Programmer’s Guide

Microsoft Video for Windows Development Kit

For the Microsoft Windows Operating System

Microsoft Corporation

SAMSUNG 1007

Information in this document is subject to change without notice. Companies, names, and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Microsoft Corporation.

©1992, 1993 Microsoft Corporation. All rights reserved.

Microsoft, MS, and MS-DOS are registered trademarks, Windows and Visual Basic are trademarks of Microsoft Corporation in the USA and other countries.

U.S. Patent No. 4955066

IBM is a registered trademark of International Business Machines Corporation.
Intel is a registered trademark of Intel Corporation.
ToolBook is a registered trademark of Asymetrix Corp.

Printed in the United States of America.

Contents

Chapter 1 Introduction and Installation . . . . 1 - 1
Installing the Software . . . . 1 - 2
Documentation Overview . . . . 1 - 2

Chapter 2 Using the Installable Compression Manager . . . . 2 - 1
Video Compression and Decompression Header Files . . . . 2 - 1
ICM Architecture . . . . 2 - 1
Using ICM Services . . . . 2 - 2
Error Returned from the ICM Functions . . . . 2 - 2
Locating and Opening Compressors and Decompressors . . . . 2 - 2
Installing and Removing Compressors and Decompressors . . . . 2 - 5
Configuring Compressors and Decompressors . . . . 2 - 7
Getting Information about Compressors and Decompressors. . . . . 2 - 8
Compressing Image Data . . . . 2 - 11
Specifying the Input Format and Determining the Compression Format. . . . . 2 - 11
Initialization for the Compression Sequence . . . . 2 - 13
Compressing the Video. . . . . 2 - 13
Decompressing Image Data . . . . 2 - 15
Specifying the Input Format and Determining the Decompression Format . . . . 2 - 16
Initialization for the Decompression Sequence . . . . 2 - 17
Decompressing the Video. . . . . 2 - 18
Using Hardware Drawing Capabilities . . . . 2 - 19
Specifying the Input Format. . . . . 2 - 20
Preparing to Decompress Video . . . . 2 - 20
Decompressing the Video. . . . . 2 - 21
Controlling Drawing Parameters . . . . 2 - 23
Video Compression and Decompression Function Reference . . . . 2 - 23
Video Compression and Decompression Functions . . . . 2 - 26
Video Compressor and Decompressor Data Structure Reference . . . . 2 - 45

Chapter 3 Using the DrawDib Functions . . . . 3 - 1
Drawing With the DrawDib Functions. . . . . 3 - 1
Supporting Palettes for the DrawDib Functions . . . . 3 - 3
Manipulating Palettes. . . . . 3 - 3
Optimizing DrawDibDraw . . . . 3 - 4
Profiling the Display Characteristics . . . . 3 - 5
DrawDib Application Reference. . . . . 3 - 5
DrawDib Function Reference . . . . 3 - 5

Chapter 4 AVI Files. . . . . 4 - 1
AVI RIFF Form . . . . 4 - 1
Data Structures for AVI Files . . . . 4 - 3
The Main AVI Header LIST . . . . 4 - 3
The Stream Header (“strl”) Chunks . . . . 4 - 5
The LIST “movi” Chunk . . . . 4 - 7
The “idx1” Chunk. . . . . 4 - 8
Other Data Chunks . . . . 4 - 9
Special Information for Interleaved Files. . . . . 4 - 9
Using VidEdit With AVI Files . . . . 4 - 10
Example Code for Writing AVI Files. . . . . 4 - 10
An Outline for Writing AVI Files. . . . . 4 - 10
Creating the File and “AVI ” Chunk. . . . . 4 - 11
Creating the LIST “hdrl ” and “avih” Chunks . . . . 4 - 11
Creating the “strl”, “strh”, “strf”, and “strd” Chunks . . . . 4 - 12
Creating the LIST “movi” and “rec ” Chunks . . . . 4 - 12
Creating the “idx1” Chunk and Ascending From the “AVI ” Chunk . . . . 4 - 13
AVI RIFF File Reference. . . . . 4 - 14

Chapter 5 DIB Format Extensions for Microsoft Windows. . . . . . 5 - 1
Windows Compression Formats . . . . 5 - 1
Existing Formats . . . . 5 - 2
Extensions to the BI_RGB Format . . . . 5 - 2
Formats Using BI_BITFIELDS and Color Masks. . . . . 5 - 3
Custom Formats . . . . 5 - 4
Determining Display Driver Capabilities. . . . . 5 - 5
Inverted DIBs . . . . 5 - 6
Definition of the Flags and Escape . . . . 5 - 6

Chapter 6 Playing AVI Files With MCI . . . . 6 - 1
MCI Overview . . . . 6 - 1
Using the MCI Command Interface . . . . 6 - 2
Using the MCI String Interface. . . . . 6 - 3
Choosing the mciSendCommand or mciSendString Interface . . . . 6 - 5

Handling MCI Notification . . . . 6 - 6
Playing AVI files with MCI . . . . 6 - 7
Opening an AVI File . . . . 6 - 7
Setting up the Playback Window . . . . 6 - 8
Playing the AVI Sequence . . . . 6 - 10
Changing the Playback State . . . . 6 - 10
Obtaining Playback Information . . . . 6 - 12
Closing the AVI File. . . . . 6 - 12

Chapter 7 MCI Command Strings for MCIAVI. . . . . 7 - 1
About the MCIAVI.DRV Driver . . . . 7 - 1
Custom Commands and Flags for MCIAVI.DRV. . . . . 7 - 1
MCI Command Strings . . . . 7 - 2

Chapter 8 MCI Command Messages for MCIAVI . . . . 8 - 1
MCI Command Messages . . . . 8 - 1

Chapter 9 Video Capture Application Reference. . . . . 9 - 1
Video Capture Function Reference. . . . . 9 - 1
Video Capture Function Summary . . . . 9 - 1
Video Capture Function Alphabetic Reference . . . . 9 - 3
Video Capture Data Structure Reference . . . . 9 - 16
Video Capture Data Structure Alphabetic Reference . . . . 9 - 16

Chapter 10 Video Compression and Decompression Drivers. 1 0 - 1
Architecture of a Video Compression and Decompression Driver . . . . 1 0 - 1
The ICSAMPLE Example Driver . . . . 1 0 - 3
The Structure of a Video Compression and Decompression Driver . . . . 1 0 - 3
Video Compression and Decompression Header Files . . . . 1 0 - 3
Naming Video Compression and Decompression Drivers . . . . 1 0 - 3
SYSTEM.INI Entries for Video Compression and Decompression Drivers . . . . 1 0 - 4
The Module-Definition File. . . . . 1 0 - 5
The Module Name Line . . . . 1 0 - 5
The Installable Driver Interface . . . . 1 0 - 6
An Example DriverProc Entry-Point Function . . . . 1 0 - 6
Handling the DRV_OPEN and DRV_CLOSE Messages. . . . . 1 0 - 8
Compressor Configuration. . . . . 1 0 - 9
Configuration Messages Sent by the System . . . . 1 0 - 9
Messages for Configuring the Driver State. . . . . 1 0 - 10
Messages Used to Interrogate the Driver . . . . 1 0 - 11
Configuration Messages for Compression Quality . . . . 1 0 - 12
Configuration Messages for Key Frame Rate and Buffer Queue. . . . . . 1 0 - 13

Video Compression and Decompression Messages 10-14
About the AVI File Format 10-14
Identifying Compression Formats 10-16
Decompressing Video Data 10-17
Setting the Driver State. 10-18
Specifying the Input Format and Determining the Decompression Format. 10-18
Preparing to Decompress Video 10-19
Decompressing the Video. 10-19
Ending Decompression. 10-20
Other Messages Received During Decompression 10-20
Compressing Video Data 10-20
Obtaining the Driver State 10-21
Specifying the Input Format and Determining the Compression Format. 10-21
Initialization for the Compression Sequence 10-23
Compressing the Video. 10-23
Ending Compression. 10-24
Decompressing Directly to Video Hardware 10-25
Setting the Driver State. 10-25
Specifying the Input Format. 10-25
Preparing to Decompress Video 10-26
Decompressing the Video. 10-27
Ending Decompression. 10-28
Rendering the Data 10-28
Using Installable Compressors for Non-video Data 10-29
Testing Video Compression and Decompression Drivers 10-30
Video Compression and Decompression Driver Reference 10-30
Video Compression and Decompression Driver Message Reference . . 10-30
Video Compression and Decompression Driver Messages 10-33

Chapter 11 Video Capture Device Drivers. 11-1
Architecture of a Video Capture Driver 11-1
Video Capture Device Driver Channels 11-2
The Video Capture Application. 11-3
Sample Device Drivers. 11-3
The Structure of a Video Capture Device Driver 11-3
Combining Video Capture and Video Compression/Decompression Drivers. 11-4
Video Capture Header Files. 11-4
Naming Video Capture Device Drivers 11-4
SYSTEM.INI Entries for Video Capture Device Drivers 11-4

The Module-Definition File 11-6
The Module Name Line 11-6
The Module Description Line 11-6
Considerations for Interrupt-Driven Drivers 11-7
Fixing Code and Data Segments 11-7
Allocating and Using Memory 11-7
Calling Windows Functions at Interrupt Time 11-8
The Installable Driver Interface 11-8
An Example DriverProc Entry-Point Function 11-8
Handling the DRV_OPEN and DRV_CLOSE Messages 11-10
Handling the DRV_ENABLE and DRV_DISABLE Messages 11-14
Driver Configuration 11-15
Video Capture Driver Messages 11-15
Configuring the Channels of a Video Capture Driver 11-15
Setting and Obtaining Video Capture Format 11-17
Setting and Obtaining the Video Source and Destination Rectangles . . 11-19
Determining Channel Capabilities 11-21
Setting and Obtaining a Video Capture Palette 11-22
Obtaining the Device Driver Version 11-25
Transferring Data From the Frame Buffer 11-25
Streaming Video Capture 11-26
The Data Transfer Model For Streaming Video Input 11-26
Initializing the Data Stream 11-28
Preparing Data Buffers 11-28
Starting and Stopping Streaming 11-29
Ending Capture 11-29
Additional Stream Messages 11-30
Video Capture Device Driver Reference 11-31
Video Capture Device Driver Message Reference 11-31
Message Summary 11-32
Video Capture Device Driver Messages 11-33
Video Capture Device Driver Data Structure Reference 11-47

CHAPTER 1

Introduction and Installation

The Microsoft® Video for Windows™ Development Kit provides the resources you need to write applications that use the following services:

- Video Capture—These functions give your application easy access to video capture drivers. Your application can use these functions to obtain video sequences that you can use in AVI movies and in other applications using Video for Windows.
- Video Compression and Decompression—These functions give your application the ability to access video compressors and decompressors that use industry standard compression formats.
- AVI Playback with MCI—The MCI commands let your application use the AVI MCI driver to play AVI movies and manage the playback window.
- Extended Display Services—These services augment the standard video services to provide access to video decompressors, provide improved dithering of true-color images to 256 colors, and dither 8-bit images to 16-color VGA displays.
- Read and Write AVI Files—The AVI file examples and information let you develop routines to read and write AVI files.

This development kit also provides the resources needed by people developing video capture device drivers, and video compression and decompression drivers.

The software support supplied in this development kit includes:

- A collection of sample applications and drivers that use and provide Video for Windows services
- Header files defining the messages, data structures, and functions
- Documentation describing the features of the components of the development kit

Installing the Software

The distribution disks included with the development kit use a batch file to install the software. The following procedure describes the installation process.

Û **To install the Video for Windows Development Software:**

1. From the MS-DOS command prompt, change to the floppy drive you are installing from and run the INSTALL batch file. The INSTALL batch file has the following syntax:

```
INSTALL  C:\VFWDK
```

   Replace C:\VFWDK with disk and path for the destination of the files.

2. When installation is complete, change your INCLUDE and LIB environment variables to include the INC and LIB directories in your destination path. For example, if you used C:\VFWDK as the path during installation, you could use the following:

```
SET INCLUDE=[previous include line];C:\VFWDK\INC

SET LIB=[previous lib line];C:\VFWDK\LIB
```

   For these examples, replace [previous include line] and [previous lib line] with any existing paths for these statements.

3. You might also want to add the BIN directory to your PATH variable. The following example shows the template for modifying your PATH statement to include the Video for Windows BIN directory:

```
SET PATH=[previous path line];C:\VFWDK\BIN
```

   As in the previous examples, replace [previous path line] with any existing path statements.

Documentation Overview

The chapters included in this guide describe the development of applications accessing Video for Windows services and development of drivers providing video capture, and video compression and decompression services. This guide contains the following chapters:

- Chapter 1, “Introduction and Installation,” provides background information about the contents of this guide.
- Chapter 2, “Using the Installable Compression Manager,” describes how applications use the Installable Compression Manager (ICM) functions for compressing or decompressing video image data. The chapter also contains a reference to the ICM functions.
- Chapter 3, “Using the DrawDib Functions,” describes how applications can use the DrawDib functions to access ICM services, and obtain improved support of low-end VGA display adapters. These functions significantly improve the speed and quality of displaying such images on display adapters with limited capabilities.

- Chapter 4, “AVI Files,” describes the AVI RIFF file format. The information in this chapter applies to applications and drivers that use this file format.
- Chapter 5, “DIB Format Extensions for Microsoft Windows,” describes the DIB format extensions for Microsoft Windows that add new compression formats, custom compression formats, and inverted DIBs. Information in this chapter applies to both applications and video drivers.
- Chapter 6, “Playing AVI Files With MCI,” describes how to play AVI files using the MCI interface for Video for Windows.
- Chapter 7, “MCI Command Strings for MCIAVI,” describes the MCI command strings for the Microsoft MCI video driver (MCIAVI.DRV) that you can use with applications that support the MCI command-string interface.
- Chapter 8, “MCI Command Messages for MCIAVI,” describes the MCI command messages for the Microsoft MCI video driver (MCIAVI.DRV) that you can use with applications that support the MCI command-message interface.
- Chapter 9, “Video Capture Application Reference,” describes functions available for video capture.
- Chapter 10, “Video Compression and Decompression Drivers,” describes the installable driver interface used by video compressors and decompressors. This information applies to developers creating these types of drivers. This chapter also contains an alphabetical reference to the messages and data structures used to write video compression and decompression drivers.
- Chapter 11, “Video Capture Drivers,” describes the installable driver interface used by video capture drivers. This information applies to developers creating these types of drivers. This chapter also contains an alphabetical reference to the messages and data structures used to write video capture drivers.

CHAPTER 2

Using the Installable Compression Manager

The Installable Compression Manager (ICM) provides services for applications that want to compress or decompress video image data stored in AVI files. This chapter explains the programming techniques used to access these services. It covers the following topics:

- General information about the ICM and the Video for Windows architecture
- Information on how to compress and decompress video image data from your application
- An alphabetic reference to the ICM functions and data structures

Before reading this chapter, you should be familiar with the video services available with Windows. For information about these Windows topics, see the *Microsoft Windows Programmer's Reference*.

Video Compression and Decompression Header Files

The function prototypes, constants, flags, and data structures applications use to access the ICM services are defined in COMPMAN.H and COMPDDK.H.

ICM Architecture

The ICM is used by the Video for Windows editing tool (VidEdit) and the playback engine (MCIAVI) to handle compression and decompression of image data. ICM is the intermediary between the application and the actual video compression and decompression drivers. It is the video compression/decompression drivers that do the real work of compressing and decompressing individual frames of data.

This chapter covers the ICM and the functions a video editing or playback application uses to communicate with it. For information on the video compression and decompression drivers, see Chapter 10, *Video Compression and Decompression Drivers.*

As the application makes calls to the ICM to compress or decompress data, the ICM translates this to a message to be sent to the appropriate compressor or decompressor which does the work of compressing or decompressing the data. The ICM gets the return from the driver and then returns back to the application.

The ICMAPP sample application illustrates routines that compress data, decompress data, and display a dialog box. You might find the helper functions defined in ICM.C useful in developing your application.

Using ICM Services

In general, an application performs the following tasks to use ICM services:

- Locate, open, or install the appropriate compressor or decompressor
- Configures or obtains configuration information about the compressor or decompressor
- Uses a series of functions to compress, decompress, and (for decompressors with drawing capabilities) draw the data

These tasks are covered in the following sections. The sample application, ICMApp, shows how to use the ICM services to do all of the above functions to compress and decompress images.

Error Returned from the ICM Functions

For most ICM functions, return values of less than zero indicate an error. Your application should check these return values to see if the ICM function encounters an error. To keep the example fragments in this chapter simple, many of them do not check for errors. For more complete examples, see the ICMAPP and ICM examples included with the development kit.

Locating and Opening Compressors and Decompressors

To use ICM, an application must open a compressor or decompressor. If your application does not know about the compressors or decompressors installed on a system, it must find a suitable compressor to open. Once your application finishes with a compressor or decompressor, it closes it to free any resources used for compression or decompression. Your application can use the following functions for finding compressors and decompressors, and opening and closing them:

ICInfo
This function obtains information about compressor or decompressor.

ICOpen
This function opens a compressor or decompressor.

ICClose
This function closes a compressor or decompressor.

ICLocate
This function locates a specific type of compressor or decompressor.

If your application knows the compressor or decompressor it needs, it can open the compressor with the **ICOpen** function. Your application uses the handle returned by this function to identify the opened compressor or decompressor when it uses other ICM functions. The **ICOpen** function has the following syntax:

BOOL ICOpen(*fccType*, *fccHandler*, *wMode*)

The *fccType* and *fccHandler* parameters are four character codes used to describe the type and handler type for the compressor. Compressor and decompressors are identified by two four-character codes. Applications open a specific compressor or decompressor by using the four-character codes for the type and handler. The first four-character code describes the type of the compressor or decompressor. For video compressors and decompressors, this is always 'vidc'. The second four-character code identifies the specific compression handler type. For example, this value is 'msvc' for the Video 1 compressor. Your application can use NULL if it does not know this four-character code.

Note: In an AVI file, the stream header contains information about the stream type and the specific handler for that stream. For video streams, the stream type is 'vidc' and the handler type is the appropriate handler four-character code. As in the previous example, Video 1 compressed streams use 'msvc'.

The *wMode* parameter specifies flags passed to the compressor or decompressor. For **ICOpen**, these flags let the compressor or decompressor know why it is opened and they can prepare for subsequent operation. The following flags are defined:

ICMODE_COMPRESS
: Advises a compressor it is opened for compression.

ICMODE_DECOMPRESS
: Advises a decompressor it is opened for decompression.

ICMODE_DRAW
: Advises a decompressor it is opened to decompress an image and draw it directly to hardware.

ICMODE_QUERY
: Advises a compressor or decompressor it is opened to obtain information.

If your application does not know which compressors and deompressors are installed on a system, it can use **ICInfo** to enumerate them. This function has the following syntax:

BOOL ICInfo(*fccType*, *fccHandler*, *lpicinfo*)

The *fccType* parameter specifies a four-character code indicating the type of compressor or decompressor. To enumerate the compressors or decompressors, your application specifies an integer for *fccHandler*. Your application receives return information for integers between 0 and the number of installed compressors or decompressors of the type specified for *fccType*. The compressor or decompressor returns information about itself in a ICINFO data structure pointed to by *lpicinfo*. The **ICInfo** function returns TRUE if it can locate the specified compressor or decompressor.

The following example enumerates the compressors or decompressors in the system to find one that can handle the format of its images. (The example uses **ICCompressQuery** and **ICDecompressQuery** to determine if a compressor or decompressor supports the image format. The use of these functions is described in “Compressing Image Data” and “Decompressing Image Data.”)

```
for (i=0; ICInfo(p->fccType, i, &p->icinfo); i++)
{
    hic = ICOpen(p->icinfo.fccType, p->icinfo.fccHandler, ICMODE_QUERY);

    if (hic)
    {
        // skip this compressor if it can't handle the specifed format
        if (p->fccType == ICTYPE_VIDEO &&
            p->pvIn != NULL &&
            ICCompressQuery(hic, p->pvIn, NULL) != ICERR_OK &&
            ICDecompressQuery(hic, p->pvIn, NULL) != ICERR_OK)
        {
            ICClose(hic);
            continue;
        }

        // find out the compressor name.
        ICGetInfo(hic, &p->icinfo, sizeof(p->icinfo));

        // stuff it into the combo box
        n = ComboBox_AddString(hwndC,p->icinfo.szDescription);
        ComboBox_SetItemData(hwndC, n, hic);
    }
}
```

Applications can use **ICLocate** to find a compressor or decompressor of a specific type, and to obtain a handle to it for use in other ICM functions. The **ICLocate** function has the following syntax:

HIC ICLocate (*fccType*, *fccHandler*, *lpbiIn*, *lpbiOut*, *wFlags***)**

The *fccType* and *fccHandler* parameters are four-character codes used to describe the type and handler type for the compressor. Your application can specify NULL for *fccHandler* if it does not know the handler type or if it can use any handler type.

The *lpbiIn* parameter contains a pointer to a BITMAPINFOHEADER structure describing the input format the compressor or decompressor will handle. The *lpbiOut* parameter contains a pointer to a BITMAPINFOHEADER structure describing the output format desired by the application. If your application does not care what output format is returned by a compressor or decompressor, you can set *lpbiOut* to NULL. The *wFlags* parameter indicates the type of operation you want the driver to do: compress, decompress or directly decompress and draw.

For example, the following fragment tries to find a compressor that can compress an 8-bit per pixel bitmap:

```
BITMAPINFOHEADER    bih;
HIC                 hIC

// inialize the Bitmap structure
bih.biSize = sizeof(BITMAPINFOHEADER);
bih.biWidth = bih.biHeight = 0;
bih.biPlanes = 1;
bih.biCompression = BI_RGB;      // standard RGB bitmap
bih.biBitcount = 8;              // 8bpp format
bih.biSizeImage = 0;
bih.biXPelsPerMeter = bih.biYPelsPerMeter = 0;
bih.biClrUsed = bih.biClrImportant = 256;

hIC = ICLocate (ICTYPE_VIDEO, 0L,
        (LPBITMAPINFOHEADER)&bih, NULL, ICMODE_COMPRESS);
```

The following fragment tries to locate a specific compressor to compress the 8-bit RGB format to an 8-bit RLE format:

```
BITMAPINFOHEADER    bihIn, bihOut;
HIC                 hIC

// initialize the Bitmap structure
bihIn.biSize = bihOut.biSize = sizeof(BITMAPINFOHEADER);
bihIn.biWidth = bihIn.biHeight = bihOut.biWidth = bihOut.biHeight = 0;
bihIn.biPlanes = bihOut.biPlanes= 1;
bihIn.biCompression = BI_RGB;              //standard RGB bitmap for input
bihOut.biCompression = BI_RLE8;            // 8-bit RLE for output format
bihIn.biBitcount = bihOut.biBitCount = 8;  // 8bpp format
bihIn.biSizeImage = bihOut.biSizeImage = 0;
bihIn.biXPelsPerMeter = bih.biYPelsPerMeter =
    bihOut.biXPelsPerMeter = bihOut.biYPelsPerMeter = 0;
bihIn.biClrUsed = bih.biClrImportant =
    bihOut.biClrUsed = bihOut.biClrImportant = 256;

hIC = ICLocate (ICTYPE_VIDEO, 0L,
        (LPBITMAPINFOHEADER)&bihIn,
        (LPBITMAPINFOHEADER)&bihOut, ICMODE_COMPRESS);
```

Installing and Removing Compressors and Decompressors

There are three methods of installing compressors and decompressors. They might be installed during the set-up of Video for Windows or other software relating to Video for Windows. Users can also install compressors and decompressors with the Drivers option of the Control Panel. Applications can also install custom compressors and decompressors functions required for their operation. Most applications will not need to install or remove compressors or decompressors. Your application can use the following functions for installing and removing compressors and decompressors:

ICInstall
This function installs compressor or decompressor.

ICRemove
This function removes an installed compressor or decompressor.

Compressors and decompressor are usually installed by a setup program or by the user with the Drivers option of the Control Panel. An application might, however, install a compressor directly or install a function as a compressor. In these cases, the application uses **ICInstall** and **ICRemove**.

The **ICInstall** function creates a new entry in the SYSTEM.INI for a compressor or decompressor. After installation, the compressor or decompressor must still be opened. The **ICInstall** function has the following syntax:

BOOL ICInstall (*fccType*, *fccHandler*, *lParam*, *szDesc*, *wFlags***)**

The *fccType* and *fccHandler* parameters specify four-character codes describing the compressor type and handler type. Flags set in *wFlags* specify the meaning of *lParam*. The following flags are defined:

ICINSTALL_DRIVER
Indicates *lParam* points a null-terminated string containing the name of a compressor or decompressor.

ICINSTALL_FUNCTION
Indicates *lParam* points to a compressor function.

If you are installing a driver, *lParam* specifies the name of the driver. If you are installing a function as a compressor or decompressor, use *lParam* to specify a far pointer to your function. This function should be structured like the **DriverProc** entry point function used by installable drivers. For more information on the **DriverProc** entry point function, see Chapter 10, “Video Compression and Decompression Drivers.”

Use the *szDesc* parameter for a descriptive name for the compressor or decompressor. This information is not used and your application does not have to supply a name.

For example, the following fragment installs the ICSample driver:

```
result = ICInstall ( ICTYPE_VIDEO, mmioFOURCC('s','a','m','p'),
        (LPARAM)(LPSTR)"icsample.drv", "Sample Codec Driver", ICINSTALL_DRIVER)
```

The following fragment shows how an application would install a function as a compressor or decompressor.

```
// This function looks like a DriverProc entry point
LRESULT MyCodecFunction(DWORD dwID, HDRVR hDriver, UINT uiMessage,
                         LPARAM lParam1, LPARAM lParam2);

// This function installs the MyCodecFunction as a compressor
result = ICInstall ( ICTYPE_VIDEO, mmioFOURCC('s','a','m','p'),
                (LPARAM)(FARPROC)&MyCodecFunction, NULL, ICINSTALL_FUNCTION);
```

Usually the Drivers option of the Control Panel is used to remove a compressor or decompressor. Applications installing a function as a compressor or decompressor must remove the function before the application terminates so other applications do not try to use the function. Applications can use **ICRemove** to remove the function installed. The **ICRemove** function has the following syntax:

BOOL ICRemove (*fccType*, *fccHandler*, *wFlags***)**

The *fccType* and *fccHandler* parameters specify four-character codes describing the compressor type and handler type. The *wFlags* parameter is not used.

Configuring Compressors and Decompressors

Applications can configure the compressor or have the compressor display a dialog box to let the user to do the configuration. The following functions are available for these operations:

ICQueryConfigure
Determines if the compressor or decompressor supports a configuration dialog box.

ICConfigure
Displays the configuration dialog box of the compressor or decompressor.

ICGetStateSize
Determines the size of the state data for the compressor or decompressor.

ICGetState
Obtains the state data from the compressor or decompressor.

ICSetState
Sends the state data to the compressor or decompressor.

If practical, your application should let the user configure the compressor with the compressor's configuration dialog box. Typically, this makes your application independent of the compressor and you do not need to consider all the options available to a compressor.

Your application uses **ICQueryConfigure** to determine if a compressor can display a configuration dialog box. If the compressor can display a configuration dialog box, your application uses **ICConfigure** to display it. Both of these functions use the handle your application obtained when it located the compressor. The **ICConfigure** function also requires a handle to the parent window. Your application can use the following fragment to test for support of the configuration dialog box and display it:

```
 if (ICQueryConfigure(hIC)){

    // If compressor handles a configuration dialog box, display it
    // using our app window as the parent window.
    ICConfigure(hIC, hwndApp);

}
```

Your application might also directly get and set the state information for a compressor. If your application creates or modifies the state data, it must know the actual layout of the compressor data before restoring a compressor state. Alternatively, if your application obtains state data from a compressor and uses it to restore the state in a subsequent session, it must make sure that it only restores state information obtained from the compressor it is currently using. The following fragments show how to obtain the state data:

```
if (size > 0) {

    dwStateSize = ICGetStateSize(hIC);        // get size of buffer required
    h = GlobalAlloc(GHND, dwStateSize);       // allocate data buffer
    lpData = GlobalLock(h);                   // lock data buffer
    ICGetState(hIC, (LPVOID)lpData, dwStateSize); // get the state data

    // Store the state data as required
}
```

The following fragments show how to restore state data:

```
ICSetState(hIC, (LPVOID)lpData, dwStateSize); // set the new state data
GlobalUnlock(h);
```

Getting Information about Compressors and Decompressors

The following functions can be used to get information about a compressor or decompressor:

ICGetInfo
Obtains general information about the compressor or decompressor.

ICGetDefaultKeyFrameRate
Determines the default key frame rate of a compressor or decompressor.

ICGetDisplayFormat
Determines the 'best' format a compressor or decompressor has for displaying on the screen.

ICGetDefaultQuality
Determines the default quality value of a compressor or decompressor.

To obtain information about a compressor or decompressor, your application can use **ICGetInfo**. This function fills an ICINFO structure with information about the compressor or decompressor. Your application must allocate the memory for the ICINFO

structure and pass a pointer to it in **ICGetInfo**. The ICINFO structure has the following definition:

```
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 **dwSize** field contains the size of the ICINFO structure.

The **fccType** field contains the four-character code 'vidc' for image compression.

The **fccHandler** field identifies the compressor or decompressor with its four-character code.

The **dwVersion** field contains the version number of the compression driver.

The **dwVersionICM** field will contain the ICM version number supported by the compressor or decompressor. This is 1.0 (0x00010000) if the compressor or decompressor is written for Video for Windows 1.0.

The **szName** field contains the short name of the compressor or decompressor. The name is used in list-boxes for choosing compression methods.

The **szDescription** field contains the long description for the compressor or decompressor.

The **szDriver** field contains the actual module name that contains the compressor or decompressor.

The **dwFlags** field contains flags indicating capabilities of the compressor or decompressor. The following flags are defined:

VIDCF_QUALITY
Indicates the compressor or decompressor supports quality levels.

VIDCF_CRUNCH
Indicates a compressor supports compressing to an arbitrary frame size.

VIDCF_TEMPORAL
Indicates the compressor or decompressor supports inter-frame compression.

VIDCF_DRAW
Indicates decompressor can draw to hardware. (These decompressors support the ICDraw functions.)

VIDCF_FASTTEMPORAL
Indicates a compressor can do temporal compression but it doesn't need previous frame.

Unless your application is looking for a particular compressor or decompressor, the flags give your application the most useful information. Your application can check the flags to determine the capabilities of the compressor or decompressor. For example, if quality is supported, your application might enable a quality selection control in a compression dialog box.

The following fragment shows how to obtain the ICINFO information from a compressor or decompressor:

```
ICINFO      ICInfo;

ICGetInfo(hIC, &ICInfo, sizeof(ICInfo));
```

The **ICGetDefaultKeyFrameRate** and **ICGetDefaultQuality** functions let your application determine the default key frame rate and default quality value. Both of these functions require only the handle to the compressor or decompressor. The following fragment uses both functions to obtain the default values:

```
DWORD    dwKeyFrameRate, dwQuality;

dwKeyFrameRate = ICGetDefaultKeyFrameRate(hIC);
dwQuality = ICGetDefaultQuality(hIC);
```

Your application can use the following functions to display the about dialog box of a compressor or decompressor:

ICQueryAbout
Determines if a compressor or decompressor supports an about dialog box.

ICAbout
Displays the about dialog box of a compressor or decompressor.

The **ICQueryAbout** function lets your application determine if a compressor or decompressor can display the about dialog box. The **ICAbout** function actually displays the dialog box. The following examples uses these two functions:

```
if ( ICQueryAbout(hIC)){

    // If the compressor has an about dialog box, show it
    ICAbout(hIC, hwndApp);
}
```

Compressing Image Data

Your application uses a series of functions to coordinate compressing video data. The coordination involves the following activities:

- Specifying the input format and determining the compression format
- Preparing the compressor for compression
- Compressing the video
- Ending compression

Your application uses the following functions for these activities:

ICCompress
Compress data.

ICCompressBegin
Prepare compressor driver for compressing data.

ICCompressEnd
Tell the compressor driver to end compression.

ICCompressGetFormat
Determine the output format of a compressor.

ICCompressGetFormatSize
Get the size of the output format data.

ICCompressGetSize
Get the size of the compressed data.

ICCompressQuery
Deterimine if a compressor can compress a specific format.

Specifying the Input Format and Determining the Compression Format

When your application wants to compress data and the output format is not important, it must first locate a compressor that can handle the input format. When the output format is not important to your application, it can use **ICCompressGetFormat** to have the compressor suggest a format. If the compressor can produce multiple formats, it returns the format 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 **ICCompressGetFormat** function has the following syntax:

LRESULT ICCompressGetFormat(*hic*, *lpbiInput*, *lpbiOutput*)

The *hic* parameter specifies the compressor handle. The *lpbiInput* parameter specifies a far pointer to a BITMAPINFO structure indicating the format of the input data. The *lpbiOutput* parameter specifies a far pointer to a buffer used to return the output format suggested by the compressor. Your application can determine the size of the buffer needed for the buffer with **ICCompressGetFormatSize**.

Your application can use the output format data as the 'strf' chunk in the AVI RIFF file. This data starts out like a BITMAPINFOHEADER data structure. The compressor can include any additional information required to decompress the file after this information. A color table (if used) follows this information. If the compressor has format data following the BITMAPINFOHEADER structure, it updates the **biSize** field to specify the number of bytes used by the structure and additional data.

The following example fragment shows how an application can determine the output format that a compressor wants to use.

```
LPBITMAPINFOHEADER  lpbiIn, lpbiOut;

// *lpbiIn must be initialized to the input format

dwFormatSize = ICCompressGetFormatSize(hIC, lpbiIn); // get output buffer size
h = GlobalAlloc(GHND, dwFormatSize);            // allocate format buffer
lpbiOut = (LPBITMAPINFOHEADER)GlobalLock(h);    // lock format buffer
ICCompressGetFormat(hIC, lpbiIn, lpbiOut);      // fill the format information
```

If your application requires a specific output format, it should use **ICCompressQuery** to interrogate a compressor to determine if it supports the output format your application suggests. This function has the following syntax:

LRESULT ICCompressQuery(*hic*, *lpbiInput*, *lpbiOutput*)

The *hic* parameter specifies the compressor handle. Your application typically obtains this with **ICLocate** or **ICOpen.** The *lpbiInput* and *lpbiOutput* parameters specify far pointers to the data structures defining the input and output formats your application prefers. If the compressor can handle both formats it returns ICERR_OK. If it cannot handle the formats, it returns ICERR_BADFORMAT. If the compressor returns ICERR_BADFORMAT and the output format is critical to your application, your application will have to find an alternate compressor. If an alternate output format is satisfactory, your application might choose to use **ICCompressQuery** with the alternate formats to determine if the compressor can handle them. Or your application can use **ICCompressGetFormat** to have the compressor suggest the output format.

If your application specifies NULL for the *lpbiOutput* parameter of **ICCompressQuery**, the compressor will select the output format. Typically, your application specifies NULL when it only wants to know if the compressor can handle the input format. The output format information is not returned to your application.

The following fragment uses **ICCompressQuery** to determine if a compressor can handle both the input and output format:

```
LPBITMAPINFOHEADER  lpbiIn, lpbiOut;

// Both *lpbiIn & *lpbiOut must be initialized to the respective formats
if (ICCompressQuery(hIC, lpbiIn, lpbiOut) == ICERR_OK){

    // format is supported - use the compressor

}
```

Your application will also need the size of the data returned from the compressor after compression is complete. Use **ICCompressGetSize** to obtain the worst case (largest) buffer required by the compressor. The number of bytes returned should be used to allocate a buffer used for subsequent compression of images. The following example determines the buffer size and allocates a buffer of that size:

```
// find the worst-case buffer size
dwCompressBufferSize = ICCompressGetSize(hIC, lpbiIn, lpbiOut);

// allocate a buffer and get lpOutput to point to it
h = GlobalAlloc(GHND, dwCompressBufferSize);
lpOutput = (LPVOID)GlobalLock(h);
```

Initialization for the Compression Sequence

Once your application selects a compressor that handles the input and output formats it needs, it can prepare the compressor to start compressing data. The **ICCompressBegin** function initializes the compressor. This function requires the compressor handle and the input and output format. It returns ICERR_OK if it initializes properly for the specified formats. If the compressor cannot handle the formats, or if they are incorrect, it returns the error ICERR_BADFORMAT.

Compressing the Video

The **ICCompress** function does the actual compression. Your application must use this function repeatedly until all the frames are compressed. This function has the following syntax:

LRESULT ICCompress(*hic*, *dwFlags*, *lpbiOutput*, *lpData*, *lpbiInput*, *lpBits*, *lpckid*, *lpdwFlags*, *lFrameNum*, *dwFrameSize*, *dwQuality*, *lpbiPrev*, *lpPrev*)

The *hic* parameter specifies the handle to the compressor.

The *dwFlags* parameter specifies any applicable flags for the compression. Your application can use ICM_COMPRESS_KEYFRAME to have the compressor make the frame a key frame. (A key frame is one that does not require data from a previous frame for decompression.) When this flag is set, compressors use this image as the initial one in a sequence.

The *lpbiInput* and *lpBits* parameters specify far pointers to the data structure defining the input format and the location of the input buffer. Similarly, the *lpbiOutput* and *lpData* parameters specify far pointers to the data structure defining the output format and the location of the buffer for the output data. Your application must allocate the memory for

these buffers. When control returns to your application, it typically stores the compressed data in *lpbiOutput* and *lpData* in a subsequent operation. If your application needs to move the compressed data, it can find the size used for the data in the **biSizeImage** field in the BITMAPINFO structure specified for *lpbiOutput*.

The *lpckid* and *lpdwFlags* are used for AVI file data returned by the compressor. The *lpckid* specifies a far pointer to a **DWORD** used to hold a chunk ID for data in the AVI file. The *lpdwFlags s*pecifies a far pointer to a **DWORD** holding the return flags used in the AVI index. The compressor will set this flag to AVIIF_KEYFRAME to correspond to the ICM_COMPRESS_KEYFRAME flag. The AVIIF_KEYFRAME flag marks the key-frames in the AVI file. If your application creates AVI files, it should save the information returned for these parameters in the file.

The *lFrameNum* parameter specifies the frame number. Your application provides and, if necessary, increments this information. Compressors use this value to check if frames are being sent out of order when they are doing fast temporal compression. If your application has a frame recompressed, it should use the same frame number used when the frame was first compressed. If your application compresses a still frame image, it can specify zero for *lFrameNum*.

The *dwFrameSize* parameter specifies the requested frame size in bytes. If set to zero, the compressor chooses the frame size. If set to a non-zero value, the compressor tries to compress the frame to within the specified size. To obtain the size goal, the compressor might have sacrificed image quality (or made some other trade-off). Compressors recognize the frame size value only if they return the VIDCF_CRUNCH flag for **ICGetInfo**.

The *dwQuality* parameter specifies the requested quality value for the frame. Compressors support this only if they set the VIDCF_QUALITY flag for **ICGetInfo.**

The *lpbiPrev* and *lpPrev* parameters specify far pointers to the data structure defining the format and the location of the previous uncompressed image. Compressors use this data if they perform temporal compression (that is, they need the previous frame to compress the current frame). Compressors need this information only if they return the VIDCF_TEMPORAL flag. Compressors returning the VIDCF_FASTTEMPORAL flag can perform temporal compression without the previous frame.

The **ICCompress** function returns ICERR_OK if successful. Otherwise, it returns an error code.

After your application has compressed its data, it uses **ICCompressEnd** to notify the compressor that it has finished. To restart compression after using this function, your application must re-initialize the compressor with **ICCompressBegin.**

The following fragment compresses image data for use in an AVI file. It assumes the compressor does not support VIDCF_CRUNCH or VIDCF_TEMPORAL flags but it does support VIDCF_QUALITY.

