally only supported by hardware that does its own asynchronous decompression, timing, and drawing. The message will also be sent only if the hardware is being

used as the synchronization master.

ICM_DRAW_QUERY

The ICM_DRAW_QUERY message is sent to a rendering driver to determine if it can render data in a specific format.

Parameters

dwParam1
Specifies a pointer to a BITMAPINFO structure describing the input format.

dwParam2
Not used.

Return Value

Returns ICERR_OK if the driver can render data in the specified format. Otherwise, returns ICERR_BADFORMAT indicating that the format is not supported.

Comments

This message asks if the driver recognizes the format for drawing operations. The ICM_DRAW_BEGIN message determines if the driver can draw the data.

ICM_DRAW_REALIZE

The ICM_DRAW_REALIZE message is sent to a rendering driver to realize its palette used while drawing.

Parameters

dwParam1
Specifies a handle to the display context used to realize the palette.

dwParam2
Specifies TRUE if the palette is to be realized in the background. Specifies FALSE if the palette is to be realized in the foreground.

Return Value

Returns ICERR_OK if the palette is realized. If this message is not supported, the driver returns ICERR_UNSUPPORTED.

Comments

Drivers need to respond to this message only if the drawing palette is different from the decompressed palette.

If this message is not supported, the palette associated with the decompressed data is realized.

ICM_DRAW_RENDERBUFFER

The ICM_DRAW_RENDERBUFFER message is sent to a rendering driver to draw the frames that have been passed to it.

Parameters

dwParam1
Not used.

dwParam2
Not used.

Return Value

None.

Comments

This message is typically used to perform a "seek" operation when, rather than playing a sequence of video frames, the driver must be specifically instructed to display each video frame passed to it.

This message is used only by hardware that does its own asynchronous decompression, timing, and drawing.

ICM_DRAW_SETTIME

The ICM_DRAW_SETTIME message is sent to a rendering driver to inform it of what frame it should be drawing if it is handling the timing of drawing frames.

Parameters

dwParam1
Specifies a LONG data type containing the sample number corresponding to the frame the driver should be rendering. The value will be specified in samples. This corresponds to frames for video.

dwParam2
Not used.

Return Value

Returns ICERR_OK if successful.

Comments

This message is generally only supported by hardware that does its own asynchronous decompression, timing, and drawing. The message will only be sent if the hardware is not being used as the synchronization master.

Typically, the driver will compare the specified "correct" value with its internal clock, and take actions to synchronize the two if the difference is significant.

ICM_DRAW_START

The ICM_DRAW_START message is sent to a rendering driver to start its internal clock for the timing of drawing frames.

Parameters

dwParam1
Not used.

dwParam2
Not used.

Return Value

None.

Comments

This message is used only by hardware that does its own asynchronous decompression, timing, and drawing.

When it receives this message, the driver should start rendering data at the rate specified with ICM_DRAW_BEGIN.

The ICM_DRAW_START and ICM_DRAW_STOP messages do not nest. If your driver receives ICM_DRAW_START before rendering is stopped with ICM_DRAW_STOP, it should restart rendering with new parameters.

ICM_DRAW_START_PLAY

The ICM_DRAW_START_PLAY message provides the start and end times of a play operation to a rendering driver.

Parameters

dwParam1
Start time.

dwParam2
End time.

Return Values

This message does not return a value.

Comments

This message precedes any frame data sent to the rendering driver.

Units for *lFrom* and *lTo* are specified with the ICM_DRAW_BEGIN message. For video data this is normally a frame number. For more information about the playback rate, see the **dwRate** and **dwScale** members of the ICDRAWBEGIN structure.

If the end time is less than the start time, the playback direction is reversed.

ICM_DRAW_STOP

The ICM_DRAW_STOP message is sent to a rendering driver to stop its internal clock for the timing of drawing frames.

Parameters

dwParam1
Not used.

dwParam2
Not used.

Return Value

None.

Comments

This message is used only by hardware that does its own asynchronous decompression, timing, and drawing.

The ICM_DRAW_STOP and ICM_DRAW_START messages do not nest. If your driver receives ICM_DRAW_START before rendering is stopped with ICM_DRAW_STOP, it should restart rendering with new parameters.

ICM_DRAW_STOP_PLAY

The ICM_DRAW_STOP_PLAY message notifies a rendering driver when a play operation is complete.

Parameters

dwParam1
Not used.

dwParam2
Not used.

Return Value

This message does not return a value.

Remarks

This message is used when the play operation is complete. The ICM_DRAW_STOP message is used to end timing.

The ICM_DRAW_SUGGESTFORMAT message is sent to a rendering driver to have it suggest a decompressed format that it can draw.

Parameters

dwParam1
Specifies a pointer to an ICDRAWSUGGEST data structure.

dwParam2
Specifies the size of the ICDRAWSUGGEST data structure.

Return Value

Returns ICERR_OK if successful. If **lpbiSuggest** is NULL, returns the amount of memory to hold the format that would have been suggested.