```
DWORD    dwCkID;
DWORD    dwCompFlags;
DWORD    dwQuality;
LONG     lNumFrames, lFrameNum;

// assume dwNumFrames is initialized to the total number of frames
// assume dwQuality holds the proper quality value (0-10000)
// assume lpbiOut, lpOut, lpbiIn and lpIn are all initialized properly.

if (ICCompressBegin(hIC, lpbiIn, lpbiOut) == ICERR_OK){

    // If o.k. to start, compress each frame
    for ( lFrameNum = 0; lFrameNum < lNumFrames; lFrameNum++){

        if (ICCompress(hIC, 0, lpbiOut, lpOut, lpbiIn, lpIn,
            &dwCkID, &dwCompFlags, lFrameNum,
            0, dwQuality, NULL, NULL) == ICERR_OK){


            // Write compressed data the AVI file.
            .
            .
            .
            // set lpIn to be the next frame in the sequence

        } else {

            // handle compressor error

        }

    }


    ICCompressEnd(hIC);    // terminate compression

} else {


    // handle error


}
```

Decompressing Image Data

Similar to compressing data, your application uses a series of functions to control the decompressor used to decompress the video data. Decompressing data involves the following activities:

- Specifying the input format and determining the decompression format
- Preparing to decompress video
- Decompressing the video
- Ending decompression

Your application uses the following functions for these activities:

ICDecompress
Decompress data.

ICDecompressBegin
Prepare decompressor for decompressing data.

ICDecompressEnd
Tell decompressor to end decompression.

ICDecompressGetFormat
Determine the output format of a decompressor.

ICDecompressGetFormatSize
Get the size of the output data format.

ICDecompressGetPalette
Get the palette for the output format of a decompressor.

ICDecompressQuery
Determine if a decompressor can decompress a specific format.

Decompression is handled very much like compression except that the input format is a compressed format and the output is a displayable format. The input format for decompression is usually obtained from the video stream header in the AVI file. After determining the input format, your application can use **ICLocate** or **ICOpen** to find a decompressor that can handle it.

Specifying the Input Format and Determining the Decompression Format

Because your application allocates the memory required for decompression, it needs to determine the maximum memory the decompressor can require for the output format. The **ICDecompressGetFormatSize** function obtains the number of bytes the decompressor uses. This function has the following syntax:

DWORD ICDecompressGetFormatSize(*hic*, *lpbi***)**

The *hic* parameter specifies a handle to a decompressor. The *lpbi* specifies a far pointer to a BITMAPINFO structure indicating the format of the input data.

If your application wants the decompressor to suggest a format, it can use **ICDecompressGetFormat** to obtain the format. This function has the following syntax:

DWORD ICDecompressGetFormat(*hic*, *lpbiInput*, *lpbiOutput*)

Like **ICDecompressGetFormatSize**, the *hic* and *lpbiInput* parameters specify a handle to the decompressor and a far pointer to the structure indicating the format of the input data. The decompressor returns its suggested format in the BITMAPINFO structure pointed to by *lpbiInput.* Your application should check that the decompressor returns ICERR_OK for the return value before accessing the *lpbiOutput* information. If the decompressor cannot handle the input format, it returns ICERR_BADFORMAT. The following fragment shows how an application can use **ICDecompressGetFormat**:

```
LPBITMAPINFOHEADER  lpbiIn, lpbiOut;

// assume *lpbiIn points to the input (compressed) format

dwFormatSize = ICDecompressGetFormatSize(hIC, lpbiIn); // get output
                                                       // buffer size
h = GlobalAlloc(GHND, dwFormatSize);         // allocate format buffer
lpbiOut = (LPBITMAPINFOHEADER)GlobalLock(h); // lock format buffer
ICDecompressGetFormat(hIC, lpbiIn, lpbiOut); // fill the format information
```

If your application needs a specific output format, in can use **ICDecompressQuery** to determine if the decompressor can handle both the input and output format. This function uses the same parameters as **ICDecompressGetFormat** except that your application sets *lpbiOutput* to point at the structure defining the desired output format. If your application is just determining if the decompressor can handle the input format, it can specify NULL for *lpbiOutput*. The following fragment shows how an application can use this function:

```
LPBITMAPINFOHEADER  lpbiIn, lpbiOut;

// assume both *lpbiIn & *lpbiOut are initialized to the respective formats
if (ICDecompressQuery( hIC, lpbiIn, lpbiOut) == ICERR_OK){

    // format is supported - use the decompressor

}
```

If your application creates its own format, it must also obtain a palette for the bitmap. (Most applications use standard formats and do not need to obtain a palette.) Your application can obtain the palette with **ICDecompressGetPalette**. This function has the following syntax:

DWORD ICDecompressGetPalette(*hic*, *lpbiInput*, *lpbiOutput*)

Like the other functions, *hic* and *lpbiInput* specify a handle to a decompressor and point to a **BITMAPINFO** structure indicating the format of the input data. The *lpbiOutput* parameter points to a **BITMAPINFO** structure used to return the color table. The space reserved for the color table must have an entire 256 color palette table reserved at the end of the structure. The following fragment shows how to get the palette information:

```
ICDecompressGetPalette(hIC, lpbiIn, lpbiOut);

// move up to the palette
lpPalette = (LPBYTE)lpbiOut + lpbi->biSize;
```

Initialization for the Decompression Sequence

Once your application selects a decompressor that handles the input and output formats it needs, it can prepare the decompressor to start decompressing data. The **ICDecompressBegin** function initializes the compressor. This function requires the compressor handle and the input and output format. It returns ICERR_OK if it initializes properly for the specified formats. If the compressor cannot handle the formats, or if they are incorrect, it returns the error ICERR_BADFORMAT.

Decompressing the Video

The **ICDecompress** function does the actual decompression. Your application must use this function repeatedly until all the frames are decompressed. This function has the following syntax:

DWORD ICDecompress(*hic*, *dwFlags*, *lpbiFormat*, *lpData*, *lpbi*, *lpBits*)

The *hic* parameter specifies the handle to the decompressor.

The *dwFlags* parameter specifies any applicable flags for decompression. If your video presentation is starting to lag other components (such as audio), your application can use ICM_DECOMPRESS_HURRYUP to have the decompressor decompress at a faster rate. To speed up decompression, a decompressor might extract only the information it needs to decompress the next frame and not fully decompress the current frame. Thus, when your application uses this flag, it should not try to draw the decompressed data.

The *lpbiFormat* and *lpData* parameters specify far pointers to the data structure defining the input format and the location of the input buffer. Similarly, the *lpbi* and *lpBits* parameters specify far pointers to the data structure defining the output format and the location of the buffer for the output data. Your application must allocate the memory for these buffers. When control returns to your application, it will use the information in *lpbi* and *lpBits* for subsequent processing of the decompressed data.

The **ICDecompress** function returns ICERR_OK if successful. Otherwise, it returns an error code.

After your application has decompressed its data, it uses **ICDecompressEnd** to notify the decompressor that it has finished. To restart decompression after using this function, your application must re-initialize the decompressor with **ICDecompressBegin**. The following fragment shows how an application can initialize a decompressor, decompress a frame sequence, and terminate decompression:

```
LPBITMAPINFOHEADER  lbpiIn, lpbiOut;
LPVOID              lpIn, lpOut;
LONG                lNumFrames, lFrameNum;

// assume lpbiIn and lpbiOut are initialized to the input and output format
// and lpIn and lpOut are pointing to the data buffers.

if (ICDecompressBegin(hIC, lpbiIn, lpbiOut) == ICERR_OK){

    for (lFrameNum = 0; lFrameNum < lNumFrames, lFrameNum++){

        if (ICDecompress(hIC, 0, lpbiIn, lpIn, lpbiOut, lpOut) == ICERR_OK){

            // frame decompressed OK so we can process it as required
```

```
            } else {

                // handle decompression error

            }
        }

        ICDecompressEnd(hIC);

    } else {

        // handle error for decompression initialization

    }
```

Using Hardware Drawing Capabilities

Some decompressors have the ability to draw directly to video hardware as they decompress video frames. These decompressors return the VIDCF_DRAW flag in response to **ICGetInfo**. When using this type of decompressor, your application does not have to handle the decompressed data. It lets the decompressor retain the decompressed data for drawing. The following functions are used to for decompressing and drawing with decompressors that have drawing capabilities:

ICDrawBegin
This function prepares a decompressor for drawing.

ICDrawEnd
This function stops a decompressor's drawing operations.

ICDrawFlush
This function flushes the buffers in the decompressor.

ICDrawQuery
This function determines if the decompressor can render data in a specific format.

ICDrawStart
This function starts the internal clock a decompressor uses for drawing.

ICDrawStop
This function stops the internal clock a decompressor uses for drawing.

ICGetBuffersWanted
This function determines the pre-buffering requirements of a compressor.

If your application uses a decompressor with drawing capabilities, it must handle the following activities:

- find a decompressor that can decompress and draw a bitmap with the input format specified
- prepare for decompression
- decompress data
- terminate the decompression process

Specifying the Input Format

Since your application no longer needs to draw the final data, it does not need to be concerned with the output format. However, it must make sure the decompressor can draw the input format. Your application can use **ICDrawQuery** to determine if a decompressor can handle the input format. While this function can determine if a deompressor can handle the format, it does not determine if the a decompressor has all the capablities needed to draw a bitmap. If your application is uncertain if the decompressor can render the bitmap as required, use this function with **ICDrawBegin** The following section describes **ICDrawBegin**. The following fragment shows how to check the input format with **ICDrawQuery**:

```
// lpbiIn points to BITMAPINFOHEADER structure indicating the input format

if (ICDrawQuery(hIC, lpbiIn) == ICERR_OK){

    // decompressor recognizes the input format

} else {

    // we need a different decompressor

}
```

Preparing to Decompress Video

The **ICDrawBegin** function initializes a decompressor and it informs the decompressor about the destination of drawing. The **ICDrawBegin** function has the following syntax:

DWORD ICDrawBegin(*hIC*, *dwFlags*, *hPal*, *hwnd*, *hdc*, *xDst*, *yDst*, *dxDst*, *dyDst*, *lpbi*, *xSrc*, *yScr*, *dxSrc*, *dySrc*, *dwRate*, *dwScale***)**

The *hIC* parameter contains the handle to the decompressor. The *dwFlags* parameter specifies any applicable flags. The following flags are defined for this function:

ICDRAW_QUERY
Use to determine if the decompressor can handle the decompression. The decompressor does not draw when this flag is used.

ICDRAW_FULLSCREEN
Indicates that the decompressor will draw to the full screen rather than to a window.

ICDRAW_HDC
Indicates that the decompressor will use a window and display context for drawing.

The *hPal* parameter specifies a handle to the palette used for drawing. Decompressor ignore this information and your application can set it to null.

The *hwnd* and *hdc* parameters define the window and display context used for drawing. Your application must set these values if it uses the ICDRAW_HDC flag.

The *xDst*, *yDst*, *dxDst* and *dyDst* parameters define the destination rectangle used for drawing. Specify the destination rectangle values relative to the current window or display context. Your application should set these parameters to the desired destination rectangle

if it uses ICDRAW_HDC. It can set them to zero if it uses the ICDRAW_FULLSCREEN flag.

The *xSrc*, *ySrc*, *dxSrc*, and *dySrc* parameters specify the source rectangle used for clipping the frames of the image. The decompressor will stretch the rectangle specified as the source into the rectangle specified by the destination when drawing.

The *lpbi* parameter should contain a pointer to the BITMAPINFO structure for the input format. Your application uses the *dwRate* and *dwScale* parameters to specify the decompression rate. The integer value specified for *dwRate* divided by the integer value specified for *dwScale* defines the play rate in frames per second. This is used by the decompressor when it is responsible for timing of frames on playback.

The following fragment shows the initialization sequence to have the decompressor draw full screen:

```
// assume lpbiIn has the input format, dwRate has the data rate
if (ICDrawBegin(hIC, ICDRAW_QUERY|ICDRAW_FULLSCREEN, NULL, NULL,
        NULL, 0, 0, 0, 0, lpbiIn, 0, 0, 0, 0, dwRate, dwScale) == ICERR_OK){

    // decompressor supports this drawing so set up to draw.
    ICDrawBegin(hIC, ICDRAW_FULLSCREEN, hPal, NULL, NULL, 0, 0, 0, 0, lpbiIn,
        0, 0, lbpi->biWidth, lpbi->biHeight, dwRate, dwScale);

    // we're ready to start decompressing and drawing frames now

    // drawing done so terminate
    ICDrawEnd(hIC);
} else {

    // do drawing myself

}
```

Some decompressors buffer the compressed data for more efficient operation. Your application can use **ICGetBuffersWanted** to determine how many data frames it should send to the decompressor before it has the decompressor draw them.

Decompressing the Video

The **ICDraw** function has the decompressor do the actual decompression. This function has the following syntax:

LRESULT ICDraw(*hIC*, *dwFlags*, *lpFormat*, *lpData*, *cbData, lTime***)**

The *hIC* is the handle to the decompressor. The *dwFlags* are flags set by the application and used by the decompressor. These flags can be:

ICDRAW_HURRYUP
Tell the decompressor to decompress at a faster rate.

ICDRAW_UPDATE
Tell the decompressor to update the screen based on the last data received. In this case the *lpData* parameter should be NULL.

The *lpFormat* parameter specifies a pointer to the format of the input data. The *lpData* parameter contains the actual data to be decompressed and later drawn.

The *cbData* parameter specifies the number of bytes in *lpData*.

The *lTime* parameter specifies the time to draw this frame. The decompressor divides this integer by the time scale specified with **ICDrawBegin** obtain the actual time. Time for the ICDraw functions is relative to **ICDrawStart**. (That is, **ICDrawStart** sets the clock to zero.) For example, if your applications specifies 1000 for the time scale and 75 for *lTime*, the decompressor draws the frame 75 milliseconds into the sequence.

The decompressor starts decompressing data in response to **ICDraw**, however, it does not start drawing data until your application calls **ICDrawStart**. (Your application should not use **ICDrawStart** until it has sent the number of frames the decompressor returned for **ICGetBuffersWanted.)** When your application uses **ICDrawStart**, the decompressor begins to draw the frames at the rate specified by *dwRate* specified with the **ICDrawBegin**. Drawing continues until your application stops the decompressor drawing clock with **ICDrawStop**. The following fragment uses the **ICDraw** functions:

```
DWORD       dwNumBuffers;

// find out how many buffers need filling before drawing starts
ICGetBuffersWanted(hIC, &dwNumBuffers);

for (dw = 0; dw < dwNumBuffers; dw++){

    ICDraw(hIC, 0, lpFormat, lpData, cbData, dw); // fill the pipeline

    // Point lpFormat and lpData to next format and data buffer

}

ICDrawStart(hIC); // start the clock

while (fPlaying){

    ICDraw(hIC, 0, lpFormat, lpData, chData, dw); // fill more buffers

    // Point lpFormat and lpData to next format and data buffer, update dw

}

ICDrawStop(hIC);  // when done stop drawing and decompressing
ICDrawFlush(hIC); // flush any existing buffers
ICDrawEnd(hIC);   // end decompression
```

Controlling Drawing Parameters

The following functions provide more control over decompressors that can draw the decompressed data:

ICDrawGetTime
This function obtains the current time from the decompressor.

ICDrawRealize
This function has the decompressor realize the palette used for drawing.

ICDrawSetTime
This function sets the value of the internal clock for the decompressor.

ICDrawWindow
This function has the decompressor redraw the window.

If your application wants to monitor or change the clock of the decompressor, it can use **ICDrawGetTime** and **ICDrawSetTime**. If your application wants to change the playback position while the decompressor is drawing, it can use **ICDrawWindow** for repositioning the decompressor. If the playback window gets a palette realize message, your application must call **ICDrawRealize** to have the decompressor realize the palette again for playback.

Video Compression and Decompression Application Reference

This section is an alphabetic reference to the functions and data structures provided by ICM for applications using video compression and decompression services. There are separate sections for functions and data structures. The COMPMAN.H and COMPDDK.H files define the functions and data structures.

Video Compression and Decompression Function Reference

Applications use the following functions for compressing video data:

ICCompress
This function compresses a single video image.

ICCompressBegin
This functions prepares a compressor for compressing data.

ICCompressEnd
This function tells a compressor to end compression.

ICCompressGetFormat
This function determines the output format of a compressor.

ICCompressGetFormatSize
This function obtains the size of the output format data.

ICCompressGetSize
This function obtains the size of the compressed data.

ICCompressQuery
This function determines if a compressor can compress a specific format.

Applications use the following functions for decompressing video data:

ICDecompress
The function decompresses a single video frame.

ICDecompressBegin
This functions prepares a decompressor for decompressing data.

ICDecompressEnd
This function tells a decompressor to end decompression.

ICDecompressGetFormat
This function determines the output format of a decompressor.

ICDecompressGetFormatSize
This function obtains the size (in bytes) of the output format data.

ICDecompressGetPalette
This function obtains the palette for the output format of a decompression.

ICDecompressQuery
This function determines if a decompressor can decompress data with a specific format.

ICDecompressQuery
This function determines if a decompressor can render a specific format.

Applications use the following functions to control video decompressors that draw directly to the display:

ICDraw
This function decompresses an image for drawing.

ICDrawBegin
This function is used to start decompressing data directly to the screen.

ICDrawEnd
This function tells a decompressor to end drawing.

ICDrawFlush
This function flushes the image buffers used for drawing.

ICDrawGetTime
This function obtains the current value of the internal clock if the decompressor is handling the timing of drawing frames.

ICDrawRealize
This function tells decompressor to realize its palette used while drawing.

ICDrawSetTime
This function sets the value of the internal clock if the decompressor is handling the timing of drawing frames.

ICDrawStart
This function tells a decompressor to start its internal clock for the timing of drawing frames.

ICDrawStop
This function tells a decompressor to stop its internal clock used for the timing of drawing frames.

ICDrawWindow
This function tells a decompressor to redraw the window when it has moved.

ICGetBuffersWanted
This function obtains information about the pre-buffering needed by a compressor.

Applications use the following functions to obtain information about a compressor or decompressor and display its dialog boxes:

ICQueryAbout
This function determines if a compressor supports an about dialog box.

ICAbout
This function instructs a compressor to display its about dialog box.

ICQueryConfigure
This functions determines if a compressor supports a configuration dialog box.

ICConfigure
This function displays the configuration dialog box of the specified compressor.

ICGetInfo
This function asks a compressor for information about itself.

ICInfo
This function returns information about specific installed compressors, or it enumerates the compressors installed.

ICGetDefaultKeyFrameRate
This function obtains the default key frame rate value.

ICGetDisplayFormat
Given an input format and optionally an open compressor handle, finds the "best" format it can for displaying on the screen.

Applications use the following functions to set and retrieve the state information of a compressor or decompressor:

ICGetState
This function gets the state of a compressor.

ICGetStateSize
This function gets the size of the state data used by a compressor.

ICSetState
This function sets the state of a compressor.

Applications use the following functions to locate, open, and close a compressor or decompressor:

ICOpen
This function opens a compressor or decompressor.

ICClose
This function closes a compressor or decompressor.

ICLocate
This function finds a compressor with specific attributes.

Applications use the following functions to install and remove a compressor or decompressor and send messages directly to it:

ICInstall
This function installs a new compressor.

ICRemove
This function removes a compressor function installed **ICInstalled**.

ICSendMessage
This function sends a message to a compressor.

Video Compression and Decompression Functions

This section contains an alphabetical list of the functions applications can use for compressing and decompressing video data. The functions are identified with the prefix IC.

ICAbout

Syntax **LRESULT ICAbout**(*hic*, *hwnd*)

This function instructs a compressor or decompressor to display its about dialog box.

Parameters HIC *hic*
Specifies the handle to the installable compressor.

HWND *hwnd*
Specifies a handle to the parent window.

Return Value Returns ICERR_OK after the compressor or decompressor displays the about dialog box. It returns ICERR_UNSUPPORTED if it does not support an about dialog box.

See Also ICQueryAbout

ICClose

Syntax

LRESULT ICClose(*hic*)

This function closes a compressor or decompressor.

Parameters

HIC *hic*
Specifies a handle to a compressor or decompressor.

Return Value

Returns ICERR_OK if successful, otherwise it returns an error number.

See Also

ICLocate ICOpen

ICCompress

Syntax

LRESULT ICCompress(*hic*, *dwFlags*, *lpbiOutput*, *lpData*, *lpbiInput*, *lpBits*, *lpckid*, *lpdwFlags*, *lFrameNum*, *dwFrameSize*, *dwQuality*, *lpbiPrev*, *lpPrev*)

This function compresses a single video image.

Parameters

HIC *hic*
Specifies the handle of the compressor to use.

DWORD *dwFlags*
Specifies applicable flags for the compression. The following flag is defined:

ICM_COMPRESS_KEYFRAME
Indicates that the compressor should make this frame a key frame.

LPBITMAPINFOHEADER *lpbiOutput*
Specifies a far pointer to a **BITMAPINFO** structure holding the output format.

LPVOID *lpData*
Specifies a far pointer to output data buffer.

LPBITMAPINFOHEADER *lpbiInput*
Specifies a far pointer to a **BITMAPINFO** structure containing the input format.

LPVOID *lpBits*
Specifies a far pointer to the input data buffer.

LPDWORD *lpckid*
Specifies a far pointer to a **DWORD** used to hold a chunk ID for data in the AVI file.

LPDWORD *lpdwFlags*
Specifies a far pointer to a **DWORD** holding the return flags used in the AVI index. The following flag is defined:

AVIIF_KEYFRAME
Indicates this frame is a key-frame.

LONG *lFrameNum*
Specifies the frame number.

DWORD *dwFrameSize*
Specifies the requested frame size in bytes. If set to zero, the compressor chooses the frame size.

DWORD *dwQuality*
Specifies the requested quality value for the frame.

LPBITMAPINFOHEADER *lpbiPrev*
Specifies a far pointer to a **BITMAPINFO** structure holding the previous frame's format.

LPVOID *lpPrev*
Specifies a far pointer to the previous frame's data buffer.

Return Value

This function returns ICERR_OK if successful. Otherwise, it returns an error code.

Comments

The *lpData* buffer should be large enough to hold a compressed frame. You can obtain the size of this buffer by calling **ICCompressGetSize**.

Set the *dwFrameSize* parameter to a requested frame size only if the compressor returns the VIDCF_CRUNCH flag in response to **ICGetInfo**. Without this flag, set this parameter to zero.

Set the *dwQuality* parameter to a quality value only if the compressor returns the VIDCF_QUALITY flag in response to **ICGetInfo**. Without this flag, set this parameter to zero.

See Also

ICCompressBegin, ICCompressEnd, ICCompressGetSize, ICGetInfo

ICCompressBegin

Syntax

LRESULT ICCompressBegin(*hic*, *lpbiInput*, *lpbiOutput*)

This function prepares a compressor for compressing data.

Parameters

HIC *hic*
Specifies a handle to a compressor.

LPBITMAPINFOHEADER *lpbiInput*
Specifies a far pointer to a **BITMAPINFO** structure holding the input format.

LPBITMAPINFOHEADER *lpbiOutput*
Specifies a far pointer to a **BITMAPINFO** structure holding the output format.

Return Value

Returns ICERR_OK if the specified compression is supported, otherwise it returns ICERR_BADFORMAT if either the input or output format is not supported.

See Also

ICCompress, ICCompressEnd, ICDecompressBegin, ICDrawBegin

ICCompressEnd

Syntax

LRESULT ICCompressEnd(*hic*)

This function ends compression by a compressor.

Parameters

HIC *hic*
Specifies a handle to the compressor.

Return Value

Returns ICERR_OK if successful, otherwise it returns an error number.

See Also ICCompressBegin, ICCompress, ICDecompressEnd, ICDrawEnd

ICCompressGetFormat

Syntax **LRESULT ICCompressGetFormat**(*hic*, *lpbiInput*, *lpbiOutput*)

This function determines the output format of a compressor.

Parameters

HIC *hic*
The compressor handle.

LPBITMAPINFOHEADER *lpbiInput*
Specifies a far pointer to a **BITMAPINFO** structure indicating the format of the input data.

LPBITMAPINFOHEADER *lpbiOutput*
Specifies a far pointer to a **BITMAPINFO** structure used to return the output format.

Return Value Returns the size of the output format.

See Also ICCompressGetFormatSize

ICCompressGetFormatSize

Syntax **LRESULT ICCompressGetFormatSize**(*hic*, *lpbi*)

This function obtains the size of the output format data.

Parameters

HIC *hic*
Specifies a handle to a compressor.

LPBITMAPINFOHEADER *lpbi*
Specifies a far pointer to a **BITMAPINFO** structure indicating the format of the input data.

Return Value Returns the size of the output data format structure.

Comments Use this function to determine the size of the output format buffer you need to allocate when using **ICCompressGetFormat**.

See Also ICCompressGetFormat

ICCompressGetSize

Syntax **LRESULT ICCompressGetSize**(*hic*, *lpbiInput*, *lpbiOutput*)

This function obtains the size of the compressed data.

Parameters

HIC *hic*
Specifies a handle to a compressor.

LPBITMAPINFOHEADER *lpbiInput*
Specifies a far pointer to a **BITMAPINFO** structure indicating the format of the input data.

LPBITMAPINFOHEADER *lpbiOutput*
Specifies a far pointer to a **BITMAPINFO** structure indicating the format of the output format.

Return Value Returns the maximum number of bytes a single compressed frame can occupy.

See Also ICCompressQuery, ICCompressGetFormat

ICCompressQuery

Syntax **LRESULT ICCompressQuery**(*hic*, *lpbiInput*, *lpbiOutput*)

This function determines if a compressor can compress a specific format.

Parameters

HIC *hic*
Specifies the compressor handle.

LPBITMAPINFOHEADER *lpbiInput*
Specifies a far pointer to a **BITMAPINFO** structure indicating the input data.

LPBITMAPINFOHEADER *lpbiOutput*
Specifies a far pointer to a **BITMAPINFO** structure indicating the format of the data output. If NULL, then any output format is acceptable.

Return Value Returns ICERR_OK if the compression is supported, otherwise it returns ICERR_BADFORMAT.

See Also ICCompressGetFormat

ICConfigure

Syntax **LRESULT ICConfigure**(*hic*, *hwnd*)

This function displays the configuration dialog box of a compressor.

Parameters

HIC *hic*
Specifies a handle to the compressor.

HWND *hwnd*
Specifies a handle to the parent window.

Return Value Returns ICERR_OK after the configuration dialog box is displayed. It returns ICERR_UNSUPPORTED if the compressor does not support a a configuration dialog box.

See Also ICQueryConfigure

ICDecompress

Syntax

LRESULT ICDecompress(*hic*, *dwFlags*, *lpbiFormat*, *lpData*, *lpbi*, *lpBits*)

The function decompresses a single video frame.

Parameters

HIC *hic*
Specifies a handle to the decompressor to use.

DWORD *dwFlags*
Specifies any applicable flags for decompression. The following flag is defined:

ICDECOMPRESS_HURRYUP
Indicates the decompressor should try to decompress at a faster rate. When an application uses this flag, it should not draw the decompressed data.

LPBITMAPINFOHEADER *lpbiFormat*
Specifies a far pointer to a **BITMAPINFO** structure containing the format of the compressed data.

LPVOID *lpData*
Specifies a far pointer to the input data.

LPBITMAPINFOHEADER *lpbi*
Specifies a far pointer to a **BITMAPINFO** structure containing the output format.

LPVOID *lpBits*
Specifies a far pointer to a data buffer for the decompressed data.

Return Value

Returns ICERR_OK on success, otherwise it returns an error code.

Comments

The *lpBits* parameter should point to a buffer large enough to hold the decompressed data. Applications can obtain the size of this buffer with **ICDecompressGetSize**.

See Also

ICDecompressBegin, ICDecompressEnd, ICDecompressGetSize

ICDecompressBegin

Syntax

LRESULT ICDecompressBegin(*hic*, *lpbiInput*, *lpbiOutput*)

This function prepares a decompressor for decompressing data.

Parameters

HIC *hic*
Specifies a handle to a decompressor.

LPBITMAPINFOHEADER *lpbiInput*
Specifies a far pointer to a **BITMAPINFO** structure indicating the format of the input data.

LPBITMAPINFOHEADER *lpbiOutput*
Specifies a far pointer to a **BITMAPINFO** structure indicating the format of the output data.

Return Value

Returns ICERR_OK if the specified decompression is supported, otherwise it returns ICERR_BADFORMAT if either the input or output format is not supported.

See Also

ICDecompress, ICDecompressEnd, ICCompressBegin, ICDrawBegin

ICDecompressEnd

Syntax **LRESULT ICDecompressEnd**(*hic*)

This function tells a decompressor to end decompression.

Parameters HIC *hic*

Specifies a handle to a decompressor.

Return Value Returns ICERR_OK if successful, otherwise it returns an error number.

See Also ICDecompressBegin, ICDecompress, ICCompressEnd, ICDrawEnd

ICDecompressGetFormat

Syntax **LRESULT ICDecompressGetFormat**(*hic*, *lpbiInput*, *lpbiOutput*)

This function determines the output format of a decompressor.

Parameters HIC *hic*

Specifies a handle to a decompressor.

LPBITMAPINFOHEADER *lpbiInput*

Specifies a far pointer to a **BITMAPINFO** structure indicating the format of the input data.

LPBITMAPINFOHEADER *lpbiOutput*

Specifies a far pointer to a **BITMAPINFO** structure used to return the format of the output data.

Return Value Returns the size (in bytes) of the output format.

See Also ICDecompressGetFormatSize

ICDecompressGetFormatSize

Syntax **LRESULT ICDecompressGetFormatSize**(*hic*, *lpbi*)

This function obtains the size (in bytes) of the output format data.

Parameters HIC *hic*

Specifies a handle to a decompressor.

LPBITMAPINFOHEADER *lpbi*

Specifies a far pointer to a **BITMAPINFO** structure indicating the format of the input data.

Return Value Returns the size of the output data format structure.

Comments Use this function before **ICDecompressGetFormat** to find the size needed to allocate the output format buffer.

See Also ICDecompressGetFormat

ICDecompressGetPalette

Syntax

LRESULT ICDecompressGetPalette(*hic*, *lpbiInput*, *lpbiOutput*)

This function obtains the palette for the output format of a decompression.

Parameters

HIC *hic*
Specifies a handle to a decompressor.

LPBITMAPINFOHEADER *lpbiInput*
Specifies a far pointer to a **BITMAPINFO** structure indicating the format of the input data.

LPBITMAPINFOHEADER *lpbiOutput*
Specifies a far pointer to a **BITMAPINFO** structure used to return the color table. The space reserved for the color table must be at least 256 bytes.

Return Value

Returns the size of the output format or an error code.

See Also

ICDecompressGetFormat

ICDecompressQuery

Syntax

LRESULT ICDecompressQuery(*hic*, *lpbiInput*, *lpbiOutput*)

This function determines if a decompressor can decompress data with a specific format.

Parameters

HIC *hic*
Specifies a handle to to a decompressor.

LPBITMAPINFOHEADER *lpbiInput*
Specifies a far pointer to a **BITMAPINFO** structure indicating the format of the input data.

LPBITMAPINFOHEADER *lpbiOutput*
Specifies a far pointer to a **BITMAPINFO** structure indicating the format of the output data. If NULL, any output format is acceptable.

Return Value

Returns ICERR_OK if the decompression is supported, otherwise it returns ICERR_BADFORMAT.

See Also

ICDecompressGetFormat

ICDecompressQuery

Syntax

LRESULT ICDecompressQuery(*hic*, *lpbiInput*)

This function determines if a decompressor can render a specific format.

Parameters

HIC *hic*
Specifies a handle to a decompressor.

LPBITMAPINFOHEADER *lpbiInput*
Specifies a far pointer to a **BITMAPINFO** structure indicating the format of the input data.

Return Value Returns ICERR_OK if the decompression is supported, otherwise it returns ICERR_BADFORMAT.

See Also ICDecompressQuery

ICDraw

Syntax **LRESULT ICDraw**(*hic*, *dwFlags*, *lpFormat*, *lpData*, *cbData*, *lTime*)

This function decompress an image for drawing.

Parameters HIC *hic*
Specifies a handle to a decompressor.

DWORD *dwFlags*
Specifies any flags for the decompression. The following flags are defined:

ICDRAW_HURRYUP
Indicates the decompressor should try to increase its decompression rate.

ICDRAW_UPDATE
Tells the decompressor to update the screen based on data previously received. Set *lpData* to NULL when this flag is used.

LPVOID *lpFormat*
Specifies a far pointer to a **BITMAPINFOHEADER** structure containing the input format of the data.

LPVOID *lpData*
Specifies a far pointer to the actual input data.

DWORD *cbData*
Specifies the size of the input data (in bytes).

LONG *lTime*
Specifies the time to draw this frame based on the time scale sent with **ICDrawBegin**.

Return Value Returns ICERR_OK on success, otherwise an appropriate error number.

Comments This function is used to decompress the image data for drawing by the decompressor. Actual drawing of frames does not occur until **ICDrawStart** is called. The application should be sure to pre-buffer the required number of frames before drawing is started (you can obtain this value with **ICGetBuffersRequired**).

See Also ICDrawBegin, ICDrawEnd, ICDrawStart, ICDrawStop, ICGetBuffersRequired

ICDrawBegin

Syntax

LRESULT ICDrawBegin(*hic*, *dwFlags*, *hpal*, *hwnd*, *hdc*, *xDst*, *yDst*, *dxDst*, *dyDst*, *lpbi*, *xSrc*, *ySrc*, *dxSrc*, *dySrc*, *dwRate*, *dwScale*)

This function starts decompressing data directly to the screen.

Parameters

HIC *hic*
Specifies a handle to the decompressor to use.

DWORD *dwFlags*
Specifies flags for the decompression. The following flags are defined:

ICDRAW_QUERY
Determines if the decompressor can handle the decompression. The decompressor does not actually decompress the data.

ICDRAW_FULLSCREEN
Tells the decompressor to draw the decompressed data on the full screen.

ICDRAW_HDC
Indicates the decompressor should use the window handle specified by *hwnd* and the display context handle specified by *hdc* for drawing the decompressed data.

HPALETTE *hpal*
Specifies a handle to the palette used for drawing.

HWND *hwnd*
Specifies a handle for the window used for drawing.

HDC *hdc*
Specifies the display context used for drawing.

int *xDst*
Specifies the x-position of the upper-right corner of the destination rectangle.

int *yDst*
Specifies the y-position of the upper-right corner of the destination rectangle.

int *dxDst*
Specifies the width of the destination rectangle.

int *dyDst*
Specifies the height of the destination rectangle.

LPBITMAPINFOHEADER *lpbi*
Specifies a far pointer to a **BITMAPINFO** structure containing the format of the input data to be decompressed.

int *xSrc*
Specifies the x-position of the upper-right corner of the source rectangle.

int *ySrc*
Specifies the y-position of the upper-right corner of the source rectangle.

int *dxSrc*
Specifies the width of the source rectangle.

int *dySrc*
Specifies the height of the source rectangle.

DWORD *dwRate*
Specifies the data rate. The data rate in frames per second equals *dwRate* divided by *dwScale*.

DWORD *dwScale*
Specifies the data rate.

Return Value Returns ICERR_OK if it can handle the decompression, otherwise it returns ICERR_UNSUPPORTED.

Comments Decompressors use the *hwnd* and *hdc* parameters only if an application sets ICDRAW_HDC flag in *dwFlags*. It will ignore these parameters if an application sets the ICDRAW_FULLSCREEN flag. When an application uses the ICDRAW_FULLSCREEN flag, it should set *hwnd* and *hdc* to NULL.

The destination rectangle is specified only if ICDRAW_HDC is used. If an application sets the ICDRAW_FULLSCREEN flag, the destination rectangle is ignored and its parameters can be set to zero.

The source rectangle is relative to the full video frame. The portion of the video frame specified by the source rectangle will be stretched to fit in the destination rectangle.

See Also ICDraw, ICDrawEnd

ICDrawEnd

Syntax **LRESULT ICDrawEnd**(*hic*)

This function tells a decompressor to end drawing.

Parameters HIC *hic*
Specifies a handle to a compressor.

Return Value Returns ICERR_OK if successful, otherwise it returns an error number.

See Also ICDrawBegin, ICDraw, ICCompressEnd, ICDecompressEnd

ICDrawFlush

Syntax **LRESULT ICDrawFlush**(*hic*)

Flush the image buffers used for drawing.

Parameters HIC *hic*
The compressor handle.

Return Value Returns ICERR_OK on success.

See Also ICDraw, ICDrawBegin, ICDrawEnd, ICDrawStart, ICDrawStop

ICDrawGetTime

Syntax

LRESULT ICDrawGetTime(*hic*, *lplTime*)

This function obtains the current value of the internal clock if the decompressor is handling the timing of drawing frames.

Parameters

HIC *hic*
Specifies a handle to a decompressor.

LPLONG *lplTime*
Specifies a far pointer to a LONG buffer used to return the current time value. The value will be in samples (frames for video).

Return Value

Returns ICERR_OK if successful.

See Also

ICDrawStart, ICDrawStop, ICDrawSetTime

ICDrawRealize

Syntax

LRESULT ICDrawRealize(*hic*, *hdc*, *fBackground*)

This function tells a decompressor to realize its palette used while drawing.

Parameters

HIC *hic*
Specifies a handle to a decompressor.

HDC *hdc*
Specifies the display context used to realize the palette.

BOOL *fBackground*
Specifies TRUE if the palette is to be realized in the background. It specifies FALSE if it is to be realized in in the foreground.

Return Value

Returns ICERR_OK if palette is realized. The compressor returns ICERR_UNSUPPORTED if it doesn't support this function.

See Also

ICDrawBegin

ICDrawSetTime

Syntax

LRESULT ICDrawSetTime(*hic*, *lpTime*)

This function sets the value of the internal clock if the installable decompressor is handling the timing of drawing frames.

Parameters

HIC *hic*
Specifies a handle to a decompressor.

LPLONG *lpTime*
Specifies the current time that the compressor should be rendering. This value should be in samples. For video, this corresponds to frames.

Return Value

Returns ICERR_OK if successful.

See Also ICDrawStart, ICDrawStop, ICDrawGetTime

ICDrawStart

Syntax **void ICDrawStart**(*hic*)

This function tells a decompressor to start its internal clock for the timing of drawing frames.

Parameters HIC *hic*
Specifies a handle to a compressor.

Comments This function should only be used with hardware decompressors that do their own asynchronous decompression, timing and drawing.

See Also ICDraw, ICDrawStop, ICDrawBegin, ICDrawEnd

ICDrawStop

Syntax **void ICDrawStop**(*hic*)

This function tells a decompressor to stop its internal clock used for the timing of drawing frames.

Parameters HIC *hic*
Specifies a handle to a decompressor.

Comments This function should only be used with hardware decompressors that do their own asynchronous decompression, timing and drawing.

See Also ICDraw, ICDrawStart, ICDrawBegin, ICDrawEnd

ICDrawWindow

Syntax **LRESULT ICDrawWindow**(*hic*, *prc*)

This function has a decompressor redraw the window when is has moved.

Parameters HIC *hic*
Specifies a handle to a decompressor.

LPRECT *prc*
Specifies a pointer to the destination rectangle. The destination rectangle is specified in screen coordinates.

Return Value Returns ICERR_OK if successful.

Comments This function is only supported by hardware which does its own asynchronous decompression, timing and drawing. The rectangle is set to empty if the window is totally hidden by other windows.

ICGetBuffersWanted

Syntax **LRESULT ICGetBuffersWanted**(*hic*, *lpdwBuffers*)

This function obtains information about the pre-buffering needed by a decompressor.

Parameters HIC *hic*
Specifies a handle to a decompressor.

LPDWORD *lpdwBuffers*
Specifies a far pointer to a DWORD used to return the number of samples the decompressor needs to get in advance of when they will be rendered.

Return Value Returns ICERR_OK if successful, otherwise it returns ICERR_UNSUPPORTED.

Comments This function is used only with a decompressor that uses hardware to render data and wants to ensure that hardware pipelines remain full.

ICGetDefaultKeyFrameRate

Syntax **LRESULT ICGetDefaultKeyFrameRate**(*hic*)

This function obtains the default key frame rate value.

Parameters HIC *hic*
Specifies a handle to a compressor.

Return Value Returns the default key frame rate.

ICGetDisplayFormat

Syntax **HIC ICGetDisplayFormat**(*hic*, *lpbiIn*, *lpbiOut*, *BitDepth*, *dx*, *dy*)

This function returns the "best" format available for display a compressed image. The function will also open a compressor if a handle to an open compressor is not specified.

Parameters HIC *hic*
Specifies the decompressor that should be used. If this is NULL, an appropriate compressor will be opened and returned.

LPBITMAPINFOHEADER *lpbiIn*
Specifies a pointer to **BITMAPINFOHEADER** structure containing the compressed format.

LPBITMAPINFOHEADER *lpbiOut*
Specifies a pointer to a buffer used to return the decompressed format. The buffer should be large enough for a **BITMAPINFOHEADER** and 256 color entries.

int *BitDepth*
If non-zero, specifies the preferred bit depth.

int *dx*
If non-zero, specifies the width to which the image is to be stretched.

int *dy*
If non-zero, specifies the height to which the image is to be stretched.

Return Value Returns a handle to a decompressor if successful, otherwise, it returns zero.

ICGetInfo

Syntax **LRESULT ICGetInfo**(*hic*, *lpicinfo*, *cb*)

This function obtains information about a compressor or decompressor.

Parameters HIC *hic*
Specifies a handle to a compressor or decompressor.

ICINFO FAR * *lpicinfo*
Specifies a far pointer to **ICINFO** structure used to return information about the compressor or decompressor.

DWORD *cb*
Specifies the size of the structure pointed to by *lpicinfo*.

Return Value Returns zero if successful.

ICGetState

Syntax **void ICGetState**(*hic*, *pv*, *cb*)

This function gets the state of a compressor or decompressor.

Parameters HIC *hic*
Specifies a handle to the compressor or decompressor.

LPVOID *pv*
Specifies a pointer to a buffer used to return the state data.

DWORD *cb*
Specifies the byte count for state buffer.

Comments Use **ICGetStateSize** before calling **ICGetState** to determine the size of buffer to allocate for the call.

See Also ICGetStateSize, ICSetState

ICGetStateSize

Syntax **LRESULT ICGetStateSize**(*hic*)

This function gets the size of the state data used by a compressor or decompressor.

Parameters HIC *hic*
Specifies a handle to the compressor or decompressor.

Return Value Returns the number of byte used by the state data.

Comments Use this function to get the size of the state data for the **ICGetState** and **ICSetState** buffers.

See Also ICGetState, ICSetState

ICInfo

Syntax **BOOL ICInfo**(*fccType*, *fccHandler*, *lpicinfo*)

This function returns information about specific installed compressors and decompressors, or it enumerates the compressors installed.

Parameters

DWORD *fccType*
Specifies a four-character code indicating the type of compressor or decompressor.

DWORD *fccHandler*
Specifies a four-character code identifying a specific compressor or decompressor, or a number between 0 and the number of installed compressors of the type specified by *fccType*.

ICINFO FAR * *lpicinfo*
Specifies a far pointer to a **ICINFO** structure used to return information about the compressor.

Return Value Returns a compressor or decompressor handle if successful, otherwise, it returns zero.

ICInstall

Syntax **BOOL ICInstall**(*fccType*, *fccHandler*, *lParam*, *szDesc*, *wFlags*)

This function installs a new compressor.

Parameters

DWORD *fccType*
Specifies a four-character code indicating the type of data used by the compressor. Use 'vidc' for video compressors.

DWORD *fccHandler*
Specifies a four-character code identifying a specific compressor.

LPARAM *lParam*
Identifies what to install. The meaning of this parameter is defined by the flags set for *wFlags*.

LPSTR *szDesc*
Specifies a pointer to a null-terminated string describing the installed compressor.

UINT *wFlags*
Specifies flags defining the contents of *lParam*. The following flags are defined:

ICINSTALL_DRIVER
Indicates *lParam* is a pointer to a null-terminated string containing the name of the compressor to install.

ICINSTALL_FUNCTION
Indicates *lParam* is a far pointer to an installable compressor function. This function should be structured like the **DriverProc** entry point function used by compressors and decompressors.

Return Value Returns a handle to a compressor or decompressor.

See Also ICRemove

ICLocate

Syntax **HIC ICLocate**(*fccType*, *fccHandler*, *lpbiIn*, *lpbiOut*, *wFlags*)

This function finds a compressor or decompressor that can handle images with the formats specified, or it finds a decompressor that can decompress an image with a specified format directly to hardware.

Parameters

DWORD *fccType*
Specifies the type of compressor or decompressor the application wants to open. For video, this is ICTYPE_VIDEO.

DWORD *fccHandler*
Specifies a single preferred handler of the given type that should be tried first. Typically, this comes from the stream header in an AVI file.

LPBITMAPINFOHEADER *lpbiIn*
Specifies a pointer to **BITMAPINFOHEADER** structure defining the input format. A compressor handle will not be returned unless it can handle this format.

LPBITMAPINFOHEADER *lpbiOut*
Specifies zero or a pointer to **BITMAPINFOHEADER** structure defining an optional decompressed format. If *lpbiOut* is nonzero, a compressor handle will not be returned unless it can create this output format.

WORD *wFlags*
Specifies any flags defining the use of the compressor or decompressor. This parameter must contain one of the following values:

ICMODE_COMPRESS
Indicates the compressor should be able to compress an image with a format defined by *lpbiIn* to the format defined by *lpbiOut*.

ICMODE_DECOMPRESS
Indicates the decompressor should be able to decompress an image with a format defined by *lpbiIn* to the format defined by *lpbiOut*.

ICMODE_DRAW
Indicates the decompressor should be able to decompress an image with a format defined by *lpbiIn* and draw it directly to hardware.

Return Value Returns a handle to a compressor or decompressor if successful, otherwise it returns zero.

ICOpen

Syntax **HIC ICOpen**(*fccType*, *fccHandler*, *wMode*)

This function opens a compressor or decompressor.

Parameters

DWORD *fccType*
Specifies the type of compressor or decompressor the application wants to open. For video, this is ICTYPE_VIDEO.

DWORD *fccHandler*
Specifies a single preferred handler of the given type that should be tried first. Typically, this comes from the stream header in an AVI file.

UINT *wMode*
Specifies any flags defining the use of the compressor or decompressor. This parameter can contain the following values:

ICMODE_COMPRESS
Advises a compressor it is opened for compression.

ICMODE_DECOMPRESS
Advises a decompressor it is opened for decompression.

ICMODE_DRAW
Advises a decompressor it is opened to decompress an image and draw it directly to hardware.

ICMODE_QUERY
Advises a compressor or decompressor it is opened to obtain information.

Return Value Returns a handle to a compressor or decompressor if successful, otherwise it returns zero.

See Also ICClose ICLocate

ICQueryAbout

Syntax **BOOL ICQueryAbout**(*hic*)

This function determines if a compressor or decompressor supports an about dialog box.

Parameters

HIC *hic*
Specifies the handle to the installable compressor.

Return Value Returns TRUE if the installable compressor supports an about dialog box, otherwise it returns FALSE.

See Also ICAbout

ICQueryConfigure

Syntax **BOOL ICQueryConfigure**(*hic*)

This function determines if a compressor or decompressor supports a configuration dialog box.

Parameters HIC *hic*

Specifies a handle to the compressor or decompressor.

Return Value Returns TRUE if compressor supports a configuration dialog box, otherwise it returns FALSE.

See Also ICConfigure

ICRemove

Syntax **BOOL ICRemove**(*fccType*, *fccHandler*, *wFlags*)

This function removes a compressor function installed with **ICInstall**.

Parameters DWORD *fccType*

Specifies a four-character code indicating the type of data used by the compressor. Use 'vidc' for video compressors.

DWORD *fccHandler*

Specifies a four-character code identifying a specific compressor.

UINT *wFlags*

Not used.

Return Value Returns TRUE if successful.

See Also ICInstall

ICSendMessage

Syntax **LRESULT ICSendMessage**(*hic*, *wMsg*, *dw1*, *dw2*)

This function sends a message to a compressor or decompressor.

Parameters HIC *hic*

Specifies the handle of the compressor or decompressor to receive the message.

UINT *wMsg*

Specifies the message to send.

DWORD *dw1*

Specifies additional message-specific information.

DWORD *dw2*

Specifies additional message-specific information.

Return Value Returns a message-specific result.

ICSetState

Syntax **void ICSetState**(*hic*, *pv*, *cb*)

This function sets the state of a compressor or decompressor.

Parameters HIC *hic*
Specifies a handle to the compressor or decompressor.

LPVOID *pv*
Specifies a pointer to the state data to set.

DWORD *cb*
Specifies the size (in bytes) of the buffer containing the state data.

See Also ICGetState, ICGetStateSize

Video Compressor and Decompressor Data Structure Reference

This section lists data structures used by video compressors and decompressors. The data structures are presented in alphabetical order. The structure definition is given, followed by a description of each field.

ICINFO

The **ICINFO** structure is filled by a video compressor when it receives the **ICM_GETINFO** message.

```
typedef struct {
    DWORD  dwSize;
    DWORD  fccType;
    DWORD  fccHandler;
    DWORD  dwFlags;
    DWORD  dwVersion;
    DWORD  dwVersionICM;
    char  szName[16];
    char  szDescription[128];
    char  szDriver[128];
} ICINFO;
```

Fields The **ICINFO** structure has the following fields:

dwSize
Should be set to the size of an **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 (**ICINFO.fccHandler** == 'vidc'):

VIDCF_QUALITY
The compressor supports quality.

VIDCF_CRUNCH
The compressor supports crunching to a frame size.

VIDCF_TEMPORAL
The compressor supports inter-frame compression.

VIDCF_DRAW
The compressor supports drawing.

VIDCF_FASTTEMPORAL
The compressor can do temporal compression and doesn't need the previous frame.

dwVersion
Specifies the version number of the compressor.

dwVersionICM
Specifies the version of the ICM supported by this compressor; it should be set to 1.0 (0x00010000)

szName[16]
Specifies the short name for the compressor. The null-terminated name should be suitable for use in list boxes.

szDescription[128]
Specifies a null-terminated string containing the long name for the compressor.

szDriver[128]
Specifies a null-terminated string for the module that contains the compressor.

CHAPTER 3

Using the DrawDib Functions

The DrawDib functions provide much of the functionality of **StretchDIBits** and adds ICM support, improved stretching capabilities, and improved support of low-end display adapters. If a display driver cannot stretch an image, the DrawDib functions can stretch it more efficiently then **StretchDIBits**. The DrawDib functions also efficiently dither true-color images to 256 colors, and dither 8-bit images on 16-color VGA displays. These functions significantly improve the speed and quality of displaying such images on display adapters with limited capabilities.

This chapter discusses the following topics:

- Drawing with the DrawDib functions
- Optimizing **DrawDibDraw**
- Profiling the display characteristics

The DrawDib functions and constants are defined in DRAWDIB.H. The ICMAPP sample application shows how your application can use these functions.

Drawing With the DrawDib Functions

Your application uses the following functions with 8, 16, and 24 bit images to access the basic DrawDib services:

DrawDibOpen
This function opens a DrawDib context for drawing.

DrawDibClose
This function closes a DrawDib context and cleans up.

DrawDibDraw
This function draws a device independent bitmap to the screen.

While your application can use **DrawDibDraw** as an almost one-to-one replacement for **StretchDIBits**, it has the following limitations:

- the DIB must have the DIB_RGB_COLORS format
- your application must use the simplest transfer mode, SRC_COPY

That is, your application cannot use **DrawDibDraw** to, say, XOR a picture with the screen.

Prior to using **DrawDibDraw**, your application must initialize the DrawDib library and let it allocate the memory it needs by calling **DrawDibOpen**. This function returns a DrawDib context handle your application uses for other DrawDib functions. If Windows cannot create a DrawDib context, **DrawDibOpen** returns NULL.

Your application can use **DrawDibOpen** to create multiple DrawDib contexts. This lets your application work with several DrawDib contexts that have different characteristics.

The **DrawDibDraw** function draws a device independent bitmap to the screen. This function replaces **StretchDIBits** and it provides transparent support of installable compressors for decompressing the bitmaps. This function has the following syntax:

BOOL DrawDibDraw(*hdd*, *hdc*, *xDst*, *yDst*, *dxDst*, *dyDst*,
lpbi, *lpBits*, *xSrc*, *ySrc*, *dxSrc*, *dySrc*, *wFlags*)

The *hdd* parameter specifies a handle to a DrawDib context. The *hdc* parameter specifies a handle to the display context.

Your application uses the *lpbi* and *lpBits* parameters to specify information about the bitmap drawn. The *lpbi* parameter points to the **BITMAPINFOHEADER** structure for the bitmap and the *lpBits* parameter points to the buffer containing the bitmap.

Your application specifies the source rectangle and destination rectangle with two sets of parameters. The *xDst*, *yDst*, *dxDst*, and *dyDst* parameters specify the X and Y coordinates of the origin of the destination rectangle, and its width and height. The *xSrc*, *ySrc*, *dxSrc*, and *dySrc* parameters specify the X and Y coordinates of the origin of the source rectangle, and its width and height.

The *wFlags* parameter specifies any applicable flags for drawing. The following flags are defined:

DDF_UPDATE
: Indicates the last bitmap is to be redrawn. Your application can specify NULL for *lpBits* when it uses this flag.

DDF_SAME_HDC
: Uses the handle to the display context specified previously. When used, DrawDib skips preparing the display context and assumes the correct palette has already been realized (possibly by **DrawDibRealize**). Your application should not use this flag until it uses a DrawDib function that specifies the display context. Your application must still specify a handle for *hdc* when it uses this flag.

DDF_SAME_DRAW
: Uses the drawing parameters previously specified for this function. Use this flag only if *lpbi*, *dxDst*, *dyDst*, *dxSrc*, and *dySrc* have not changed since last using **DrawDibDraw**.

DDF_DONTDRAW
: Indicates the frame is only to be decompressed and not drawn. Your application can use the DDF_UPDATE flag to draw the image later with **DrawDibDraw**.

DDF_ANIMATE
Allows palette animation. If this flag is present, the palette DrawDib creates will have the PC_RESERVED flag set for as many entries as possible, and your application can subsequently animate the palette with **DrawDibChangePalette**. The **DrawDibDraw** function passes this flag to **DrawDibBegin** implicitly.

When your application has finished using the DrawDib context it uses **DrawDibClose** to close the context and clean up. This function uses the handle to the DrawDib context as its only argument. It returns TRUE if the context closed successfully, otherwise it returns FALSE.

Supporting Palettes for the DrawDib Functions

If your application uses DrawDib, it should be palette-aware. That is, it must respond to WM_QUERYNEWPALETTE and WM_PALETTECHANGED messages. If your application does not already use palettes, it will need two short message handlers for these messages. In response to WM_PALETTECHANGED, your application should invalidate the destination window to let DrawDib redraw. In response to WM_QUERYNEWPALETTE, your application uses the following function to respond to this message:

DrawDibRealize
This function realizes palette for drawing.

This function has the following syntax:

UINT DrawDibRealize(*hdd*, *hdc*, *fBackground*)

The *hdd* parameter specifies a handle to a DrawDib context and the *hdc* parameter specifies a handle to the display context. The *fBackground* parameter specifies whether the logical palette is always to be the background palette. If this parameter is nonzero, the selected palette is always a background palette. If this parameter is zero and the device context is attached to a window, the logical palette is a foreground palette when the window has the input focus. It returns number of entries in the logical palette that were mapped to different values in the system palette.

Manipulating Palettes

Applications use the following functions for handling palettes associated with a DrawDib context:

DrawDibSetPalette
This function sets the palette used for drawing device independent bitmaps.

DrawDibChangePalette
This function sets the palette entries used for drawing device independent bitmaps.

DrawDibGetPalette
This function obtains the palette used by a DrawDib context.

If your application is already palette-aware, it might already have realized a palette and it needs to prevent DrawDib from realizing its own palette. Your application can use **DrawDibSetPalette** to notify DrawDib of the palette it would like to use. If your substitute palette does not contain the colors required by images displayed other applications, color shifts will appear in those images. This function has the following syntax:

BOOL DrawDibSetPalette(*hdd*, *hpal***)**

The *hdd* parameter specifies a handle to a DrawDib context. The *hpal* parameter specifies a handle to the palette your application wants to use.

If your application wants to modify the colors in the DrawDib palette, it can use **DrawDibChangePalette**. This function has the following syntax:

BOOL DrawDibChangePalette(*hdd*, *iStart*, *iLen*, *lppe*)

The *hdd* parameter specifies a handle to a DrawDib context.

The *iStart* parameter specifies the starting palette entry number and the *iLen* specifies the number of palette entries to change. The *lppe* parameter specifies a pointer to an array of palette entries.

If the DDF_ANIMATE flag was not set in the previous call to **DrawDibBegin**, this function will not animate the palette. In this case, use **DrawDibRealize** to realize the updated palette.

If your application needs the current palette, it can use **DrawDibGetPalette** to obtain a handle to the palette. If you want to realize the correct palette in response to a window message, just use **DrawDibRealize** instead of using this function to obtain the pallette and resending it. If your application uses this function, it should remember that it does not have exclusive use of the handle. Thus, your application should not free the palette when it is done and it should anticipate that some other application can invalidate the handle. Your application should rarely need to use **DrawDibGetPalette**.

Optimizing DrawDibDraw

If your application uses **DrawDibDraw** to display a series of bitmaps with the same dimensions and formats, it can use the following function to improve the efficiency of **DrawDibDraw**.

DrawDibBegin
This function prepares **DrawDibDraw** for drawing.

If your application does not use **DrawDibBegin**, **DrawDibDraw** implicitly executes it prior to drawing. When your application uses **DrawDibBegin** prior to **DrawDibDraw**, **DrawDibDraw** does not have to take the time to process the function and wait for it to complete. The **DrawDibBegin** function has the following syntax:

BOOL DrawDibBegin(*hdd*, *hdc*, *dxDest*, *dyDest*, *lpbi*, *dxSrc*, *dySrc*, *wFlags*)

The *hdd* parameter specifies a handle to a DrawDib context and the *hdc* parameter specifies a handle for the display context. Your application should use the values specified for these parameters in **DrawDibDraw**. When your application subsequently uses **DrawDibDraw**, it should set the DDF_SAME_HDC flag to indicate the handle to the display context has not changed.

The *dxDest*, *dyDest*, *dxSrc*, and *dySrc* parameters specify the width and height of the destination rectangle. The *lpbi* parameter points to the BITMAPINFOHEADER structure indicating the format of the image. Your application needs to use this same information when it specifies the parameters for **DrawDibDraw**. If your application changes these values, **DrawDibDraw** will sense the changes and implicitly use **DrawDibBegin** again. While your application must specify the width and height of the source and destination rectangles, it does not need to specify the origins. Thus, your application can redefine the origins in **DrawDibDraw** to use different portions of the image or update different portions of the display.

The *wFlags* parameter specifies applicable flags for the operation. Your application can use the DDF_ANIMATE if it will animate the palette.

Profiling the Display Characteristics

The very first time an application uses DrawDib capabilities, Windows displays a clock face and executes a series of tests to determine the display characteristics. Once the DrawDib libraries characterize the display, they will not run the tests again unless the information is removed (for example, by reinstalling Windows) or the user changes the display driver.

DrawDib Application Reference

This section is an alphabetic reference to the functions provided by Windows for applications using the DrawDib functions. The DRAWDIB.H file defines the functions, constants, and flags used by the DrawDib functions.

DrawDib Function Reference

Applications use the following functions for basic DrawDib operation:

DrawDibOpen
This function opens a DrawDib context for drawing.

DrawDibDraw
This function draws a device independent bitmap to the screen.

DrawDibClose
This function closes a DrawDib context and cleans up.

Applications use the following functions to improve the efficiency of **DrawDibDraw** when they draw a series of images:

DrawDibBegin
This function prepares **DrawDibDraw** for drawing.

DrawDibEnd
This function frees the resources allocated by **DrawDibBegin**.

Applications use the following functions for handling palettes associated with a DrawDib context:

DrawDibChangePalette
This function sets the palette entries used for drawing device independent bitmaps.

DrawDibGetPalette
This function obtains the palette used by a DrawDib context.

DrawDibRealize
This function realizes palette for drawing.

DrawDibSetPalette
This function sets the palette used for drawing device independent bitmaps.

DrawDib Functions

This section contains an alphabetical list of the functions applications can use for accessing the DrawDib services. The functions are identified with the prefix DrawDib.

DrawDibBegin

Syntax **BOOL DrawDibBegin**(*hdd*, *hdc*, *dxDest*, *dyDest*, *lpbi*, *dxSrc*, *dySrc*, *wFlags*)

This function prepares **DrawDibDraw** for drawing.

Parameters

HDRAWDIB *hdd*
Specifies a handle to a DrawDib context.

HDC *hdc*
Specifies a handle for the display context.

int *dxDest*
Specifies the width of the destination rectangle.

int *dyDest*
Specifies the height of the destination rectangle.

LPBITMAPINFOHEADER *lpbi*
Specifies a pointer to a **BITMAPINFOHEADER** structure containing the format of the data to be drawn.

int *dxSrc*
Specifies the width of the source rectangle.

int *dySrc*
Specifies the height of the source rectangle.

UNIT *wFlags*
Specifies applicable flags for the operation. The following flags are defined:

DDF_ANIMATE
Allows palette animation.

Return Value Returns a handle to a DrawDib context if successful, otherwise it returns NULL.

Comments This function prepares to draw a bitmap specified by *lpbi* to the display context *hdc* with stretching to the size *dxDest* and *dyDest*. If *dxDest* or *dyDest* is set to -1, the bitmap is drawn to a 1:1 scale with no stretching.

If *dxSrc* or *dySrc* is set to -1, **DrawDibDraw** uses the whole width or height of the source bitmap.

Use this function only if you want to prepare for drawing an image before you actually have data to draw. If you do not use this function, **DrawDibDraw** implicitly uses it when it draws the image.

See Also DrawDibEnd, DrawDibDraw

DrawDibChangePalette

Syntax **BOOL DrawDibChangePalette**(*hdd*, *iStart*, *iLen*, *lppe*)

This function sets the palette entries used for drawing device independent bitmaps.

Parameters

HDRAWDIB *hdd*
Specifies a handle to a DrawDib context.

int *iStart*
Specifies the starting palette entry number.

int *iLen*
Specifies the number of palette entries to change.

LPPALETTEENTRY *lppe*
Specifies a pointer to an array of palette entries.

Return Value Returns TRUE if successful, otherwise it returns FALSE.

Comments If the DDF_ANIMATE flag was not set in the previous call to **DrawDibBegin** or **DrawDibDraw**, this function will not animate the palette. In this case, use **DrawDibRealize** to realize the updated palette.

See Also DrawDibSetPalette, DrawDibGetPalette

DrawDibClose

Syntax **BOOL DrawDibClose**(*hdd*)

This function closes a DrawDib context and cleans up.

Parameters

HDRAWDIB *hdd*
Specifies a handle to a DrawDib context.

Return Value

Returns TRUE if the context closed successfully, otherwise it returns FALSE.

Comments

Use this function to free the **HDRAWDIB** handle and clean up after you have finished drawing.

See Also

DrawDibOpen

DrawDibDraw

Syntax

BOOL DrawDibDraw(*hdd*, *hdc*, *xDst*, *yDst*, *dxDst*, *dyDst*, *lpbi*, *lpBits*, *xSrc*, *ySrc*, *dxSrc*, *dySrc*, *wFlags*)

This function draws a device independent bitmap to the screen.

Parameters

HDRAWDIB *hdd*
Specifies a handle to a DrawDib context.

HDC *hdc*
Specifies a handle to the display context.

int *xDst*
Specifies the x-coordinate of the origin of the destination rectangle.

int *yDst*
Specifies the y-coordinate of the origin of the destination rectangle.

int *dxDst*
Specifies the width of the destination rectangle.

int *dyDst*
Specifies the height of the destination rectangle.

LPBITMAPINFOHEADER *lpbi*
Specifies a pointer to the **BITMAPINFOHEADER** structure for the bitmap.

LPVOID *lpBits*
Specifies a pointer to the buffer containing the bitmap bits. A null value indicates a packed-DIB. The data bits for a packed-DIB follow the color table in the **BITMAPINFOHEADER** structure pointed to by *lpbi*.

int *xSrc*
Specifies the x-coordinate of the source rectangle.

int *ySrc*
Specifies the y-coordinate of the source rectangle.

int *dxSrc*
Specifies the width of the source rectangle.

int *dySrc*
Specifies the height of the source rectangle.

UINT *wFlags*
Specifies any applicable flags for drawing. The following flags are defined:

DDF_UPDATE
Indicates the last bitmap is to be redrawn. If this flag is used, *lpBits* can be NULL.

DDF_SAME_HDC
Uses the handle to the display context specified previously. When used, DrawDib skips preparing the display context and assumes the correct palette has already been realized (possibly by **DrawDibRealize**). Your application should not use this flag until it uses a DrawDib function that specifies the display context. Your application must still specify a handle for *hdc* when it uses this flag.

DDF_SAME_DRAW
Uses the drawing parameters previously specified for this function. Use this flag only if the data specified for the *lpbi* structure, and the *dxDst*, *dyDst*, *dxSrc*, and *dySrc* parameters has not changed since last using **DrawDibDraw**.

DDF_DONTDRAW
Indicates the frame is only to be decompressed and not drawn. The *DDF_UPDATE* flag can be used later to actually draw the image.

DDF_ANIMATE
Allows palette animation. If this flag is present, the palette DrawDib creates will have the PC_RESERVED flag set for as many entries as possible, and the palette can be animated by with **DrawDibChangePalette**. The **DrawDibDraw** function passes this flag to **DrawDibBegin** implicitly.

Return Value Returns TRUE if successful, FALSE otherwise.

Comments This function replaces **StretchDIBits** and allows decompression of bitmaps by installable compressors. The function will dither true color bitmaps properly on 8-bit display devices.

DrawDibEnd

Syntax **BOOL DrawDibEnd**(*hdd*)

This function frees the resources allocated by **DrawDibBegin**.

Parameters HDRAWDIB *hdd*
Specifies the handle to the DrawDib context to free.

Return Value Returns TRUE if successful, otherwise it returns FALSE.

Comments Applications do not need to use this function.

DrawDibGetPalette

Syntax **HPALETTE DrawDibGetPalette**(*hdd*)

This function obtains the palette used by a DrawDib context.

Parameters HDRAWDIB *hdd*
Specifies a handle to a DrawDib context.

Return Value Returns a handle for the palette if successful, otherwise it returns NULL.

Comments If you want to realize the correct palette in response to a window message, use **DrawDibRealize** instead of this function. You should rarely need to call this function.

See Also DrawDibSetPalette, DrawDibRealize

DrawDibOpen

Syntax **HDRAWDIB DrawDibOpen**()

This function opens a DrawDib context for drawing.

Return Value Returns a handle to a DrawDib context if successful, otherwise it returns NULL.

Comments Call this function to obtain a handle to a DrawDib context before drawing DIBs.

See Also DrawDibClose

DrawDibRealize

Syntax **UINT DrawDibRealize**(*hdd*, *hdc*, *fBackground*)

This function realizes the palette of *hdc* into the DrawDib context specfied by *hdd*.

Parameters

HDRAWDIB *hdd*
Specifies a handle to a DrawDib context.

HDC *hdc*
Specifies a handle to the display context.

BOOL *fBackground*
Specifies whether the logical palette is always to be the background palette. If this parameter is nonzero, the selected palette is always a background palette. If this parameter is zero and the device context is attached to a window, the logical palette is a foreground palette when the window has the input focus.

Return Value Returns number of entries in the logical palette that were mapped to different values in the system palette. If an error occurs, it returns 0.

DrawDibSetPalette

Syntax **BOOL DrawDibSetPalette**(*hdd*, *hpal*)

This function sets the palette used for drawing device independent bitmaps.

Parameters

HDRAWDIB *hdd*
Specifies a handle to a DrawDib context.

HPALETTE *hpal*
Specifies a handle to the palette.

Return Value Returns TRUE if successful, otherwise it returns FALSE.

Comments Use this function when the application needs to realize an alternate palette. The function forces the DrawDib context to use the specified palette, possibly at the expense of image quality.

See Also DrawDibGetPalette

C H A P T E R 4

AVI Files

The Microsoft Audio/Video Interleaved (AVI) file format is a RIFF file specification used with applications that capture, edit, and playback audio/video sequences. In general, AVI files contain multiple streams of different types of data. Most AVI sequences will use both audio and video streams. A simple variation for an AVI sequence uses video data and does not require an audio stream. Specialized AVI sequences might include a control track or MIDI track as an additional data stream. The control track could control external devices such as an MCI videodisc player. The MIDI track could play background music for the sequence. While a specialized sequence requires a specialized control program to take advantage of all its capabilities, applications that can read and play AVI sequences can still read and play an AVI sequence in a specialized file. (These applications ignore the non-AVI data in the specialized file.) This chapter primarily describes AVI files containing only audio and video data.

This chapter covers the following topics:

- The required chunks of an AVI file
- The optional chunks of an AVI file
- Developing routines to write AVI files

For additional information about RIFF files, see the *Microsoft Windows Multimedia Programmer's Guide* and *Microsoft Windows Multimedia Programmer's Reference*.

For additional information about installable compressors and decompressors, see chapter 10, "Video Compression and Decompression Drivers."

AVI RIFF Form

AVI files use the AVI RIFF form. The AVI RIFF form is identified by the four-character code "AVI ". All AVI files include two mandatory LIST chunks. These chunks define the format of the streams and stream data. AVI files might also include an index chunk. This

optional chunk specifies the location of data chunks within the file. An AVI file with these components has the following form:

```
RIFF ('AVI '
     LIST ('hdrl'
            .
            .
            .
          )
     LIST ('movi'
            .
            .
            .
          )
     ['idx1'<AVI Index>]
     )
```

The LIST chunks and the index chunk are subchunks of the RIFF “AVI ” chunk. The “AVI ” chunk identifies the file as an AVI RIFF file. The LIST “hdrl” chunk defines the format of the data and is the first required list chunk. The LIST “movi” chunk contains the data for the AVI sequence and is the second required list chunk. The “idx1” chunk is the optional index chunk. AVI files must keep these three components in the proper sequence.

The LIST “hdrl” and LIST “movi” chunks use subchunks for their data. The following example shows the AVI RIFF form expanded with the chunks needed to complete the LIST “hdrl” and LIST “movi” chunks:

```
RIFF ('AVI '
       LIST ('hdrl'
             'avih'(<Main AVI Header>)
             LIST ('strl'
                   'strh'(<Stream header>)
                   'strf'(<Stream format>)
                   'strd'(additional header data)
                       .
                       .
                       .
             )


             .
             .
             .
   )


LIST ('movi'
             {SubChunk | LIST('rec '
                              SubChunk1
                              SubChunk2
                                 .
                                 .
                                 .
                         )
```

```
                        .
                        .
                        .
                    }


               .
               .
               .
          )

          ['idx1'<AVIIndex>]
     )
```

The following sections describe the chunks contained in the LIST “hdrl” and LIST “movi” chunks as well as the “idx1” chunk.

Data Structures for AVI Files

Data structures used in the RIFF chunks are defined in the AVIFMT.H header file. The reference section at the end of this chapter describes the data structures that can be used for the main AVI header, stream header, AVIIndex, and palette change chunks.

The Main AVI Header LIST

The file begins with the main header. In the AVI file, this header is identified with “avih” four-character code. The header contains general information about the file, such as the number of streams within the file and the width and height of the AVI sequence. The main header has the following data structure defined for it:

```
typedef struct {
    DWORD  dwMicroSecPerFrame;
    DWORD  dwMaxBytesPerSec;
    DWORD  dwReserved1;
    DWORD  dwFlags;
    DWORD  dwTotalFrames;
    DWORD  dwInitialFrames;
    DWORD  dwStreams;
    DWORD  dwSuggestedBufferSize;
    DWORD  dwWidth;
    DWORD  dwHeight;
    DWORD  dwScale;
    DWORD  dwRate;
    DWORD  dwStart;
    DWORD  dwLength;
} MainAVIHeader;
```

The **dwMicroSecPerFrame** field specifies the period between video frames. This value indicates the overall timing for the file.

The **dwMaxBytesPerSec** field specifies the approximate maximum data rate of the file. This value indicates the number of bytes per second the system must handle to present an

AVI sequence as specified by the other parameters contained in the main header and stream header chunks.

The **dwFlags** field contains any flags for the file. The following flags are defined:

AVIF_HASINDEX

Indicates the AVI file has an "idx1" chunk.

AVIF_MUSTUSEINDEX

Indicates the index should be used to determine the order of presentation of the data.

AVIF_ISINTERLEAVED

Indicates the AVI file is interleaved.

AVIF_WASCAPTUREFILE

Indicates the AVI file is a specially allocated file used for capturing real-time video.

AVIF_COPYRIGHTED

Indicates the AVI file contains copyrighted data.

The AVIF_HASINDEX and AVIF_MUSTUSEINDEX flags applies to files with an index chunk. The AVI_HASINDEX flag indicates an index is present. The AVIF_MUSTUSEINDEX flag indicates the index should be used to determine the order of the presentation of the data. When this flag is set, it implies the physical ordering of the chunks in the file does not correspond to the presentation order.

The AVIF_ISINTERLEAVED flag indicates the AVI file has been interleaved. The system can stream interleaved data from a CD-ROM more efficiently than non-interleaved data. For more information on interleaved files, see "Special Information for Interleaved Files."

The AVIF_WASCAPTUREFILE flag indicates the AVI file is a specially allocated file used for capturing real-time video. Typically, capture files have been defragmented by user so video capture data can be efficiently streamed into the file. If this flag is set, an application should warn the user before writing over the file with this flag.

The AVIF_COPYRIGHTED flag indicates the AVI file contains copyrighted data. When this flag is set, applications should not let users duplicate the file or the data in the file.

The **dwTotalFrames** field of the main header specifies the total number of frames of data in file.

The **dwInitialFrames** is used for interleaved files. If you are creating interleaved files, specify the number of frames in the file prior to the initial frame of the AVI sequence in this field.

The **dwStreams** field specifies the number of streams in the file. For example, a file with audio and video has 2 streams.

The **dwSuggestedBufferSize** field specifies the suggested buffer size for reading the file. Generally, this size should be large enough to contain the largest chunk in the file. If set to

zero, or if it is too small, the playback software will have to reallocate memory during playback which will reduce performance. For an interleaved file, the buffer size should be large enough to read an entire record and not just a chunk.

The **dwWidth** and **dwHeight** fields specify the width and height of the AVI file in pixels.

The **dwScale** and **dwRate** fields are used to specify the general time scale that the file will use. In addition to this time scale, each stream can have its own time scale. The time scale in samples per second is determined by dividing **dwRate** by **dwScale**.

The **dwStart** and **dwLength** fields specify the starting time of the AVI file and the length of the file. The units are defined by **dwRate** and **dwScale**. The **dwStart** field is usually set to zero.

The Stream Header ("strl") Chunks

The main header is followed by one or more "strl" chunks. (A "strl" chunk is required for each data stream.) These chunks contain information about the streams in the file. Each "strl" chunk must contain a stream header and stream format chunk. Stream header chunks are identified by the four-character code "strh" and stream format chunks are identified with the four-character code "strf". In addition to the stream header and stream format chunks, the "strl" chunk might also contain a stream data chunk. Stream data chunks are identified with the four-character code "strd".

The stream header has the following data structure defined for it:

```
typedef struct {
    FOURCC  fccType;
    FOURCC  fccHandler;
    DWORD   dwFlags;
    DWORD   dwReserved1;
    DWORD   dwInitialFrames;
    DWORD   dwScale;
    DWORD   dwRate;
    DWORD   dwStart;
    DWORD   dwLength;
    DWORD   dwSuggestedBufferSize;
    DWORD   dwQuality;
    DWORD   dwSampleSize;
} AVIStreamHeader;
```

The stream header specifies the type of data the stream contains, such as audio or video, by means of a four-character code. The **fccType** field is set to “vids” if the stream it specifies contains video data. It is set to “auds” if it contains audio data.

The **fccHandler** field contains a four-character code describing the installable compressor or decompressor used with the data.

The **dwFlags** field contains any flags for the data stream. The AVISF_DISABLED flag indicates that the stream data should be rendered only when explicitly enabled by the user. The AVISF_VIDEO_PALCHANGES flag indicates palette changes are embedded in the file.

The **dwInitialFrames** is used for interleaved files. If you are creating interleaved files, specify the number of frames in the file prior to the initial frame of the AVI sequence in this field.

The remaining fields describe the playback characteristics of the stream. These factors include the playback rate (**dwScale** and **dwRate**), the starting time of the sequence (**dwStart**), the length of the sequence (**dwLength**), the size of the playback buffer (**dwSuggestedBuffer**), an indicator of the data quality (**dwQuality**), and sample size (**dwSampleSize**). See the reference section for more information on these fields.

Some of the fields in the stream header structure are also present in the main header structure. The data in the main header structure applies to the whole file while the data in the stream header structure applies only to a stream.

A stream format (“strf”) chunk must follow a stream header (“strh”) chunk. The stream format chunk describes the format of the data in the stream. For video streams, the information in this chunk is a BITMAPINFO structure (including palette information if appropriate). For audio streams, the information in this chunk is a WAVEFORMATEX or PCMWAVEFORMAT structure. (The WAVEFORMATEX structure is an extended version of the WAVEFORMAT structure.) For more information on this structure, see the *New Multimedia Data Types and Data Techniques Standards Update*.

The “strl” chunk might also contain a stream data (“strd”) chunk. If used, this chunk follows the stream format chunk. The format and content of this chunk is defined by installable compression or decompression drivers. Typically, drivers use this information

for configuration. Applications that read and write RIFF files do not need to decode this information. They transfer this data to and from a driver as a memory block.

An AVI player associates the stream headers in the LIST “hdrl” chunk with the stream data in the LIST “movi” chunk by using the order of the “strl” chunks. The first “strl” chunk applies to stream 0, the second applies to stream 1, and so forth. For example, if the first “strl” chunk describes the wave audio data, the wave audio data is contained in stream 0. Similarly, if the second “strl” chunk describes video data, then the video data is contained in stream 1.

The LIST “movi” Chunk

Following the header information is a LIST “movi” chunk that contains chunks of the actual data in the streams; that is, the pictures and sounds themselves. The data chunks can reside directly in the LIST “movi” chunk or they might be grouped into “rec ” chunks. The “rec ” grouping implies that the grouped chunks should be read from disk all at once. This is used only for files specifically interleaved to play from CD-ROM.

Like any RIFF chunk, the data chunks contain a four-character code to identify the chunk type. The four-character code that identifies each chunk consists of the stream number and a two-character code that defines the type of information encapsulated in the chunk. For example, a waveform chunk is identified by a two-character code of “wb”. If a waveform chunk corresponded to the second LIST “hdrl” stream description, it would have a four-character code of “01wb”.

Since all the format information is in the header, the audio data contained in these data chunks does not contain any information about its format. An audio data chunk has the following format (the ## in the format represents the stream identifier):

```
WAVE  Bytes   '##wb'
      BYTE    abBytes[];
```

Video data can be compressed or uncompressed DIBs. An uncompressed DIB has BI_RGB specified for the **biCompression** field in its associated BITMAPINFO structure. A compressed DIB has a value other than BI_RGB specified in the **biCompression** field. For more information about compression formats, see the description of the BITMAPINFOHEADER data structure in the *Microsoft Windows Programmers Reference* and Chapter 5, “DIB Format Extensions for Microsoft Windows.”

A data chunk for an uncompressed DIB contains RGB video data. These chunks are identified with a two-character code of “db” (db is an abbreviation for DIB bits). Data chunks for a compressed DIB are identified with a two-character code of “dc” (dc is an abbreviation for DIB compressed). Neither data chunk will contain any header information about the DIBs. The data chunk for an uncompressed DIB has the following form:

```
DIB  Bits   '##db'
     BYTE   abBits[];
```

The data chunk for a compressed DIB has the following form:

```
Compressed DIB   '##dc'
     BYTE         abBits[];
```

Video data chunks can also define new palette entries used to update the palette during an AVI sequence. These chunks are identified with a two-character code of “pc” (pc is an abbreviation for palette change). The following data structure is defined palette information:

```
typedef struct {
    BYTE          bFirstEntry;
    BYTE          bNumEntries;
    WORD          wFlags;
    PALETTEENTRY  peNew;
} AVIPALCHANGE;
```

The **bFirstEntry** field defines the first entry to change and the **bNumEntries** field specifies the number of entries to change. The **peNew** field contains the new color entries.

If you include palette changes in a video stream, set the AVITF_VIDEO_PALCHANGES flag in the **dwFlags** field of the stream header. This flag indicates that this video stream contains palette changes and warns the playback software that it will need to animate the palette.

The “idx1” Chunk

AVI files can have an index chunk after the LIST “movi” chunk. The index chunk essentially contains a list of the data chunks and their location in the file. This provides efficient random access to the data within the file, because an application can locate a particular sound sequence or video image in a large AVI file without having to scan it.

Index chunks use the four-character code “idx1”. The following data structure is defined for index entries:

```
typedef struct {
    DWORD  ckid;
    DWORD  dwFlags;
    DWORD  dwChunkOffset;
    DWORD  dwChunkLength;
} AVIINDEXENTRY;
```

The **ckid**, **dwFlags**, **dwChunkOffset**, and **dwChunkLength** entries are repeated in the AVI file for each data chunk indexed. If the file is interleaved, the index will also have these entries for each “rec” chunk. The “rec” entries should have the AVIIF_LIST flag set and the list type in the **ckid** field.

The **ckid** field identifies the data chunk. This field uses four-character codes for identifying the chunk.

The **dwFlags** field specifies any flags for the data. The AVIIF_KEYFRAME flag indicates key frames in the video sequence. Key frames do not need previous video information to be decompressed. The AVIIF_NOTIME flag indicates a chunk does not

affect the timing of a video stream. For example, changing palette entries indicated by a palette chunk should occur between displaying video frames. Thus, if an application needs to determine the length of a video sequence, it should not use chunks with the AVIIF_NOTIME flag. In this case, it would ignore a palette chunk. The AVIIF_LIST flag indicates the current chunk is a LIST chunk. Use the **ckid** field to identify the type of LIST chunk.

The **dwChunkOffset** and **dwChunkLength** fields specify the position of the chunk and the length of the chunk. The **dwChunkOffset** field specifies the position of the chunk in the file relative to the 'movi' list. The **dwChunkLength** field specifies the length of the chunk excluding the eight bytes for the RIFF header.

If you include an index in the RIFF file, set the AVIF_HASINDEX in the **dwFlags** field of the AVI header. (This header is identified by “avih” chunk ID.) This flag indicates that the file has an index.

Other Data Chunks

If you need to align data in your AVI file you can add a “JUNK” chunk. (This chunk is a standard RIFF type.) Applications reading these chunks ignore their contents. Files played from CD-ROM use these chunks to align data so they can be read more efficiently. You might want to use this chunk to align your data for the 2 kilobyte CD-ROM boundaries. The “JUNK” chunk has the following form:

```
AVI Padding   'JUNK'
    Byte      data[]
```

As with any other RIFF files, all applications that read AVI files should ignore the non-AVI chunks that it does not recognize. Applications that read and write AVI files should preserve the non-AVI chunks when they save files they have loaded.

Special Information for Interleaved Files

Files that are interleaved for playback from CD-ROM require some special handling. While they can be read similarly to any other AVI files, they require special care when produced.

The audio has to be separated into single-frame pieces, and audio and video for each frame needs to be grouped together into “rec ” chunks. The record chunks should be padded so that their size is a multiple of 2 kilobytes and so that the beginning of the actual data in the LIST chunk lies on a 2 kilobyte boundary in the file. (This implies that the LIST chunk itself begins 12 bytes before a 2 kilobyte boundary.)

To give the audio driver enough audio to work with, the audio data has to be skewed from the video data. Typically, the audio data should be moved forward enough frames to allow approximately 0.75 seconds of audio data to be preloaded. The **dwInitialRecords** field of the main header and the **dwInitialFrames** field of the audio stream header should be set to the number of frames the audio is skewed.

Additionally, you must ensure that CD-ROM drive is capable of reading the data fast enough to support your AVI sequence. Non-MPC CD-ROM drives can have a data rate of less than 150 kilobytes per second.

Using VidEdit With AVI Files

VidEdit lets you create and edit audio-visual sequences consisting of a series of frames that contain digital audio and video data. You can use VidEdit to create and edit AVI files that contain one audio and one video stream. Each stream in the file must start at the beginning of the file (that is, the **dwStart** field in each stream header must be zero).

Example Code for Writing AVI Files

The WRITEAVI.C and AVIEASY.C files contain example code for writing AVI files. For simplicity, the examples assume that all video frames are uncompressed DIBs of the same size. While the DIBS can have any bit depth; 8, 16, and 24 bits are preferred.

These examples also assume all wave data is in memory. A more generalized procedure should work with wave data that is in memory as well as in a disk file. These examples do not restrict wave data to PCM. It should work with any format.

An Outline for Writing AVI Files

Like other RIFF files, AVI files are created with the **mmioOpen**, **mmioCreateChunk**, **mmioWrite**, **mmioAscend**, and **mmioClose** functions. These functions have the following definitions:

---

mmioOpen

Opens a file for reading or writing, and returns a handle to the open file.

mmioCreateChunk

Creates a new chunk in a RIFF file.

mmioWrite

Writes a specified number of bytes to an open file.

mmioAscend

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

mmioClose

Closes an open file.

In addition to these functions, you can use **mmioFOURCC** to convert four individual characters into a four-character code. For more information on these functions and macros, see the *Microsoft Windows Multimedia Programmer’s Guide* and *Microsoft Windows Multimedia Programmer’s Reference*.

The AVIFMT.H file contains macro definitions for creating the two- and four-character codes described in this chapter. It also defines the **aviTWOCC** and **TWOCCFromFOURCC** macros. These macros create two-character codes from individual characters or from four-character codes.

Unlike many other RIFF files, AVI files use many nested chunks and subchunks. This makes them more complicated than most RIFF files. Use the following tables as a checklist to help you decide when to create a chunk, when to write data to a chunk, and when to ascend from a chunk. The tables do not include information about writing non-AVI data chunks to the file. The information in the chunk column of the table mirrors the example in the “AVI RIFF Form” section presented previously.

Creating the File and “AVI ” Chunk

The “AVI ” chunk is the first chunk in the file. You will not ascend from this chunk until all other chunks have been created.

Chunk	How to Handle
RIFF ('AVI '	Use **mmioOpen** to open the file. Seek to the beginning of the file with **mmioSeek**. Create the AVI chunk with **mmioCreateChunk**. (Use the “AVI ” four-character code and the MMIO_CREATERIFF flag.) Do not ascend from this chunk in preparation for writing the remaining chunks.

Creating the LIST “hdrl ” and “avih” Chunks

The LIST “hdrl ” chunk contains the stream format header chunks. Because it contains other chunks, you will not ascend from it until the other header chunks are created.

The “avih” chunk contains the main header list. This is written as a complete chunk.

Chunk	How to Handle
LIST ('hdrl'	Create the LIST “hdrl” chunk with **mmioCreateChunk**. (Use the “hdrl” four-character code and the MMIO_CREATELIST flag.)
'avih'(<Main AVI Header>)	Create the Main AVI Header chunk with **mmioCreateChunk**. (Use the “avih” four-character code.) Write the header information with **mmioWrite**. Ascend from the “avih” chunk with **mmioAscend**. Do not ascend from the LIST “hdrl” chunk.

Creating the “strl”, “strh”, “strf”, and “strd” Chunks

The “strl”, “strh”, “strf”, and “strd” chunks are written as complete chunks. You write a set of the “strh”, “strf”, and “strd” chunks for each stream in the file. After all the stream descriptions are written, you ascend from LIST “hdrl” chunk.

Chunk	How to Handle
LIST ('strl'	Create the LIST “strl” chunk with **mmioCreateChunk**. (Use the “strl” four-character code and the MMIO_CREATELIST flag.)
'strh'(<Stream header>)	Create the stream header chunk with **mmioCreateChunk**. (Use the “strh” four-character code.) Write the stream header information with **mmioWrite**. Ascend from the “strh” chunk with **mmioAscend**.
'strf'(<Stream format>)	Create the stream format chunk with **mmioCreateChunk**. (Use the “strf” four-character code.) Write the stream format information with **mmioWrite**. Ascend from the “strf” chunk with **mmioAscend**.
'strd'(additional header data)	If needed, create chunks for any additional header data with **mmioCreateChunk**. (Use the “strd” four-character code.) Write the additional header data with **mmioWrite**. Ascend from the “strd” chunk.
. . .	If needed, add stream header, stream format, and additional header data chunks for other streams in the file.
)	Ascend from the LIST “strl” chunk with **mmioAscend**.
. . .)	Ascend from the LIST “hdrl” chunk with **mmioAscend**. If needed, create and write padding chunks or other data chunks.

Creating the LIST “movi” and “rec ” Chunks

The LIST “movi” chunk contains other chunks. After you create this chunk, you will not ascend from it until the other chunks are written.

You can write the data as an individual chunk or as part of a “rec ” chunk. Like the LIST “movi” chunk, you will not ascend from a “rec ” chunk until you write all of its subchunks.

Chunk	How to Handle
LIST ('movi'	Create the LIST “movi” chunk with **mmioCreateChunk**. (Use the LIST “movi” four-character code and the MMIO_CREATELIST flag.)
{ SubChunk | LIST('rec ' SubChunk1 SubChunk2 . . .) . . . } . . .	You can add your movie data directly at this point in a subchunk or include it in a “rec ” chunk. The following steps summarize creating these chunks: Create a data chunk with **mmioCreateChunk**. (Use the four-character code appropriate for the data chunk and stream.) If you are adding an index chunk to the end of the file, save the location of the subchunks for it.
)	Ascend from the LIST “movi ” chunk.

Creating the “idx1” Chunk and Ascending From the “AVI ” Chunk

The optional index chunk is written as a complete chunk. After you have completed this chunk, you can ascend from the “AVI ” chunk and close the file.

Chunk	How to Handle
['idx1'<AVIIndex>]	If used, create the AVI index chunk with **mmioCreateChunk**. (Use the “idx1” four-character code.) Write the index information with **mmioWrite**. Ascend from the “idx1” chunk with **mmioAscend**. Although the “idx1” is the last chunk used in an AVI sequence, you can add non-AVI chunks after it. These subchunks will still be part of the “AVI ” chunk.
)	Ascend from the “AVI ” chunk with **mmioAscend**. Close the file with **mmioClose**.

AVI RIFF File Reference

This section lists data structures used to support AVI RIFF files. (These structures are defined in AVIFMT.H.) The data structures are presented in alphabetical order. The structure definition is given, followed by a description of each field.

AVIINDEXENTRY

The AVI file index consists of an array of **AVIINDEXENTRY** structures contained within an 'idx1' chunk at the end of an AVI file. This chunk follows the main LIST 'movi' chunk which contains the actual data.

```
typedef struct {
    DWORD  ckid;
    DWORD  dwFlags;
    DWORD  dwChunkOffset;
    DWORD  dwChunkLength;
} AVIINDEXENTRY;
```

Fields

The **AVIINDEXENTRY** structure has the following fields:

ckid
Specifies a four-character code corresponding to the chunk ID of a data chunk in the file.

dwFlags
Specifies any applicable flags. The flags in the low-order word are reserved for AVI, while those in the high-order word can be used for stream- and compressor/decompressor-specific information.

The following values are currently defined:

AVIIF_LIST
Indicates the specified chunk is a 'LIST' chunk, and the **ckid** field contains the list type of the chunk.

AVIIF_KEYFRAME
Indicates this chunk is a key frame. Key frames do not require additional preceding chunks to be properly decoded.

AVIIF_FIRSTPART
Indicates this chunk needs the frames following it to be used; it cannot stand alone.

AVIIF_LASTPART
Indicates this chunk needs the frames preceding it to be used; it cannot stand alone.

AVIIF_NOTIME
Indicates this chunk should have no effect on timing or calculating time values based on the number of chunks. For example, palette change chunks in a video stream should have this flag set, so that they are not counted as taking up a frame's worth of time.

dwChunkOffset
Specifies the position in the file of the specified chunk. The position value includes the eight byte RIFF header.

dwChunkLength
Specifies the length of the specified chunk. The length value does not include the eight byte RIFF header.

AVIPALCHANGE

The **AVIPALCHANGE** structure is used in video streams containing palettized data to indicate the palette should change for subsequent video data.