Comments

The driver should examine the format specified in the **lpbiIn** member of the ICDRAWSUGGEST structure and use the **lpbiSuggest** member to return a format it can draw. The returned format should be as similar as possible to the input format.

For example, if the driver can draw only 8 or 16-bit RGB data, and the input format is 320-by-240-by-16-bit MSVideo1 compressed data, the renderer suggests the 320-by-240-by-16-bit RGB format.

Optionally, the driver can use the installable compressor handle passed in **hicDecompressor** to make more complex selections. For example, if the input format is 24-bit JPEG data, a renderer could query the compressor to find out if it can decompress to a YUV format (which might be drawn more efficiently) before selecting the format to suggest.

ICM_DRAW_WINDOW

The ICM_DRAW_WINDOW message is sent to a rendering driver when the window specified for ICM_DRAW_BEGIN has moved, or has become totally obscured. This message is used by overlay drivers so they can draw when the window is obscured or moved.

Parameters

dwParam1
Points to the destination rectangle in screen coordinates. If *dwParam1* points to an empty rectangle, drawing should be turned off.

dwParam2
Not used.

Return Value

Returns ICERR_OK if successful.

Comments

This message is only supported by hardware that does its own asynchronous decompression, timing, and drawing.

The rectangle is empty if the window is totally hidden by other windows. Drivers should turn off overlay hardware when the rectangle is empty.

ICM_GET

The ICM_GET message retrieves a buffer of driver-defined status information from a video compression driver.

Parameters

dwParam
Address of a block of memory to be filled with driver-defined status information. If NULL, the

driver should return the amount of memory required by the driver information.

DwParam2

Size, in bytes, of the block of memory.

Return Value

Returns the amount of memory, in bytes, required to store the status information.

Comments

The structure used to represent status information is driver-specific and defined by the driver.

ICM_GETBUFFERSWANTED

The ICM_GETBUFFERSWANTED message is sent to a video compression driver to have it return information about how many samples the driver will pre-buffer.

Parameters

dwParam1

Specifies a pointer to a DWORD. The driver uses the DWORD to return the number of samples it needs to get in advance of when they will be presented.

dwParam2

Not used.

Return Value

Returns ICERR_OK if successful. Otherwise, returns ICERR_UNSUPPORTED.

Comments

Typically, this message is only used by a driver that uses hardware to render data and must ensure hardware pipelines remain full. For example, if a driver controls a video decompression board that can hold ten frames of video, it could return ten for this message. This instructs an application to try and stay ten frames ahead of the frame it currently needs.

ICM_GETDEFAULTKEYFRAMERATE

The ICM_GETDEFAULTKEYFRAMERATE message is sent to a video compression driver to obtain its default (or preferred) key frame spacing.

Parameters

dwParam1

Specifies a pointer to a DWORD used to return the preferred key frame spacing.

dwParam2

Not used.

Return Value

Returns ICERR_OK if the driver supports this message. Otherwise, returns ICERR_UNSUPPORTED.

ICM_GETDEFAULTQUALITY

The ICM_GETDEFAULTQUALITY message is sent to a video compression driver to obtain its default quality setting.

Parameters

dwParam1

Specifies a pointer to a DWORD used to return the default quality value.

dwParam2

Not used.

Return Value

Returns ICERR_OK if the driver supports this message. If not, returns ICERR_UNSUPPORTED.

Comments

Quality values range from 0 through 10,000.

ICM_GETINFO

The ICM_GETINFO message is set to a video compression driver to have it return information describing the driver.

Parameters

dwParam1

Specifies a pointer to an ICINFO data structure used to return information.

dwParam2

Specifies the size of the ICINFO data structure.

Return Value

Returns the size of the ICINFO data structure, or zero if an error occurs.

Comments

Typically, this message is sent by applications to display a list of the installed compressors.

The driver should fill in all members of the ICINFO structure except the **szDriver** member.

ICM_GETQUALITY

The ICM_GETQUALITY message is sent to a video compression driver to obtain its current quality setting.

Parameters

dwParam1

Specifies a pointer to a DWORD used to return the current quality value.

dwParam2

Not used.

Return Value

Returns ICERR_OK if the driver supports this message. If not, returns ICERR_UNSUPPORTED.

Comments

Quality values range from 0 through 10,000.

ICM_GETSTATE

The ICM_GETSTATE message is sent to a video compression driver to have it fill a block of memory describing the compressor's current configuration.

Parameters

dwParam1

Specifies a pointer to a block of memory to be filled with the current state, or NULL. If NULL, return the amount of memory required by the state information.

dwParam2

Specifies the size of the block of memory.

Return Value

If *dwParam1* is NULL, the driver returns the size of the configuration information. If *dwParam1* is not NULL, and the value received in *dwParam2* is less than size of the configuration information, the driver returns zero. Otherwise the driver returns the size of the information it returns in the structure supplied by *dwParam1*.