```
typedef struct {
    BYTE  bFirstEntry;
    BYTE  bNumEntries;
    WORD  wFlags;
    PALETTEENTRY  peNew;
} AVIPALCHANGE;
```

Fields

The **AVIPALCHANGE** structure has the following fields:

bFirstEntry
Specifies the first palette entry to change.

bNumEntries
Specifies the number of entries to change.

wFlags
Reserved. (This should be set to 0.)

peNew
Specifies an array of new palette entries.

AVIStreamHeader

The **AVIStreamHeader** structure contains header information for a single stream of an file. It is contained within an 'strh' chunk within a LIST 'strl' chunk that is itself contained within the LIST 'hdrl' chunk at the beginning of an AVI RIFF file.

```
typedef struct {
    FOURCC  fccType;
    FOURCC  fccHandler;
    DWORD  dwFlags;
    DWORD  dwReserved1;
    DWORD  dwInitialFrames;
    DWORD  dwScale;
    DWORD  dwRate;
    DWORD  dwStart;
    DWORD  dwLength;
    DWORD  dwSuggestedBufferSize;
    DWORD  dwQuality;
    DWORD  dwSampleSize;
} AVIStreamHeader;
```

Fields

The **AVIStreamHeader** structure has the following fields:

fccType
Contains a four-character code which specifies the type of data contained in the stream. The following values are currently defined for AVI data:

'vids'
Indicates the stream contains video data. The stream format chunk contains a **BITMAPINFO** structure which can include palette information.

'auds'
Indicates the stream contains video data. The stream format chunk contains a **WAVEFORMAT** or **PCMWAVEFORMAT** structure.

Other four-character codes can identify non-AVI data.

fccHandler
Optionally, contains a four-character code that identifies a specific data handler. The data handler is the preferred handler for the stream.

dwFlags
Specifies any applicable flags. The bits in the high-order word of these flags are specific to the type of data contained in the stream. The following flags are currently defined:

AVISF_DISABLED
Indicates this stream should not be enabled by default.

AVISF_VIDEO_PALCHANGES
Indicates this video stream contains palette changes. This flag warns the playback software that it will need to animate the palette.

dwReserved1
Reserved. (Should be set to 0.)

dwInitialFrames
Specifies how far audio data is skewed ahead of the video frames in interleaved files. Typically, this is about 0.75 seconds.

dwScale
This field is used together with **dwRate** to specify the time scale that this stream will use.

Dividing **dwRate** by **dwScale** gives the number of samples per second.

For video streams, this rate should be the frame rate.

For audio streams, this rate should correspond to the time needed for **nBlockAlign** bytes of audio, which for PCM audio simply reduces to the sample rate.

dwRate
See **dwScale**.

dwStart
Specifies the starting time of the AVI file. The units are defined by the **dwRate** and **dwScale** fields in the main file header. Normally, this is zero, but it can specify a delay time for a stream which does not start concurrently with the file.

Note: The 1.0 release of the AVI tools does not support a non-zero starting time.

dwLength
Specifies the length of this stream. The units are defined by the **dwRate** and **dwScale** fields of the stream's header.

dwSuggestedBufferSize
Suggests how large a buffer should be used to read this stream. Typically, this contains a value corresponding to the largest chunk present in the stream. Using the correct buffer size makes playback more efficient. Use zero if you do not know the correct buffer size.

dwQuality
Specifies an indicator of the quality of the data in the stream. Quality is represented as a number between 0 and 10000. For compressed data, this typically represent the value of the quality parameter passed to the compression software. If set to -1, drivers use the default quality value.

dwSampleSize
Specifies the size of a single sample of data. This is set to zero if the samples can vary in size. If this number is non-zero, then multiple samples of data can be grouped into a single chunk within the file. If it is zero, each sample of data (such as a video frame) must be in a separate chunk.

For video streams, this number is typically zero, although it can be non-zero if all video frames are the same size.

For audio streams, this number should be the same as the **nBlockAlign** field of the **WAVEFORMAT** structure describing the audio.

MainAVIHeader

The **MainAVIHeader** structure contains global information for the entire AVI file. It is contained within an 'avih' chunk within the LIST 'hdrl' chunk at the beginning of an AVI RIFF file.

```
typedef struct {
    DWORD  dwMicroSecPerFrame;
    DWORD  dwMaxBytesPerSec;
    DWORD  dwReserved1;
    DWORD  dwFlags;
    DWORD  dwTotalFrames;
    DWORD  dwInitialFrames;
    DWORD  dwStreams;
    DWORD  dwSuggestedBufferSize;
    DWORD  dwWidth;
    DWORD  dwHeight;
    DWORD  dwScale;
    DWORD  dwRate;
    DWORD  dwStart;
    DWORD  dwLength;
} MainAVIHeader;
```

Fields

The **MainAVIHeader** structure has the following fields:

dwMicroSecPerFrame
Specifies the number of microseconds between frames.

dwMaxBytesPerSec
Specifies the approximate maximum data rate of file.

dwReserved1
Reserved. (This field should be set to 0.)

dwFlags
Specifies any applicable flags. The following flags are defined:

AVIF_HASINDEX
Indicates the AVI file has an 'idx1' chunk containing an index at the end of the file. For good performance, all AVI files should contain an index.

AVIF_MUSTUSEINDEX
Indicates that the index, rather than the physical ordering of the chunks in the file, should be used to determine the order of presentation of the data. For example, this could be used for creating a list frames for editing.

AVIF_ISINTERLEAVED
Indicates the AVI file is interleaved.

AVIF_WASCAPTUREFILE
Indicates the AVI file is a specially allocated file used for capturing real-time video. Applications should warn the user before writing over a file with this flag set because the user probably defragmented this file.

AVIF_COPYRIGHTED
Indicates the AVI file contains copyrighted data and software. When this flag is used, software should not permit the data to be duplicated.

dwTotalFrames
Specifies the number of frames of data in file.

dwInitialFrames
Specifies the initial frame for interleaved files. Non-interleaved files should specify zero.

dwStreams
Specifies the number of streams in the file. For example, a file with audio and video has 2 streams.

dwSuggestedBufferSize
Specifies the suggested buffer size for reading the file. Generally, this size should be large enough to contain the largest chunk in the file. If set to zero, or if it is too small, the playback software will have to reallocate memory during playback which will reduce performance.

For an interleaved file, this buffer size should be large enough to read an entire record and not just a chunk.

dwWidth
Specifies the width of the AVI file in pixels.

dwHeight
Specifies the height of the AVI file in pixels.

dwScale
This field is used with **dwRate** to specify the time scale that the file as a whole will use. In addition, each stream can have its own time scale.

Dividing **dwRate** by **dwScale** gives the number of samples per second.

dwRate
See **dwScale**.

dwStart
Specifies the starting time of the AVI file. The units are defined by **dwRate** and **dwScale**. This field is usually set to zero.

dwLength
Specifies the length of the AVI file. The units are defined by **dwRate** and **dwScale**. This length is returned by MCIAVI when using the frames time format.

CHAPTER 5

DIB Format Extensions for Microsoft Windows

The DIB format extensions for Microsoft Windows add the capabilities to handle new compression formats, custom compression formats, and inverted DIBs. The extensions also include an escape message to let applications interrogate display drivers to determine their capabilities. This chapter includes the following topics related to these extensions:

- 16 and 32 bit extensions to the BI_RGB compression format
- 16 and 32 bit BI_BITFIELDS compression format extensions
- Extensions for custom compression formats
- Determining display driver capabilities
- Inverted DIBs

Windows Compression Formats

Compression flags for a bitmap are specified in the BITMAPINFOHEADER data structure defined by Windows. This structure has the following fields:

```
typedef struct tagBITMAPINFOHEADER {
    DWORD biSize;
    LONG  biWidth;
    LONG  biHeight;
    WORD  biPlanes;
    WORD  biBitCount;
    DWORD biCompression;
    DWORD biSizeImage;
    LONG  biXPelsPerMeter;
    LONG  biYPelsPerMeter;
    DWORD biClrUsed;
    DWORD biClrImportant;
} BITMAPINFOHEADER;
```

Information about the compression format is specified in the **biCompression** and **biBitCount** fields. The **biCompression** field specifies the type of compression used or requested. Both existing and new compression formats use this field.

The **biBitCount** field specifies the number of bits per pixel. Some compression formats need this information to properly decode the colors in the pixel.

When the value in the **biBitCount** field is set to less than or equal to eight, video drivers can assume the bitmap uses a palette or color table defined in the BITMAPINFO data structure. This data structure has the following fields:

```
typedef struct tagBITMAPINFO {
    BITMAPINFOHEADER bmiHeader;
    RGBQUAD bmiColors[1]
} BITMAPINFO;
```

When the value in the **biBitCount** field is set to greater than eight, video drivers can assume bitmaps are true color and they do not use a color table. For more information on these data structures, see the *Microsoft Windows Programmer's Reference*.

Existing Formats

Windows defines the following compression formats:

BI_RGB
: Specifies the bitmap is not compressed. (Valid for **biBitCount** set to 1, 4, 8, 16, 24, or 32.)

BI_RLE8
: Specifies a run-length encoded format for bitmaps with 8 bits per pixel. (Valid for **biBitCount** set to 8.)

BI_RLE4
: Specifies a run-length encoded format for bitmaps with 4 bits per pixel. (Valid for **biBitCount** set to 4.)

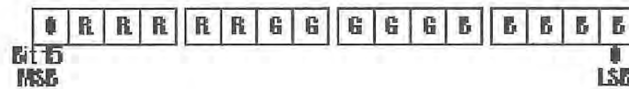
For more information on these formats, see the *Microsoft Windows Programmer's Reference*.

Extensions to the BI_RGB Format

Extensions to the BI_RGB format include 16 and 32 bits per pixel bitmap formats. These formats do not use a color table. They embed the colors in the WORD or DWORD representing each pixel.

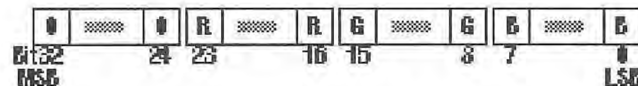
The 16 bit BI_RGB format is identified by setting **biCompression** to BI_RGB and **biBitCount** to 16. For this format, each pixel in the bitmap is represented by a 16 bit RGB color value. The high-bit of this value is zero. The remaining bits are divided into three groups of 5-bits to represent the red, green, and blue color values. The group containing the five most significant bits represents red. The group containing the five least significant bits represents blue. (This format is also referred to as the RGB555 format.

This format supports 32K colors.) The following illustration shows the bit organization of the RGB555 format:



16 bit BI_RGB format.

The 32 bit BI_RGB format is identified by setting **biCompression** to BI_RGB and **biBitCount** to 32. For this format, each pixel is represented by a 32 bit (4 byte) RGB color value. The first byte is zero. The second byte represents red, the third byte represents green, and the last byte represents blue. The following illustration shows the bit organization of this format:



32 bit BI_RGB format.

Display drivers must support the BI_RGB format for 1, 4, 8, and 24 bits per pixel bitmaps. If practical, they should also support this format for 16 and 32 bits per pixel bitmaps.

Formats Using BI_BITFIELDS and Color Masks

In addition to the 16 and 32 bits per pixel BI_RGB format, the BI_BITFIELDS flag has been defined for 16 and 32 bit bitmaps. This flag is recognized only by enhanced display drivers and does not need to be supported by most display drivers. The BI_BITFIELDS flag has the following definition:

BI_BITFIELDS
: Specifies the bitmap is not compressed and a color mask is defined in the **bmiColors** field of the BITMAPINFO data structure. (Valid for **biBitCount** set to 16 or 32.)

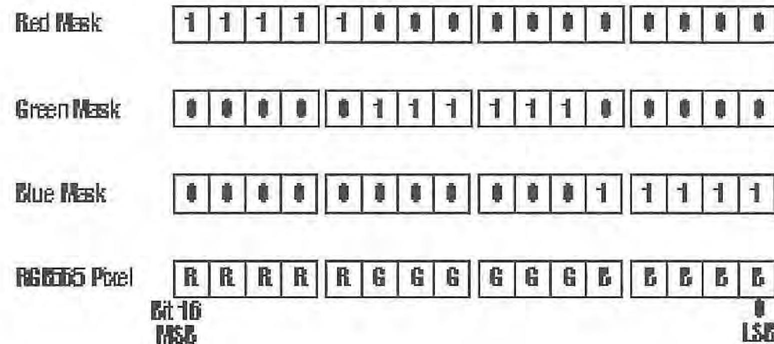
Setting the **biCompression** field to BI_BITFIELDS indicates the **bmiColors** field contains three DWORDS used to mask each pixel in the bitmap. The masks are used to obtain the RGB color values of the pixel. The first DWORD contains the red mask, the second DWORD contains the green mask, and the third DWORD contains the blue mask. The image bits follow the three DWORDs. The color masks have the following characteristics:

- The bits in a mask must not overlap any bits in another mask.
- The set of bits defined for each mask must be contiguous.

These characteristics do not restrict any one mask to a particular location in a DWORD. For example, the red mask can occupy the least significant, most significant, or central position of the ORed combination of all three masks. The position of each mask corresponds to the color position defined for the appropriate RGB component of each pixel. This implies for a 16 bit image, the color masks will reside in the low-ordered word of the DWORD. (For 16 bit images, set the **biBitCount** field of the BITMAPINFOHEADER data structure to 16; for 32 bit images set it to 32.)

Additionally, you need to set the bits in a mask only for the bit positions in a pixel that represent color. Because unused bits in a pixel will always be masked, you can set the unused bits in a pixel to either zero or one.

For example, color masks can be used to decode the colors of a 16 bit pixel divided into three unequal groups of bits to represent the red, green, and blue color values. The group containing the five most significant bits represents red. The group containing the five least significant bits represents blue. The group containing the middle six bits represents green. (This format is also referred to as the RGB565 format.) The following illustration shows the definitions of the color masks and the bit organization of a pixel with the RGB565 format:



RGB565 format using BI_BITFIELDS.

Drivers obtain the RGB values for a pixel by masking the pixel with the DWORD corresponding to each color mask and then they map the colors to the appropriate registers for display. (If an application needs to retrieve the individual color values for a pixel, it can use the color masks to separate the color components and then right shift each color component by the number of least significant zeros in the mask.)

Custom Formats

Your driver can define custom compression and bitmap formats by assigning a four-character code to the **biCompression** field in place of the standard constants. When you define a custom format, you must specify the number of bytes in the image in the **biSizeImage** field.

The compression type four-character code must be unique. If you want to create a new four-character code for a compression type, register it with Microsoft to set up a standard definition of it and avoid any conflicts with other compression codes that might be defined. To register a code for a compression type, request a *Multimedia Developer Registration Kit* from the following group:

Microsoft Corporation
Multimedia Systems Group
Product Marketing
One Microsoft Way
Redmond, WA 98052-6399

The following is a list of the currently reserved compression types:

Four-character Code	biBitCount	Compression Method	Registered by
CRAM Cram	8, 16	Video compression	Microsoft
JPEG	24	JPEG format for images	Microsoft
YUV9	24, 16	411 YUV format for images	Microsoft
TYUV	8, 16	YUV	Microsoft
RYUV	8	Delta YUV	Microsoft

For more information on four character codes, see the *Microsoft Windows Multimedia Programmer's Guide*, *Microsoft Windows Multimedia Programmer's Reference*, and Chapter 10, "Video Compression and Decompression Drivers.".

Determining Display Driver Capabilities

You can determine if a display driver can handle a DIB with the QUERYDIBSUPPORT escape. The following syntax statement illustrates the use of this escape:

short **Escape**(*hdc*, QUERYDIBSUPPORT, *nSize*, *lpbi*, *lpFlags*)

The following parameter descriptions apply to the QUERYDIBSUPPORT escape:

Parameter	Data Type	Description
hdc	HDC	Identifies the device context.
nSize	int	Specifies the size of the BITMAPINFO data structure passed.
lpbi	LPBITMAPINFO	Points to a BITMAPINFO data structure containing the characteristics of the bitmap.

(Continued)

Parameter	Data Type	Description
lpFlags	LPINT	Points to an integer containing the return flags. Drivers set these flags to indicate which capabilities they support. The following flags are defined: **Flag** — **Description** QDI_SETDIBITS — Device can convert DIB to bitmap QDI_GETDIBITS — Device can convert bitmap to DIB QDI_DIBTOSCREEN — Device can draw DIB QDI_STRETCHDIB — Device can stretch DIB

When a display driver gets this escape it should examine the BITMAPINFO structure indicated by *lpbi* and determine if it supports the DIB. The driver checks **biBitCount** for the proper bit depth, **biCompression** for the proper compression type, and **biHeight** for a positive or negative value (negative values indicate an inverted DIB). If **biCompression** is set to BI_BITFIELDS, the driver also checks the bit masks in the color table.

A display driver will set flags for *lpFlags* if it provides either partial or complete functionality corresponding to the flag. For example, a driver sets the QDI_STRETCHDIB flag if can stretch a DIB by integer amounts (partial functionality) or if it can stretch a DIB by both integer and non-integer values (complete functionality).

Inverted DIBs

Video drivers incorporating the DIB format extensions will let you specify negative values for **biHeight**. If **biHeight** is negative then the origin of the bitmap is the upper-left corner and the height is the absolute value of **biHeight**.

Applications determine if a driver supports inverted DIBs by sending the QUERYDIBSUPPORT flag with **biHeight** set to a negative value. Drivers return the QDI_DIBTOSCREEN flag in response to this if they support inverted DIBs.

Definition of the Flags and Escape

The flags, constants, and escape values described in this chapter are defined in MMREG.H. Use this header file until these flags, escapes, and constants are added to the header files distributed with Microsoft Windows.

CHAPTER 6

Playing AVI Files With MCI

This chapter describes how to play Video for Windows AVI files using the MCI interface. It contains the following topics:

- MCI Overview
- Using the MCI Command Interface
- Using the MCI String Interface
- Handling MCI notification
- Playing AVI files using MCI

Sample code for AVI playback is in the MOVPLAY1.C and MOVPLAY2.C files. MOVPLAY1.C uses the MCI command interface while MOVPLAY2.C uses the string interface. Both applications look the same to the end user, they just illustrate the different methods of using MCI to send commands.

MCI Overview

MCI provides a high-level interface to control various media devices through generalized commands such as play, pause and stop as well as through specific command sets defined for different device types. MCI uses the MCIAVI.DRV driver to handle AVI playback.

Your application uses MCI commands from the Digital Video Command Set to control MCIAVI.DRV. Since most of the work is done by the commands and not by MCI directly, the interface to MCI itself is very simple and just passes commands down to MCIAVI. In fact, MCI only has five functions that applications use for MCI operation. Of these five functions, the following two functions are commonly used for sending commands:

mciSendCommand
Sends a command message to MCI.

mciSendString
Sends a string command to MCI.

Your application must link with MMSYSTEM.LIB to use MCI. It must also include the MMSYSTEM.H and DIGITALV.H files. The MMSYSTEM.H file included with the SDK for Microsoft Windows defines the prototypes for these functions and defines the messages, flags, constants, and data structures needed for their use. The DIGITALV.H file

defines the digital video command set specifically used to control MCIAVI. For a summary of the command messages and command strings used with MCIAVI, see Chapter 7, “MCI Command Strings for MCIAVI” and Chapter 8, “MCI Command Messages for MCIAVI.”

For full information on the MCI commands see the *Microsoft Multimedia Programmer's Reference* and the *Microsoft Multimedia Programmer's Guide* of the Windows 3.1 Software Development Kit. For full information on the MCI Digital Video Command Set see the *Digital Video Command Set for the Media Control Interface* standards update.

Using the MCI Command Interface

One method of sending MCI commands to MCIAVI uses **mciSendCommand** to send a *command message*. Command messages include a message corresponding to the command, a set of flags, and a data structure defining the parameters for that command. This function has the following syntax:

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

The *wDeviceID* parameter defines the device ID of the MCI device that will receive the command. (This parameter is returned for the open command, which does not require the device ID.) The *wMessage* parameter specifies the message your application wants to send. The *dwParam1* parameter defines the flags for the command, and *dwParam2* points to a data structure for the command. This function returns 0 on success or an MCI error code on failure.

The programming example for sending a command message has MCIAVI.DRV play an AVI file. The command message for playing an AVI file is MCI_PLAY. For this command, MCIAVI.DRV accepts the following flags in *dwParam1:*

MCI_FROM
Indicates the **dwFrom** field of the structure identified by *dwParam2* specifies a starting position for the file.

MCI_TO
Indicates the **dwTo** field of the structure identified by *dwParam2* specifies an ending position for the file.

MCI_DGV_PLAY_WINDOW
Indicates playing should occur in the window associated with a device instance (the default).

MCI_MCIAVI_PLAY_FULLSCREEN
Indicates playing should use a full-screen display, typically with a 320x200 resolution.

The MCI_PLAY command uses the following data structure to pass information (*dwParam2* points to this structure):

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwFrom;
    DWORD  dwTo;
} MCI_DGV_PLAY_PARMS;
```

Prior to using **mciSendCommand** to send the MCI_PLAY message, your application allocates the memory for the data structure, initializes the fields it wants to use, and sets the flags corresponding to the fields used in the data structure. (If your application does not set a flag for a data structure field, MCI drivers ignore the data structure field.) For example, the following function plays a movie from the starting position specified by *dwFrom* to the an ending position specified by *dwTo* (if either position is 0 then it is considered not used):

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

    // check dwFrom, if it is != 0 then set parameters and flags
    if (dwFrom){
        mciPlay.dwFrom = dwFrom; // set parameter
        dwFlags |= MCI_FROM;     // set corresponding flag to validate field
    }

    // check dwTo, if it is != 0 then set parameters and flags
    if (dwTo){
        mciPlay.dwTo = dwTo;    // set parameter
        dwFlags |= MCI_TO;      // set corresponding flag to validate field
    }

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

Using the MCI String Interface

Another method of sending MCI commands to MCIAVI uses **mciSendString** to send a *command string*. This function uses text strings to represent the command. It has the following syntax:

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

The *lpstrCommand* parameter specifies a far pointer to the actual command string. Each command string includes a command, a device identifier, and command arguments. Arguments are optional on some commands and required on other commands. A command string has the following form:

command device_id arguments

The *command* parameter represents the command name (for example, **open**, **close**, or **play**). The *device_id* parameter identifies the MCI driver or a file. The *arguments* parameter indicates any flags and values associated with the command.

When your application opens MCIAVI, it uses a device name, a keyword from the [MCI] section of the SYSTEM.INI file, or filename as the *device_id* used to identify the MCI device. Your application can avoid using the formal *device_id* argument in subsequent commands by specifying the **alias** flag when it opens MCIAVI. (The alias your application wants to use in subsequent commands is specified after the **alias** flag.) Most string examples in this section use an alias.

The *lpstrReturnString* parameter of **mciSendString** specifies a far pointer to a buffer for return information. (Your application can set it NULL if a command does not return information.) The *wReturnLength* parameter specifies the size of the return buffer, or 0 if no buffer is specified. The *hCallback* parameter specifies a window handle if your applications wants to receive MCI notify messages.

For example, the string **play** command used with MCIAVI.DRV has the following definition and arguments:

play *items*	Starts playing the video sequence. The following optional *items* modify the command:	
	from *position*	Specifies a starting position for the play.
	to *position*	Specifies an ending position for the play.
	fullscreen	Specifies playing should use a full-screen display.
	window	Specifies playback should be in the window associated with the device instance (the default).

The following example uses the string interface to send the **play** command with the **from** and **to** flags:

```
DWORD PlayMovie(LPSTR lpstrAlias, DWORD dwFrom, DWORD dwTo)
{
    char    achCmndBuff[128];

    wsprintf(achCmndBuff, "play %s from %u to %u", lpstrAlias, dwFrom, dwTo);

    return mciSendString(achCommandBuff, NULL, 0, NULL);
}
```

When using the string interface, all values passed with the command and all return values are text strings so your application needs conversion routines to translate from variables to strings or back again. For example, the following fragment gets the size of an AVI sequence and uses the size to allocate memory for a RECT structure:

```
void GetSourceRect(LPSTR lpstrAlias, LPRECT lpRect)
{
    char    achRetBuff[128];
    char     achCommandBuff[128];

    // build the command string "where name source"
    sprintf(achCommandBuff, "where %s source", lpstrAlias);


    SetRectEmpty(lpRect);    // clear the RECT


    // send the command
    if (mciSendString(achCommandBuff, achRetBuff,
               sizeof(achRetBuff), NULL)== 0){


        // The rectangle is returned in our buffer as "x y dx dy" and we
        // know that x and y are both 0 since this is the source rectangle.
        // The following lines translate the string into the RECT structure.
        char    *p;
        p = achRetBuff;            // point to the return string
        while (*p != ' ') p++;     // go past the x (0)
        while (*p == ' ') p++;     // go past spaces
        while (*p != ' ') p++;     // go past the y (0)
        while (*p == ' ') p++;     // go past spaces

        // now get the width
        for ( ; *p != ' '; p++)
            lpRect->right = (10 * lpRect->right) +
                            (*p - '0');

        while (*p == ' ') p++;     // go past spaces

        // now get the height
        for ( ; *p != ' '; p++)
            lpRect->bottom = (10 * lpRect->bottom) +
                            (*p - '0');

    }

}
```

Choosing the mciSendCommand or mciSendString Interface

Since there are two interfaces to send commands to MCIAVI, you must select the most appropriate one for your application's needs. With the command interface, your application must fill a data structure and make sure that the flags it sets match the data structure fields it uses. With the string interface, however, your application must handle the conversion of string data for anything that might be variable within the application. Your application might choose to mix the two methods for the most efficient operation.

For straightforward commands your application might use the string interface, and for commands that return information or commands your application passes information (such as window or palette handles), it might use the command interface.

You will probably find the string interface the easiest command set to understand and read. While the structure of the string commands is simple, it still retains the capabilities of the message commands to control MCI devices. This makes the command set extremely useful in planning your application and discussing the MCI capabilities of your application.

The examples in the rest of this chapter use a combination of the string and command interfaces. You can find other examples of using MCI with the digital video command set in the MOVPLAY1.C and MOVPLAY2.C files. For examples using command messages, see MOVPLAY1.C. For examples using command strings, see MOVPLAY2.C.

Handling MCI Notification

Whichever interface your application uses, it can have MCI send notification messages when an action completes. With the string interface, your application requests notification by adding the **notify** flag to the string command it sends. It prepares to receive the notification messages by setting the *hCallback* parameter to its window handle. With the command interface, your application requests notification by adding **MCI_NOTIFY** to the flags sent in *dwParam1*. It prepares to receive the notification messages by setting the **dwCallback** field associated with *dwParam2* to the callback window handle. In both cases the callback window procedure must be able to handle the MM_MCINOTIFY message.

An MCI notification message indicates one of the following results:

- The command completed successfully—MCI_NOTIFY_SUCCESSFUL
- The command was superseded—MCI_NOTIFY_SUPERCEDED
- The command was aborted—MCI_NOTIFY_ABORTED
- The command fails—MCI_NOTIFY_FAILURE

The MOVPLAY sample applications uses notification on the play command to determine when playing stops at the end of the sequence. Once started this way, the sequence plays independently of MOVPLAY and MCI notifies MOVPLAY when the sequence completes. MOVPLAY uses the notification message to rewind the sequence. The main window procedure of MOVPLAY includes the following fragment to handle the notification:

```
case MM_MCINOTIFY:
    // Check the status of an AVI movie that might have been playing.
    // By using MCI_NOTIFY, we will get the MCI_NOTIFY_SUCCESSFUL flag
    // if the play completes on it's own.
    switch(wParam){
        case MCI_NOTIFY_SUCCESSFUL:
            // Playing finished, let's rewind and clear our flag
            fPlaying = FALSE;
            mciSendCommand(wMCIDeviceID, MCI_SEEK,
                MCI_SEEK_TO_START, (DWORD)(LPVOID)NULL);
            return 0;
    }
```

The following fragment shows how the notify flag is used with the **play** command. To use the previous fragment to process the notification message, the handle to the window procedure containing it is specified in *hwnd.*

```
MCI_DGV_PLAY_PARMS    mciPlay;
DWORD                dwFlags;

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

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

For the string interface, the following line uses **mciSendString** to send the **play** command and request notification. The *hwnd* parameter in it would also specify the handle to the window procedure containing the handler for notification.

```
mciSendString("play movie notify", NULL, 0, hwnd);
```

Playing AVI files with MCI

To play an AVI file, your application will perform the following actions:

1. Open the AVI file
2. Set up the playback window
3. Play the AVI sequence
4. Optionally change the playback state
5. Optionally get playback information
6. Close the AVI file

Opening an AVI File

To open an AVI file, your application sends the **open** command to MCIAVI. This command lets your application specify the file. If desired, your application can also specify information about the window used for playback.

If your application plans on opening multiple AVI files, it might open the MCIAVI driver initially by specifying the driver identifier and then open each file separately. This saves time because MCI will not load the MCIAVI driver for each file open command.

If your application will open multiple files, it should include routines like **initAVI** and **termAVI** found in MOVPLAY2.C. The application would use **initAVI** during its initialization and **termAVI** during its termination.

```
// Initialize the MCIAVI driver. This returns TRUE if OK, FALSE on error.
BOOL    initAVI(void)
{
    return mciSendString("open avivideo", NULL, 0, NULL) == 0;
}


// Close the MCIAVI driver
void termAVI(void)
{
    mciSendString("close avivideo", NULL, 0, NULL);
}
```

When your application uses a filename to open a device, MCI uses the file extension to locate the driver. For example, the following fragment opens MCIAVI using the file "YOSEMITE.AVI" and the alias movie. Subsequent commands for this file can use the alias movie to reference it.

```
if (mciSendString("open yosemite.avi alias movie", NULL, 0, NULL) == 0){

    // open is OK

} else {

    // handle the error

}
```

The **open** command has options to set some playback window characteristics. These options are covered in the next section. For a full example of using the **open** command, see the **fileOpenMovie** function in MOVPLAY1.C and MOVPLAY2.C.

Setting up the Playback Window

Your application can specify several options to define the playback window for playing the AVI sequence. The following options are available to your application:

- Use the default pop-up window of MCIAVI for playing
- Specify a parent window and window style that MCIAVI can use create the playback window
- Specify a playback window for MCIAVI to use for playback
- Play the AVI sequence on a full screen display

If your application does not specify any window options, MCIAVI creates a default window for playing the sequence. MCIAVI creates this playback window for the **open** command but it does not display the window until your application either sends a command to display the window or sends a command to play the file. This default playback window is a sizable pop-up window with a caption, a thick frame, a system menu, and a minimize box.

The application can also specify a parent window handle and a window style when it opens MCIAVI. When opened this way, MCIAVI creates a window based on these specifications instead of the default pop-up window. Your application can specify any window style available for the **CreateWindow** function. Those styles that require a parent window, like WS_CHILD, should include a parent window handle. The following fragment shows how to use the **open** command to set a parent window and create a child of that window:

```
MCI_DGV_OPEN_PARMS    mciOpen;

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

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

    // open is OK, continue operation

}
```

Your application can also create its own window and supply the handle to MCIAVI with the **window** command. MCIAVI uses this window instead of the one it might have created for playback. The following fragment finds the dimensions needed to play an AVI file, creates a window corresponding to that size, and has MCIAVI to play the file in the window:

```
HWND        hwnd;
MCI_DGV_RECT_PARMS mciRect;

// Get the movie dimensions with MCI_WHERE
mciSendCommand(wDeviceID, MCI_WHERE, MCI_DGV_WHERE_SOURCE,
    (DWORD)(LPSTR)&mciRect);


// Create the playback window. Make it bigger for the border.
hwndMovie = CreateWindow("mywindow", "Playback",
                         WS_CHILD|WS_BORDER, 0,0,
                         mciRect.rc.right+(2*GetSystemMetric(SM_CXBORDER)),
                         mciRect.rc.bottom+(2*GetSystemMetric(SM_CYBORDER)),
                         hwndParent, hInstApp, NULL);
```

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

    MCI_DGV_WINDOW_PARMS    mciWindow;

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

}
```

When MCIAVI creates the playback window or obtains window handle from your application, it does not display the window until your application either plays the sequence or sends a command to display the window. Your application can use the **window** command to display the window without playing the sequence. The "**window movie state show**" command displays the window using the command string interface. The following fragment shows how to display the window using the command message interface:

```
MCI_DGV_WINDOW_PARMS    mciWindow;

mciWindow.nCmdShow = SW_SHOW;    // set command - see ShowWindow()
mciSendCommand(wDeviceID, MCI_WINDOW, MCI_DGV_WINDOW_STATE,
            (DWORD)(LPSTR)&mciWindow;
```

Your application can also play an AVI sequence full screen instead of in a window. To play full screen, modify the **play** command with the **fullscreen** flag. (Use the MCI_MCIAVI_PLAY_FULLSCREEN flag for the message interface.) When your application uses this flag, MCIAVI uses a 320x240 full screen format for playing the sequence. For example, "**play movie fullscreen**" plays a movie full screen.

With the **fullscreen** flag, movies with 160x120 dimensions play back centered in the 320x240 screen. If your application wants to play these moves in a full 320x240 screen, it can use "**play movie fullscreen by 2**" command to stretch the 160x120 movie to full screen.

Playing the AVI Sequence

Playing an AVI sequence is straightforward using the MCI **play** command. This command can play the entire sequence or portions of it. The previous examples show how use the play command with **mciSendString** and **mciSendCommand**.

Changing the Playback State

Your application can control many of the play back capabilities of MCIAVI. The **pause**, **resume**, **stop,** and **seek** commands let your application control the video sequence. Using these, the application can pause a play in progress, seek to a location within the video sequence, and resume play from that point. The following string command examples show how to use these commands:

```
// assume the file was opened with the alias 'movie'

// pause playing
mciSendString("pause movie", NULL, 0, NULL);

// resume play
mciSendString("resume movie", NULL, 0, NULL);

// stop play
mciSendString("stop movie", NULL, 0, NULL);

// seek to the beginning
mciSendString("seek movie to start", NULL, 0, NULL);
```

Your application can use the **seek** command to move the play position to the beginning, the end, or an arbitrary position in the AVI file. There are two seek modes for the MCIAVI driver—exact or non-exact—which affect the seek position. When seek exactly is enabled (**seek exactly on**), MCIAVI seeks exactly to the frame your application specifies. This might cause a delay if the file is temporally encoded and your application does not specify a key frame. With seek exactly disabled (**seek exactly off**), MCIAVI seeks to the nearest key frame in a temporally encoded file. Your application can change the seek mode with the **set** command. The following example shows how to use the string interface to change the seek mode:

```
// Set seek mode with the string interface
// assume the file was opened with the alias 'movie'
void SetSeekMode(BOOL fExact)
{
    if (fExact)

        mciSendString("set movie seek exactly on", NULL, 0, NULL);

    else

        mciSendString("set movie seek exactly off", NULL, 0,
                    NULL);
}
```

Other MCI commands let your application alter the play other than altering the control flow of the play. For example, an AVI sequence by default plays at its normal rate of speed. Your application can change the play rate to speed up or slow down the playback. The **speed** flag for the **set** command lets your application control the play rate. For AVI sequences, a speed value of 1000 is considered normal. Thus, to play a movie at half-speed, your application can use the command string "**set movie speed 500.**" Alternatively, it can use "**set movie speed 2000**" to play the sequence at twice the normal rate.

The **setaudio** command lets your application control the audio portion of an AVI sequence. You application can mute audio during playback, or in the case of multiple audio stream files, select the audio stream played. For example, the "**setaudio movie off**" command string turns audio off during playback. The "**setaudio movie stream to n**" command string specifies the audio stream number (specified by **n**) played for the sequence.

MCIAVI has a dialog box to control some of its playback options. Some of the important option available to the user include selection of windowed or full screen playback, selection of the seek mode, and zooming the image. Your application can have MCIAVI display this dialog box with the **configure** command. For more information on this dialog box, see "MCI String Messages for MCIAVI."

Obtaining Playback Information

Your application can get the status on the playback of an AVI sequence with the **status** command. This command obtains information on the state of the audio, state of the video, mode of the play, position of the play, seek mode, as well as other parameters. Your application might monitor playback so that it can update the state and position of the play in a routine that gets called through a timer call-back. The information returned by the **status** command can depend on the time format used. The device can specify the return values for position, length, and start in milliseconds or frames. Your application can use the **set** command to set alternate time formats and modes. The following fragment shows an example of such a function:

```
MCI_DGV_SET_PARMS       mciSet;
MCI_DGV_STATUS_PARMS    mciStatus;

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

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

// Update mode based on mciStatus.dwReturn

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

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

Closing the AVI File

When finished with a file, your application closes it with the **close** command. With the string interface a "**close movie**" command is sent, with the command interface a **MCI_CLOSE** command is used and all parameters may be NULL.

CHAPTER 7

MCI Command Strings for MCIAVI

Applications such as Visual Basic™ and Asymetrix ToolBook use MCI command strings to provide control for MCI devices. This chapter describes the MCI command strings for the Microsoft MCI video driver (MCIAVI.DRV) that you can use with applications that support the MCI command-string interface. (Applications with this interface send the command strings to MCI with the **mciSendString** function.) For more information on using command strings and the command-string interface, see the *Multimedia Programmer's Guide* and *Multimedia Programmer's Reference* in the Microsoft Windows Software Development Kit. The same information is also available in the *Multimedia Programmer's Workbook* and *Multimedia Programmer's Reference* in the Microsoft Windows Multimedia Development Kit.

The following list summarizes the MCI command strings supported by MCIAVI. This command set is taken from the digital-video command set for MCI. Any device-specific behavior affecting the MCI commands is also noted in this chapter.

About the MCIAVI.DRV Driver

The MCIAVI.DRV driver plays video sequences under the control of MCI commands. These video sequences can contain images, audio, and palettes. The image data is implemented either with color palettes or true-color information.

MCIAVI.DRV supports 11, 22, or 44 kHz audio in an 8- or 16-bit format. Audio is synchronized with the video within 1/30 of a second. However, if audio hardware is not available, the driver will silently play the video sequence. MCIAVI.DRV can drop video frames, if necessary, to play a sequence without audio interruption.

Custom Commands and Flags for MCIAVI.DRV

MCIAVI.DRV uses a subset of the digital-video command set except for the **configure** command and two custom flags used by the **play** command. The **configure**

command displays a dialog box for setting the operating options of the MCIAVI. This dialog box contains the following selections:

Option	Description
Window	Displays the video in a window.
Full Screen	Displays the video using the full screen with 320-by-240 resolution. This allows a 256-color display on 16-color devices.
Zoom By 2	Stretches the video to twice its normal size.
Skip Video Frames If Behind	Specifies to drop frames if the video falls behind. If this isn't selected, all the video frames will be shown and the audio will break up as necessary.
Seek To Nearest Full Frame	Specifies to go to the nearest frame on a **seek**. If this isn't selected, **seek** will go to the closest key frame prior to the specified frame.
Play Only If Audio Device Available	Returns an error if an audio device is not available to play wave data. If this isn't selected and there isn't a wave-audio device available, audio data is ignored and the video is played.
Don't Buffer Offscreen	Specifies that a copy of the display window should not be maintained.

The custom flags used by the **play** command are **fullscreen** and **window**. These flags specify the display mode used for playing a video sequence. These flags are described with the **play** command.

MCI Command Strings

In general, a command string has another field following the command verb not explicitly indicated in the descriptions below. For most commands, this field contains a device name or alias as specified by a prior **open** command. The device name is used by MCI to route the command to the appropriate device driver and device-driver instance.

All commands accept the optional items **wait** and **notify**, although they are not explicitly listed in the command-string table. All commands, except **open** and **close**, also accept the optional item **test**.

The MCIAVI driver uses the **AVIVideo** keyword to identify the driver type.

The MCIAVI driver supports the following command set:

Command	Description	
capability *item*	Fills an application-supplied buffer with a string containing additional information about the capabilities of MCIAVI. The following optional *items* modify **capability**:	
	can eject	Returns **false**.
	can freeze	Returns **false**.
	can lock	Returns **false**.
	can play	Returns **true**.
	can record	Returns **false**.
	can reverse	Returns **false**.
	can save	Returns **false**.
	can stretch	Returns **true**.
	can stretch input	Returns **false**.
	can test	Returns **true**.
	compound device	Returns **true**.
	device type	Returns **digitalvideo**.
	has audio	Returns **true**.
	has still	Returns **false**.
	has video	Returns **true**.
	uses files	Returns **true**.
	uses palettes	Returns **true**.
close	Closes this instance of the MCIAVI and releases all resources associated with it.	
configure	Displays a dialog box used to configure MCIAVI.	
cue *items*	Prepares MCIAVI for playback and leaves it in a paused state. This command is modified by the following optional *items*:	
	output	Prepares MCIAVI for playing.
	to *position*	Positions the workspace to the specified position.

Command	Description	
info *items*	Fills a user-supplied buffer with a string containing information about MCIAVI. The following optional *items* modify **info**:	
	file	Returns the name of the file currently loaded.
	product	Returns **Video for Windows**.
	version	Returns the release level of MCIAVI.
	window text	Returns the text string in the title bar of the window associated with MCIAVI.
open *items*	Initializes MCIAVI. The following *items* modify **open**:	
	alias *alias*	Specifies an *alias* used to reference this instance of MCIAVI.
	elementname	Specifies the name of the device element (file) loaded when MCIAVI opens.
	parent *hwnd*	Specifies the parent of the default window.
	style *stylevalue*	Specifies the style used for the default window. The following constants are defined for *stylevalue*: **overlapped**, **popup**, and **child**.
	type AVIVideo	Specifies the device type of the device element.
pause	Pauses the playing of motion video or audio.	
play *items*	Starts playing the video sequence. The following optional *items* modify **play**:	
	from *position*	Specifies the position to seek to before beginning the **play**.
	to *position*	Specifies the position at which to stop playing.
	fullscreen	Specifies playing should use a full-screen display.
	window	Specifies that playing should use the window associated with a device instance (the default).

Command	Description	
put *items*	Specifies a rectangular region that describes a cropping or scaling option. One of the following *items* must be present to indicate the specific type of rectangle:	
	destination	Specifies that the full client window associated with this instance of MCIAVI is used to show the image or video.
	destination at *rectangle*	Specifies which portion of the client window associated with this instance of MCIAVI is used to show the image or video.
	source	Specifies that the full frame buffer is scaled to fit in the **destination** rectangle.
	source at *rectangle*	Specifies which portion of the frame buffer, in frame-buffer coordinates, is scaled to fit in the **destination** rectangle.
realize *items*	Tells MCIAVI to select and realize its palette into a display context of the displayed window. One of the following *items* modifies **realize**:	
	background	Realizes the palette as a background palette.
	normal	Realizes the palette normally used for a top level window (the default).
	window at *rectangle*	Changes the size and location of the display window. The rectangle specified with the **at** flag is relative to the parent window of the display window (usually the desktop).
resume	Specifies that operation should continue from where it was interrupted by a **pause** command.	
seek *items*	Positions and cues the workspace to the specified position showing the specified frame. One of the following *items* modifies **seek**:	
	to *position*	Specifies the desired new position, measured in units of the current time format.
	to end	Moves the position after the last frame of the workspace.
	to start	Moves the position to the first frame of the workspace.

Command	Description	
set *items*	Sets the state of various control items. One of the following *items* must be included:	
	seek exactly on **seek exactly off**	Selects one of two **seek** modes. With **seek exactly on**, **seek** will always move to the frame specified. With **seek exactly off**, **seek** will move to the closest key frame prior to frame specified.
	speed *factor*	Sets the relative speed of video and audio playback from the workspace. *Factor* is the ratio between the nominal frame rate and the desired frame rate where the nominal frame rate is designated as 1000.
	time format *format*	Sets the time format to *format*. The default time format is **frames**. **Milliseconds** can be abbreviated as **ms**. MCIAVI supports **frames** and **milliseconds**.
	audio off	Disables audio.
	audio on	Enables audio.
	video off	Disables video.
	video on	Enables video.
setaudio *items*	Sets various values associated with audio playback and capture. Only one of the following *items* can be present in a single command, unless otherwise noted:	
	off	Disables audio.
	on	Enables audio.
	volume to *factor*	Sets the average audio volume for both audio channels.
setvideo *items*	Sets various values associated with playback. The following *items* modify **setvideo**:	
	off	Disables video display in the window.
	on	Enables video display in the window.
	palette handle to *handle*	Specifies the handle to a palette.

Command	Description	
signal *items*	Marks a specified position in the workspace. MCIAVI supports only one active signal at a time. The following items modify **signal**:	
	at *position*	Specifies the first frame to be marked.
	cancel	An optional parameter which indicates that the **signal** indicated by the **uservalue** should be removed from the workspace.
	every *interval*	Specifies the period in the current time format after which the succeeding marks should be placed.
	return position	An optional parameter which indicates that the MCIAVI should send the position value instead of the **uservalue** value in the Window message.
	uservalue *id*	Specifies a value associated with this **signal** request that is reported back with the Windows message.
status *item*	Returns status information about this instance MCIAVI. One of the following *items* modifies **status**:	
	audio	Returns **on** if either or both speakers are enabled, and **off** otherwise.
	forward	Returns **true**.
	length	Returns the length of the loaded video sequence in the current time format.
	media present	Returns **true**.
	mode	Returns one of the following: **not ready**, **paused**, **playing**, **recording**, or **stopped**.
	monitor	Returns **file**.
	nominal frame rate	Returns the nominal frame rate associated with the file in units of frames per second times1000.
	number of tracks	Returns the number of tracks in a video sequence (normally 1).
	palette handle	Returns the palette handle.
	position	Returns the current position in the workspace in the current time format.
	ready	Returns **true** if this instance of MCIAVI is ready accept another command.
	reference *frame*	Returns the nearest key-frame number that precedes *frame*.

Command	Description	
	seek exactly	Returns **on** or **off** indicating whether or not **seek exactly** is set.
	speed	Returns the current playback speed.
	start position	Returns the start of the media.
	time format	Returns the current time format (frames or milliseconds).
	unsaved	Returns **false**.
	video	Returns **on** or **off** depending on the most recent **setvideo**.
	window handle	Returns the ASCII decimal value for the window handle associated with this instance of MCIAVI.
	window visible	Returns **true** if the window is not hidden.
	window minimized	Returns **true** if the window is minimized.
	window maximized	Return **true** if the window is maximized.
step *items*	Advances the sequence to the specified image. This command is modified by the following options:	
	by *frames*	Specifies the number of frames to advance before showing another image. You can specify negative values for *frames*.
	reverse	Requests that the **step** be taken in the reverse direction.
stop *item*	Stops playing.	
update *items*	Repaints the current frame into the specified display context. The following *items* modify **update**:	
	at *rect*	Specifies the clipping rectangle relative to the client rectangle.
	hdc *hdc*	Specifies the handle of the display context to paint.
	paint	An application uses the paint flag with **update** when it receives a WM_PAINT message intended for a display DC.

Command	Description	
where *items*	Returns the rectangular region that has been previously specified, or defaulted, using the **put** command. The following *items* modify **where**:	
	destination	Returns a description of the rectangular region used to display video and images in the client area of the current window.
	destination max	Returns the current size of the client rectangle.
	source	Returns a description of the rectangular region cropped from the frame buffer which is stretched to fit the **destination** rectangle on the display.
	source max	Returns the maximum size of the frame buffer.
	window	Returns the current size and position of the display-window frame.
	window max	Returns the size of the entire display.
window *items*	Provides an instance of MCIAVI with a window handle to the window that will be used to display images or motion video. The following *items* modify **window**:	
	handle *hwnd*	Specifies a window to be used with this instance.
	handle default	Specifies that the window associated with this instance should be the default window created during the **open**.
	state *showvalue*	This command issues a **ShowWindow** call for the current window. The following constants are defined for *showvalue*: **hide** **minimize** **restore** **show** **show maximized** **show minimized** **show min noactive** **show na** **show noactivate** **show normal**.
	text *caption*	Specifies the text placed in the title bar of the window.

C H A P T E R 8

MCI Command Messages for MCIAVI

This chapter describes the MCI command messages for the Microsoft MCI video device driver (MCIAVI.DRV) that you can use with the **mciSendCommand** function that supports the MCI command-message interface. For more information on using command messages, see the *Microsoft Multimedia Programmer's Guide* and the *Microsoft Multimedia Programmer's Referenc*e in the Microsoft Windows Software Development Kit. The same information is also available in the *Multimedia Programmer's Workbook* and *Multimedia Programmer's Reference* in the Microsoft Windows Multimedia Development Kit.

The following list summarizes the MCI command messages supported by MCIAVI. This command set is taken from the digital-video command set for MCI. Any device-specific behavior affecting the MCI commands is also noted in this appendix.

MCI Command Messages

All commands accept the optional flags MCI_NOTIFY, MCI_WAIT, and MCI_TEST flags as described in the digital-video command set.

MCIAVI.DRV uses the AVIVideo keyword to identify the driver type.

MCI_CLOSE

This message releases access to a device or device element.

Parameters

DWORD *lParam1*
Specifies the MCI_NOTIFY or MCI_WAIT flag.

LPMCI_GENERIC_PARMS *lParam2*
Specifies a far pointer to the **MCI_GENERIC_PARMS** data structure.

MCI_CONFIGURE

This message displays a dialog box for setting the operating options.

Parameters

DWORD *lParam1*
Specifies the MCI_NOTIFY, MCI_WAIT, and MCI_TEST flags.

LPMCI_GENERIC_PARMS *lParam2*
Specifies a far pointer to the **MCI_GENERIC_PARMS** data structure.

MCI_CUE

This message prepares a device instance so that it can begin playback with minimum delay.

Parameters

DWORD *lParam1*
The following flags apply to MCIAVI.DRV:

MCI_DGV_CUE_OUTPUT
Specifies an instance should be cued for playing.

MCI_TO
Specifies that a workspace position is included in the **dwTo** field of the data structure identified by *lParam2*.

LPMCI_DGV_CUE_PARMS *lParam2*
Specifies a far pointer to the **MCI_DGV_CUE_PARMS** data structure.

MCI_GETDEVCAPS

This message obtains static information about a device.

Parameters

DWORD *lParam1*
The following flags apply to the MCIAVI:

MCI_GETDEVCAPS_ITEM
Specifies that the **dwItem** field of the data structure identified by *lParam2* contains a constant indicating which device capability to obtain. The following constants are recognized by MCIAVI.DRV:

MCI_GETDEVCAPS_CAN_EJECT

MCIAVI.DRV sets the **dwReturn** field to FALSE.

MCI_GETDEVCAPS_CAN_PLAY

MCIAVI.DRV sets the **dwReturn** field to TRUE.

MCI_GETDEVCAPS_CAN_RECORD

MCIAVI.DRV sets the **dwReturn** field to FALSE.

MCI_GETDEVCAPS_CAN_SAVE

MCIAVI.DRV sets the **dwReturn** field to FALSE.

MCI_GETDEVCAPS_COMPOUND_DEVICE

MCIAVI.DRV sets the **dwReturn** field to TRUE.

MCI_GETDEVCAPS_DEVICE_TYPE

MCIAVI.DRV sets the **dwReturn** field to MCI_DEVTYPE_DIGITAL_VIDEO.

MCI_GETDEVCAPS_HAS_AUDIO

MCIAVI.DRV sets the **dwReturn** field to TRUE.

MCI_GETDEVCAPS_HAS_VIDEO

MCIAVI.DRV sets the **dwReturn** field to TRUE.

MCI_GETDEVCAPS_USES_FILES

MCIAVI.DRV sets the **dwReturn** field to TRUE.

MCI_DGV_GETDEVCAPS_CAN_FREEZE

MCIAVI.DRV sets the **dwReturn** field to FALSE.

MCI_DGV_GETDEVCAPS_CAN_LOCK

MCIAVI.DRV sets the **dwReturn** field to FALSE.

MCI_DGV_GETDEVCAPS_CAN_REVERSE

MCIAVI.DRV sets the **dwReturn** field to FALSE.

MCI_DGV_GETDEVCAPS_CAN_STRETCH

MCIAVI.DRV sets the **dwReturn** field to TRUE.

MCI_DGV_GETDEVCAPS_CAN_STR_IN

MCIAVI.DRV sets the **dwReturn** field to FALSE.

MCI_DGV_GETDEVCAPS_CAN_TEST

MCIAVI.DRV sets the **dwReturn** field to TRUE.

MCI_DGV_GETDEVCAPS_HAS_STILL

MCIAVI.DRV sets the **dwReturn** field to FALSE.

MCI_DGV_GETDEVCAPS_PALETTES

MCIAVI.DRV sets the **dwReturn** field to TRUE.

LPMCI_GETDEVCAPS_PARMS *lParam2*
Specifies a far pointer to the **MCI_GETDEVCAPS_PARMS** data structure.

MCI_INFO

This message obtains string information from a device.

Parameters

DWORD *lParam1*
The following flags apply to MCIAVI.DRV:

MCI_INFO_PRODUCT
MCIAVI.DRV returns Video for Windows.

MCI_DGV_INFO_TEXT
Returns the text string in the title bar of the window associated with the device instance.

MCI_INFO_FILE
Obtains the path and filename of the last file specified with the **MCI_OPEN** command.

MCI_INFO_VERSION
Returns the release level of the device driver and hardware.

LPMCI_DGV_INFO_PARMS *lParam2*
Specifies a far pointer to the **MCI_DGV_INFO_PARMS** data structure.

MCI_OPEN

This message initializes an instance of the device or device element.

Parameters

DWORD *lParam1*
The following flags apply to MCIAVI.DRV:

MCI_OPEN_ALIAS
Specifies that an alias is referenced in the **lpstrAlias** field of the data structure identified by *lParam2*.

MCI_OPEN_TYPE
Specifies that a device-type constant or a pointer to a device-type name is included in the **lpstrDeviceType** field of the data structure identified by *lParam2*.

MCI_OPEN_TYPE_ID
Specifies that the **lpstrDeviceType** field of the data structure identified by *lParam2* contains a standard MCI device-type ID and the optional ordinal index for the device.

MCI_OPEN_ELEMENT
Specifies that an element name is included in the **lpstrElementName** field of the data structure identified by *lParam2*.

MCI_OPEN_ELEMENT_ID
Specifies that the **lpstrElementName** field of the data structure identified by *lParam2* has meaning defined by the device.

MCI_DGV_OPEN_PARENT
Indicates the parent window handle is specified in the **hWndParent** field of the data structure identified by *lParam2*.

MCI_DGV_OPEN_WS
Indicates a window style is specified in the **dwStyle** field of the data structure identified by *lParam2*.

LPMCI_DGV_OPEN_PARMS *lParam2*
Specifies a far pointer to the **MCI_DGV_OPEN_PARMS** data structure.

MCI_PAUSE

This message pauses the current action.

Parameters

DWORD *lParam1*
Specifies the MCI_NOTIFY, MCI_TEST, and MCI_WAIT flags.

LPMCI_DGV_PAUSE_PARMS *lParam2*
Specifies a far pointer to the **MCI_DGV_PAUSE_PARMS** data structure.

MCI_PLAY

This message begin play of the audio and video.

Parameters

DWORD *lParam1*
The following flags apply to MCIAVI.DRV:

MCI_FROM
Specifies that a starting position is included in the **dwFrom** field of the data structure identified by *lParam2*.

MCI_TO
Specifies that an ending position is included in the **dwTo** field of the data structure identified by *lParam2*. MCIAVI.DRV returns an error if the “to” position is less than the “from” position.

MCI_DGV_PLAY_WINDOW
Specifies that playing should occur in the window associated with a device instance (the default). (This flag is specific to the MCIAVI.DRV.)

MCI_MCIAVI_PLAY_FULLSCREEN
Specifies that playing should use a full-screen display, typically, with a 320-by-200 resolution. The full-screen display takes over the entire desktop. (This flag is specific to MCIAVI.DRV.)

LPMCI_DGV_PLAY_PARMS *lParam2*
Specifies a far pointer to an **MCI_DGV_PLAY_PARMS** data structure.

MCI_PUT

This message specifies a rectangular region that describes a cropping or scaling option.

Parameters

DWORD *lParam1*
The following flags apply to MCIAVI.DRV:

MCI_DGV_RECT
Specifies that the **rc** field of the data structure identified by *lParam2* contains a valid rectangle.

MCI_DGV_PUT_DESTINATION
Indicates the rectangle defined for MCI_DGV_RECT specifies a destination rectangle. The destination rectangle specifies the portion of the client window associated with this device driver instance that shows the image or video.

MCI_DGV_PUT_SOURCE
Indicates the rectangle defined for MCI_DGV_RECT specifies a source rectangle. The source rectange specifies which portion of the frame buffer is to be scaled to fit into the destination rectangle.

MCI_DGV_PUT_WINDOW
Indicates that the rectangle defined for MCI_DGV_RECT applies to the display window. This rectangle is relative to the parent window of the display window (usually the desktop). If the window is not specified, it defaults to the initial window size and position.

LPMCI_DGV_PUT_PARMS *lParam2*
Specifies a far pointer to a **MCI_DGV_PUT_PARMS** data structure.

MCI_REALIZE

This message tells MCIAVI to select and realize its palette into a display context of the displayed window. You should use this message when your application receives the WM_QUERYNEWPALETTE message from Windows.

Parameters

DWORD *lParam1*
The following flags apply to MCIAVI.DRV:

MCI_DGV_REALIZE_BKGD
Realizes the palette as a background palette.

MCI_DGV_REALIZE_NORM
Realizes the palette normally (the default).

LPMCI_GENERIC_PARMS *lParam2*
Specifies a far pointer to a **MCI_GENERIC_PARMS** data structure.

MCI_RESUME

This message resumes MCIAVI operation when it is paused .

Parameters

DWORD *lParam1*
Specifies the MCI_NOTIFY, MCI_WAIT, and MCI_TEST flags.

LPMCI_GENERIC_PARMS *lParam2*
Specifies a far pointer to the **MCI_GENERIC_PARMS** data structure.

MCI_SEEK

This message positions and cues the workspace to the specified position showing the specified frame.

Parameters

DWORD *lParam1*
The following flags apply to MCIAVI.DRV:

MCI_SEEK_TO_END
Specifies the seek should move to the end of the workspace.

MCI_SEEK_TO_START
Specifies the seek should move to the beginning of the workspace.

MCI_TO
Specifies a position is included in the **dwTo** field of the **MCI_SEEK_PARMS** data structure.

LPMCI_SEEK_PARMS *lParam2*
Specifies a far pointer to the **MCI_SEEK_PARMS** data structure.

MCI_SET

This message sets device information.

Parameters

DWORD *lParam1*
The following flags apply to MCIAVI.DRV:

MCI_SET_AUDIO
Specifies an audio-channel number is included in the **dwAudio** field of the data structure identified by *lParam2*. This flag must be used with MCI_SET_ON or MCI_SET_OFF. Specify the constant MCI_SET_AUDIO_ALL for the channel number.

MCI_SET_TIME_FORMAT
Specifies a time-format parameter is included in the **dwTimeFormat** field of the data structure identified by *lParam2*. Constants defined for time formats include:

MCI_FORMAT_FRAMES

Specifies frames.

MCI_FORMAT_MILLISECONDS

Specifies milliseconds.

MCI_SET_VIDEO
Sets the video signal on or off. This flag must be used with either MCI_SET_ON or MCI_SET_OFF.

MCI_SET_ON
Enables the video or audio channel, or enables the seek-exactly mode.

MCI_SET_OFF
Disables the video or audio channel, or disables the seek-exactly mode.

MCI_DGV_SET_SEEK_EXACTLY
Sets the format used for positioning. This flag must be used with MCI_SET_ON or MCI_SET_OFF.

MCI_DGV_SET_SPEED
Specifies that a speed parameter is included in the **dwSpeed** field of the data structure identified by *lParam2*.

LPMCI_DGV_SET_PARMS *lParam2*
Specifies a far pointer to the **MCI_DGV_SET_PARMS** data structure.

MCI_SETAUDIO

This messageets various values associated with audio playback and capture.

Parameters

DWORD *lParam1*

The following flags apply to MCIAVI.DRV:

MCI_DGV_SETAUDIO_ITEM

Indicates an audio constant is specified in the **dwAdustParm** field of the data structure identified by *lParam2*. The following constant is supported by MCIAVI.DRV:

MCI_DGV_SETAUDIO_VOLUME

Indicates that the audio level is specified as a factor in the **dwValue** field of the data structure identified by *lParam2*. The volume level ranges between 0 and 1000.

MCI_DGV_SETAUDIO_VALUE

Indicates that an audio value is specified in the **dwValue** field of the data structure identified by *lParam2*.

MCI_SET_ON

Enables the audio channel.

MCI_SET_OFF

Disables the audio channel.

LPMCI_DGV_SETAUDIO_PARMS *lParam2*

Specifies a far pointer to the **MCI_DGV_SETAUDIO_PARMS** data structure.

MCI_SETVIDEO

This message sets various values associated with video playback.

Parameters

DWORD *lParam1*

The following flags apply to MCIAVI.DRV:

MCI_DGV_SETVIDEO_ITEM

Indicates a video constant is specified in the **dwAdustParm** field of the data structure identified by *lParam2*. MCIAVI.DRV supports the following constant:

MCI_DGV_SETVIDEO_PALHANDLE

Indicates that a palette-handle value is specified in the **dwValue** field of the data structure identified by *lParam2*.

MCI_DGV_SETVIDEO_SRC_VALUE

Specifies a value is included in the **dwValue** field of the data structure identified by *lParam2*.

MCI_SET_ON

Enables video output.

MCI_SET_OFF

Disables video output.

LPMCI_DGV_SETVIDEO_PARMS *lParam2*
Specifies a far pointer to the **MCI_DGV_SETVIDEO_PARMS** data structure.

MCI_SIGNAL

This message sets a specified position in the workspace. MCIAVI.DRV supports only one active signal at a time.

Parameters

DWORD *lParam1*
The following flags apply to all devices supporting **MCI_SIGNAL** :

MCI_DGV_SIGNAL_AT
Specifies a signal position is included in the **dwPosition** field of the data structure identified by *lParam2*.

MCI_DGV_SIGNAL_EVERY
Specifies a signal-period value is included in the **dwEvery** field of the data structure identified by *lParam2*.

MCI_DGV_SIGNAL_USERVAL
Specifies a data value is included in the **dwUserParm** field of the data structure identified by *lParam2*. The data value associated with this request is reported back with the Windows message.

MCI_DGV_SIGNAL_CANCEL
Removes the signal position specified by the value associated with the MCI_DGV_SIGNAL_USERVAL flag.

MCI_DGV_SIGNAL_POSITION
Specifies that the device should send the position value with the Windows message instead of the user-specified value.

LPMCI_DGV_SIGNAL_PARMS *lParam2*
Specifies a far pointer to the **MCI_DGV_SIGNAL_PARMS** structure.

MCI_STATUS

This message obtains information about an instance of an MCI device.

Parameters

DWORD *lParam1*
The following flags apply to MCIAVI.DRV:

MCI_STATUS_ITEM
Specifies that the **dwItem** field of the data structure identified by *lParam2* contains a constant specifying which status item to obtain. MCIAVI.DRV supports the following constants:

MCI_STATUS_LENGTH

MCIAVI.DRV sets the **dwReturn** field to the media length. (It returns an error for any track but 1.)

MCI_STATUS_MODE

MCIAVI.DRV sets the **dwReturn** field to the current mode of the device.

MCI_STATUS_NUMBER_OF_TRACKS

MCIAVI.DRV sets the **dwReturn** field to 1.

MCI_STATUS_POSITION

MCIAVI.DRV sets the **dwReturn** field to the position of the track. (It returns an error for any track but 1.)

MCI_STATUS_READY

MCIAVI.DRV sets the **dwReturn** field to TRUE if the device is ready to accept another command.

MCI_STATUS_TIME_FORMAT

MCIAVI.DRV sets the **dwReturn** to the current time format.

MCI_DGV_STATUS_AUDIO

MCIAVI.DRV sets the **dwReturn** field to MCI_ON or MCI_OFF, depending on the most recent MCI_SET_AUDIO option for the **MCI_SET** command.

MCI_DGV_STATUS_FILEFORMAT

MCIAVI.DRV returns the constant for AVI RIFF in the **dwReturn** field.

MCI_DGV_STATUS_FORWARD

MCIAVI.DRV sets the **dwReturn** field to TRUE.

MCI_DGV_STATUS_MEDIA_PRESENT

MCIAVI.DRV sets the **dwReturn** field to TRUE.

MCI_DGV_STATUS_MONITOR

MCIAVI.DRV sets the **dwReturn** field to MCI_DGV_MONITOR_FILE.

MCI_DGV_STATUS_HPAL

MCIAVI.DRV sets the **dwReturn** field to the current palette handle.

MCI_DGV_STATUS_HWND

MCIAVI.DRV sets the **dwReturn** field to the window handle.

MCI_DGV_STATUS_NOMINAL_RATE

MCIAVI.DRV sets the **dwReturn** field to the nominal frame rate associated with the file.

MCI_DGV_STATUS_SIZE

MCIAVI.DRV sets the **dwReturn** field to zero.

MCI_DGV_STATUS_SEEK_EXACTLY

MCIAVI.DRV sets the **dwReturn** field to TRUE or FALSE indicating whether or not seek exactly is set.

MCI_DGV_STATUS_SPEED

MCIAVI.DRV sets the **dwReturn** field to the current playback speed.

MCI_DGV_STATUS_UNSAVED

MCIAVI.DRV sets the **dwReturn** field to FALSE.

MCI_DGV_STATUS_VIDEO

MCIAVI.DRV indicates whether the video is enabled or disabled in the **dwReturn** field.

MCI_DGV_STATUS_WINDOW_VISIBLE

MCIAVI.DRV sets the **dwReturn** field to TRUE if the window is not hidden.

MCI_DGV_STATUS_WINDOW_MINIMIZED

MCIAVI.DRV sets the **dwReturn** field to TRUE if the window is minimized.

MCI_DGV_STATUS_WINDOW_MAXIMIZED

MCIAVI.DRV sets the **dwReturn** field to TRUE if the window is maximized.

MCI_STATUS_START
Obtains the starting position of the media. To get the starting position, combine this flag with MCI_STATUS_ITEM and set the **dwItem** field of the data structure, identified by *lParam2*, to MCI_STATUS_POSITION.

MCI_DGV_STATUS_REFERENCE
The **dwReference** field returns the nearest previous keyframe.

LPMCI_DGV_STATUS_PARMS *lParam2*
Specifies a far pointer to the **MCI_DGV_STATUS_PARMS** data structure.

MCI_STEP

This message steps the player one or more frames.

Parameters

DWORD *lParam1*
The following flags apply to MCIAVI.DRV:

MCI_DGV_STEP_FRAMES
Indicates that the **dwFrames** field of the data structure identified by *lParam2* specifies the number of frames to advance before displaying another image.

MCI_DGV_STEP_REVERSE
Steps in reverse.

LPMCI_DGV_STEP_PARMS *lParam2*
Specifies a far pointer to the **MCI_DGV_STEP_PARMS** data structure.

MCI_STOP

This message stops all play sequences and ceases display of video images.

Parameters

DWORD *lParam1*
Specifies the MCI_NOTIFY, MCI_WAIT, and MCI_TEST flags.

LPMCI_DGV_STOP_PARMS *lParam2*
Specifies a far pointer to the **MCI_DGV_STOP_PARMS** data structure.

MCI_UPDATE

This message repaints the current frame into the specified display context.

Parameters

DWORD *lParam1*

The following flags apply to MCIAVI.DRV:

MCI_UPDATE_HDC
Specifies that the **hDC** field of the data structure identified by *lParam2* contains a valid window of the display context to paint.

MCI_DGV_UPDATE_PAINT
An application uses this flag when it receives a WM_PAINT message that is intended for a display DC. A frame-buffer device will usually paint the key color. If the display device does not have a frame buffer, it might ignore the MCI_UPDATE message when the MCI_DGV_UPDATE_PAINT flag is used, because the display will be repainted during the playback operation.

LPMCI_DGV_UPDATE_PARMS *lParam2*

Specifies a far pointer to a **MCI_DGV_UPDATE_PARMS** data structure.

MCI_WHERE

This message returns the rectangular region that has been specified with the **MCI_PUT** command.

Parameters

DWORD *lParam1*

The following flags apply to MCIAVI.DRV:

MCI_DGV_WHERE_DESTINATION
Obtains a description of the rectangular region used to display video and images in the client area of the current window.

MCI_DGV_WHERE_SOURCE
Obtains a description of the rectangular region (cropped from the frame buffer) that is stretched to fit the destination rectangle on the display.

MCI_DGV_WHERE_MAX
When used with MCI_DGV_WHERE_DESTINATION or MCI_DGV_WHERE_SOURCE, the rectangle returned indicates the maximum width and height of the specified region.

MCI_DGV_WHERE_WINDOW
Obtains a description of the display window frame.

LPMCI_DGV_RECT_PARMS *lParam2*

Specifies a far pointer to a **MCI_DGV_RECT_PARMS** data structure.

MCI_WINDOW

This message specifies the window and the window characteristics for graphic devices.

Parameters

DWORD *lParam1*

The following flags apply to MCIAVI.DRV:

MCI_DGV_WINDOW_HWND

Indicates that the handle of the window needed for use as the destination is included in the **hWnd** field of the data structure identified by *lParam2*.

MCI_DGV_WINDOW_STATE

Indicates the **nCmdShow** field of the **MCI_DGV_WINDOW_PARMS** data structure contains parameters for setting the window state.

MCI_DGV_WINDOW_TEXT

Indicates the **lpstrText** field of the **MCI_DGV_WINDOW_PARMS** data structure contains a pointer to a buffer containing the caption used in the window title bar.

LPMCI_DGV_WINDOW_PARMS *lParam2*

Specifies a far pointer to a **MCI_DGV_WINDOW_PARMS** data structure.

MM_MCISIGNAL

This messag is sent to a window to notify an application that an MCI device has reached a position defined in a previous MCI_SIGNAL to the device.

Parameters

WORD *wParam*

Contains the ID of the device initiating the signal message.

LONG *lParam*

Normally this contains the value passed in **dwUserParm** when the MCI_SIGNAL message has defined this callback. Alternatively, it might contain the position value.

Data Structures for MCI Command Messages

The following data structures are used by the MCI command messages for MCIAVI.DRV.

MCI_DGV_CUE_PARMS

The **MCI_DGV_CUE_PARMS** structure contains parameters used by the **MCI_CUE** message for digital video devices. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwTo;
} MCI_DGV_CUE_PARMS;
```

Fields

The **MCI_DGV_CUE_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwTo
Specifies the cue position.

MCI_DGV_INFO_PARMS

The **MCI_DGV_INFO_PARMS** structure contains parameters used by the **MCI_INFO** message for digital video devices. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD  dwCallback;
    LPSTR  lpstrReturn;
    DWORD  dwRetSize;
} MCI_DGV_INFO_PARMS;
```

Fields

The **MCI_DGV_INFO_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

lpstrReturn
Specifies a long pointer to a user-supplied buffer for the return string.

dwRetSize
Specifies the size in bytes of the buffer for the return string.

MCI_DGV_OPEN_PARMS

The **MCI_DGV_OPEN_PARMS** structure contains information used by **MCI_OPEN** message for digital video devices. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD dwCallback;
    WORD  wDeviceID;
    WORD  wReserved0;
    LPSTR lpstrDeviceType;
    LPSTR lpstrElementName;
    LPSTR lpstrAlias;
    DWORD dwStyle;
    WORD  hWndParent;
    WORD  wReserved1;
} MCI_DGV_OPEN_PARMS;
```

Fields The **MCI_DGV_OPEN_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

wDeviceID
Contains the device ID returned to user.

wReserved0
Reserved.

lpstrDeviceType
Specifies the name or constant ID of the device type.

lpstrElementName
Specifies the device-element name (usually a path).

lpstrAlias
Specifies an optional device alias.

dwStyle
Specifies the window style.

hWndParent
Specifies the handle to use as the window parent.

wReserved1
Reserved.

MCI_DGV_PAUSE_PARMS

The **MCI_DGV_PAUSE_PARMS** structure contains information used by the **MCI_PAUSE** command. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD  dwCallback;
} MCI_DGV_PAUSE_PARMS;
```

Fields The **MCI_DGV_PAUSE_PARMS** structure has the following field:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

MCI_DGV_PLAY_PARMS

The **MCI_DGV_PLAY_PARMS** structure contains parameters use by the **MCI_PLAY** message for digital video devices. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwFrom;
    DWORD  dwTo;
} MCI_DGV_PLAY_PARMS;
```

Fields The **MCI_DGV_PLAY_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwFrom
Specifies the position to play from.

dwTo
Specifies the position to play to.

MCI_DGV_PUT_PARMS

The **MCI_DGV_PUT_PARMS** structure contains parameters used by the **MCI_PUT** message for digital video devices. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD  dwCallback;
    RECT   rc;
} MCI_DGV_PUT_PARMS;
```

Fields The **MCI_DGV_PUT_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Specifies a rectangle.

MCI_DGV_SIGNAL_PARMS

The **MCI_DGV_SIGNAL_PARMS** structure contains parameters for the **MCI_SIGNAL** message used by digital video devices. When assigning data to the fields in this data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field.

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwPosition;
    DWORD  dwPeriod;
    DWORD  dwUserParm;
} MCI_DGV_SIGNAL_PARMS;
```

Fields

The **MCI_DGV_SIGNAL_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwPosition
Specifies the position to be marked.

dwPeriod
Specifies the interval of the position marks.

dwUserParm
Specifies a value associated with the signals being set.

MCI_DGV_RECT_PARMS

The **MCI_DGV_RECT_PARMS** structure contains parameters used by the **MCI_FREEZE**, **MCI_PUT**, **MCI_UNFREEZE**, and **MCI_WHERE** messages for digital video devices. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD  dwCallback;
    RECT  rc;
} MCI_DGV_RECT_PARMS;
```

Fields

The **MCI_DGV_RECT_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Specifies a rectangle.

MCI_DGV_SET_PARMS

The **MCI_DGV_SET_PARMS** structure contains parameters used by the **MCI_SET** message for digital video devices. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwTimeFormat;
    DWORD  dwAudio;
    DWORD  dwFileFormat;
    DWORD  dwSpeed;
} MCI_DGV_SET_PARMS;
```

Fields

The **MCI_DGV_SET_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwTimeFormat
Specifies the time format used by the device.

dwAudio
Specifies the channel used for audio output.

dwFileFormat
Specifies the file format.

dwSpeed
Specifies the playback speed.

MCI_DGV_SETAUDIO_PARMS

The **MCI_DGV_SETAUDIO_PARMS** structure contains parameters used by the **MCI_SETAUDIO** message for digital video devices. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwItem;
    DWORD  dwValue;
    DWORD  dwOver;
    LPSTR  lpstrAlgorithm;
    LPSTR  lpstrQuality;
} MCI_DGV_SETAUDIO_PARMS;
```

Fields

The **MCI_DGV_SETAUDIO_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwItem
Specifies the constant indicating the target adjustment.

dwValue
Specifies the adjustment level.

dwOver
Specifies the transition-length parameter.

lpstrAlgorithm
Specifies a long pointer to a null-terminated string containing the name of the audio-compression algorithm.

lpstrQuality
Specifies a long pointer to a null-terminated string containing a descriptor of the audio-compression algorithm.

MCI_DGV_SETVIDEO_PARMS

The **MCI_DGV_SETVIDEO_PARMS** structure contains parameters used by the **MCI_SETVIDEO** message for digital video devices. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwItem;
    DWORD  dwValue;
    DWORD  dwOver;
    LPSTR  lpstrQuality;
    LPSTR  lpstrAlgorithm;
    DWORD  dwSourceNumber;
} MCI_DGV_SETVIDEO_PARMS;
```

Fields The **MCI_DGV_SETVIDEO_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwItem
Specifies the constant indicating the target adjustment.

dwValue
Specifies the adjustment level.

dwOver
Specifies the transition-length parameter.

lpstrQuality
Specifies a long pointer to a null-terminated string containing a descriptor of the video-compression algorithm.

lpstrAlgorithm
Specifies a long pointer to a null-terminated string containing the name of the video-compression algorithm.

dwSourceNumber
Specifies the index of input source.

MCI_DGV_STATUS_PARMS

The **MCI_DGV_STATUS_PARMS** structure contains parameters used by the **MCI_STATUS** message for digital video devices. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwReturn;
    DWORD  dwItem;
    DWORD  dwTrack;
    LPSTR  lpstrDrive;
    DWORD  dwReference;
} MCI_DGV_STATUS_PARMS;
```

Fields The **MCI_DGV_STATUS_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwReturn
Contains the return information on exit.

dwItem
Identifies the capability being queried.

dwTrack
Specifies the length or number of tracks.

lpstrDrive
Specifies the approximate amount of disk space that can be obtained by a **MCI_RESERVE** command.

dwReference
Specifies the approximate location of nearest, previous intraframe-encoded image.

MCI_DGV_STEP_PARMS

The **MCI_DGV_STEP_PARMS** structure contains parameters used by the **MCI_STEP** message for digital video devices. When assigning data the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwFrames;
} MCI_DGV_STEP_PARMS;
```

Fields The **MCI_DGV_STEP_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwFrames
Specifies the number of frames to step.

MCI_DGV_STOP_PARMS

The **MCI_DGV_STOP_PARMS** structure contains the information used by **MCI_STOP** command message for digital video devices. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD  dwCallback;
} MCI_DGV_STOP_PARMS;
```

Fields The **MCI_DGV_STOP_PARMS** structure has the following field:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

MCI_DGV_UPDATE_PARMS

The **MCI_DGV_UPDATE_PARMS** structure contains parameters used by the **MCI_UPDATE** message. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD dwCallback;
    RECT  rc;
    HDC   hDC;
    WORD  wReserved0;
} MCI_DGV_UPDATE_PARMS;
```

Fields The **MCI_DGV_UPDATE_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

rc
Specifies a rectangle.

hDC
Specifies a handle to a display context.

wReserved0
Reserved.

MCI_DGV_WINDOW_PARMS

The **MCI_DGV_WINDOW_PARMS** structure contains parameters used by the **MCI_WINDOW** message for digital video devices. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD  dwCallback;
    WORD   hWnd;
    WORD   wReserved1;
    WORD   nCmdShow;
    WORD   wReserved2;
    LPSTR  lpstrText;
} MCI_DGV_WINDOW_PARMS;
```

Fields The **MCI_DGV_WINDOW_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

hWnd
Specifies a handle to the display window.

wReserved1
Reserved.

nCmdShow
Specifies how the window is displayed.

wReserved2
Reserved.

lpstrText
Specifies a long pointer to a null-terminated string containing the window caption.

MCI_GENERIC_PARMS

The **MCI_GENERIC_PARMS** structure contains the information used by MCI command messages that have empty parameter lists. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

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

Fields The **MCI_GENERIC_PARMS** structure has the following field:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

MCI_GETDEVCAPS_PARMS

The **MCI_GETDEVCAPS_PARMS** structure contains parameters for the **MCI_GETDEVCAPS** message. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

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

Fields

The **MCI_GETDEVCAPS_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwReturn
Contains the return information on exit.

dwItem
Identifies the capability being queried.

MCI_SEEK_PARMS

The **MCI_SEEK_PARMS** structure contains parameters used by the **MCI_SEEK** message. When assigning data to the fields in the following data structure, set the corresponding MCI flags in the *lParam1* parameter of **mciSendCommand** to validate each field:

```
typedef struct {
    DWORD  dwCallback;
    DWORD  dwTo;
} MCI_SEEK_PARMS;
```

Fields

The **MCI_SEEK_PARMS** structure has the following fields:

dwCallback
The low-order word specifies a window handle used for the MCI_NOTIFY flag.

dwTo
Specifies the seek position.

CHAPTER 9

Video Capture Application Reference

This section is an alphabetic reference to the functions and data structures provided by Video for Windows for use by video capture applications. There are separate sections for functions, messages, and data structures. The messages and data structures are defined in MSVIDEO.H.

Extensions are being added to the video capture functions to make it easier for applications to access video capture drivers. If your application needs video capture services, it should use the extensions rather than these functions. If you are developing video capture device drivers, you might use these functions for testing your drivers.

If you need information on the video capture extensions to develop your application, you can request the latest information from the following group:

Microsoft Corporation
Multimedia Systems Group
Product Marketing
One Microsoft Way
Redmond, WA 98052-6399

FAX: (206) 93MSFAX

Video Capture Function Reference

This section contains a listing of the functions used by video capture applications. The function definition is given, followed by a description of each parameter.

Video Capture Function Summary

The following function operates on a single frame:

videoFrame
This function transfers a single frame from or to a video device channel.

The following functions are used to open, close, and communicate with a video capture device:

videoClose
This function closes the specified video device channel.

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

videoMessage
This function sends messages to a video device channel.

videoOpen
This function opens a channel on the specified video device.

The following functions control the configuration of a video capture device:

videoConfigure
This function sets or retrieves a configurable driver option.

videoConfigureStorage
This function saves or loads all configurable options for a channel.

videoDialog
This function displays a channel specific dialog box used to set configuration parameters.

videoGetChannelCaps
This function retrieves a description of the capabilities of a channel.

videoGetNumDevs
This function returns the number of MSVIDEO devices installed.

videoUpdate
This function directs a channel to repaint the display.

The following functions control video capture streaming:

videoStreamAddBuffer
This function sends a buffer to a video device channel.

videoStreamFini
This function terminates streaming from the specified device channel.

videoStreamGetError
This function returns the most recent error encountered.

videoStreamGetPosition
This function retrieves the current position of the specified video device channel.

videoStreamInit
This function initializes a video device channel for streaming.

videoStreamPrepareHeader
This function prepares a buffer for video streaming.

videoStreamReset
This function stops streaming on the specified video device channel, returns all video buffers from the driver, and resets the current position to zero.

videoStreamStart
This function starts streaming on the specified video device channel.

videoStreamStop
This function stops streaming on a video channel.

videoStreamUnprepareHeader
This function cleans up the preparation performed by **videoStreamPrepareHeader**.

Video Capture Function Alphabetic Reference

videoClose

Syntax DWORD **videoClose**(*hVideo*)

This function closes the specified video device channel.

Parameters HVIDEO *hVideo*
Specifies a handle to the video device channel. If the function is successful, the handle will be invalid after this call.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_INVALHANDLE
Specified device handle is invalid.

DV_ERR_NONSPECIFIC
The driver failed to close the channel.

Comments If buffers have been sent with **videoStreamAddBuffer** and they haven't been returned to the application, the close operation fails. You can use **videoStreamReset** to mark all pending buffers as done.

See Also videoOpen, videoStreamInit, videoStreamFini, videoStreamReset

videoConfigure

Syntax DWORD **videoConfigure**(*hVideo*, *msg*, *dwFlags*, *lpdwReturn*, *lpData1*, *dwSize1*, *lpData2*,

This function sets or retrieves a configurable driver option.

Parameters HVIDEO *hVideo*
Specifies a handle to the video device channel.

UINT *msg*
Specifies the option to set or retrieve.

DVM_PALETTE
Indicates a palette is being sent to the driver or retrieved from the driver.

DVM_PALETTERGB555
Indicates an RGB555 palette is being sent to the driver.

DVM_FORMAT
Indicates format information is being sent to the driver or retrieved from the driver.

DWORD *dwFlags*
Specifies flags for configuring or interrogating the device driver. The following flags are defined:

VIDEO_CONFIGURE_SET
Indicates values are being sent to the driver.

VIDEO_CONFIGURE_GET
Indicates values are being obtainded from the driver.

VIDEO_CONFIGURE_QUERY
This flag is used to determine if the driver supports the option specified by *msg*. This flag should be combined with either the VIDEO_CONFIGURE_SET or VIDEO_CONFIGURE_GET flag. If this flag is set, the *lpData1*, *dwSize1*, *lpData2*, and *dwSize2* parameters are ignored.

VIDEO_CONFIGURE_QUERYSIZE
Returns the size, in bytes, of the configuration option in *lpdwReturn*. This flag is only valid if the VIDEO_CONFIGURE_GET flag is also set.

VIDEO_CONFIGURE_CURRENT
Requests the current value. This flag is only valid if the VIDEO_CONFIGURE_GET flag is also set.

VIDEO_CONFIGURE_NOMINAL
Requests the nominal value. This flag is only valid if the VIDEO_CONFIGURE_GET flag is also set.

VIDEO_CONFIGURE_MIN
Requests the minimum value. This flag is only valid if the VIDEO_CONFIGURE_GET flag is also set.

VIDEO_CONFIGURE_MAX
Get the maximum value. This flag is only valid if the VIDEO_CONFIGURE_GET flag is also set.

LPDWORD lpdwReturn
Points to a DWORD used for returning information from the driver. If the VIDEO_CONFIGURE_QUERYSIZE flag is set, *lpdwReturn* is filled with the size of the configuration option.

LPVOID *lpData1*
Specifies a pointer to message specific data.

DWORD *dwSize1*
Specifies the size of the *lpData1* buffer in bytes.

LPVOID *lpData2*
Specifies a pointer to message specific data.

DWORD *dwSize2*
Size of the *lpData2* buffer in bytes.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_INVALHANDLE
Specified device handle is invalid.

DV_ERR_NOTSUPPORTED
Function is not supported.

See Also videoOpen, videoMessage

videoConfigureStorage

Syntax DWORD **videoConfigureStorage**(*hVideo*, *lpstrIdent*, *dwFlags*)

This function saves or loads all configurable options for a channel. Options can be saved and recalled for each application or each application instance.

Parameters HVIDEO *hVideo*
Specifies a handle to the video device channel.

LPSTR lpstrIdent
Identifies the application or instance. Use an arbitrary string which uniquely identifies your application or instance.

DWORD *dwFlags*
Specifies flags for the storage. The following flags are defined:

VIDEO_CONFIGURE_GET
Requests that the values be loaded.

VIDEO_CONFIGURE_SET
Requests that the values be saved.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_INVALHANDLE
Specified device handle is invalid.

DV_ERR_NOTSUPPORTED
Function is not supported.

Comments The method used by a driver to save configuration options is device dependent.

See Also videoOpen

videoDialog

Syntax DWORD **videoDialog**(*hVideo*, *hWndParent*, *dwFlags*)

This function displays a channel-specific dialog box used to set configuration parameters.

Parameters

HVIDEO *hVideo*
Specifies a handle to the video device channel.

HWND hWndParent
Specifies the parent window handle.

DWORD *dwFlags*
Specifies flags for the dialog box. The following flag is defined:

VIDEO_DLG_QUERY
If this flag is set, the driver immediately returns zero if it supplies a dialog box for the channel, or DV_ERR_NOTSUPPORTED if it does not.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_INVALHANDLE
Specified device handle is invalid.

DV_ERR_NOTSUPPORTED
Function is not supported.

Comments

Typically, each dialog box displayed by this function lets the user select options appropriate for the channel. For example, a VIDEO_IN channel dialog box lets the user select the image dimensions and bit depth.

See Also

videoOpen, videoConfigureStorage

videoFrame

Syntax

DWORD **videoFrame**(*hVideo*, *lpVHdr*)

This function transfers a single frame from or to a video device channel.

Parameters

HVIDEO *hVideo*
Specifies a handle to the video device channel. The channel must be of type VIDEO_IN or VIDEO_OUT.

LPVIDEOHDR *lpVHdr*
Specifies a far pointer to an **VIDEOHDR** structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_INVALHANDLE
Specified device handle is invalid.

Comments

Use this function with a VIDEO_IN channel to transfer a single image from the frame buffer. Use this function with a VIDEO_OUT channel to transfer a single image to the frame buffer.

To determine the size of the buffer needed for capturing data, use **videoConfigure**. Set the size of the buffer in the **dwBufferLength** field of the **VIDEOHDR** structure. If the buffer size is not large enough, the function can fail.

See Also

videoOpen

videoGetChannelCaps

Syntax

DWORD **videoGetChannelCaps**(*hVideo*, *lpChannelCaps*, *dwSize*)

This function retrieves a description of the capabilities of a channel.

Parameters

HVIDEO *hVideo*
Specifies a handle to the video device channel.

LPCHANNEL_CAPS *lpChannelCaps*
Specifies a far pointer to a **CHANNEL_CAPS** structure.

DWORD *dwSize*
Specifies the size of the **CHANNEL_CAPS** structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_UNSUPPORTED
Function is not supported.

Comments

The channel capabilities structure returns the capability information. For example, capability information might include whether or not the channel can crop and scale images.

videoGetErrorText

Syntax

DWORD **videoGetErrorText**(*hVideo*, *wError*, *lpText*, *wSize*)

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

Parameters

HVIDEO *hVideo*
Specifies a handle to the video device channel. This might be NULL if the error is not device specific.

UINT *wError*
Specifies the error number.

LPSTR *lpText*
Specifies a far pointer to a buffer which is filled with a null-terminated string corresponding to the error number.

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

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_BADERRNUM
Specified error number is out of range.

Comments

If the error description is longer than the buffer, the description is truncated. The returned error string is always null-terminated. If *wSize* is zero, nothing is copied and the function returns zero.

videoGetNumDevs

Syntax DWORD **videoGetNumDevs**()

This function returns the number of MSVIDEO devices installed.

Parameters None

Return Value Returns the number of MSVIDEO devices listed in the [drivers] section of the SYSTEM.INI file.

See Also videoOpen

videoMessage

Syntax DWORD **videoMessage**(*hVideo*, *wMsg*, *dwP1*, *dwP2*)

This function sends a message to a video device channel.

Parameters

HVIDEO *hVideo*
Specifies the handle to the device.

UINT *wMsg*
Specifies the message to send.

DWORD *dwP1*
Specifies the first parameter for the message.

DWORD *dwP2*
Specifies the second parameter for the message.

Return Value Returns the message specific value returned from the driver.

Comments This function is used for configuration messages such as **DVM_SRC_RECT** and **DVM_DST_RECT** and device specific messages.

See Also videoConfigure

videoOpen

Syntax DWORD **videoOpen**(*lphvideo*, *dwDeviceID*, *dwFlags*)

This function opens a channel on the specified video device.

Parameters

LPHVIDEO *lphvideo*
Specifies a far pointer to a **HVIDEO** handle. The video capture driver uses this location to return a handle that uniquely identifies the opened video device channel. Use this handle to identify the device channel when calling other video functions.

DWORD dwDeviceID
Identifies the video device to open. The value of *dwDeviceID* varies from zero to one less than the number of video capture devices installed in the system.

DWORD *dwFlags*
Specifies flags for opening the device. The following flags are defined:

VIDEO_EXTERNALIN
Specifies the channel is opened for external input. Typically, external input channels capture images into a frame buffer.

VIDEO_EXTERNALOUT
Specifies the channel is opened for external output. Typically, external output channels display images stored in a frame buffer on an auxiliary monitor or overlay.

VIDEO_IN
Specifies the channel is opened for video input. Video input channels transfer images from a frame buffer to system memory buffers.

VIDEO_OUT
Specifies the channel is opened for video output. Video output channels transfer images from system memory buffers to a frame buffer.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_BADDEVICEID
Specified device ID is out of range.

DV_ERR_ALLOCATED
Specified resource is already allocated.

DV_ERR_NOMEM
Unable to allocate or lock memory.

Comments At a minimum, all capture drivers support a **VIDEO_EXTERNALIN** and a **VIDEO_IN** channel. Use **videoGetNumDevs** to determine the number of video devices present in the system.

See Also videoClose, videoGetNumDevs

videoStreamAddBuffer

Syntax DWORD **videoStreamAddBuffer**(*hVideo*, *lpvideoHdr*, *dwSize*)

This function sends a buffer to a video device channel. After the buffer is filled by the device, the device sends it back to the application.

Parameters HVIDEO *hVideo*
Specifies a handle to the video device channel.

LPVIDEOHDR *lpvideoHdr*
Specifies a far pointer to a **VIDEOHDR** structure that identifies the buffer.

DWORD *dwSize*
Specifies the size of the **VIDEOHDR** structure.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_INVALHANDLE
The device handle specified is invalid.

DV_ERR_UNPREPARED
The *lpvideoHdr* structure hasn't been prepared.

Comments

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

To determine the size of the buffer needed for capturing data, use **videoConfigure**. Set the size of the buffer in the **dwBufferLength** field of the **VIDEOHDR** structure. If the buffer size is not large enough, the driver might return DV_ERR_NONSPECIFIC.

See Also

videoStreamPrepareHeader

videoStreamFini

Syntax

DWORD **videoStreamFini**(*hVideo*)

This function terminates streaming from the specified device channel.

Parameters

HVIDEO *hVideo*
Specifies a handle to the video device channel.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_INVALHANDLE
The device handle specified is invalid.

DV_ERR_STILLPLAYING
There are still buffers in the queue.

Comments

If there are buffers that have been sent with **videoStreamAddBuffer** that haven't been returned to the application, this operation will fail. Use **videoStreamReset** to mark all pending buffers as done.

Each call to **videoStreamInit** must be matched with a call to **videoStreamFini**.

For **VIDEO_EXTERNALIN** channels, this function is used to halt capturing of data to the frame buffer.

For **VIDEO_EXTERNALOUT** channels that support overlay, this function is used to disable the overlay.

See Also

videoStreamInit

videoStreamGetError

Syntax

DWORD **videoStreamGetError**(*hVideo*, *lpdwErrorID*, *lpdwErrorValue*)

This function returns the error most recently encountered.

Parameters

HVIDEO *hVideo*
Specifies a handle to the video device channel.

LPDWORD lpdwErrorID
Specifies a far pointer to the DWORD to be filled with the error ID.

LPDWORD lpdwErrorValue
Specifies a far pointer to the DWORD to be filled with the number of frames skipped.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_INVALHANDLE
The device handle specified is invalid.

Comments

While streaming video data, a capture driver can fill buffers faster than the client application can save the buffers to disk. In this case, the **DV_ERR_NO_BUFFERS** error is returned in *lpdwErrorID* and *lpdwErrorValue* contains a count of the number of frames missed. After receiving this message and returning the error status, the driver should reset its internal error flag to **DV_ERR_OK** and the count of missed frames to zero.

Applications should send this message frequently during capture since some drivers which do not have access to interrupts use this message to trigger buffer processing.

See Also

videoOpen

videoStreamGetPosition

Syntax

DWORD **videoStreamGetPosition**(*hVideo*, *lpInfo*, *dwSize*)

This function retrieves the current position of the specified video device channel.

Parameters

HVIDEO *hVideo*
Specifies a handle to the video device channel.

LPMMTIME *lpInfo*
Specifies a far pointer to an **MMTIME** structure.

DWORD *dwSize*
Specifies the size of the **MMTIME** structure.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_INVALHANDLE
Specified device handle is invalid.

Comments

Before using **videoStreamGetPosition**, set the **wType** field of the **MMTIME** structure to indicate the time format you desire. After **videoStreamGetPosition** returns, check the **wType** field to determine if the your time format is supported. If not, **wType** will specify an alternate format. Video capture drivers typically provide the milliseconds time format.

The position is set to zero when streaming is started with **videoStreamStart**.

videoStreamInit

Syntax

DWORD **videoStreamInit**(*hVideo*, *dwMicroSecPerFrame*, *dwCallback*, *dwCallbackInstance*, *dwFlags*)

This function initializes a video device channel for streaming.

Parameters

HVIDEO *hVideo*
Specifies a handle to the video device channel.

DWORD dwMicroSecPerFrame
Specifies the number of microseconds between frames.

DWORD dwCallback
Specifies the address of a callback function or a handle to a window called during video streaming. The callback function or window processes messages related to the progress of streaming.

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

DWORD *dwFlags*
Specifies flags for opening the device channel. The following flags are defined:

CALLBACK_WINDOW
If this flag is specified, *dwCallback* is a window handle.

CALLBACK_FUNCTION
If this flag is specified, *dwCallback* is a callback procedure address.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_BADDEVICEID
The device ID specified in *hVideo* is not valid.

DV_ERR_ALLOCATED
The resource specified is already allocated.

DV_ERR_NOMEM
Unable to allocate or lock memory.

Comments

If a window is chosen to receive callback information, the following messages are sent to the window procedure function to indicate the progress of video input: **MM_DRVM_OPEN** at the time of videoStreamInit, **MM_DRVM_CLOSE** at the time of videoStreamFini, **MM_DRVM_DATA** when a buffer of image data is available, **MM_DRVM_ERROR** when an error occurs.

If a function is chosen to receive callback information, the following messages are sent to the function to indicate the progress of video input: **MM_DRVM_OPEN**, **MM_DRVM_CLOSE**, **MM_DRVM_DATA**, **MM_DRVM_ERROR**. The callback function must reside in a DLL. You do not have to use **MakeProcInstance** to get a procedure-instance address for the callback function.

Callback

void CALLBACK **videoFunc**(*hVideo*, *wMsg*, *dwInstance*, *dwParam1*, *dwParam2*)

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

Callback Parameters

HVIDEO *hVideo*
Specifies a handle to the video device channel associated with the callback.

DWORD *wMsg*
Specifies the **MM_DRVM_** message.

DWORD *dwInstance*
Specifies the user instance data specified with **videoOpen**.

DWORD *dwParam1*
Specifies a parameter for the message.

DWORD *dwParam2*
Specifies a parameter for the message.

Callback Comments

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

For **VIDEO_EXTERNALIN** channels, this function is used to initiate capturing of data to the frame buffer.

For **VIDEO_EXTERNALOUT** channels which support overlay, this function is used to enable the overlay.

See Also

videoOpen, videoStreamFini, videoClose

videoStreamPrepareHeader

Syntax

DWORD **videoStreamPrepareHeader**(*hVideo*, *lpvideoHdr*, *dwSize*)

This function prepares a buffer for video streaming.

Parameters

HVIDEO *hVideo*
Specifies a handle to the video device channel.

LPVIDEOHDR *lpvideoHdr*
Specifies a pointer to a **VIDEOHDR** structure that identifies the buffer to be prepared.

DWORD *dwSize*
Specifies the size of the **VIDEOHDR** structure.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_INVALHANDLE
Specified device handle is invalid.

DV_ERR_NOMEM
Unable to allocate or lock memory.

Comments Use this function after **videoStreamInit** or after **videoStreamReset** to prepare the data buffers for streaming data.

The **VIDEOHDR** data structure and the data block pointed to by its **lpData** field must be allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with **GlobalLock**. Preparing a header that has already been prepared will have no effect and the function will return zero. Typically, this function is used to insure that the buffer will be available for use at interrupt time.

See Also videoStreamUnprepareHeader

videoStreamReset

Syntax DWORD **videoStreamReset**(*hVideo*)

This function stops streaming on the specified video device channel and resets the current position to zero. All pending buffers are marked as done and are returned to the application.

Parameters HVIDEO *hVideo*
Specifies a handle to the video device channel.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_INVALHANDLE
The device handle specified is invalid.

See Also videoStreamReset, videoStreamStop, videoStreamAddBuffer, videoStreamClose

videoStreamStart

Syntax DWORD **videoStreamStart**(*hVideo*)

This function starts streaming on the specified video device channel.

Parameters HVIDEO *hVideo*
Specifies a handle to the video device channel.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_INVALHANDLE
The device handle specified is invalid.

See Also videoStreamReset, videoStreamStop, videoStreamAddBuffer, videoStreamClose

videoStreamStop

Syntax DWORD **videoStreamStop**(*hVideo*)

This function stops streaming on a video channel.

Parameters HVIDEO *hVideo*
Specifies a handle to the video device channel.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_INVALHANDLE
Specified device handle is invalid.

Comments If there are any buffers in the queue, the current buffer will be marked as done (the **dwBytesRecorded** field in the **VIDEOHDR** header will contain the actual length of data), but any empty buffers in the queue will remain there. Calling this function when the channel is not started has no effect, and the function returns zero.

See Also videoStreamStart, videoStreamReset

videoStreamUnprepareHeader

Syntax DWORD **videoStreamUnprepareHeader**(*hVideo*, *lpvideoHdr*, *dwSize*)

This function cleans up the preparation performed by **videoStreamPrepareHeader**.

Parameters HVIDEO *hVideo*
Specifies a handle to the video device channel.

LPVIDEOHDR *lpvideoHdr*
Specifies a pointer to a **VIDEOHDR** structure identifying the data buffer to be cleaned up.

DWORD *dwSize*
Specifies the size of the **VIDEOHDR** structure.

Return Value Returns zero if the function was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_INVALHANDLE
The device handle specified is invalid.

DV_ERR_STILLPLAYING
The structure identified by *lpvideoHdr* is still in the queue.

Comments

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

See Also

videoStreamPrepareHeader

videoUpdate

Syntax

DWORD **videoUpdate**(*hVideo*, *hWnd*, *hDC*)

This function directs a channel to repaint the display. It applies only to VIDEO_EXTERNALOUT channels.

Parameters

HVIDEO *hVideo*
Specifies a handle to the video device channel.

HWND *hWnd*
Specifies the handle of the window to be used by the channel for image display.

HDC *hDC*
Specifies a handle to a device context.

Return Value

Returns zero if the function was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_UNSUPPORTED
Specified message is unsupported.

Comments

This message is normally sent whenever the client window receives a **WM_MOVE**, **WM_SIZE**, or **WM_PAINT** message.

Video Capture Data Structure Reference

This section lists data structures used by video capture applications. The data structures are presented in alphabetical order. The structure definition is given, followed by a description of each field.

Video Capture Data Structure Alphabetic Reference

CHANNEL_CAPS

The **CHANNEL_CAPS** structure is used with **videoGetChannelCaps** to return the capabilities of a channel to an application.

```
typedef struct channel_caps_tag {
    DWORD  dwFlags;
    DWORD  dwSrcRectXMod;
    DWORD  dwSrcRectYMod;
    DWORD  dwSrcRectWidthMod;
    DWORD  dwSrcRectHeightMod;
    DWORD  dwDstRectXMod;
    DWORD  dwDstRectYMod;
    DWORD  dwDstRectWidthMod;
    DWORD  dwDstRectHeightMod;
} CHANNEL_CAPS;
```

Fields

The **CHANNEL_CAPS** structure has the following fields:

dwFlags
Returns flags giving information about the channel. The following flags are defined:

VCAPS_OVERLAY
Indicates the channel is capable of overlay. This flag is used only for VIDEO_EXTERNALOUT channels.

VCAPS_SRC_CAN_CLIP
Indicates that the source rectangle can be set smaller than the maximum dimensions.

VCAPS_DST_CAN_CLIP
Indicates that the destination rectangle can be set smaller than the maximum dimensions.

VCAPS_CAN_SCALE
Indicates that 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

Some channels can only use source and destination rectangles which fall on 2, 4, or 8 pixel boundaries. Similarly, some channels only accept capture rectangles widths and heights that are multiples of a fixed value. Rectangle dimensions indicated by modulus operators are considered advisory. When requesting a particular rectangle, the application must always check the return value to insure the request was accepted by the driver. For example, if **dwDstRectWidthMod** is set to 64, the application might try to set destination rectangles with widths of 64, 128, 192, 256, ..., and 640 pixels. The driver might actually support a subset of these sizes and indicates the supported sizes with the return value of the **DVM_DST_RECT** message. If a channel supports arbitrarily positioned rectangles, with arbitrary sizes, the values above should all be set to 1.

VIDEOHDR

The **VIDEOHDR** structure defines the header used to identify a video data buffer.

```
typedef struct videohdr_tag {
    LPSTR  lpData;
    DWORD  dwBufferLength;
    DWORD  dwBytesUsed;
    DWORD  dwTimeCaptured;
    DWORD  dwUser;
    DWORD  dwFlags;
    DWORD  dwReserved[4];
} VIDEOHDR;
```

The **VIDEOHDR** structure has the following fields:

lpData
Specifies a far pointer to the video data buffer.

dwBufferLength
Specifies the length of the data buffer.

dwBytesUsed
Specifies the number of bytes used in the data buffer.

dwTimeCaptured
Specifies the time (in milliseconds) when the frame was captured relative to the first frame in the stream.

dwUser
Specifies 32 bits of user data.

dwFlags
Specifies flags giving information about the data buffer. The following flags are defined for this field:

VHDR_DONE
Set by the device driver to indicate it is finished with the data buffer and it is returning the buffer to the application.

VHDR_PREPARED
Set by Windows to indicate the data buffer has been prepared with **videoStreamPrepareHeader**.

VHDR_INQUEUE
Set by Windows to indicate the data buffer is queued for playback.

VHDR_KEYFRAME
Set by the device driver to indicate a key frame.

dwReserved[4]
Reserved for use by the device driver. Typically, these maintain a linked list of buffers in the queue.

CHAPTER 10

Video Compression and Decompression Drivers

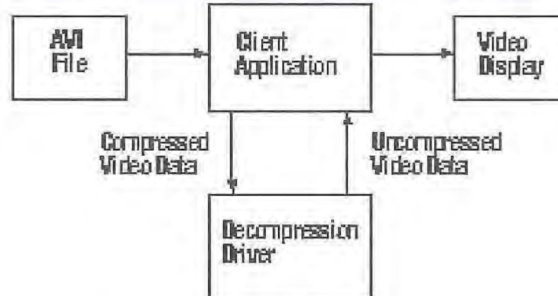
Video compression and decompression drivers provide low-level video compression and decompression services for Microsoft Video for Windows. The compression and decompression algorithms used can be hardware or software based. This chapter explains the Windows interface for these drivers. It covers the following topics:

- General information about writing a video compression and decompression driver
- Information on how a video compression and decompression driver handles the system messages for the installable driver interface
- Information on how a video compression and decompression driver handles messages specific to compressing and decompressing video data
- An alphabetical reference to the messages and data structures used to write video compression and decompression drivers

Before reading this chapter, you should be familiar with the video services available with Windows. You should also be familiar with the Windows installable driver interface. For information about these Windows topics, see the *Microsoft Windows Programmer's Reference*.

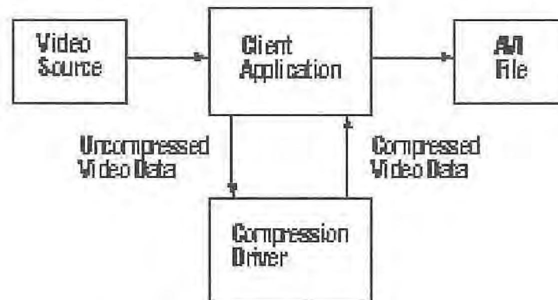
Architecture of a Video Compression and Decompression Driver

The following two block diagrams show the architecture of a video compression and decompression driver. While the diagrams show separate compression and decompression drivers, an actual driver usually combines both functions. The following illustration shows the architecture of a decompression driver:



Architecture for a decompression driver.

The following illustration shows a similar architecture for the compression driver:



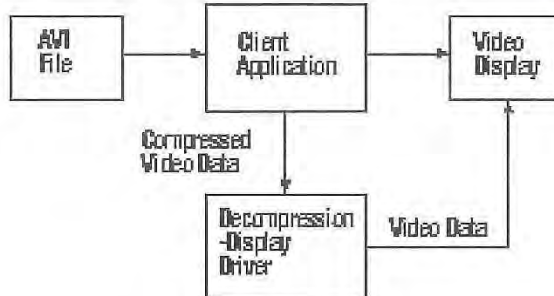
Architecture for a compression driver.

The decompression driver and compression driver blocks represent your compression and decompression driver. The client-application block represents the system and application software that uses the services of your compression and decompression driver. Application software will always use the system software to access compression and decompression drivers.

The source of information used for decompression is represented by the AVI file block. Other sources of images can be used in place of this block. AVI files are RIFF files that contain audio and video data. The client-application maintains the RIFF format when it reads and writes the file. (Your driver will send and receive video data. The client-application will add and remove the RIFF tags.)

Compression drivers receive uncompressed data from the video source. Typically the video source is a disk file but it could also come from other video sources such as a video capture device. The video data can be either still bitmaps or motion video frames.

While a previous block diagram showed the decompression driver returning the uncompressed video to the client-application, your driver can have the capability to write directly to the display or display driver. These devices can replace a Windows video driver or work in conjunction with it. The following illustration shows a decompression driver with the ability to write to the video display:



Architecture for a decompression-video 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.

The ICSAMPLE Example Driver

The examples in this chapter were extracted from the ICSAMPLE example driver. This sample illustrates the interface between Windows and video compression and decompression drivers. The sample compresses data by extracting every tenth pixel from the source and discarding the other nine. It decompresses by replacing the nine missing pixels with their retained neighbor.

The Structure of a Video Compression and Decompression Driver

Video compression and decompression drivers are dynamic-link libraries (DLLs) usually written in C or assembly language, or a combination of the two languages.

As installable drivers, these drivers will provide a **DriverProc** entry point. For general information about installable drivers, the **DriverProc** entry point, and system messages sent to this entry point, see the *Microsoft Windows Programmer's Reference*. This chapter includes supplemental information for the system messages. This information describes specifically how compression and decompression drivers should respond to the system messages that are critical to their proper operation.

Video compression and decompression drivers also use the **DriverProc** entry point to process messages specifically for video compression and decompression. Information on how drivers use the **DriverProc** entry point to process these messages is contained in this chapter.

Video Compression and Decompression Header Files

The messages and data structures used exclusively by video compression and decompression drivers are defined in COMPDDK.H.

Naming Video Compression and Decompression Drivers

The filenames for driver DLLs are not required to have a file extension of ".DLL"—you can name your driver using any file extension you want. It is suggested that you use the extension ".DRV" for your drivers to follow the convention set by Windows.

SYSTEM.INI Entries for Video Compression and Decompression Drivers

The SYSTEM.INI file contains information for loading and configuring drivers. Your driver must be identified in the [Drivers] or [Installable Compressors] section. This entry lets Windows load the driver. If an entry for your driver is absent, it won't be recognized. While installation applications normally add the necessary entry for completed drivers, you might have to manually add it while you develop your driver. The final version of your driver should use an installation application to create and delete the entries in these two sections.

Identify your driver in the [Drivers] section if you want to use the Drivers option of the Control Panel to install or configure it. (This is the recommended method of installation.) The Drivers option obtains the information it needs to install the driver from an OEMSETUP.INF file you create for your driver. This file should be included on the distribution disk for your driver. For information about the files needed to install your driver, see the *Microsoft Windows Device Driver Adaptation Guide* and *Microsoft Windows Programmer's Reference*. For information on the Drivers option, see the *Microsoft Windows Programmer's Reference*.

Identify your driver in the [Installable Compressors] section if you want to use a custom installation application. If you use a custom application, it should update the [Installable Compressors] section when your driver is installed or removed.

Video compression and decompression drivers are identified by a key name of "VIDC." followed by its four-character code identifier. For example, the following [Installable Compressors] section of SYSTEM.INI identifies one video compression and decompression driver:

```
[Installable Compressors]
VIDC.SAMP = ICSAMPLE.DRV
```

SAMP is the four-character code identifier of the compressor. This driver has a file name of "ICSAMPLE.DRV".

The four-character code identifier must be unique. If you want to create a new four-character code identifier, register it with Microsoft to set up a standard definition and avoid any conflicts with other codes that might be defined. To register a code for a compression and decompression driver, request a *Multimedia Developer Registration Kit* from the following group:

Microsoft Corporation
Multimedia Systems Group
Product Marketing

One Microsoft Way
Redmond, WA 98052-6399

For more information on four character codes, see the *Microsoft Windows Multimedia Programmer's Guide*, *Microsoft Windows Multimedia Programmer's Reference*, and Chapter 4, "AVI Files."

For more information on storing configuration information in the SYSTEM.INI file, see "The Installable Driver Interface," later in this chapter.

The Module-Definition File

To build a driver DLL, you must have a module-definition (.DEF) file. In this file, you must export the **DriverProc** entry-point function. Functions are exported by ordinal, as shown in the following example ICSAMPLE.DEF file:

```
LIBRARY      ICSAMPLE

DESCRIPTION 'VIDC.SAMP:Sample Decompression Driver'

STUB         'WINSTUB.EXE'
EXETYPE      WINDOWS

CODE         MOVEABLE  DISCARDABLE  LOADONCALL
DATA         MOVEABLE  SINGLE  PRELOAD

SEGMENTS      _TEXT   DISCARDABLE   PRELOAD

HEAPSIZE     128

EXPORTS
      WEP
      DriverProc
```

If you are using the Drivers option of the Control Panel, include the key name of VIDC, a period (.), and the four-character code for your driver in the DESCRIPTION entry. (Use a colon (:) to separate this entry from the text description.) The Drivers option uses this description when it lists the driver in the [Drivers] section of the SYSTEM.INI file. For example, the previous description for ICSAMPLE.DRV uses VIDC.SAMP: in the DESCRIPTION line. If you are using a custom installation application, you do not need to include this description information.

For more information on the entry-point function listed in this example, see "An Example DriverProc Entry-Point Function" later in this chapter.

The Module Name Line

The module name line should specify a unique module name for your driver. Windows will not load two different modules with the same module name. It's a good idea to use the base of your driver filename since, in certain instances, **LoadLibrary** will assume that to be your module name. For example, the previous fragment used LIBRARY ICSAMPLE.

The Installable Driver Interface

The entry-point function, **DriverProc**, 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** should return ICERR_UNSUPPORTED for any messages that it does not handle.

Note: Your driver should respond to all system messages. If supplemental information is not provided for them in this chapter, use the definitions provided in the *Microsoft Windows Programmer's Reference*.

An Example DriverProc Entry-Point Function

A video compression and decompression driver uses the **DriverProc** function for its entry-point. The following example is derived from the ICSAMPLE driver:

```
LRESULT CALLBACK _loadds DriverProc(DWORD dwDriverID, HDRVR hDriver,
UINT uiMessage, LPARAM lParam1, LPARAM lParam2)
{
    INSTINFO *pi = (INSTINFO *)(UINT)dwDriverID;

    switch (uiMessage)
    {

       case DRV_LOAD:
            return (LRESULT) Load();

       case DRV_FREE:
            Free();
            return (LRESULT)1L;

       case DRV_OPEN:
            // If being opened without an open structure, return a non-zero
            // value without actually opening.
            if (lParam2 == 0L)
                return 0xFFFF0000;
```

```
            return (LRESULT)(DWORD)(WORD)Open((ICOPEN FAR *) lParam2);

        case DRV_CLOSE:
            if (pi)
                Close(pi);

            return (LRESULT)1L;

      /*********************************************************************
          system configuration messages
       *********************************************************************/

        case DRV_QUERYCONFIGURE:    // For configuration with Drivers option.
            return (LRESULT)0L;

        case DRV_CONFIGURE:
            return DRV_OK;


     /*********************************************************************
          device specific messages
      *********************************************************************/

            .
            .
            .


     /*********************************************************************
          standard driver messages
      *********************************************************************/


         case DRV_DISABLE:
         case DRV_ENABLE:
             return (LRESULT)1L;

         case DRV_INSTALL:
         case DRV_REMOVE:
             return (LRESULT)DRV_OK;
     }

     if (uiMessage << DRV_USER)
         return DefDriverProc(dwDriverID, hDriver,
              uiMessage,lParam1,lParam2);
     else
         return ICERR_UNSUPPORTED;
}
```

Handling the DRV_OPEN and DRV_CLOSE Messages

Like other installable drivers, client applications must open a video compression and decompression driver before using it, and they must close it when finished using it 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 do this, 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 far pointer to an ICOPEN data structure. The ICOPEN data structure has the following fields:

```
typedef struct {
     DWORD  fccType;
     DWORD  fccHandler;
     DWORD  dwVersion;
     DWORD  dwFlags;
 } ICOPEN;
```

The **fccType** field specifies a four-character code representing the type of stream being compressed or decompressed. For video streams, this will be 'vidc'.

Because video capture drivers can rely on video compression and decompression drivers for efficient operation, a single driver can handle both video capture, and video compression and decompression services. Video capture drivers use the VIDEO_OPEN_PARMS data structure when it is opened. This structure has the same field definitions as the ICOPEN structure. By examining the **fccType** field, a combined driver can determine whether it is being opened as a video capture driver or a video compression and decompression driver. Video capture devices contain the four-character code 'vcap' in this field. For more information on video capture drivers, see Chapter 11, “Video Capture Device Drivers.”

Other drivers that require close coordination with video compression and decompression drivers can also be combined with video compression and decompression drivers if they use a similar interface.

The **fccHandler** field specifies a four-character code identifying a specific compressor. The client-application obtains the four-character code from the entry in the SYSTEM.INI file used to open your driver. Your driver should not fail the open if it does not recognize the four-character code.

The **dwVersion** field specifies the version of the compressor interface used to open the driver. Your driver can use this information to determine the capabilities of the system software when future versions of it are available.

The **dwFlags** field contains a constant indicating the function of the driver. The following constants are defined:

ICMODE_COMPRESS
The driver is opened to compress data.

ICMODE_DECOMPRESS
The driver is opened to decompress data.

ICMODE_DRAW
The device driver is opened to decompress data directly to hardware.

ICMODE_QUERY
The driver is opened for informational purposes, rather than for actual compression.

The ICMODE_COMPRESS, ICMODE_DECOMPRESS, and ICMODE_DRAW flags indicate your driver is opened to compress or decompress data. Depending on the flag, your driver should prepare to handle ICM_COMPRESS, ICM_DECOMPRESS, or ICM_DRAW messages. Your driver should also prepare to handle all messages used to configure and interrogate your driver.

The ICMODE_QUERY flag indicates your driver is opened to obtain information. It should prepare to handle the ICM_ABOUT, ICM_GETINFO, and ICM_GETDEFAULTQUALITY messages.

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

The following system messages are used by video compression and decompression drivers for hardware configuration:

DRV_QUERYCONFIGURE
This system message is sent to determine if the driver supports configuration.

DRV_CONFIGURE
This Control Panel message is sent to let the driver display a custom configuration dialog box for hardware configuration.

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 a section with the same name as the driver in the user's SYSTEM.INI file.

Like other installable drivers, your driver will receive DRV_QUERYCONFIGURE and DRV_CONFIGURE messages from the Drivers option of the Control Panel. If your driver controls hardware that needs to be configured, it should return a non-zero 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 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. The following messages apply specifically to video compression and decompression drivers:

ICM_CONFIGURE
This message displays a custom configuration dialog box for driver configuration.

ICM_GETSTATE
This message obtains the current driver configuration.

ICM_SETSTATE
This message sets the state of the compressor.

If your driver is configurable, it should support the ICM_CONFIGURE message for driver configuration. In addition, 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 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. For information on the ICM_DECOMPRESS_GET_FORMAT message, see "Decompressing Video Data" later in this chapter.

Messages Used to Interrogate the Driver

The client-application uses the following messages to obtain or display information about your driver:

ICM_ABOUT
This message displays the about dialog box for the driver.

ICM_GETINFO
This message obtains information about the driver.

The client-application sends the ICM_ABOUT message to display your driver's about box. If the client-application sets *dwParam1* to -1, it wants to know if 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 tells the client-application what capabilities the driver supports. The ICINFO structure has the following fields:

```
typedef struct {
     DWORD  dwSize;
     DWORD  fccType;
     DWORD  fccHandler;
     DWORD  dwFlags;
     DWORD  dwVersion;
     DWORD  dwVersionICM;
     char  szName[16];
     char  szDescription[128];
     char  szDriver[128];
 } ICINFO;
```

Set the **dwSize** field to the size of the ICINFO structure.

Set the **fccType** field to the four-character code to 'vidc' for video streams.

Set **fccHandler** to the four-character code identifying your driver. Your driver should use the four-character code used to install your driver and used in the description line of the .DEF file.

Specify the version number of the driver in the **dwVersion** field.

Set the **dwVersionICM** field to 1.0 (0x00010000). This specifies the version of the compression manager supported by this driver.

Use the **szName[16]** field to specify the short name of the compressor. The null-terminated name should be suitable for use in list boxes.

Use the **szDescription[128]** field to specify a null-terminated string containing a long name for the compressor.

Your driver will not normally use the **szDriver[128]** field. This field 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** field. You can use the high-ordered word for driver-specific flags. The following flags are defined for video compression and decompression drivers:

VIDCF_QUALITY
The driver supports quality levels.

VIDCF_CRUNCH
The driver supports compressing to an arbitrary frame size.

VIDCF_TEMPORAL
The driver supports inter-frame compression.

VIDCF_DRAW
The driver supports drawing to hardware with the ICM_DRAW messages.

VIDCF_FASTTEMPORAL
The driver can do temporal compression and doesn't need the previous frame.

Configuration Messages for Compression Quality

The client-application sends the following messages to obtain and set image quality values:

ICM_GETDEFAULTQUALITY
This message obtains the default quality settings of the driver.

ICM_GETQUALITY
This message obtains the current driver quality settings.

ICM_SETQUALITY
This message sets the driver quality settings.

For the video compression and decompression interface, quality is indicated by an integer ranging from 0 to 10,000. A quality level of 7,500 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 10,000 that corresponds to a good default quality level for your compressor. Your 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 sends the following messages to obtain the key frame rate and size of the buffer queue desired by the device driver:

ICM_GETDEFAULTKEYFRAMERATE
This message obtains the default key frame rate of the driver used during compression.

ICM_GETBUFFERSWANTED
This message obtains the number of buffers the driver wants for pre-buffering when drawing data.

The client-application uses ICM_GETDEFAULTKEYFRAMERATE to obtain the drivers 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 ICM_COMPRESS_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 wants to maintain a queue of buffers. Your driver might maintain a queue of buffers if it renders the decompressed data and it wants 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.

Video Compression and Decompression Messages

This section discusses the driver specific messages for video compression and decompression. The messages are covered by the three basic operations of these drivers: video compression, video decompression using the client-application, and video decompression directly to video hardware. Because video compression and decompression drivers typically use AVI files and bitmaps, this section includes a brief overview of the AVI RIFF format, the BITMAPINFO data structure, and the BITMAPINFOHEADER data structure.

About the AVI File Format

Many of the video compression and decompression messages rely on information embedded in the AVI RIFF file. Drivers do not typically access this information directly. They rely on the client-application to read and write the AVI file and maintain the RIFF file structure. While your driver should not have to manipulate an AVI file, understanding its structure helps identify the purpose of the information your driver will supply and receive.

AVI files have the following general structure:

```
RIFF('AVI'
    LIST('hdrl'
        avih(<<MainAVIHeader>>)
         LIST ('strl'
            strh(<<Stream header>>)
            strf(<<Stream format>>)
            strd(<<Stream data>>)
        )
    )

    LIST('movi'
      '00??'(<<driver Data>>)
        .
        .
        .
     '00??'(<<driver Data>>)
    )
    ' idx1'(<<AVIIndex>>)
)
```

The following table summarizes the entries in the AVI RIFF file:

RIFF Chunk	Description
RIFF 'AVI '	Identifies the file as AVI RIFF file.
LIST 'hdrl'	Identifies a chunk containing subchunks that define the format of the data.
'avih'	Identifies a chunk containing general information about the file. This includes the number of streams and the width and height of the AVI sequence.
LIST 'strl'	Identifies a chunk containing subchunks that describe the streams in a file. This chunk exists for each stream.
'strh'	Identifies a chunk containing a stream header. This includes the type of stream.
'strf'	Identifies a chunk describing the format of the data in the stream. For video streams, the information in this chunk is a BITMAPINFO structure. It includes palette information if appropriate.
'strd'	Identifies a chunk containing information used by compressor and decompressors. For video compressors and decompressors, this includes the state formation.
LIST 'movi '	Identifies a chunk containing subchunks used for the audio and video data.
'00??'	Identifies a chunk containing the audio or video data. For this example, both the zeros (00) and the question marks (??) are used as place holders. The zeros are replaced by stream numbers. The question marks are replaced by codes indicating the type of data in the chunk. For example, a stream for a compressed DIB might use '01dc'.
'idx1'	Identifies a chunk containing the file index.

For more information on the AVI file format, see Chapter 4, "AVI Files."

Identifying Compression Formats

The BITMAPINFO data structure defined by Windows is used with many of the compression and decompression messages to pass information about the bitmaps being compressed and decompressed. This structure has the following fields:

```
typedef struct tagBITMAPINFO {
     BITMAPINFOHEADER bmiHeader;
     RGBQUAD bmiColors[];
 } BITMAPINFO;
```

The **bmiColors** field is used for the color table. The BITMAPINFOHEADER data defined for the **bmiHeader** field is used to pass information about the format of the bitmaps being compressed and decompressed. This structure has the following fields:

```
typedef struct tagBITMAPINFOHEADER {
     DWORD biSize;
     LONG  biWidth;
     LONG  biHeight;
     WORD  biPlanes;
     WORD  biBitCount;
     DWORD biCompression;
     DWORD biSizeImage;
     LONG  biXPelsPerMeter;
     LONG  biYPelsPerMeter;
     DWORD biClrUsed;
     DWORD biClrImportant;
 } BITMAPINFOHEADER;
```

The **biCompression** field specifies the type of compression used or requested. Windows defines the following compression formats:

BI_RGB
Specifies the bitmap is not compressed.

BI_RLE8
Specifies a run-length encoded format for bitmaps with 8 bits per pixel.

BI_RLE4
Specifies a run-length encoded format for bitmaps with 4 bits per pixel.

Extensions to the BI_RGB format include 16 and 32 bits per pixel bitmap formats. These formats do not use a color table. They embed the colors in the WORD or DWORD representing each pixel.

The 16 bit BI_RGB format is identified by setting **biCompression** to BI_RGB and setting **biBitCount** to 16. For this format, each pixel is represented by a 16-bit RGB color value. The high-bit of this value is zero. The remaining bits are divided into 3 groups of 5-bits to represent the red, green, and blue color values.

The 32 bit BI_RGB format is identified by setting **biCompression** to BI_RGB and setting **biBitCount** to 32. For this format, each pixel is represented by a 32 bit (4 byte) RGB color value. One byte is used for each red, green, and blue color value. The fourth byte is set to zero.

Your driver should support the BI_RGB format for 8 bit per pixel bitmaps. If practical, it should also support this format for 16, 24, and 32 bits per pixel bitmaps.

In addition to the new BI_RGB formats, the BI_BITFIELDS format adds new compression capabilities. This format specifies a bitmap is not compressed and color masks are defined in the **bmiColors** field of the BITMAPINFO data structure. The first DWORD in the **bmiColors** field is the red mask, the second DWORD is the green mask, and the third DWORD is the blue mask.

Your driver can also extend the format set by defining custom formats. Custom formats use a four character code for the format in the **biCompression** field in place of the standard constants. Your driver can use a custom format to support a unique or nonstandard compression type. When you define a custom format, you can specify values other than 1, 4, 8, 16, 24, or 32 for the **biBitCount** field.

For more information about the new formats and registering custom formats, see Chapter 5, "DIB Format Extensions for Microsoft Windows." For more information about the existing formats, see the *Microsoft Windows Programmer's Reference*.

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:

ICM_DECOMPRESS
This message tells the driver to decompress a frame of data into a buffer provided by the client-application.

ICM_DECOMPRESS_BEGIN
This message tells the driver to prepare for decompressing data.

ICM_DECOMPRESS_END
This message tells the driver to clean up after decompressing.

ICM_DECOMPRESS_GET_FORMAT
This message asks the driver to suggest a good format for the decompressed data.

ICM_DECOMPRESS_QUERY
This message asks the driver if it can decompress a specific input format.

ICM_DECOMPRESS_GET_PALETTE
This message asks the driver to return the color table of the output data structure.

The video decompressed with these messages is returned to the client-application and it 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 explained in "Decompressing Directly to Video Hardware." 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.

Setting 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 it reads from the 'strd' data chunk 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_DECOMPRESS_QUERY to specify the input format and determine the decompression format. The client-application sends ICM_DECOMPRESS_GET_FORMAT if it wants your driver to suggest the decompressed format. The client-application sends ICM_DECOMPRESS_QUERY to determine if your driver supports a format it is suggesting.

Both messages send 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 gets 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 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** field 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 gets ICM_DECOMPRESS_QUERY, *dwParam1* 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 decompressed format the client-application wants to use.

If *dwParam2* is NULL, your decompression driver can use any output format. In this case, the client-application wants to know if you can decompress the input format and it doesn't care about the output format. If *dwParam2* 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), 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 formats suggested. If you fail the message by returning ICERR_BADFORMAT, the client-application will suggest alternate formats until your driver accepts one. If your driver exhausts the list of formats normally used, the client-application requests a format with ICM_DECOMPRESS_GET_FORMAT.

If you are decompressing to 8-bit data, your driver will also receive 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** field. The space reserved for the color table will always be 256 colors.

Preparing to Decompress Video

When the client-application is ready, it sends the ICM_DECOMPRESS_BEGIN message to the driver. The client-application sets *dwParam1* and *dwParam2* to the BITMAPINFO data structures describing 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_DECOMPRESS 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 ICDECOMPRESS data structure specified in *dwParam1* contains the decompression parameters. The value specified in *dwParam2* specifies the size of the structure. The ICDECOMPRESS data structure has the following fields:

```
typedef struct {
     DWORD   dwFlags;
     LPBITMAPINFOHEADER  lpbiInput;
     LPVOID  lpInput;
     LPBITMAPINFOHEADER  lpbiOutput;
     LPVOID  lpOutput;
     DWORD   ckid
 } ICDECOMPRESS;
```

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** fields contain pointers to the format data and buffer used for the output data.

The client-application sets the ICDECOMPRESS_HURRYUP flag in the **dwFlags** field if it wants your driver to try and decompress the data at a faster rate. The client-application will not display any data decompressed with this flag. 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.

Ending Decompression

Your driver receives ICM_DECOMPRESS_END when the client-application no longer needs data decompressed. For this message, your driver should free the resources it allocated for the ICM_DECOMPRESS_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.

Compressing Video Data

Similar to decompressing video data, your driver will receive a series of messages when it is used to compress data. The client-application will send 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:

ICM_COMPRESS
This message tells the driver to compress a frame of data into the buffer provided by the client-application.

ICM_COMPRESS_BEGIN
This message tells the driver to prepare for compressing data.

ICM_COMPRESS_END
This message tells the driver to clean up after compressing.

ICM_COMPRESS_GET_FORMAT
This message asks the driver to suggest the output format of the compressed data.

ICM_COMPRESS_GET_SIZE
This message requests the maximum size of one frame of data when it is compressed in the output format.

ICM_COMPRESS_QUERY
This message asks the driver if it 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 AVI 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 user with the ICM_CONFIGURE dialog box. 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 if it wants your driver to 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* depends on the message.

If the client-application wants your driver to suggest the format, it 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** field 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 gets 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 the client-application wants to use. If *dwParam2* is NULL, your compression driver can use any output format. (The client-application just wants to know if 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 normally 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.

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 fields of ICCOMPRESS for returning information about the compressed data.

Your driver returns the actual size of the compressed data in the **biSizeImage** field in the BITMAPINFOHEADER data structure pointed to by the **lpbiOutput** field of ICCOMPRESS. The ICCOMPRESS data structure has the following fields:

```
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 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** fields 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** field in the BITMAPINFO structure specified for **lpbiOutput**.

The **dwFlags** field specifies flags used for compression. The client-application sets ICM_COMPRESS_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** field 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. For more information on chunk IDs, see Chapter 4, "AVI Files."

The **lpdwFlags** field 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 field. Your driver can define its own flags but they must be set in the high word only.

The **lFrameNum** field specifies the frame number of the frame to compress. If your driver is performaing fast temporal compression, check this field to see if frames are being sent out of order or if the client-application is having a frame recompressed.

The **dwFrameSize** field indicates the maximum size (in bytes) desired for the compressed frame. If it specifies zero, your driver determines the size of the compressed image. If it is non-zero, 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** field 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** fields, it should set the VIDCF_FASTTEMPORAL flag when it responds to the ICM_GETINFO message. The VIDCF_FASTEMPORAL 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 decompressing 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. Your driver should not expect the client-application to send a ICM_COMPRESS_END message for each ICM_COMPRESS_BEGIN message. The client-application can use ICM_COMPRESS_BEGIN to restart compression without sending ICM_COMPRESS_END.

When the driver is no longer needed, the system will close it by sending DRV_CLOSE.

Decompressing 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 decompress data directly to hardware rather than into a data buffer returned to the client-application by the decompression driver.

Your driver will receive a series of messages from the client-application to coordinate the following activities to decompress 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 video decompression drivers for these decompression activities:

ICM_DRAW
This message tells the driver to decompress a frame of data and draw it to the screen.

ICM_DRAW_BEGIN
This message tells the driver to get ready to draw data.

ICM_DRAW_END
This message tells the driver to clean up after decompressing an image to the screen.

ICM_DRAW_REALIZE
This message realizes a palette.