Comments

Client applications send this message with *dwParam1* set to NULL to determine the size of the memory block required for obtaining the state information.

The data structure used to represent state information is driver-specific and is defined by the driver.

ICM_SET_STATUS_PROC

The ICM_SET_STATUS_PROC message is periodically sent to an application's status callback function during lengthy operations.

Parameters

dwParam1

Specifies a pointer to an ICSETSTATUSPROC structure.

dwParam2

Specifies the size of the ICSETSTATUSPROC data structure.

Return Value

Returns ICERR_OK if successful.

Comments

Support of this function is optional but highly recommended if compression or decompression takes longer than approximately one tenth of one second.

ICM_SETQUALITY

The ICM_SETQUALITY message is sent to a video compression driver to set the quality level for compression.

Parameters

dwParam1

Specifies the new quality value.

dwParam2

Not used.

Return Value

Returns ICERR_OK if the driver supports this message. If not, returns ICERR_UNSUPPORTED.

Comments

Quality values range from 0 through 10,000.

ICM_SETSTATE

The ICM_SETSTATE message is sent to a video compression driver to set the state of the compressor.

Parameters

dwParam1

Specifies a pointer to a block of memory containing configuration data, or NULL. If NULL, the driver should return to its default state.

dwParam2

Specifies the size of the block of memory.

Return Value

Returns the number of bytes actually used by the compressor. A return value of zero generally indicates an error.

Comments

Because the information used by ICM_SETSTATE is private and specific to a given compressor, client applications should use this message only to restore information previously obtained with the ICM_GETSTATE message.

Callback Functions, Compression Drivers

The following topics describe the callback functions used by compression drivers.

Status

LONG Status(LPARAM *lParam*, **UINT** *message*, **LONG** *l*);

This is the prototype for a status function used by the ICM_SET_STATUS_PROC message.

Parameters

lParam

Contains the constant specified with the status callback address.

message

Specifies one of the following values:

Value	Meaning
ICSTATUS_START	Indicates a lengthy operation is starting.
ICSTATUS_STATUS	Indicates the operation is proceeding, and is *l* percent done.
ICSTATUS_END	Indicates a lengthy operation is finishing.
ICSTATUS_YIELD	Indicates a lengthy operation is proceeding. Essentially, this is just the same as ICSTATUS_STATUS without a specific percentage value.

l

Message-specific data.

Return Value

Returns zero if processing should continue, or a nonzero value if it should end.

Structures, Compression Drivers

The following topics describe the structures used by compression drivers.

ICCOMPRESS

```
typedef struct {
    DWORD dwFlags;
    LPBITMAPINFOHEADER lpbiOutput;
    LPVOID lpOutput;
    LPBITMAPINFOHEADER lpbiInput;
    LPVOID lpInput;
    LPDWORD lpckid;
    LPDWORD lpdwFlags;
    LONG lFrameNum;
    DWORD dwFrameSize;
    DWORD dwQuality;
    LPBITMAPINFOHEADER lpbiPrev;
    LPVOID lpPrev;
} ICCOMPRESS;
```

The ICCOMPRESS structure is used with the ICM_COMPRESS message to specify compression

parameters.

Members

dwFlags
Specifies flags used for compression. At present, only the ICCOMPRESS_KEYFRAME flag is defined and indicates that the input data should be treated as a key frame.

lpbiOutput
Specifies a pointer to a BITMAPINFOHEADER structure containing the output (compressed) format. The **biSizeImage** member of the BITMAPINFOHEADER structure must be filled in with the size of the compressed data.

lpOutput
Specifies a pointer to the buffer where the driver should write the compressed data.

lpbiInput
Specifies a pointer to a BITMAPINFOHEADER structure containing the input format.

lpInput
Specifies a pointer to the buffer containing input data.

lpckid
Specifies a pointer to a buffer used to return the chunk ID for data in the AVI file.

lpdwFlags
Specifies a pointer to a buffer used to return flags for the AVI index.

lFrameNum
Specifies the frame number of the frame to compress.

dwFrameSize
Specifies zero, or the desired maximum size (in bytes) for compressing this frame.

dwQuality
Specifies the compression quality.

lpbiPrev
Specifies a pointer to a BITMAPINFOHEADER structure containing the format of the previous frame. Normally, this is the same as the input format.

lpPrev
Specifies a pointer to the buffer containing the previous frame.

ICCOMPRESSFRAMES

```
typedef struct {
    DWORD dwFlags;
    LPBITMAPINFOHEADER lpbiOutput;
    LPARAM lOutput;
    LPBITMAPINFOHEADER lpbiInput;
    LPARAM lInput;
    LONG lStartFrame;
    LONG lFrameCount;
    LONG lQuality;
    LONG lDataRate;
    LONG lKeyRate;
    DWORD dwRate;
    DWORD dwScale;
    DWORD dwOverheadPerFrame;
    DWORD dwReserved2;
    LONG (CALLBACK* GetData) (LPARAM lInput, LONG lFrame, LPVOID lpBits,
      LONG len);
    LONG (CALLBACK* PutData) (LPARAM lOutput, LONG lFrame, LPVOID lpBits,
      LONG len);
} ICCOMPRESSFRAMES;
```

The ICCOMPRESSFRAMES structure is used with the ICM_COMPRESS_FRAMES_INFO message to specify compression parameters.

dwFlags
Specifies flags used for compression. At present, only the ICDECOMPRESSFRAMES_PADDING flag is defined and indicates that padding is used with the frame.

lpbiOutput
Specifies a pointer to a BITMAPINFOHEADER structure containing the output format.

lOutput
Specifies a parameter returned by the function specified by the **PutData** member.

lpbiInput
Specifies a pointer to a BITMAPINFOHEADER structure containing the input format.

lInput
Specifies a parameter returned by the function specified by the **GetData** member.

lStartFrame
Specifies the starting frame number to be compressed.

lFrameCount
Specifies the number of frames to compress.

lQuality
Specifies the quality.

lDataRate
Specifies the maximum data rate in bytes per second.

lKeyRate
Specifies the maximum spacing, in frames, between key frames.

dwRate
Specifies the compression rate in an integer format. To obtain the rate in frames-per-second, divide this value by the value in *dwScale*.

dwScale
Specifies the value used to scale *dwRate* to frames-per-second.

dwOverheadPerFrame
Reserved.

dwReserved2
Reserved.

GetData
Specifies the callback function used get frames of data to compress. (Not used.)

PutData
Specifies the callback function used to send frames of compressed data. (Not used.)

ICDECOMPRESS

```
typedef struct {
    DWORD dwFlags;
    LPBITMAPINFOHEADER lpbiInput;
    LPVOID lpInput;
    LPBITMAPINFOHEADER lpbiOutput;
    LPVOID lpOutput;
    DWORD ckid;
} ICDECOMPRESS;
```

The ICDECOMPRESS structure is used with the ICM_DECOMPRESS message to specify the parameters for decompressing the data.

Members

dwFlags
Specifies applicable flags. The following flags are defined:

Flag	Meaning
ICDECOMPRESS_HURRYUP	Indicates the data is just buffered and not drawn to the screen. Use this flag for the

	fastest decompression.
ICDECOMPRESS_UPDATE	Indicates the screen is being updated.
ICDECOMPRESS_PREROLL	Indicates this frame is not drawn because it is prior to the point in the movie where play begins.
ICDECOMPRESS_NULLFRAME	Indicates this frame does not have any data, and the decompressed image should be left the same.
ICDECOMPRESS_NOTKEYFRAME	Indicates that this frame is not a key frame.

lpbiInput
Specifies a pointer to a BITMAPINFOHEADER structure containing the input format.

lpInput
Specifies a pointer to a data buffer containing the input data.

lpbiOutput
Specifies a pointer to a BITMAPINFOHEADER structure containing the output format.

lpOutput
Specifies a pointer to a data buffer where the driver should write the decompressed image.

ckid
Specifies the chunk ID from the AVI file.

ICDECOMPRESSEX

```
typedef struct {
    DWORD dwFlags;
    LPBITMAPINFOHEADER lpbiSrc;
    LPVOID lpSrc;
    LPBITMAPINFOHEADER lpbiDst;
    LPVOID lpDst;
    int xDst;
    int yDst;
    int dxDst;
    int dyDst;
    int xSrc;
    int ySrc;
    int dxSrc;
    int dySrc;
} ICDECOMPRESSEX;
```

The ICDECOMPRESSEX structure is used with the ICM_DECOMPRESSEX message to specify the parameters for decompressing the data.

Members

dwFlags
Specifies applicable flags. The following flags are defined:

Flag	Meaning
ICDECOMPRESS_HURRYUP	Indicates the data is just buffered and not drawn to the screen. Use this flag for the fastest decompression.
ICDECOMPRESS_UPDATE	Indicates the screen is being updated.
ICDECOMPRESS_PREROLL	Indicates this frame will not actually be drawn, because it is before the point in the movie where play will start.
ICDECOMPRESS_NULLFRAME	Indicates this frame does not have any data, and the decompressed image should be left the same.

ICDECOMPRESS_NOTKEYFRAME	Indicates that this frame is not a key frame.

lpbiSrc
Specifies a pointer to a BITMAPINFOHEADER structure containing the input format.

lpSrc
Specifies a pointer to a data buffer containing the input data.

lpbiDst
Specifies a pointer to a BITMAPINFOHEADER structure containing the output format.

lpDst
Specifies a pointer to a data buffer where the driver should write the decompressed image.

xDst
Specifies the x-coordinate of the destination rectangle within the DIB specified by **lpbiDst**.

yDst
Specifies the y-coordinate of the destination rectangle.

dxDst
Specifies the width of the destination rectangle.

dyDst
Specifies the height of the destination rectangle.

xSrc
Specifies the x-coordinate of the source rectangle, within the DIB specified by **lpbiSrc**.

ySrc
Specifies the y-coordinate of the source rectangle.

dxSrc
Specifies the width of the source rectangle.

dySrc
Specifies the height of the source rectangle.