ICM_DRAW_QUERY
This message determines if the driver can render data in a specific format.

The video decompressed with the ICM_DRAW messages is retained by your driver and it 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 Format

Because your driver handles the drawing of video, the client-application does not need to determine the output format. The client-application needs to know only 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 formats. If your driver does not support the format, return ICERR_BADFORMAT.

Preparing to Decompress Video

When the client-application is ready, it sends the ICM_DRAW_BEGIN message to the driver to prepare the driver for decompressing the video 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*. The ICDRAWBEGIN structure has the following fields:

```
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 **dwFlags** field can specify any of the following flags:

ICDRAW_QUERY
Set when the client-application wants to determine if the driver can handle the decompression. The driver does not actually decompress the data.

ICDRAW_FULLSCREEN
Indicates the full screen is used to draw the decompressed data.

ICDRAW_HDC
Indicates a window and DC are used to draw the decompressed data.

If the ICDRAW_QUERY flag is set, 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 without this flag to start the actual decompression sequence.

The ICDRAW_FULLSCREEN and ICDRAW_HDC flags are described with the data structure fields associated with them.

Your driver can ignore the palette handle specified in the **hpal** field.

The **hwnd** and **hdc** field specifies the handle of the window and DC used for drawing. These fields are valid only if the ICDRAW_HDC flag is set in the **dwFlags** field.

The **xDst** and **yDst** fields 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** fields specifies the width and height of the destination rectangle. These fields are valid only if 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 fields.

The **xSrc**, **ySrc**, **dxSrc**, and **dySrc** fields 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** fields specify 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** fields 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 on handling the source and destination rectangles, see the **StretchDIBits** function in the *Microsoft Windows Programmer's Reference*.

The **lpbi** field specifies a pointer to a **BITMAPINFOHEADER** data structure containing the input format.

The **dwRate** field 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 will use 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 will not get an ICM_DRAW_END message so it should not prepare its resources for other ICM_DRAW messages.

Decompressing the Video

The client-application sends ICM_DRAW each time it has an image to decompress. (Your driver should use this message to decompress images. It should wait for the ICM_DRAW_START message before it begins to render them.) The client-application uses the flags in the file index to ensure the first frame in a series of frames decompressed starts with a key frame boundary. Your driver must allocate the memory it needs for the decompressed image.

The ICDRAW data structure specified in *dwParam1* contains the decompression parameters. The value specified in *dwParam2* specifies the size of the structure. The ICDRAW data structure has the following fields:

```
typedef struct {
     DWORD    dwFlags;
     LPVOID   lpFormat;
     LPVOID   lpData;
     DWORD    cbData;
 } ICDRAW;
```

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** field when it wants your driver to try 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 if it wants your driver to update the screen based on data previously received.

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 Decompression

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. Your driver should not expect the client-application to send a 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:

ICM_DRAW_GETTIME
This message obtains the value of the driver's internal clock if it is handling the timing of drawing frames.

ICM_DRAW_SETTIME
This message sets the driver's internal clock if it is handling the timing of drawing frames.

ICM_DRAW_START
This message tells the driver to start its internal clock if it is handling the timing of drawing frames.

ICM_DRAW_STOP
This message tells the driver to stop its internal clock if it is handling the timing for drawing frames.

ICM_DRAW_WINDOW
This message tells the driver that the display window has been moved, hidden, or displayed.

ICM_DRAW_FLUSH
This message tells the driver to flush any frames that are waiting to be drawn.

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 normally frame numbers for video) in the DWORD indicated by the pointer specified 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 normally 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.

Using Installable Compressors for Non-video Data

Installable compressors are not necessarily limited to video data. By using a different value than 'vidc' in the **fccType** field, you can specify that your installable driver expects to handle a type of data that is not video. (Four-character codes for non-video data should also be registered. See the "Architecture of a Video Compression and Decompression Driver" section for information on registering four-character codes.)

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 will be used to search for a driver capable of handling the data. The search will follow the same procedure used for installable compressor drivers.

Writing a driver to render non-video 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. For this function, VidEdit uses MCIAVI to decompress the file.)

Video Compression and Decompression Driver Reference

This section is an alphabetic reference to the messages and data structures provided by Windows for use by video compression and decompress drivers. There are separate sections for messages and data structures. The COMPDDK.H file defines the messages and data structures.

Video Compression and Decompression Driver Message Reference

Windows communicates with video compression and decompression drivers through messages sent to the driver. The driver processes these messages with its **DriverProc** entry-point function.

The following messages are used by video compression and decompression drivers for compressing data:

ICM_COMPRESS
This message tells the driver to compress a frame of data into the buffer provided by the calling application.

ICM_COMPRESS_BEGIN
This message prepares the driver for compressing data.

ICM_COMPRESS_END
This message tells the driver to clean up after compressing.

ICM_COMPRESS_GET_FORMAT
This message returns the output format of the compressed data.

ICM_COMPRESS_GET_SIZE
This message obtains the maximum size of one frame of data when it is compressed in the output format.

ICM_COMPRESS_QUERY
This message asks the driver if it can compress a specific input format.

The following messages are used by video compression and decompression drivers for decompression:

ICM_DECOMPRESS
This message tells the driver to decompress a frame of data into a buffer provided by the calling application.

ICM_DECOMPRESS_BEGIN
This message prepares the driver for decompressing data.

ICM_DECOMPRESS_END
This message tells the driver to clean up after decompressing.

ICM_DECOMPRESS_GET_FORMAT
This message asks the driver to suggest the format of the decompressed data.

ICM_DECOMPRESS_GET_PALETTE
This message asks the driver to return the color table of the output data structure.

ICM_DECOMPRESS_QUERY
This message asks the driver if it can decompress a specific input format.

The following messages are used by video compression and decompression drivers for drawing with the compressed data:

ICM_DRAW
This message tells the driver to decompress a frame of data and draw it to the screen.

ICM_DRAW_BEGIN
This message tells the driver to get ready to draw data.

ICM_DRAW_END
This message tells the driver to clean up after decompressing an image to the screen.

ICM_DRAW_GETTIME
This message obtains the value of the driver's internal clock if it is handling the timing of drawing frames.

ICM_DRAW_QUERY
This message determines if the driver can render data in a specific format.

ICM_DRAW_REALIZE
This message obtains a palette from the driver.

ICM_DRAW_SETTIME
This message informs a video compression driver of what frame it should be drawing.

ICM_DRAW_START
This message tells the driver to start its internal clock if it is handling the timing of drawing frames.

ICM_DRAW_STOP
This message tells the driver to stop its internal clock if it is handling the timing for drawing frames.

ICM_DRAW_WINDOW
This message tells the driver when a window has physically moved, or has become totally obscured.

ICM_DRAW_FLUSH
This message is sent to a video compression driver to flush any frames it has that are waiting to be drawn.

The following messages are used to configure video compression and decompression drivers:

ICM_ABOUT
This message displays an about dialog box for a compressor driver.

ICM_CONFIGURE
This message displays a configuration dialog box for a compressor driver.

ICM_GETBUFFERSWANTED
This message obtains information about how much pre-buffering the driver wants.

ICM_GETDEFAULTKEYFRAMERATE
This message obtains the preferred key frame spacing of the driver.

ICM_GETDEFAULTQUALITY
This message obtains the default quality setting of the driver.

ICM_GETINFO
This message returns information about the driver.

ICM_GETQUALITY
This message obtains the current quality setting of the driver.

ICM_GETSTATE
This message fills in a compressor-specific block of memory describing the compressor's current configuration.

ICM_SETQUALITY
This message sets the quality level of the compressor.

ICM_SETSTATE
This message sets the quality level for compression.

The following system message is used to open video compression and decompression drivers:

DRV_OPEN
This system message is sent to a video compression driver each time it is opened.

Video Compression and Decompression Driver Messages

This section contains an alphabetical list of the video compression and decompression messages that can be received and sent by video capture drivers. Each message name contains a prefix, identifying the type of the message.

A message consists of three parts: a message number and two DWORD parameters. Message numbers are identified by predefined message names. The two DWORD parameters contain message-dependent values.

DRV_OPEN

This system message is sent to a video compression driver each time it is opened.

Parameters

DWORD *dwDriverIdentifier*
Specifies the handle returned to the application opening the driver.

HANDLE *hDriver*
Specifies the handle created by the system. This handle is returned to the application. A unique handle is created each time the driver is opened.

LONG *lParam1*
Specifies a pointer to a NULL-terminated string. The string contains any characters that follow the filename in the SYSTEM.INI file. If the device driver was opened by filename, or if there is no additional information, a NULL string or a NULL pointer is passed. Device drivers should verify that *lParam1* is not NULL before dereferencing it.

LONG *lParam2*
Specifies a far pointer to an **ICOPEN** structure, or zero if the driver is opened only for configuration by the Drivers option of the Control Panel. If an **ICOPEN** structure is passed, the driver should verify that the **fccType** field contains 'vidc'. This indicates the driver is opened as a video compressor.

Return Value

The driver should return zero to fail the call. A non-zero return value is passed back to the driver in the ID field each time **DriverProc** is sent a message with **SendDriverMessage** or **CloseDriver**.

ICM_ABOUT

This message is sent to a video compression driver to display its about dialog box.

Parameters

DWORD *dwParam1*
Specifies an **HWND** which should be the parent of the displayed dialog box.

If *dwParam1* is -1, the driver should return ICERR_OK if it has an about dialog box, however, the driver should not display it. The driver should return ICERR_UNSUPPORTED if it does not display a dialog box.

DWORD *dwParam2*
Not Used.

Return Value Return ICERR_OK if the driver supports this message. Otherwise, return ICERR_UNSUPPORTED.

See Also ICM_CONFIGURE, ICM_GETINFO

ICM_COMPRESS

This message is sent to a video compression driver to compress a frame of data into the application-supplied buffer.

Parameters

DWORD *dwParam1*

Specifies a far pointer to an **ICCOMPRESS** data structure. The following fields of the **ICCOMPRESS** specify the compression parameters:

The **lpbiInput** field of **ICCOMPRESS** contains the format of the uncompressed data; the data itself is in a buffer pointed to by **lpInput**.

The **lpbiOutput** field of the **ICCOMPRESS** data structure contains a pointer to the output (compressed) format, and **lpOutput** contains a pointer to a buffer used for the compressed data.

The **lpbiPrev** field of the **ICCOMPRESS** data structure contains a pointer to the format of the previous frame, and **lpPrev** contains a pointer to a buffer used for the previous data. These fields are used by drivers that do temporal compression.

The driver should use the **biSizeImage** field of the **BITMAPINFOHEADER** structure associated with **lpbiOutput** to return the size of the compressed frame.

The **lpdwFlags** field 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** field contains the size the compressor should try to make the frame fit within. This size is used for compression methods that can make tradeoffs between compressed image size and image quality.

The **dwQuality** field contains the specific quality the compressor should use if it supports it.

DWORD *dwParam2*

Specifies the size of the **ICCOMPRESS** structure.

Return Value Return ICERR_OK if successful. Otherwise, return an error number.

See Also ICM_COMPRESS_BEGIN, ICM_COMPRESS_END, ICM_DECOMPRESS, ICM_DRAW

ICM_COMPRESS_BEGIN

This message is sent to a video compression driver to prepare it for compressing data.

Parameters

DWORD *dwParam1*

Specifies a far pointer to a **BITMAPINFO** data structure indicating the input format.

DWORD *dwParam2*

Specifies a far pointer to a **BITMAPINFO** data structure indicating the output format.

Return Value Return ICERR_OK if the specified compression is supported. Otherwise, return 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** messages.

ICM_COMPRESS_BEGIN and ICM_COMPRESS_END 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.

See Also ICM_COMPRESS, ICM_COMPRESS_END, ICM_DECOMPRESS_BEGIN, ICM_DRAW_BEGIN

ICM_COMPRESS_END

This message is sent to a video compression driver to end compression. The driver should clean-up after compressing, and release any memory allocated during processing of an **ICM_COMPRESS_BEGIN** message.

Parameters DWORD *dwParam1*
Not used.

DWORD *dwParam2*
Not used.

Return Value Return ICERR_OK if successful. Otherwise, return an error number.

Comments ICM_COMPRESS_BEGIN and ICM_COMPRESS_END 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.

See Also ICM_COMPRESS_BEGIN, ICM_COMPRESS, ICM_DECOMPRESS_END, ICM_DRAW_END

ICM_COMPRESS_GET_FORMAT

This message is sent to a video compression driver to suggest the output format of the compressed data.

Parameters DWORD *dwParam1*
Specifies a far pointer to a **BITMAPINFO** data structure indicating the input format.

DWORD *dwParam2*
Specifies zero or a far pointer to a **BITMAPINFO** data structure used by the driver to return the output format.

Return Value Return the size of the output format.

Comments If *dwParam2* is zero, the driver should simply return the size of the output format.

If *dwParam2* is non-zero, 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 which will preserve 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 by this message.

See Also ICM_COMPRESS_QUERY, ICM_DECOMPRESS_GET_FORMAT, ICM_DRAW_GET_FORMAT

ICM_COMPRESS_GET_SIZE

This 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

DWORD *dwParam1*
Specifies a far pointer to a **BITMAPINFO** data structure indicating the input format.

DWORD *dwParam2*
Specifies a far pointer to a **BITMAPINFO** data structure indicating the output format.

Return Value Return the maximum number of bytes a single compressed frame can occupy.

Comments Typically, applications use 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.

See Also ICM_COMPRESS_QUERY, ICM_COMPRESS_GET_FORMAT

ICM_COMPRESS_QUERY

This 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

DWORD *dwParam1*
Specifies a far pointer to a **BITMAPINFO** data structure describing the input format.

DWORD *dwParam2*
Specifies a far pointer to a **BITMAPINFO** data structure describing the output format, or zero. Zero indicates any output format is acceptable.

Return Value Return ICERR_OK if the specified compression is supported. Otherwise, return an error. The following errors are defined:

ICERR_OK
No error.

ICERR_BADFORMAT
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.)

See Also ICM_COMPRESS_GET_FORMAT

ICM_CONFIGURE

This message is sent to a video compression driver to display its configuration dialog box.

Parameters

DWORD *dwParam1*

Specifies an **HWND** which should be the parent of the displayed dialog.

If *dwParam1* is -1, the driver should return ICERR_OK if it has a configuration dialog box, however, the driver should not display it. The driver should return ICERR_UNSUPPORTED if it does not display a dialog box.

DWORD *dwParam2*

Not Used.

Return Value

Return ICERR_OK if the driver supports this message. Otherwise, return ICERR_UNSUPPORTED.

Comments

This message is distinct from the **DRV_CONFIGURE** message which is used for hardware configuration. This message should let the user configure the internal state referenced by **ICM_GETSTATE** and **ICM_SETSTATE**. For example, this dialog box can let the user change parameters affecting the quality level and other similar compression options.

See Also

DRV_CONFIGURE, ICM_ABOUT, ICM_GETINFO

ICM_DECOMPRESS

This message is sent to a video compression driver to decompress a frame of data into an application-supplied buffer.

Parameters

DWORD *dwParam1*

Specifies a far pointer to an **ICDECOMPRESS** structure.

DWORD *dwParam2*

Specifies the size of the **ICDECOMPRESS** structure.

Return Value

Return ICERR_OK if successful. Otherwise, return an error number.

Comments

If the driver is supposed to decompress data directly to the screen instead of a buffer, it will receive the **ICM_DRAW** message rather than this one.

The driver should return an error if this message is received before the **ICM_DECOMPRESS_BEGIN** message.

See Also

ICM_COMPRESS_BEGIN, ICM_DECOMPRESS_BEGIN, ICM_DECOMPRESS_END, ICM_DRAW_BEGIN

ICM_DECOMPRESS_BEGIN

This 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_DECOMPRESS** messages efficiently.

Parameters

DWORD *dwParam1*

Specifies a far pointer to a **BITMAPINFO** data structure describing the input format.

DWORD *dwParam2*
Specifies a far pointer to a **BITMAPINFO** data structure describing the output format.

Return Value
Return ICERR_OK if the specified decompression is supported. Otherwise, return an error number. The following errors are defined:

ICERR_OK
No error.

ICERR_BADFORMAT
The input or output format is not supported.

Comments
If the calling application wants the driver to decompress data directly to the screen, it 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.

See Also
ICM_COMPRESS_BEGIN, ICM_DECOMPRESS, ICM_DECOMPRESS_END, ICM_DRAW_BEGIN

ICM_DECOMPRESS_END

This message is sent to a video compression driver to have it clean-up after decompressing.

Parameters
DWORD *dwParam1*
Not used.

DWORD *dwParam2*
Not used.

Return Value
Return ICERR_OK if successful. Otherwise, return an error number.

Comments
The driver should free any resources allocated in response to the **ICM_DECOMPRESS_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.

See Also
ICM_COMPRESS_END, ICM_DECOMPRESS_BEGIN, ICM_DECOMPRESS, ICM_DRAW_END

ICM_DECOMPRESS_GET_FORMAT

This message is sent to a video compression driver to obtain the format of the decompressed data.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a **BITMAPINFO** data structure describing the input format.

DWORD *dwParam2*
Specifies zero or a far pointer to a **BITMAPINFO** data structure used by the driver to describe the output format.

Return Value

Return the size of the output format.

Comments

If *dwParam2* is zero, the driver should simply return the size of the output format. Applications set *dwParam2* to zero to determine the size of the buffer it needs to allocate.

If *dwParam2* is non-zero, 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 which will preserve 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 re-compressed.

See Also

ICM_COMPRESS_GET_FORMAT, ICM_DECOMPRESS_GET_PALETTE, ICM_DECOMPRESS_QUERY

ICM_DECOMPRESS_GET_PALETTE

This message is sent to a video compression driver to have it fill in the color table of the output **BITMAPINFOHEADER** structure.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a **BITMAPINFO** data structure indicating the input format.

DWORD *dwParam2*
Specifies zero or a far pointer to a **BITMAPINFO** data structure used to return the color table. The space reserved for the color table will always be at least 256 colors.

Return Value

Return the size of the output format or an error code.

Comments

If *dwParam2* is zero, the driver should simply return the size of the output format. Applications set this value to zero when they want to determine the size of the output format.

If *dwParam2* is non-zero, the driver should set the **biClrUsed** field of the **BITMAPINFOHEADER** data structure to the number of colors in the color table. The driver fills the **bmiColors** fields 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 in the input format.

See Also ICM_DECOMPRESS_GET_FORMAT

ICM_DECOMPRESS_QUERY

This 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

DWORD *dwParam1*
Specifies a far pointer to a **BITMAPINFO** data structure describing the input format.

DWORD *dwParam2*
Specifies zero or a far pointer to a **BITMAPINFO** data structure used by the driver to describe the output format. Zero indicates that any output format is acceptable.

Return Value

Return ICERR_OK if the specified decompression is supported. Otherwise, return an error number. The following errors are defined:

ICERR_OK
No error.

ICERR_BADFORMAT
The input or output format is not supported.

See Also ICM_COMPRESS_QUERY

ICM_DRAW

This message is sent to a video compression driver to decompress a frame of data and draw it to the screen.

Parameters

DWORD *dwParam1*
Specifies a far pointer to an **ICDRAW** structure.

DWORD *dwParam2*
Specifies the size of the **ICDRAW** structure.

Return Value

Return ICERR_OK if successful. Otherwise, return an error number.

Comments

If the **ICDRAW_UPDATE** flag is set in **dwFlags** field of the **ICDRAW** data structure, the area of the screen used for drawing is invalid and needs to be updated.

If the **ICDRAW_HURRYUP** flag is set in the **dwFlags** field, the calling application wants the driver to proceed as quickly as possible, possibly not even updating the screen.

If the **ICDRAW_PREROLL** flag is set in the **dwFlags** field, 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 keyframe, 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, the **ICM_DECOMPRESS** message is sent instead.

See Also

ICM_DECOMPRESS, ICM_DRAW_BEGIN, ICM_DRAW_END, ICM_DRAW_START, ICM_DRAW_STOP

ICM_DRAW_BEGIN

This message is sent to a video compression driver to prepare it for drawing data.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a **ICDRAWBEGIN** data structure describing the input format.

DWORD *dwParam2*
Specifies the size of the **ICDRAWBEGIN** data structure describing the output format.

Return Value

Return ICERR_OK if the driver supports drawing the data to the screen in the manner and format specified. Otherwise, return an error number. The following errors are defined:

ICERR_OK
No error.

ICERR_BADFORMAT
The input or output format is not supported.

ICERR_NOTSUPPORTED
The message is not supported.

Comments

If the driver is supposed to decompress data into a buffer instead of drawing directly to the screen, the **ICM_DECOMPRESS_BEGIN** message is sent rather than this one.

If the driver does not support drawing directly to the screen, return ICERR_NOTSUPPORTED.

ICM_DRAW_BEGIN and ICM_DRAW_END do not nest. If your driver receives an ICM_DRAW_BEGIN message before decompression is stopped with ICM_DRAW_END, it should restart decompression with new parameters.

See Also

ICM_DECOMPRESS_BEGIN, ICM_DRAW, ICM_DRAW_END, ICM_DRAW_START

ICM_DRAW_END

This message is sent to video compression drivers to clean-up after decompressing an image to the screen.

Parameters

DWORD *dwParam1*
Not used.

DWORD *dwParam2*
Not used.

Return Value

Return ICERR_OK if successful. Otherwise, return an error number.

Comments

ICM_DRAW_BEGIN and ICM_DRAW_END do not nest. If your driver receives an ICM_DRAW_BEGIN message before decompression is stopped with ICM_DRAW_END, it should restart decompression with new parameters.

See Also ICM_DECOMPRESS_END, ICM_DRAW, ICM_DRAW_BEGIN, ICM_DRAW_STOP

ICM_DRAW_FLUSH

This message is sent to a video compression driver to flush any frames it has that are waiting to be drawn.

Parameters

DWORD *dwParam1*
Not used.

DWORD *dwParam2*
Not used.

Return Value None.

Comments This message is used only by hardware which does its own asynchronous decompression, timing, and drawing.

See Also ICM_DRAW, ICM_DRAW_END, ICM_DRAW_STOP, ICM_GETBUFFERSWANTED

ICM_DRAW_GETTIME

This message is sent to a video compression driver to obtain the current value of its internal clock if it is handling the timing of drawing frames.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a LONG to be used by the driver to return the current time. The return value should be specified in samples. This corresponds to frames for video.

DWORD *dwParam2*
Not used.

Return Value Return ICERR_OK if successful.

Comments This message is generally only supported by hardware which does its own asynchronous decompression, timing, and drawing. The message will also only be sent if the hardware is being used as the synchronization master.

See Also ICM_DRAW_START, ICM_DRAW_STOP, ICM_DRAW_SETTIME

ICM_DRAW_QUERY

This message is sent to a video compression driver to determine if it can render data in a specific format.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a **BITMAPINFO** data structure describing the input format.

DWORD *dwParam2*
Not used.

Return Value Return ICERR_OK if the compressor can render data in the specified format. Otherwise, return an error number. The following errors are defined:

ICERR_OK
  No error.

ICERR_BADFORMAT
  The format is not supported.

Comments This message asks if the compressor recognizes the format for draw operations. The ICM_DRAW_BEGIN is sent to see if the compressor can draw the data.

See Also ICM_DECOMPRESS_QUERY

ICM_DRAW_REALIZE

This message is sent to a video compression driver to realize its palette used while drawing.

Parameters DWORD *dwParam1*
  Specifies a handle to the display context used to realize palette.

DWORD *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 Return ICERR_OK if palette realized.

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 (returns ICERR_UNSUPPORTED), the palette associated with the decompressed data is realized.

See Also ICM_DRAW_BEGIN

ICM_DRAW_RENDERBUFFER

This message is sent to a video compression driver to tell it to draw the frames that have been passed to it.

Parameters DWORD *dwParam1*
  Not used.

DWORD *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 a single video frame passed to it.

This message is used only by hardware which does its own asynchronous decompression, timing, and drawing.

See Also ICM_DRAW, ICM_DRAW_END, ICM_DRAW_START

ICM_DRAW_SETTIME

This message is sent to a video compression driver to inform it of what frame it should be drawing if it is handling the timing of drawing frames.

Parameters

DWORD *dwParam1*
Specifies a LONG containing the sample which the driver should now be rendering. The value will be specified in samples. This corresponds to frames for video.

DWORD *dwParam2*
Not used.

Return Value

Return ICERR_OK if successful.

Comments

This message is generally only supported by hardware which 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 own internal clock, and take actions to synchronize the two if the difference is great enough.

See Also

ICM_DRAW_START, ICM_DRAW_STOP, ICM_DRAW_GETTIME

ICM_DRAW_START

This message is sent to a video compression driver to start its internal clock for the timing of drawing frames.

Parameters

DWORD *dwParam1*
Not used.

DWORD *dwParam2*
Not used.

Return Value

None.

Comments

This message is typically used by hardware which does its own asynchronous decompression, timing, and drawing.

When it receives this message, the driver should start rendering data at the rate specified in the **ICM_DRAW_BEGIN** message.

ICM_DRAW_START and ICM_DRAW_STOP do not nest. If your driver receives an ICM_DRAW_START message before rendering is stopped with ICM_DRAW_STOP, it should restart rendering with new parameters.

See Also

ICM_DRAW, ICM_DRAW_BEGIN, ICM_DRAW_END, ICM_DRAW_STOP

ICM_DRAW_STOP

This message is sent to a video compression driver to stop its internal clock for the timing of drawing frames.

Parameters

DWORD *dwParam1*
Not used.

DWORD *dwParam2*
Not used.

Return Value

None.

Comments

This message is typically used by hardware which does its own asynchronous decompression, timing, and drawing.

See Also

ICM_DRAW, ICM_DRAW_END, ICM_DRAW_START

ICM_DRAW_WINDOW

This message is sent to a video compression driver when the window specifed in the **ICM_DRAW_BEGIN** message has physically 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

DWORD *dwParam1*
Points to a RECT structure containing the destination rectangle. The destination rectangle is specified in screen coordinates. If *dwParam1* points to a empty rectangle drawing should be turned off.

DWORD *dwParam2*
Not used.

Return Value

Return ICERR_OK if successful.

Comments

This message is only supported by hardware which does its own asynchronous decompression, timing, and drawing.

The rectangle is set empty if the window is totally hidden by other windows. Drivers should turn off overlay hardware when the rectangle is empty.

See Also

ICM_DRAW_BEGIN

ICM_GETBUFFERSWANTED

This message is sent to a video compression driver to have the driver return information about how much pre-buffering it wishes to do.

Parameters

DWORD *dwParam1*
Specifies a far 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.

DWORD *dwParam2*
Not used.

Return Value Return ICERR_OK if successful. Otherwise, return ICERR_UNSUPPORTED.

Comments Typically, this message is only used by a driver that uses hardware to render data and wants to 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 applications to try and stay exactly ten frames ahead of the frame it currently needs.

ICM_GETDEFAULTKEYFRAMERATE

This message is sent to a video compression driver to request that it return its default (or preferred) key frame spacing.

Parameters DWORD *dwParam1*
Specifies a far pointer to a DWORD used by the driver to return its preferred key frame spacing.

DWORD *dwParam2*
Not used.

Return Value Return ICERR_OK if the driver supports this message. Otherwise, return ICERR_UNSUPPORTED.

ICM_GETDEFAULTQUALITY

This message is sent to a video compression driver to request that the driver return its default quality setting.

Parameters DWORD *dwParam1*
Specifies a far pointer to a DWORD used by the driver to return its default quality.

DWORD *dwParam2*
Not used.

Return Value Return ICERR_OK if the driver supports this message. If not, return ICERR_UNSUPPORTED.

Comments Quality values range between 0 and 10,000.

See Also ICM_SETQUALITY, ICM_GETQUALITY

ICM_GETINFO

This message is set to a video compression driver to have it return information describing the driver.

Parameters DWORD *dwParam1*
Specifies a far pointer to an **ICINFO** data structure used by the driver to return information.

DWORD *dwParam2*
Specifies the size of the **ICINFO** data structure.

Return Value Return the size of the **ICINFO** data structure, or zero if an error occurs.

Comments Typically, this message is sent by applications that want to display a list of the installed compressors.

The driver should fill in all fields of the **ICINFO** structure except the **szDriver** field.

See Also ICM_ABOUT

ICM_GETQUALITY

This message is sent to a video compression driver to request that it return its current quality setting.

Parameters DWORD *dwParam1*
Specifies a far pointer to a DWORD used by the driver to return the current quality value.

DWORD *dwParam2*
Not used.

Return Value Return ICERR_OK if the driver supports this message. If not, return ICERR_UNSUPPORTED.

Comments Quality values range between 0 and 10,000.

See Also ICM_SETQUALITY, ICM_GETDEFAULTQUALITY

ICM_GETSTATE

This message is sent to a video compression driver to have it fill a block of memory describing the compressor's current configuration.

Parameters DWORD *dwParam1*
Specifies a far 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.

DWORD *dwParam2*
Specifies the size of the block of memory.

Return Value Return the amount of memory required by the state information.

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.

See Also DRV_CONFIGURE, ICM_SETSTATE

ICM_SETQUALITY

This message is sent to a video compression driver to set the quality level for compression.

Parameters

DWORD *dwParam1*
Specifies the new quality value.

DWORD *dwParam2*
Not used.

Return Value

Return ICERR_OK if the driver supports this message. If not, return ICERR_UNSUPPORTED.

Comments

Quality values range between 0 and 10,000.

See Also

ICM_GETQUALITY, ICM_GETDEFAULTQUALITY

ICM_SETSTATE

This message is sent to a video compression driver to set the state of the compressor.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a block of memory containing configuration data or NULL. If NULL, the driver should return to its default state.

DWORD *dwParam2*
Specifies the size of the block of memory.

Return Value

Return the number of bytes actually used by the compressor. A return value of zero generally indicates an error.

Comments

Since the information used by **ICM_SETSTATE** is private and specific to a given compressor, client applications should use this message only to pass information previously returned for the **ICM_GETSTATE** message.

See Also

DRV_CONFIGURE, ICM_GETSTATE

Video Compression and Decompression Driver Data Structure Reference

This section lists data structures used by video compression and decompression drivers for Windows. The data structures are presented in alphabetical order. The structure definition is given, followed by a description of each field.

ICCOMPRESS

The **ICCOMPRESS** structure is used with the **ICM_COMPRESS** message to specify the parameters used for compression.

```
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  lpInput;
} ICCOMPRESS;
```

Fields

The **ICCOMPRESS** structure has the following fields:

dwFlags
Specifies flags used for compression. The following flag is defined.

ICCOMPRESS_KEYFRAME
Treat input data as a keyframe.

lpbiOutput
Specifies a pointer to a **BITMAPINFOHEADER** structure containing the output (compressed) format. The **biSizeImage** field 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. Device drivers can ignore this field.

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) to compress this frame to.

dwQuality
Specifies the compression quality.

lpbiPrev
Specifies a pointer to a **BITMAPINFOHEADER** structure containing the format of the previous frame. Normally, this will be the same as the input format.

lpInput
Specifies a pointer to the buffer containing the previous frame.

ICDECOMPRESS

The **ICDECOMPRESS** structure is used with the **ICM_DECOMPRESS** message to specify the parameters for decompressing the data.

```
typedef struct {
    DWORD  dwFlags;
    LPBITMAPINFOHEADER  lpbiInput;
    LPVOID  lpInput;
    LPBITMAPINFOHEADER  lpbiOutput;
    LPVOID  lpOutput;
    DWORD  ckid;
} ICDECOMPRESS;
```

Fields

The **ICDECOMPRESS** structure has the following fields:

dwFlags
Specifies flags.

The following flags in **dwFlags** specify the operation for this data:

ICDECOMPRESS_HURRYUP
Indicates the data is just buffered and not drawn to the screen. Use this flag for the fastest decompression.

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.

ICDRAW

The **ICDRAW** structure is used with the **ICM_DRAW** message to specify the parameters for drawing video data to the screen.

```
typedef struct {
    DWORD  dwFlags;
    LPVOID  lpFormat;
    LPVOID  lpData;
    DWORD  cbData;
    LONG  lTime;
} ICDRAW;
```

Fields

The **ICDRAW** structure has the following fields:

dwFlags
Specifies the flags from the AVI file index.

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.

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 keyframe, frames 0 through 9 are sent to the driver with the **ICDRAW_PREROLL** flag set. The driver needs this data so that it can display frame 10 properly, but frames 0 through 9 need not be individually displayed.

lpFormat
Specifies a pointer to a structure containing the data format. For video, this will be 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 that this data should be drawn. For video data this is normally a frame number. See **dwRate** and **dwScale** of the **ICDRAW** structure.

See Also

ICM_DRAW_BEGIN, ICDRAWBEGIN

ICDRAWBEGIN

The **ICDRAWBEGIN** structure is used with the **ICM_DRAW_BEGIN** message to specify the parameters used to decompress the data.

```
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;
```

Fields

The **ICDRAWBEGIN** structure has the following fields:

dwFlags
Specifies any of the following flags:

ICDRAW_QUERY
Set when an application wants to 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 DC is used to draw the decompressed data.

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.

xDst
Specifies the x-position of destination rectangle.

yDst
Specifies the y-position of destination rectangle.

dxDst
Specifies the width of destination rectangle.

dyDst
Specifies the height of destination rectangle.

lpbi
Specifies a pointer to a **BITMAPINFOHEADER** data structure containing the input format.

xSrc
Specifies the x-position of source rectangle.

ySrc
Specifies the y-position of source rectangle.

dxSrc
Specifies the width of source rectangle.

dySrc
Specifies the height of 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.

ICINFO

The **ICINFO** structure is filled by a video compression driver when it receives the **ICM_GETINFO** message.

```
typedef struct {
    DWORD  dwSize;
    DWORD  fccType;
    DWORD  fccHandler;
    DWORD  dwFlags;
    DWORD  dwVersion;
    DWORD  dwVersionICM;
    char  szName[16];
    char  szDescription[128];
    char  szDriver[128];
} ICINFO;
```

Fields

The **ICINFO** structure has the following fields:

dwSize
Should be set to the size of an **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 (**ICINFO.fccHandler** == 'vidc'):

VIDCF_QUALITY
The driver supports quality.

VIDCF_CRUNCH
The driver supports crunching to a frame size.

VIDCF_TEMPORAL
The driver supports inter-frame compression.

VIDCF_DRAW
The driver supports drawing.

VIDCF_FASTTEMPORAL
The driver can do temporal compression and doesn't need the previous frame.

dwVersion
Specifies the version number of the driver.

dwVersionICM
Specifies the version of the ICM supported by this driver; it should be set to 1.0 (0x00010000)

szName[16]
Specifies the short name for the compressor. The null-terminated name should be suitable for use in list boxes.

szDescription[128]
Specifies a null-terminated string containing the long name for the compressor.

szDriver[128]
Specifies a null-terminated string for the module that contains the driver. Normally, a driver will not need to fill this out.

ICOPEN

The **ICOPEN** structure is sent to a video compression driver with the **DRV_OPEN** message.

```
typedef struct {
    DWORD  fccType;
    DWORD  fccHandler;
    DWORD  dwVersion;
    DWORD  dwFlags;
} ICOPEN;
```

Fields

The **ICOPEN** structure has the following fields:

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:

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.

Comments

This structure is the same as that passed to video capture drivers when they are opened. This lets a single installable driver to function as either an installable compressor or a video capture device. By examining the **fccType** field of the **ICOPEN** structure, the driver can determine its function. For example, a **fccType** value of 'vidc' indicates that it is opened as an installable video compressor.

CHAPTER 11

Video Capture Device Drivers

Video capture device drivers provide low-level video capture services for Windows multimedia applications. Both applications and MCI device drivers can use these services to control video capture devices. These devices can provide services such as the following:

- Single frame video capture
- Real-time (streaming) video capture
- Video overlay
- Produce data in a standard or proprietary compressed format

Video capture devices must have a corresponding video capture device driver to be used with Windows. This chapter explains the Windows interface for video capture device drivers. It covers the following topics:

- The different types of video capture channels
- General information about writing a video capture device driver
- How a video capture device driver handles the system messages for the installable driver interface
- How a video capture device driver handles device specific messages for video capture
- An alphabetical reference to the messages and data structures used to write video capture device drivers

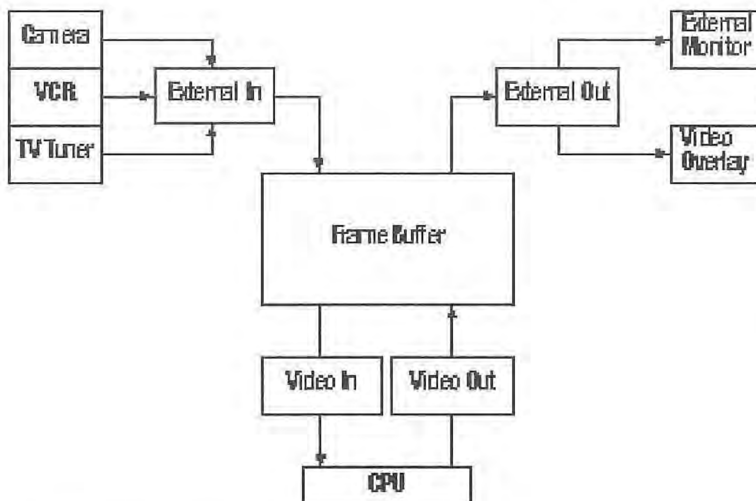
Before reading this chapter, you should be familiar with the video services available with Windows. You should also be familiar with the Windows installable driver interface. For information about the video services and the installable driver interface, see the *Microsoft Windows Programmer's Reference*. For information on other drivers using the installable driver interface, see the *Microsoft Windows Multimedia Device Adaptation Guide*.

Architecture of a Video Capture Driver

The MSVIDEO.DLL module provides the interface between client applications and video capture device drivers—applications do not call the drivers directly. When a client application calls a video capture function, MSVIDEO.DLL translates the call into a message and sends the message to the device driver.

Video Capture Device Driver Channels

Video capture device drivers can transfer data through four different channels. The destination or source of each channel is the frame buffer that is part of the video capture hardware. The four channels and the frame buffer are shown in the following illustration:



Data channels in the video capture driver.

The video capture channel (External In) is a source of video information placed in the frame buffer. The video source might be a video camera, video player, or television tuner. The format of both the incoming signal and the data placed in the frame buffer is controlled by the video capture hardware.

The video capture device can display the frame buffer data by using the video display channel (External Out). In practice, this could be with a second monitor or a video overlay device.

The device driver and application will use the video in channel (Video In) to transfer the video data to application supplied buffers.

The device driver and application can play captured data by using the video out channel (Video Out) to transfer data back into the frame buffer. Playback through this channel might be to review a sequence just captured or to play data from a file.

To supply minimum services, video capture drivers must support the External In and Video In channels. These channels provide services for video capture but not for video playback. Drivers with only External In and Video In channels rely on other system components (such as video compression and decompression drivers) for video playback.

Note: The Video Out channel is not currently used. The interface for video compression and decompression drivers is currently used to display this information.

The Video Capture Application

The application controlling the video capture driver is an integral part of the capture process. The application has the responsibility of allocating the memory used for video capture and managing the data buffers used for the transfer of data. If the user wants to capture audio simultaneously with video, the application also controls the audio driver used for capturing the input audio. Once the video and audio drivers capture the data, the application has the responsibility for any post-processing of this data. For example, if the application wants to save it as an AVI file, it must add the appropriate headers and create the AVI RIFF structure saved in the disk file.

Sample Device Drivers

The examples in this chapter were extracted from a sample device driver (BRAVADO.DRV) for the Truevision Bravado video capture hardware. The examples also apply to the Creative Labs Video Blaster (VBLASTER.DRV) capture hardware. The sample source code for this driver shares or parallels the sample source code for the Bravado device driver. (The files that are unique for the two samples include the .H, .RC, .DEF, .DLL, .LIB, and MAKEFILEs.)

Like many of the newer frame grabbers, these devices use the PCVIDEO 9001 chipset from Chips and Technologies. The sample driver is designed to support any video capture device based on the PCVIDEO chipset. You can develop a device driver for this chipset in as little as a single day if the following assumptions are true:

- A DLL is available which is modeled after PCVIDEO.DLL from Chips and Technologies. Functions exported by the DLL may have different names, but they should have similar functionality. For example, Truevision supplies a DLL called VW.DLL. The sample driver, BRAVADO.DRV calls on the services of this DLL to access most low-level hardware functions.
- Internally, images are captured to memory using YUV 4:1:1 encoding.

Video capture devices that are not based on the PCVIDEO chipset, or that use alternate internal formats, will require additional work to develop routines to convert between formats and control the device. Devices which capture data with the RGB format can be readily supported by modifications to the sample code.

The Structure of a Video Capture Device Driver

Video capture device drivers are dynamic-link libraries (DLLs) usually written in C or assembly language, or a combination of the two languages. You should combine operations for different video capture channels in a single DLL. For example, the Bravado video capture driver module, BRAVADO.DRV, has operations for video capture as well as the display of live video using key color or overlay.

As installable drivers, these drivers will provide a **DriverProc** entry point used to process system messages. For general information about installable drivers, the **DriverProc** entry point, and system messages sent to this entry point, see the *Microsoft Windows Programmer's Reference*. This chapter includes supplemental information for the system

messages. This information describes specifically how video capture drivers should respond to the system messages that are critical to their proper operation.

Video capture drivers also use the **DriverProc** entry point to process messages specifically for video capture. Information on how drivers use the **DriverProc** entry point to process these messages is contained in this chapter.

Combining Video Capture and Video Compression/Decompression Drivers

If the same hardware is required or used for a combination of video capture and video compression, you might combine both of these functions into a common DLL and use a single **DriverProc** entry point to service them. The common entry point will simplify the coordination of the different functions.

Note: Because video capture drivers can rely on video compression and decompression drivers for efficient operation, a single driver can handle both video capture, and video compression and decompression services. Video capture drivers use the VIDEO_OPEN_PARMS data structure when they are opened. This structure has the same field definitions as the ICOPEN structure used by video compression and decompression drivers. By examining the **fccType** field, a combined driver can determine whether it is being opened as a video capture driver or a video compression and decompression driver. (Video capture devices contain the four-character code 'vcap' in this field.) For more information on video compression and decompression drivers, see Chapter 10, "Video Compression and Decompression Drivers."

Video Capture Header Files

The messages and data structures used exclusively by video capture device drivers are defined in MSVIDDRV.H. Functions, error returns, and constants used by both video capture device drivers and applications are defined in MSVIDEO.H

Naming Video Capture Device Drivers

The filenames for device driver DLLs are not required to have a file extension of ".DLL"—you can name your driver using any file extension you want. It is suggested that you use the extension ".DRV" for your device drivers to follow the convention set by Windows.

SYSTEM.INI Entries for Video Capture Device Drivers

The SYSTEM.INI file contains information for loading and configuring device drivers. Your device driver must be identified in the [drivers] section. Your device driver might also have entries in the [386enh] section if it requires any VxDs for operation. Your driver might also reserve a device-specific section in the SYSTEM.INI to store configuration information. For more information on this device-specific section, see "The Installable Driver Interface," later in this chapter. The [drivers] and [386enh] sections are updated by an installation program when your device is installed or removed.

The preferred method for installing device drivers uses the Drivers option in the Control Panel. The Drivers option uses information in the OEMSETUP.INF file for your driver to add the entries in the [drivers] section as well as entries in the [386enh] section to install any VxDs you require. The procedures for creating an OEMSETUP.INF file are described in the Windows DDK.

The entry that identifies your driver in the [drivers] section lets Windows load the driver. If this entry is absent, your driver won't be recognized. While installation programs normally add the necessary entry for completed device drivers, you might have to manually add it while you are developing your device driver. You might also have to manually add any [386enh] entries you need. The final version of your device driver should use an installation program to create and delete the entries in these two sections.

For video capture devices, a key name of "MSVideo" specifies the name of your driver in the [drivers] section of SYSTEM.INI. For example, the following extract identifies one video capture device driver named "BRAVADO.DRV".