ICDRAW

```
typedef struct {
    DWORD dwFlags;
    LPVOID lpFormat;
    LPVOID lpData;
    DWORD cbData;
    LONG lTime;
} ICDRAW;
```

The ICDRAW structure is used with the ICM_DRAW message to specify the parameters for drawing video data to the screen.

Members

dwFlags
Specifies the flags from the AVI file index. The following flags are defined:

Flag	Meaning
ICDRAW_HURRYUP	Indicates the data is just buffered and not drawn to the screen. Use this flag for the fastest decompression.
ICDRAW_UPDATE	Indicates the driver should update the screen based on data previously received. In this case, the *lpData* parameter should be ignored.
ICDRAW_PREROLL	Indicates that this frame of video occurs before actual playback should start. For instance, if playback is to begin on frame 10, and frame 0 is the nearest previous key frame, frames 0 through 9 are sent to the driver with the ICDRAW_PREROLL flag set. The driver needs this data so it can display frame 10 properly, but frames 0

	through 9 need not be individually displayed.
ICDRAW_NULLFRAME	Indicates the previous frame should be repeated.
ICDRAW_NOTKEYFRAME	Indicates the image is not a key frame.

lpFormat
Specifies a pointer to a structure containing the data format. For video, this is a BITMAPINFOHEADER structure.

lpData
Specifies the data to be rendered.

cbData
Specifies the number of bytes of data to be rendered.

lTime
Specifies the time, in samples, when this data should be drawn. For video data, this is usually a frame number. See **dwRate** and **dwScale** of the ICDRAW structure for details.

ICDRAWBEGIN

```
typedef struct {
    DWORD dwFlags;
    HPALETTE hpal;
    HWND hwnd;
    HDC hdc;
    int xDst;
    int yDst;
    int dxDst;
    int dyDst;
    LPBITMAPINFOHEADER lpbi;
    int xSrc;
    int ySrc;
    int dxSrc;
    int dySrc;
    DWORD dwRate;
    DWORD dwScale;
} ICDRAWBEGIN;
```

The ICDRAWBEGIN structure is used with the ICM_DRAW_BEGIN message to specify the parameters used to decompress the data.

Members

dwFlags
Specifies any of the following flags:

Flag	Meaning
ICDRAW_QUERY	Set when an application must determine if the device driver can handle the operation. The device driver does not actually perform the operation.
ICDRAW_FULLSCREEN	Indicates the full screen is used to draw the decompressed data.
ICDRAW_HDC	Indicates a window or display context is used to draw the decompressed data.
ICDRAW_ANIMATE	Indicates the palette might be animated.
ICDRAW_CONTINUE	Indicates drawing is a continuation of the previous frame.
ICDRAW_MEMORYDC	Indicates the display context is offscreen.
ICDRAW_UPDATING	Indicates the frame is being updated rather than being played.

hpal
Specifies a handle of the palette used for drawing.

hwnd
Specifies the handle of the window used for drawing.

hdc
Specifies the handle of the display context used for drawing. If NULL is specified, a display context to the specified window should be used.

xDst
Specifies the x-position of the destination rectangle.

yDst
Specifies the y-position of the destination rectangle.

dxDst
Specifies the width of the destination rectangle.

dyDst
Specifies the height of the destination rectangle.

lpbi
Specifies a pointer to a BITMAPINFOHEADER data structure containing the input format.

xSrc
Specifies the x-position of the source rectangle.

ySrc
Specifies the y-position of the source rectangle.

dxSrc
Specifies the width of the source rectangle.

dySrc
Specifies the height of the source rectangle.

dwRate
Specifies the decompression rate in an integer format. To obtain the rate in frames-per-second, divide this value by the value in **dwScale**.

dwScale
Specifies the value used to scale **dwRate** to frames-per-second.

ICDRAWSUGGEST

```
typedef struct {
    DWORD dwFlags;
    LPBITMAPINFOHEADER lpbiIn;
    LPBITMAPINFOHEADER lpbiSuggest;
    int dxSrc;
    int dySrc;
    int dxDst;
    int dyDst;
    HIC hicDecompressor;
} ICDRAWSUGGEST;
```

The ICDRAWSUGGEST structure is used with the ICM_DRAW_SUGGESTFORMAT message.

Members

dwFlags
Specifies applicable flags. Set this to zero.

lpbiIn
Specifies a pointer to the structure containing the compressed input format.

lpbiSuggest
Specifies a pointer to a buffer used to return the suggested format that the draw device would like to receive.

dxSrc
Specifies the source width.

dySrc

Specifies the source height.

dxDst
Specifies the destination width.

dyDst
Specifies the destination height.

hicDecompressor
Specifies a decompressor that can work with the format of data in **lpbiIn**.

ICINFO

```
typedef struct {
    DWORD dwSize;
    DWORD fccType;
    DWORD fccHandler;
    DWORD dwFlags;
    DWORD dwVersion;
    DWORD dwVersionICM;
    char szName[16];
    char szDescription[128];
    char szDriver[128];
} ICINFO;
```

The ICINFO structure is filled by a video compression driver when it receives the ICM_GETINFO message.

Members

dwSize
Should be set to the size of the ICINFO structure.

fccType
Specifies a four-character code representing the type of stream being compressed or decompressed. Set this to 'vidc' for video streams.

fccHandler
Specifies a four-character code identifying a specific compressor.

dwFlags
Specifies any flags. The following flags are defined for video compressors:

Flag	Meaning
VIDCF_QUALITY	Indicates the driver supports quality values.
VIDCF_CRUNCH	Indicates the driver supports crunching to a frame size.
VIDCF_TEMPORAL	Indicates the driver supports interframe compression.
VIDCF_DRAW	Indicates the driver supports drawing.
VIDCF_FASTTEMPORALC	Indicates the driver can do temporal compression and doesn't need the previous frame.
VIDCF_FASTTEMPORALD	Indicates the driver can do temporal decompression and doesn't need the previous frame.
VIDCF_COMPRESSFRAMES	Indicates the driver wants the "compress all frames" message.

dwVersion
Specifies the version number of the driver.

dwVersionICM
Specifies the version of the ICM supported by this driver; it should be set to ICVERSION.

szName[16]

Specifies the short name for the compressor. The name in the zero-terminated string should be suitable for use in list boxes.

szDescription[128]
Specifies a zero-terminated string containing the long name for the compressor.

szDriver[128]
Specifies a zero-terminated string for the module that contains the driver. Typically, a driver will not need to fill this out.

ICOPEN

```
typedef struct {
     DWORD dwSize;
     DWORD fccType;
     DWORD fccHandler;
     DWORD dwVersion;
     DWORD dwFlags;
     DWORD dwError;
     LPVOID pV1Reserved;
     LPVOID pV2Reserved;
     DWORD dnDevNode;
} ICOPEN;
```

The ICOPEN structure is sent to a video compression or decompression driver with the DRV_OPEN message.

Members

dwSize
Specifies the size of the structure.

fccType
Specifies a four-character code representing the type of stream being compressed or decompressed. For video streams, this should be 'vidc'.

fccHandler
Specifies a four-character code identifying a specific compressor.

dwVersion
Specifies the version of the installable driver interface used to open the driver.

dwFlags
Contains flags indicating why the driver is opened. The following flags are defined:

Flag	Meaning
ICMODE_COMPRESS	The driver is opened to compress data.
ICMODE_DECOMPRESS	The driver is opened to decompress data.
ICMODE_QUERY	The driver is opened for informational purposes rather than for actual compression.
ICMODE_DRAW	The device driver is opened to decompress data directly to hardware.

dwError
Specifies error return values.

pV1Reserved
Reserved.

pV2Reserved
Reserved.

dnDevNode
Device node for Plug and Play devices.

Comments

This structure is the same as that passed to video-capture drivers when they are opened. This lets a single installable driver function as either an installable compressor or a video-capture device.

By examining the **fccType** member of the ICOPEN structure, the driver can determine its function. For example, an **fccType** value of 'vidc' indicates that it is opened as an installable video compressor.

ICSETSTATUSPROC

```
typedef struct {
    DWORD   dwFlags;
    LPARAM  lParam;
    LONG (CALLBACK *Status) (LPARAM lParam, UINT message, LONG l);
} ICSETSTATUSPROC;
```

The ICSETSTATUSPROC structure is used with the ICM_SET_STATUS_PROC message.

Members

dwFlags
Specifies applicable flags. Set this to zero.

lParam
Specifies a constant passed to the status procedure when it is called.

Status
Specifies a pointer to the status function. This is NULL if status messages should not be sent.

Joystick Drivers

The joystick is an input device that provides absolute position information. It is an additional supported input device and not a replacement for the mouse. For this chapter, the term *joystick* refers to any absolute position device; for example, a light pen, a digitizing tablet, and a touch screen could all use the joystick driver interface.

All joystick function calls are routed through the WINMM module. WINMM loads the joystick driver and passes application requests to it. The joystick driver must handle the standard tasks handled by all installable drivers, as well as the following joystick-specific tasks:

- Returning the device's button configuration and movement range
- Returning position and button-press information
- Accepting calibration values and adjusting position information accordingly

The joystick driver interface enables a driver to handle one or two devices. Each device can have one to three axes and one to four buttons. The joystick interface accommodates both analog and digital devices.

Note: Providing drivers for non-interrupt, analog joysticks under Windows NT is difficult because the device polling required by such drivers is incompatible with NT's operation. The use of digital joysticks is recommended.

Writing a Joystick Driver

A joystick driver must include the standard **DriverProc** entry point function. This function handles the standard and joystick-specific messages sent by WINMM.

For joystick messages, the *hDriver* parameter contains a handle to the driver and the *dwDriverID* parameter contains the driver ID created by the driver.

Handling Joystick Driver Errors

The following list summarizes the joystick error codes defined in *mmsystem.h*:

MMSYSERR_NOERROR
JOYERR_PARMS
JOYERR_NOCANDO
JOYERR_UNPLUGGED

The DRV_OPEN Message

On the DRV_OPEN system message, the *lParam2* parameter specifies which device is being opened. This value is JOYSTICKID1 for the first device and JOYSTICKID2 for the second device.

The value that **DriverProc** returns for the DRV_OPEN message becomes the *dwDriverID* parameter for all subsequent messages sent to the driver. The driver can use this value for any purpose; for example, to identify a structure containing information about the client.

Joystick-Specific Messages Handled by DriverProc

The **DriverProc** function for a joystick driver handles the following joystick-specific messages:

JDD_GETDEVCAPS
JDD_GETNUMDEVS
JDD_GETPOS
JDD_GETPOSEX
JDD_SETCALIBRATION

Number of Devices

WINMM sends a JDD_GETNUMDEVS message to the driver to determine how many devices the driver supports. The *lParam1* and *lParam2* parameters are not used. In response to the JDD_GETNUMDEVS message, return the maximum number of devices supported by the driver.

The driver should return the number of supported devices, which is not necessarily the same as the number of connected devices. This allows a user to connect a device after the driver has been loaded. The joystick interface provides a JOYERR_UNPLUGGED error value that a driver should return when an application polls an unplugged device.

Device Capability Information

The JDD_GETDEVCAPS message requests information about the coordinate range and button configuration of a device. In response to this message, the driver fills in the JOYCAPS structure, which is described in the Win32 SDK.. (A pointer to it is passed to the *lParam1* parameter.) The JOYCAPS structure contains information that describes the coordinate range, button configuration, and manufacturer of the device.

The minimum and maximum position values should reflect a set of logical coordinates that is independent of minute differences between copies of a hardware device. For example, users of joystick services should always see a full range of logical coordinates returned from a calibrated joystick. Calibration values should not affect the minimum and maximum values returned by a driver.

If a hardware device does not support a given axis, return zero values for the minimum and maximum coordinate values for that axis. For example, a driver for a two-dimensional device would set the **wZmin** and **wZmax** members of the JOYCAPS structure to zero.

Accepting New Calibration Settings

The Joystick application in the Control Panel calculates calibration values for the device. Use the calibration values to convert the actual values returned by the hardware device to the logical values expected by the joystick interface. The driver establishes the logical value range in its response to the JDD_GETDEVCAPS message.

The calibration settings consist of base and delta values for each coordinate. The base value represents the lowest logical value the driver returns; the delta value is the multiplier to use when converting the actual value returned by the device to a logical value appropriate for the established value range.

Immediately after WINMM loads the joystick driver, it reads the calibration values from the registry and sends them to the driver with the JDD_SETCALIBRATION message. WINMM sends the JDD_SETCALIBRATION message with two JOYCALIBRATE structures (pointers to them are passed in the *lParam1* and *lParam2* parameters). The first structure contains the new calibration settings that the driver should adopt. The driver should fill the second structure with the previous calibration settings.

Calculation of Calibration Values

To calculate the base and delta calibration values, the Joystick application sets the base value to 0 and the delta values to 1. It then polls the joystick driver while the user holds the joystick at each corner of the device coordinate space. This produces the actual value range returned by the device for each coordinate.

After retrieving the coordinate ranges specified by the **joyGetDevCaps** function, the application uses the following formulas to calculate new base and delta values.

```
wDelta = (wTargetMax - wTargetMin) / (wActualMax - wActualMin)
wBase = wActualMin * wDelta - wTargetMin
```

In the following example, the joystick returns *x* coordinate values in the range 43 to 345, compared to a logical value range of 0 to 65535:

```
wDelta = (65535-0) / (345-43)
       = 216
wBase  = 43 * 216 - 0
       = 9288
```

Providing Position and Button-State Information

The JDD_GETPOS and JDD_GETPOSEX messages request device coordinate and button information. The driver fills in JOYINFO or JOYINFOEX structure, described in the Win32 SDK, and passes a pointer to it in the *lParam1* parameter.

The coordinate values should be within the range the driver established when responding to the JDD_GETDEVCAPS message. When calculating the coordinate values, use the base- and delta-calibration values passed to the driver with the JDD_SETCALIBRATION message.

Messages, Joystick Drivers

The following topics describe the messages used by Win32-based joystick drivers.

JDD_GETDEVCAPS

The JDD_GETDEVCAPS message is sent to get joystick device capability information.

Parameters

LPARAM *lParam1*
Specifies a pointer to a JOYCAPS structure, which is described in the Win32 SDK. The driver fills this structure with the capabilities of the device.

LPARAM *lParam2*
Specifies the size of the structure pointed to by *lParam1* in bytes.

Return Value

If JDD_GETDEVCAPS succeeds, it returns MMSYSERR_NOERROR. Otherwise, it returns JOYERR_PARMS.

Comments

Compare the structure size value passed to *lParam2* with the size of the JOYCAPS structure with which the driver was compiled. If *lParam2* is larger than the expected structure size, fill the remaining with zeros.

If *lParam2* is zero, return MMSYSERR_NOERROR without writing anything to the location specified by *lParam1*.

JDD_GETNUMDEVS

The JDD_GETNUMDEVS message requests the number of devices supported by the joystick driver.

Parameters

LPARAM *lParam1*
Not used.

LPARAM *lParam2*
Not used.

Return Value

JDD_GETNUMDEVS returns the maximum number of devices supported by the driver. This value might not match the number of physical devices actually connected to the computer.

JDD_GETPOS

The JDD_GETPOS message is sent to get the current joystick position and button-state information.

Parameters

LPARAM *lParam1*
Specifies a pointer to a JOYINFO structure, which is described in the Win32 SDK.

LPARAM *lParam2*
Not used.

Return Value

If the driver successfully retrieves the information, JDD_GETPOS returns MMSYSERR_NOERROR. If the joystick is unplugged, JDD-GETPOS returns JOYERR_UNPLUGGED.

Comments

The coordinate values returned in the JOYINFO structure should be mapped into the coordinate range identified by the driver. The minimum and maximum values for each axis are defined in the JOYCAPS structure, which the driver fills out in response to the JDD_GETDEVCAPS message.

JDD_GETPOSEX

The JDD_GETPOSEX message is sent to get the current joystick position and button-state information.

Parameters

LPARAM *lParam1*
Specifies a pointer to a JOYINFOEX structure, which is described in the Win32 SDK.

LPARAM *lParam2*
Not used.

Return Value

If the driver successfully retrieves the information, JDD_GETPOSEX returns MMSYSERR_NOERROR. If the joystick is unplugged, JDD_GETPOSEX returns JOYERR_UNPLUGGED.

Comments

The JDD_GETPOSEX message, along with the JOYINFOEX structure is used to obtain extended information not available with the JDD_GETPOS message and the JOYINFO structure.

JDD_SETCALIBRATION

The JDD_SETCALIBRATION message requests the driver to set calibration information. The driver uses the calibration values to map the physical device coordinates to the logical coordinate range established by the driver.

Parameters

LPARAM *lParam1*
Specifies a pointer to a JOYCALIBRATE structure containing the new calibration values to adopt.

LPARAM *lParam2*
Specifies a pointer to a JOYCALIBRATE structure. The driver should fill this structure with the existing calibration values.

Return Value

JDD_SETCALIBRATION returns MMSYSERR_NOERROR.

Comments

The calibration settings consist of base and delta values for each coordinate. The base value represents the lowest logical value the driver returns; the delta value is the multiplier to use when converting the actual value returned by the device to a logical value appropriate for the established value range.

Structures, Joystick Drivers

The following topics describe the structures used by Win32-based joystick drivers.

JOYCALIBRATE

```
typedef struct joycalibrate_tag {
    WORD wXbase;
    WORD wXdelta;
    WORD wYbase;
    WORD wYdelta;
    WORD wZbase;
    WORD wZdelta;
} JOYCALIBRATE;
```

The JOYCALIBRATE structure contains calibration values for the three axes of an absolute position device.

Members

wXbase
Specifies a base calibration value for the x axis.

wXdelta
Specifies a delta calibration value for the x axis.

wYbase

Specifies a base calibration value for the y axis.

wYdelta
Specifies a delta calibration value for the y axis.

wZbase
Specifies a base calibration value for the z axis.

wZdelta
Specifies a delta calibration value for the z axis.

Comments

Use the base and delta values returned in this structure to convert actual device values to the logical value range specified by the driver.

The base values represent the lowest logical value that the driver returns for a given axis. The delta values are multipliers the driver should use when mapping the value returned by the device into the value range established by the driver.

The following formula is used to calculate the delta values:

```
Delta = (LogicalMax - LogicalMin) / (DeviceMax - DeviceMin)
```

The **LogicalMax** and **LogicalMin** members represent the maximum and minimum logical coordinate values for the axis, as defined by the JOYCAPS structure, which is described in the Win32 SDK. **DeviceMax** and **DeviceMin** represent the actual values returned by the device.

The following formula is used to calculate the base values:

```
Base = (DeviceMin * Delta) - LogicalMin
```

The **DeviceMin** member represents the minimum value returned by the device, **Delta** represents the delta value calculated using the first formula, and **LogicalMin** represents the minimum value returned by the driver (as defined by JOYCAPS).