```
[drivers]
timer=timer.drv
joystick=ibmjoy.drv
MSVideo=bravado.drv
```

If there is more than one driver for a given device type, append a number from 1 to 9 after the key name. (When you have multiple drivers, use sequential numbers to identify them.)

While you can have more than one driver of the MSVIDEO type in the [drivers] section, the Drivers option in the Control Panel cannot install multiple drivers of this type. To work with more than one video driver, you might use the Drivers option to remove the existing driver and install an alternate, or you might manually edit SYSTEM.INI file to include the additional MSVIDEO entries. If you manually edit SYSTEM.INI, you can select the driver used when you execute the video capture application. The following example shows a [drivers] section with entries for five video capture drivers:

```
[drivers]
msvideo=targa16.drv
msvideo1=testdrv.drv
msvideo2=bravado.drv
msvideo3=vblaster.drv
msvideo4=MYDRVR.DRV
```

If you are using the VIDCAP video capture application, you can select the video capture driver it uses with the -d command line option. The integer specified after the -d corresponds to the video capture driver entry. For example, VIDCAP -d0 uses the TARGA16.DRV driver associated with the msvideo entry. VIDCAP -d3 uses the VBLASTER.DRV associated with the msvideo3 entry.

Note: Video capture device drivers are loaded only when needed by an application.

The Module-Definition File

To build a device-driver DLL, you must have a module-definition (.DEF) file. In this file, you must export the **DriverProc** entry-point function. Functions are exported by ordinal, as shown in the following example BRAVADO.DEF file:

```
LIBRARY      AVIBRAV

DESCRIPTION 'MSVIDEO:Truevision Bravado Driver'

EXETYPE      WINDOWS

PROTMODE

CODE         MOVEABLE  DISCARDABLE  LOADONCALL
DATA         FIXED  SINGLE  PRELOAD

SEGMENTS     _TEXT    FIXED                     PRELOAD
             INIT     MOVEABLE    DISCARDABLE   PRELOAD
             VCAP     MOVEABLE    DISCARDABLE   PRELOAD

HEAPSIZE     1024

EXPORTS      WEP                    @1   RESIDENTNAME
             DriverProc             @2   RESIDENTNAME
```

The actual ordinal values you assign to each exported function are not significant, though each must be unique within the DLL.

For more information on the entry-point function listed in this example, see "Entry-Point Function" later in this chapter.

The Module Name Line

The module name line should specify a unique module name for your device driver. Windows will not load two different modules with the same module name. It's a good idea to use the base of your driver filename since, in certain instances, **LoadLibrary** will assume that to be your module name.

The Module Description Line

The module description line in the module-definition file should specify the type of device the driver supports. For example, here's the module description line from the module-definition file for the Bravado video capture driver:

```
DESCRIPTION 'MSVIDEO: Truevision Bravado Driver'
```

Use MSVIDEO followed by a colon (:) to indicate the type of device your driver supports.

The Drivers option in the Control Panel uses these names to identify different types of drivers and to create the entry in the [Drivers] section of SYSTEM.INI when installing a driver.

Considerations for Interrupt-Driven Drivers

Most video capture device drivers will be interrupt-driven. For example, a video input device interrupts when the device receives a new video frame. Driver code accessed during an interrupt service routine must adhere to the guidelines discussed in the following sections.

Fixing Code and Data Segments

Any code segments or data segments a driver accesses at interrupt-time must be fixed segments. For best overall system performance, you should minimize the amount of code and data in fixed segments. To minimize the amount of fixed code, isolate all interrupt-time code in a few source modules and put this code into a single fixed code segment. Unless your driver has a large amount of data not accessed at interrupt time, use a single fixed data segment.

The Bravado video capture driver is a medium-model DLL, using a single data segment and multiple code segments. The following example fragment is from the module-definition file for the Bravado device driver:

```
CODE        MOVEABLE  DISCARDABLE  LOADONCALL
DATA        FIXED  SINGLE  PRELOAD

SEGMENTS    _TEXT   FIXED                      PRELOAD
            INIT    MOVEABLE   DISCARDABLE     PRELOAD
            VCAP    MOVEABLE   DISCARDABLE     PRELOAD
```

This example fixes the data segment and the code segment named _TEXT. All other code segments are moveable.

The code segment _TEXT is used as a safety measure. The compiler places code for which you do not specify a segment in the _TEXT segment. This way any code that is missed will be placed into a fixed segment preventing possible problems at interrupt time. However, you should check your segmentation to ensure that only code that is required to be FIXED goes into the FIXED code segment.

Allocating and Using Memory

You can allocate either local memory or global memory for use at interrupt time.

To allocate local memory for use at interrupt time, follow these steps:

1. Use **LocalAlloc** with the LMEM_FIXED flag to get a handle to the memory block. (This assumes fixed data segments.)
2. Pass this handle to **LocalLock** to get a near pointer to the memory block.

   Any global memory a driver uses at interrupt-time must be page-locked. To allocate and page-lock global memory, follow these steps:
3. Use **GlobalAlloc** with the GMEM_MOVEABLE and GMEM_SHARE flags to get a handle to the memory block.
4. Pass this handle to **GlobalLock** to get a far pointer to the memory block.
5. Pass the handle to **GlobalPageLock** to page-lock the memory block.

Calling Windows Functions at Interrupt Time

The only Windows functions a driver can call at interrupt time are **PostMessage**, **PostAppMessage**, **DriverCallback**, **timeGetSystemTime**, **timeGetTime**, **timeSetEvent**, **timeKillEvent**, **midiOutShortMsg**, **midiOutLongMsg**, and **OutputDebugStr**.

The Installable Driver Interface

The entry-point function, **DriverProc**, processes messages sent by the system to the driver as the result of an application call to a low-level video capture function. For example, when an application opens a video capture device, the system sends the specified video capture device driver a DRV_OPEN message. The driver's **DriverProc** function receives and processes this message.

Note: Your driver should respond to all system messages. If supplemental information is not provided for them in this chapter, use the definitions provided in the *Microsoft Windows Programmer's Reference*.

An Example DriverProc Entry-Point Function

The video capture driver uses the **DriverProc** function for its entry-point. The following example is extracted from the Bravado video capture driver.

```
LRESULT FAR PASCAL _loadds DriverProc(DWORD dwDriverID, HDRVR hDriver,
UINT uiMessage, LPARAM lParam1, LPARAM lParam2)
{
switch (uiMessage)
    {

        case DRV_LOAD:
            return (LRESULT)1L; //Device loaded successfully


        case DRV_FREE:
            return (LRESULT)1L; //Device freed successfully

        case DRV_OPEN:
            // lParam2 is NULL when the user configures
            // the device driver with the Drivers Option of the
            // Control Panel. If opened without an open structure,
            // return a dummy (non-zero) ID so OpenDriver will work.
            if (lParam2 == NULL)
                return BOGUS_DRIVER_ID;


            // Verify this open is for a video capture driver, and
            // not for an installable compressor/decompressor
            if (((LPVIDEO_OPEN_PARMS) lParam2) -> fccType != OPEN_TYPE_VCAP)
                return 0L;
```

```
            return (DWORD)(WORD)
                VideoOpen ((LPVIDEO_OPEN_PARMS) lParam2);


case DRV_CLOSE:
            //Device opened without an open structure
            if (dwDriverID == BOGUS_DRIVER_ID || dwDriverID == 0)
                return 1L;      // Device closed

            //Close device if termination routine executed successfully
            return ((VideoClose((PCHANNEL)dwDriverID)
                == DV_ERR_OK) ? 1L : 0);


case DRV_ENABLE:
            // Enable the driver: initialize hardware, hook
            // interrupts, allocate DMA buffer, etc.
            return (LRESULT)1L;


case DRV_DISABLE:
            //Disable the driver: free DMA buffer, unhook
            //interrupts, reset hardware, etc.
            return (LRESULT)1L;


case DRV_QUERYCONFIGURE:
            return (LRESULT)1L;        // Driver supports configuration


case DRV_CONFIGURE:
            // The Drivers option of the Control Panel sends this
            // message to display a dialog box that lets the user configure
            // the driver. For example, set the port base and interrupt.
            return (LRESULT)Config((HWND)lParam1, ghModule);


case DRV_INSTALL:
            return (LRESULT)DRV_OK; //Driver installed OK


case DRV_REMOVE:
            // The driver is being removed from the installed drivers list.
            // The driver should remove its .INI section, etc.
            ConfigRemove();
            return (LRESULT)DRV_OK; //Driver removed OK

        .
        .
        .
```

```
        default:
            if (dwDriverID == BOGUS_DRIVER_ID || dwDriverID == 0)
                return DefDriverProc(dwDriverID, hDriver, uiMessage,
                                     lParam1, lParam2);


            // Process video capture driver specific messages
            return VideoProcessMessage((PCHANNEL)dwDriverID,
                                     uiMessage, lParam1, lParam2);
    }
}
```

Handling the DRV_OPEN and DRV_CLOSE Messages

Like other installable drivers, client applications must open a video capture device before using it and close it when finished using it, so the device 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 device driver receives other messages, it can use this value to identify instance data needed for operation. Drivers can use the instance data for information related to the client that opened a device.

It's up to you to decide if your device driver will support more than one client simultaneously. If you do this, though, remember to check the *dwDriverID* parameter to determine which channel is being accessed.

For DRV_OPEN, the *lParam2* parameter contains a pointer to a VIDEO_OPEN_PARMS data structure containing information about the open. This structure has the following fields:

```
typedef struct {
    DWORD    dwSize;
    FOURCC   fccType;
    FOURCC   fccComp;
    DWORD    dwVersion;
    DWORD    dwFlags;
    DWORD    dwError;
} VIDEO_OPEN_PARMS;
```

The **fccType** field of this structure will contain the four character code 'vcap'. Because of the four video capture channels, video capture drivers must examine the flags set in the **dwFlags** field of the VIDEO_OPEN_PARMS data structure to determine the type of channel being opened. Your driver should be prepared to open (and conversely, close) the video channels in any order.

The following flags are defined for the video channels:

VIDEO_EXTERNALIN
: An external input channel responsible for loading images into the frame buffer.

VIDEO_EXTERNALOUT
: An external output channel responsible for displaying images in the frame buffer to an external or system monitor, or to an overlay device.

VIDEO_IN
A video input channel responsible for transferring images from the frame buffer to system memory. This might include a translation step or reformatting of the image. For example, reformatting a 16-bit RGB image to an 8 bit palette image.

VIDEO_OUT
A video output channel responsible for transferring images into the frame buffer from the CPU. (The sample driver does not use this channel type.)

The **dwVersion** field specifies the version of the video capture command set used by MSVIDEO.DLL. The version number lets your driver identify the command set to determine its capabilities. For the initial release of the video capture command set, your driver does not have to detect and adjust itself for multiple versions of the command set. Future versions of your driver can use this value to enable new features that depend on new capabilities of the video capture command set.

The **dwSize** field specifies the size of the **VIDEO_OPEN_PARMS** structure.

The **fccComp** field is unused.

The **dwError** field specifies an error value the driver might return to the client application if it fails the open.

The following code fragment illustrates the routines the Bravado device driver uses to handle the DRV_OPEN and DRV_CLOSE messages. This device driver supports only one instance of each video channel.

```
PCHANNEL NEAR PASCAL VideoOpen( LPVIDEO_OPEN_PARMS lpOpenParms)
{
    PCHANNEL    pChannel;
    DEVICE_INIT di;
    LPDWORD     lpdwError = &lpOpenParms->dwError;
    DWORD       dwFlags = lpOpenParms-> dwFlags;

    *lpdwError = DV_ERR_OK;
```

```
// Initialize hardware on first call
if (!fDeviceInitialized) {

    // Get Port/IRQ/Base/etc. in INI file
    GetHardwareSettingsFromINI (&di);

    // Perform hardware initialization
    if (! HardwareInit (&di)) {
        *lpdwError = DV_ERR_NOTDETECTED;
        return NULL;
    }

    ConfigGetSettings();   // Get global hue, sat, channel, zoom

    .
    .
    .
    // Deleted code initializes hardware & sets remaining global values
    .
    .
    .
}

// Get instance memory. On exit this function assigns this value
// to dwDeviceID. By using this value for dwDeviceID,
// the device driver can easily retrieve the instance data
// when it needs it to process subsequent messages.
pChannel = (PCHANNEL)LocalAlloc (LPTR, sizeof(CHANNEL));
if (pChannel == NULL)
    return (PCHANNEL) NULL;

//  make sure the channel is not already in use
switch ( dwFlags &
       ( VIDEO_EXTERNALIN | VIDEO_EXTERNALOUT | VIDEO_IN | VIDEO_OUT) ) {

    case VIDEO_EXTERNALIN:
        if( gwCaptureUsage >= MAX_CAPTURE_CHANNELS)
            goto error;
        gwCaptureUsage++;
        break;

    case VIDEO_EXTERNALOUT:
        if( gwDisplayUsage >= MAX_DISPLAY_CHANNELS)
            goto error;
        gwDisplayUsage++;
        break;
```

```
        case VIDEO_IN:
            if( gwVideoInUsage >= MAX_IN_CHANNELS)
                goto error;
            gwVideoInUsage++;
            break;

        case VIDEO_OUT:
            if( gwVideoOutUsage >= MAX_OUT_CHANNELS)
                goto error;
            gwVideoOutUsage++;
            break;

        default:
            goto error;
    }

    //  Now that the hardware is allocated initialize instance structure

    pChannel->fccType            = OPEN_TYPE_VCAP;
    pChannel->dwOpenType         =
        (dwFlags & (VIDEO_EXTERNALIN|VIDEO_EXTERNALOUT|VIDEO_IN|VIDEO_OUT));
    pChannel->dwOpenFlags        = dwFlags;
    pChannel->lpVHdr             = NULL;
    pChannel->dwError            = 0L;

    gwDriverUsage++;
    return pChannel;

error:
    if (pChannel)
        LocalFree((HLOCAL)pChannel);

    *lpdwError = DV_ERR_ALLOCATED;
    return NULL;
}
```

The following example shows the function used to close the example video capture device driver:

```
DWORD NEAR PASCAL VideoClose(PCHANNEL pChannel)
{
    // Decrement the channel open counters

    switch (pChannel-> dwOpenType) {
```

```
        case VIDEO_EXTERNALIN:
            gwCaptureUsage--;
            break;


        case VIDEO_EXTERNALOUT:
            gwDisplayUsage--;
            break;


        case VIDEO_IN:
            // If started, or buffers in the queue,
            // return error and don't close
            if (gfVideoInStarted || lpVHdrFirst)
                return DV_ERR_STILLPLAYING;

            gwVideoInUsage--;
            break;


        case VIDEO_OUT:
            gwVideoOutUsage--;
            break;

        default:
            break;
    }


    gwDriverUsage--;  // Overall driver useage count

    if (gwDriverUsage == 0) {
        HardwareFini ();         // Shut down the device
        TransFini ();            // Free the translation table
        FreeFrameBufferSelector ();  // Free the frame buffer selector
        fDeviceInitialized = FALSE;
    }

    // Free the instance data
    LocalFree((HLOCAL)pChannel);

    return DV_ERR_OK;
}
```

Handling the DRV_ENABLE and DRV_DISABLE Messages

The example **DriverProc** function calls the functions **Enable** and **Disable** to do the work of enabling and disabling the driver. These functions are device dependent.

Generally, when a driver is enabled, you initialize the hardware, hook interrupts, allocate any memory that you need, and set a flag to indicate the driver is enabled. The exact sequence your device driver will follow is determined by the requirements and structure of your device driver. For example, the Bravado device driver uses interrupts only for

streaming data. When enabled, it will hook its interrupts only if it was disabled while streaming data.

If your driver has not been enabled by MMSYSTEM, or if it failed the enable process, the driver should return MMSYSERR_NOTENABLED for any messages it receives from client applications. When a driver is disabled, you free any memory that you allocated, unhook interrupts, reset the hardware, and set a flag to indicate the driver is not enabled.

It's possible for a driver to receive a DRV_DISABLE message while it is in the process of capturing data. For example, this can happen when the user switches to a MS-DOS application when Windows is running in standard mode. Video capture device drivers should behave as if the driver were stopped with a DVM_STREAM_STOP message and then restarted with a DVM_STREAM_START message when it receives a DRV_DISABLE/DRV_ENABLE message pair.

Driver Configuration

Installable drivers can supply a configuration dialog box for users to access through the Drivers option in the Control Panel. The Drivers option sends the DRV_CONFIGURE message to your driver to display the dialog box.

The dialog box should display the name and version number of your device driver. If your device driver supports different interrupt-level and port assignments, it should also support user configuration through the Drivers option in the Control Panel.

Interrupt-level and port assignments, and any other hardware-related settings, can be stored in a section with the same name as the driver in the user's SYSTEM.INI file. For example, the following SYSTEM.INI section created by the Bravado example driver specifies interrupt level 9 and memory base E:

```
[Bravado.drv]
Interrupt=9
MemoryBase=E
```

Alternatively, your driver might use its own INI file for this information.

Video Capture Driver Messages

This section gives the device driver specific messages for video capture device drivers. See "Video Capture Device Driver Reference," later in this chapter, for detailed information on these messages.

Configuring the Channels of a Video Capture Driver

In addition to the configuration dialog box displayed for the DRV_CONFIGURE message, video capture drivers can display a dialog box for each channel. These dialog boxes are the primary means of setting parameters in your device driver. The following message requests that the device driver display a dialog box:

DVM_DIALOG
Displays a dialog box which controls a video channel.

When your device driver first gets this message, use the handle in *lParam1* to determine which channel is being configured. The Bravado example driver determines the channel from the flags used to open it. It saves these flags as part of its instance data created when it was opened.

The dialog box displayed for each channel sets the characteristics for each channel. If a channel does not support configuration, return DV_ERR_NOTSUPPORTED. The following table suggests the contents of each dialog box:

Channel	Dialog Box Description
VIDEO_EXTERNALIN	Displays a dialog box which controls how video (either analog or digital) is captured. The dialog box might set attributes such as contrast and brightness.
VIDEO_EXTERNALOUT	Displays a dialog box which controls how video is displayed on a second monitor or video adapter such as a video overlay card.
VIDEO_IN	Displays a dialog box which controls how video is transferred from the frame buffer.
VIDEO_OUT	Displays a dialog box which controls how video is transferred to the frame buffer.

When processing the DVM_DIALOG message, check *lParam2* for the VIDEO_DLG_QUERY flag prior to displaying the dialog box. If an application uses this flag, it is only determining if a video channel supports a dialog box. For this flag, return DV_ERR_OK if the video channel supports a dialog box. If not, return DV_ERR_NOTSUPPORTED in response to the message.

The Bravado example driver uses the following function to handle the DVM_DIALOG message (this function is called from the Bravado VideoProcessMessage function):

```
DWORD NEAR PASCAL VideoDialog (DWORD dwOpenType, HWND hWndParent, DWORD
dwFlags)
{

    switch (dwOpenType) {

        case VIDEO_EXTERNALIN:
            if (dwFlags & VIDEO_DLG_QUERY)
                return DV_ERR_OK;        // Channel has a dialog box
            DialogBox(ghModule, MAKEINTRESOURCE(DLG_VIDEOSOURCE),
                    (HWND)hWndParent, VideoSourceDlgProc);
            break;
```

```
        case VIDEO_IN:
            if (dwFlags & VIDEO_DLG_QUERY)
                return DV_ERR_OK;          // Channel has a dialog box
            DialogBox(ghModule, MAKEINTRESOURCE(DLG_VIDEOFORMAT),
                    (HWND)hWndParent, VideoFormatDlgProc);
            break;


        case VIDEO_OUT:
            return DV_ERR_NOTSUPPORTED; //Channel does not have a dialog box


        case VIDEO_EXTERNALOUT:
            if (dwFlags & VIDEO_DLG_QUERY)
                return DV_ERR_OK;          // Channel has a dialog box
            DialogBox(ghModule, MAKEINTRESOURCE (DLG_VIDEODISPLAY),
                    (HWND)hWndParent, VideoMonitorDlgProc);
            break;

        default:
            return DV_ERR_NOTSUPPORTED;
    }
    return DV_ERR_OK;
}
```

Video capture drivers might save the settings from these dialog boxes in the section reserved for your device driver in the SYSTEM.INI file. Your driver should append this information to the entries created for the DVR_CONFIGURE messages to this section. For example, the example Bravado driver might have this section in the SYSTEM.INI file:

```
[Bravado.drv]
Interrupt=9
MemoryBase=E
Hue=10
Saturation=6
InputChannel=2
Contrast=24
```

Alternatively, a device driver might implement its own method of storing configuration information for each channel.

Setting and Obtaining Video Capture Format

The video capture format globally defines the attributes of the images transferred from the frame buffer with the video in channel. Attributes include image dimensions, color depth, and the compression format of images transferred. Applications use the following message to set or retrieve the format of the digitized image:

DVM_FORMAT
  Assigns or obtains format information.

The calling application must modify this message with flags to indicate its purpose. Your driver must examine the flags sent with the message to determine the proper response. The flags are specified in *lParam1*. The following flags help define the meaning of the messages:

VIDEO_CONFIGURE_QUERY
: Determines if the driver supports the message.

VIDEO_CONFIGURE_QUERYSIZE
: Requests the size of the format data structure.

VIDEO_CONFIGURE_SET
: Indicates values are being sent to the driver.

VIDEO_CONFIGURE_GET
: Indicates the application is interrogating the driver.

The VIDEO_CONFIGURE_GET or VIDEO_CONFIGURE_SET flag indicates if the DVM_FORMAT message is being used to obtain the format or set the format. The DVM_FORMAT message and these flags are sent to your driver when it is opened, and when it is configured with DVM_DIALOG.

When an application opens your driver, it retrieves the initial driver format. (Video capture drivers initially default to a format that efficiently uses the capabilities of the video capture hardware or, if they have been previously configured, they restore the last user specified configuration saved in a disk file.) If this format is acceptable, the application continues its operations. If the format is not acceptable, the application will either immediately close your driver or suggest a very limited format. If the limited format is not acceptable to your driver, the application closes it. (Typically, applications do not accept a format because they cannot allocate enough memory to capture video. A limited format might free enough memory for operation.)

Applications also get the format when the user changes the format. (Users change the format with the VIDEO_IN channel dialog box displayed with the DVM_DIALOG message.) In this case, applications get and retain a copy of the current format prior to sending the DVM_DIALOG message. After the user exits from the DVM_DIALOG dialog box, applications get the new format from your driver. If the application accepts the new format, it uses the VIDEO_CONFIGURE_SET flag to return the format back to your driver. (Your driver should verify that the application has not changed the format information.) If the application does not accept the new format, it restores the format it obtained prior to displaying the dialog box.

The DVM_FORMAT messages uses *lParam2* to pass the format information. This parameter contains a pointer to a VIDEOCONFIGPARMS structure. This structure has the following fields:

```
typedef struct tag_video_configure_parms {
    LPDWORD  lpdwReturn;
    LPVOID   lpData1;
    DWORD    dwSize1;
    LPVOID   lpData2;
    DWORD    dwSize2;
} VIDEOCONFIGPARMS;
```

The **lpData1** field points to a BITMAPINFOHEADER data structure. The size of this structure is specified in **dwSize1**.

Changing the format can affect overall dimensions of the active frame buffer as well as bit depth and color space representation. Since changing between NTSC and PAL video standards can also affect image dimensions, applications should request the current format following display of the EXTERNAL_IN channel dialog box.

If an application just wants to know if your driver supports DVM_FORMAT, it sends the VIDEO_CONFIGURE_QUERY flag with the message. (Using the VIDEO_CONFIGURE_QUERY flag without VIDEO_CONFIGURE_GET or VIDEO_CONFIGURE_SET is invalid.) Your device driver should return DV_ERR_OK if it supports the message. Otherwise, it should return DV_ERR_NOTSUPPORTED.

If an application wants to determine the amount of memory it needs to allocate for the format information, it sends the DVM_FORMAT message with the VIDEO_CONFIGURE_GET and VIDEO_CONFIGURE_QUERYSIZE flags set. Your driver should specify the format size in the **lpdwReturn** field of the VIDEOCONFIGUREPARMS structure.

Setting and Obtaining the Video Source and Destination Rectangles

Video capture drivers might support a source rectangle to specify a portion of an image that is digitized or transferred to the display. External in ports use the source rectangle to specify the portion of the analog image digitized. External out ports use the source rectangle to specify the portion of frame buffer shown on the external output.

Similarly, video capture drivers might support a destination rectangle to specify the portion of the frame buffer or screen used to receive the image. External in ports can use a destination rectangle to specify the portion of the frame buffer used for the digitized video input. External out ports can use the rectangle to specify the client rectangle on the display.

The following messages are used to set and obtain the video source and destination rectangles:

DVM_DST_RECT
Sets and retrieves the destination rectangle used by video devices.

DVM_SRC_RECT
Sets and retrieves the source rectangle used by video devices.

The calling application must modify these messages with flags to indicate their exact meaning. Your driver must examine the flags sent with the messages to determine the proper response. The flags are specified in *lParam2*. The following flags define the meaning of the DVM_DST_RECT and DVM_SRC_RECT messages:

VIDEO_CONFIGURE_SET
: Indicates values are being sent to the driver.

VIDEO_CONFIGURE_GET
: Indicates the application is interrogating the driver.

VIDEO_CONFIGURE_QUERY
: Determines if the driver supports the message.

The VIDEO_CONFIGURE_SET flag indicates the application is setting a source or destination rectangle. The rectangle coordinates are specified in a RECT structure pointed to by *lParam1*.

If an application sets a source or destination rectangle for an external out channel, your driver will normally receive a series of messages. For these channels, applications normally send both DVM_SRC_RECT and DVM_DST_RECT to your driver to properly set the rectangles. The application follows these messages with DVM_UPDATE. Video overlay devices should paint their key color in response to DVM_UPDATE.

Applications use VIDEO_CONFIGURE_GET to determine the coordinates of the source and destination rectangles. Applications use additional flags with VIDEO_CONFIGURE_GET to indicate if they want the coordinates of the rectangle currently defined, the maximum size of the rectangle, or the minimum size of the rectangle. The following flags are defined for these operations:

VIDEO_CONFIGURE_MIN
: Used with VIDEO_CONFIGURE_GET to determine the minimum rectangle supported.

VIDEO_CONFIGURE_MAX
: Used with VIDEO_CONFIGURE_GET to determine the maximum rectangle supported.

VIDEO_CONFIGURE_CURRENT
: Used with VIDEO_CONFIGURE_GET to determine the current rectangle.

Your driver should return the coordinates for the appropriate rectangle in the RECT structure pointed to by *lParam1*.

An application uses VIDEO_CONFIGURE_QUERY to determine if your driver supports VIDEO_CONFIGURE_QUERY or VIDEO_CONFIGURE_SET. (The VIDEO_CONFIGURE_QUERY flag without VIDEO_CONFIGURE_GET or VIDEO_CONFIGURE_SET is invalid.) Your device driver should return DV_ERR_OK if it supports the flag. Otherwise, it should return DV_ERR_NOTSUPPORTED.

Determining Channel Capabilities

Channel capabilities include overlaying video, scaling of images with the source and destination rectangles, and clipping of images with the source and destination rectangles. The following message retrieves the channel capabilities of a driver:

DVM_GET_CHANNEL_CAPS
Return the capabilities of a channel to the application.

Applications use DVM_GET_CHANNEL_CAPS to obtain information about the capabilities of a channel. The *lParam1* parameter specifies a far pointer to a

CHANNEL_CAPS data structure and the *lParam2* parameter specifies its size. The **CHANNEL_CAPS** structure has the following fields:

```
typedef struct channel_caps_tag {
    DWORD  dwFlags;
    DWORD  dwSrcRectXMod;
    DWORD  dwSrcRectYMod;
    DWORD  dwSrcRectWidthMod;
    DWORD  dwSrcRectHeightMod;
    DWORD  dwDstRectXMod;
    DWORD  dwDstRectYMod;
    DWORD  dwDstRectWidthMod;
    DWORD  dwDstRectHeightMod;
} CHANNEL_CAPS;
```

Your driver should use the **dwFlags** field to return flags indicating its capabilities for overlaying video, and clipping and scaling images with the source and destination rectangles. The following flags are defined:

VCAPS_OVERLAY
Indicates the channel is capable of overlay. This flag is used only for **EXTERNAL_OUT** channels.

VCAPS_SRC_CAN_CLIP
Indicates that the source rectangle can be set smaller than the maximum dimensions.

VCAPS_DST_CAN_CLIP
Indicates that the destination rectangle can be set smaller than the maximum dimensions.

VCAPS_CAN_SCALE
Indicates that the source rectangle can be a different size than the destination rectangle.

If your driver supports changing the size and position of the source rectangle, it should indicate the finest granularity used for changes to the rectangle in the **dwSrcRectXMod**, **dwSrcRectYMod**, **dwSrcRectWidthMod**, and **dwSrcRectHeightMod** fields.

If your driver supports changing the size and position of the destination rectangle, it should indicate the finest granularity used for changes to the rectangle in the **dwDstRectXMod**, **dwDstRectYMod**, **dwDstRectWidthMod**, and

dwDstRectHeightMod fields. If a channel supports arbitrarily positioned rectangles, with arbitrary sizes, the values above should all be set to 1.

Your driver returns DV_ERR_OK if the message was processed successfully. It returns DV_ERR_NOTSUPPORTED if the message is not supported.

Setting and Obtaining a Video Capture Palette

Applications can set and retrieve the palette used with captured video. This gives applications the ability to control and modify the palette used for video sequences. The palette messages apply only to the video in and video out channels. The following messages apply to the video capture palette:

DVM_PALETTE
Assigns or obtains palette information.

DVM_PALETTERGB555
Associates an RGB555 palette with a video device channel.

The calling application must modify these messages with flags to indicate their purpose. Your driver must examine the flags sent with the messages to determine the proper response. The flags are specified in *lParam1*. The following flags define the meaning of the messages:

VIDEO_CONFIGURE_SET
Indicates values are being sent to the driver.

VIDEO_CONFIGURE_GET
Indicates the application is interrogating the driver.

VIDEO_CONFIGURE_QUERY
Determines if the driver supports the message.

VIDEO_CONFIGURE_QUERYSIZE
Requests the size of the palette data structure.

The VIDEO_CONFIGURE_GET or VIDEO_CONFIGURE_SET flag modifies the DVM_PALETTE message to indicate that the driver should return the current palette or that the driver should set a new palette. The *lParam2* parameter used with DVM_PALETTE contains a pointer to a VIDEOCONFIGPARMS data structure. This structure has the following fields:

```
typedef struct tag_video_configure_parms {
    LPDWORD  lpdwReturn;
    LPVOID   lpData1;
    DWORD    dwSize1;
    LPVOID   lpData2;
    DWORD    dwSize2;
} VIDEOCONFIGPARMS;
```

If the VIDEO_CONFIGURE_SET flag is used with DVM_PALETTE, the **lpData1** field points to a LOGPALETTE structure containing the new palette. The size of the memory allocated for the LOGPALETTE structure is specified in the **dwSize1** field.

If the VIDEO_CONFIGURE_GET flag is used with DVM_PALETTE, the **lpData1** field points to a LOGPALETTE structure used to retrieve the palette. The size of the memory allocated for the LOGPALETTE structure is specified in the **dwSize1** field. Your driver should transfer the palette to the structure indicated by **lpData1**.

If an application just wants to determine the size of the palette, it sends the DVM_PALETTE message with both the VIDEO_CONFIGURE_GET and VIDEO_CONFIGURE_QUERYSIZE flags. Your driver should return the palette size in the **lpdwReturn** field.

If an application just wants to know if your driver supports DVM_PALETTE and its flags, it also sets the VIDEO_CONFIGURE_QUERY flag with VIDEO_CONFIGURE_GET or VIDEO_CONFIGURE_SET. (The VIDEO_CONFIGURE_QUERY flag without VIDEO_CONFIGURE_GET or VIDEO_CONFIGURE_SET is invalid.) Your device driver should return DV_ERR_OK if it supports the DVM_PALETTE message and the operation indicated with the set or get flag. Otherwise, it should return DV_ERR_NOTSUPPORTED.

DVM_PALETTE does not use the **lpData2** and **dwSize2** fields.

Applications use the DVM_PALETTERGB555 message to associate an RGB555 palette with a video device channel. Only the VIDEO_CONFIGURE_SET and VIDEO_CONFIGURE_QUERY flags apply to this message. The VIDEO_CONFIGURE_SET flag modifies the DVM_PALETTERGB555 message to indicate that the driver should set a new palette. The *lParam2* parameter used with DVM_PALETTERGB555 contains a pointer to a VIDEOCONFIGPARMS data structure.

When setting the palette, the **lpData1** field points to a LOGPALETTE structure containing the new palette. The **lpData2** field points to a 32 kilobyte RGB555 translation table. The device driver uses this table to translate the RGB555 triples into palette colors when capturing data in an 8 bit palette mode. The **dwSize1** and **dwSize2** fields specify the size of the structures indicated by **lpData1** and **lpData2**.

If an application just wants to know if your driver supports DVM_PALETTERGB555, it sends the VIDEO_CONFIGURE_QUERY flag with VIDEO_CONFIGURE_SET. (The VIDEO_CONFIGURE_QUERY flag without VIDEO_CONFIGURE_SET is invalid.) Your device driver should return DV_ERR_OK if it supports the DVM_PALETTERGB555 message. Otherwise, it should return DV_ERR_NOTSUPPORTED.

The following example shows how the Bravado device driver handles the DVM_PALETTE message. The structure for DVM_PALETTERGB555 is similar.

```
DWORD NEAR PASCAL VideoConfigureMessage(PCHANNEL pChannel, UINT msg, LONG
lParam1, LONG lParam2)
{
    LPVIDEOCONFIGPARMS lpcp;
    LPDWORD      lpdwReturn;  // Return parameter from configure
    LPVOID       lpData1;     // Pointer to data1
    DWORD        dwSize1;     // size of data buffer1
    LPVOID       lpData2;     // Pointer to data2
    DWORD        dwSize2;     // size of data buffer2
    DWORD        dwFlags;


    if (pChannel-> dwOpenType != VIDEO_IN)
        return DV_ERR_NOTSUPPORTED;

    dwFlags = lParam1;

    lpcp = (LPVIDEOCONFIGPARMS) lParam2;
    lpdwReturn = lpcp-> lpdwReturn;
    lpData1 = lpcp-> lpData1;
    dwSize1 = lpcp-> dwSize1;
    lpData2 = lpcp-> lpData2;
    dwSize2 = lpcp-> dwSize2;


switch (msg) {

    case DVM_PALETTE:

        switch (dwFlags) {

            case (VIDEO_CONFIGURE_QUERY | VIDEO_CONFIGURE_SET):
            case (VIDEO_CONFIGURE_QUERY | VIDEO_CONFIGURE_GET):
                return DV_ERR_OK;

            case VIDEO_CONFIGURE_QUERYSIZE:
            case (VIDEO_CONFIGURE_QUERYSIZE | VIDEO_CONFIGURE_GET):
               *lpdwReturn = sizeof(LOGPALETTE) +
                    (palCurrent.palNumEntries-1) *
                    sizeof(PALETTEENTRY);
               break;


            case VIDEO_CONFIGURE_SET:
            case (VIDEO_CONFIGURE_SET | VIDEO_CONFIGURE_CURRENT):
                if (!lpData1)     // points to a LOGPALETTE structure.
                    return DV_ERR_PARAM1;
                return (SetDestPalette ( (LPLOGPALETTE) lpData1,
                   (LPBYTE) NULL));
                break;
```

```
            case VIDEO_CONFIGURE_GET:
            case (VIDEO_CONFIGURE_GET | VIDEO_CONFIGURE_CURRENT):
               return (GetDestPalette ( (LPLOGPALETTE) lpData1,
                        (WORD) dwSize1));
               break;


            default:
               return DV_ERR_NOTSUPPORTED;

        } // end of DVM_PALETTE switch

        return DV_ERR_OK;
        .
        .
        .


    default:        // Not a message that we understand
        return DV_ERR_NOTSUPPORTED;

    } // end of message switch

    return DV_ERR_NOTSUPPORTED;
}
```

Obtaining the Device Driver Version

The following message lets an application interrogate your device driver to determine the version of the video capture command set.

DVM_GETVIDEOAPIVER
  Obtains the version of the video capture command set.

Your driver should return VIDEOAPIVERSION in the DWORD buffer that *lParam1* points to. This message does not have any flags associated with it.

Transferring Data From the Frame Buffer

The following message lets an application obtain a single frame from the frame buffer:

DVM_FRAME
  Obtains a single frame from the frame buffer.

This message is the basis for the simplest form of video capture. Applications might use this to record animated sequences created frame-by-frame or to capture a single still image such as a photograph. The following sequence of operations occurs when a client application requests the transfer of a single video frame:

1. The client allocates the memory for the data buffer.
2. The client sets a pointer to the empty data buffer in the VIDEOHDR data structure.

3. The client sends the device driver a pointer to the VIDEOHDR data structure with the **videoFrame** function. (The destination channel must be a VIDEO_IN channel.)
4. When the device driver receives the DVM_FRAME messages that Windows sends in response to **videoFrame**, it fills the data buffer with information from the frame buffer and updates the VIDEOHDR data structure. Note that the buffer might not have been prepared.
5. When the device driver has filled a data buffer, it returns from the DVM_FRAME message. This returns control back to the client.
6. After the client has finished with the data, it frees the memory used for the data.

Streaming Video Capture

Video capture device drivers use the DVM_STREAM messages sent to a VIDEO_IN channel to stream full motion video to the client application. Your device driver will use the following messages while it is streaming video:

DVM_STREAM_INIT
Initializes a video input stream.

DVM_STREAM_PREPAREHEADER
Requests that the driver prepare a data buffer for input.

DVM_STREAM_ADDBUFFER
Adds a buffer to the video input stream queue.

DVM_STREAM_START
Begins streaming video input.

DVM_STREAM_STOP
Ends video input streaming.

DVM_STREAM_UNPREPAREHEADER
Requests that a driver clean up the preparation previously done on a data buffer.

DVM_STREAM_FINI
Closes and deallocates a video stream.

The Data Transfer Model For Streaming Video Input

The data transfer model for streaming video input is similar to the model defined for the waveform device drivers. If you have worked with the waveform device drivers, many of the concepts used there will be usable with video capture device drivers.

The following sequence of operations occurs when streaming video data between a video capture device driver and a client application:

1. The client allocates the memory buffers for the video data.
2. The client initializes the data stream (DVM_STREAM_INIT).
3. The client requests that the driver prepare the data buffers (DVM_STREAM_PREPAREHEADER).
4. The client sends the empty data buffers to the driver (DVM_STREAM_ADDBUFFER).

5. The driver puts the data buffers in its input queue.
6. When the streaming operation begins with DVM_STREAM_START, the driver fills a data buffer and sets the done bit for the data buffer. The driver will then release the buffer from its queue and proceed to fill the next buffer.
7. When the client is ready for data, it uses the done bit or callback to see if the data in the buffer is ready.
8. After the client empties the buffer, it resets the done bit and sends the empty buffer back to the driver for it to add to its queue (DVM_STREAM_ADDBUFFER).

Once the stream starts, the client application and the video capture driver do not communicate directly. The video capture driver fills the data buffers at the rate specified by the client application using the frame rate information provided with the DVM_STREAM_INIT message. It fills the buffers without waiting for any synchronization signal from the application as long as buffers are available and it is not paused or stopped by the application. The buffers are filled in the order that the driver receives them from the application. (If the device driver runs out of buffers, it should set an error flag. A client application can use the DVM_STREAM_GETERROR message to test for this condition.)

The client application expects the buffers back in the order that it sends them to the device driver. When it is ready for more data, it will check the done bit of the next buffer it expects to receive from the device driver. If the done bit is set, the application continues operation with that buffer. If the done bit is not set, the application will periodically check the done bit while it waits for the buffer.

Streaming continues until it is stopped by the application. The following sequence of operations occurs when the application has finished capturing data:

- When the client stops the streaming operation with DVM_STREAM_STOP, the driver stops filling buffers.
- If the client wants to restart streaming, it sends DVM_STREAM_START. If the client is finished streaming, it requests that the driver unprepare the data buffers (DVM_STREAM_UNPREPAREHEADER).
- The client releases the data stream (DVM_STREAM_FINI) and frees the memory allocated for the video data.

Initializing the Data Stream

The DVM_STREAM_INIT message initializes a video device for data streaming. This message must precede all other streaming messages for a channel.

The *lParam1* parameter of DVM_STREAM_INIT specifies a far pointer to a VIDEO_STREAM_INIT_PARMS structure and the *lParam2* specifies its size in bytes. The VIDEO_STREAM_INIT_PARMS structure has the following fields:

```
typedef struct tag_video_stream_init_parms {
    DWORD  dwMicroSecPerFrame;
    DWORD  dwCallback;
    DWORD  dwCallbackInst;
    DWORD  dwFlags;
    DWORD  hVideo;
} VIDEO_STREAM_INIT_PARMS;
```

The different channels handle the message and data structure in different ways.

For external in channels, DVM_STREAM_INIT enables capture of images into the frame buffer. External in channels should expect this message at any time. They can ignore the **dwMicroSecPerFrame, dwCallback**, and **dwCallbackInst** fields. The **dwFlags** field must contain the VIDEO_ALLOWSYNC flag for synchronous devices.

For video in channels, DVM_STREAM_INIT sets the capture rate and callback information. The **dwMicroSecPerFrame** field specifies the number of microseconds between successive capture frames. The **dwCallback** field contains the address of a callback function or the handle to a window called during video streaming. (This parameter is set to NULL if a callback function or window is not used.) The callback procedure processes any messages related to the progress of recording. If a callback function address is specified, **dwFlags** is set to CALLBACK_FUNCTION. If the application has any data to pass to the callback function, it specifies the data in **dwCallbackInst**. If a callback window handle is specified, **dwFlags** is set to CALLBACK_WINDOW. Drivers can also use **DriverCallback** to send a message to a window or callback function. For more information on **DriverCallback**, see the *Windows Multimedia Device Adaptation Guide.* For more information on using the video capture callback, see the "Video Capture Device Driver Reference."

For external out channels, DVM_STREAM_INIT enables overlay display. External out channels should expect this message at any time. They can ignore the **dwMicroSecPerFrame, dwCallback**, and **dwCallbackInst** fields. The **dwFlags** field contains any flags that might affect the external out channel.

All channels return DV_ERR_OK if the message was processed successfully. All channels should return DV_ERR_ALLOCATED if the channel is already allocated or DV_ERR_NOMEM if they are unable to allocate or lock memory.

Preparing Data Buffers

Because video data buffers must be accessed at interrupt time, the memory allocated for them is subject to the requirements mentioned previously in "Considerations for Interrupt-Driven Drivers." Rather than have the client application prepare the memory before sending data blocks to the driver, the client requests that the driver do the preparation.

Most drivers can respond to the DVM_STREAM_PREPAREHEADER and DVM_STREAM_UNPREPAREHEADER messages) by returning a DV_ERR_UNSUPPORTED error. When your driver returns DV_ERR_UNSUPPORTED, the system will perform the necessary preparation on the data block. This consists of page locking the header and data sections so the driver can access them at interrupt time.

If your device driver does not need the data to be page locked (for example, if you immediately copy the data to an on-card buffer) or if you have additional preparation to do to the buffer, you might respond to these messages yourself instead of having the system handle them. You should respond to both DVM_STREAM_PREPAREHEADER and DVM_STREAM_UNPREPAREHEADER, or to neither.

Starting and Stopping Streaming

DVM_STREAM_START starts a video stream. For video in channels, this message begins transferring the contents of the frame buffer to the system supplied buffers. In response to DVM_STREAM_START, your driver should enable the interrupts it needs and begin capturing the images and copying them to the application supplied buffers.

DVM_STREAM_STOP stops a video stream. When a video in channel receives this message, it stops filling buffers and retains any empty buffers remaining in the queue. Your driver can disable any interrupts it needs to capture data, however, it should be prepared to receive the DVM_STREAM_START message to resume capturing data. If data capture has not started, this message has no effect and the device driver returns DV_ERR_OK.

Neither DVM_STREAM_START nor DVM_STEAM_STOP use *lParam1* or *lParam2*. Your driver should return DV_ERR_OK if it processed the message successfully. It should return DV_ERR_NOTSUPPORTED if it does not support the message.

Ending Capture

The DVM_STREAM_FINI message terminates data streaming. This should always be the last streaming message received by a channel.

For external in channels, DVM_STREAM_FINI disables capture of images into the frame buffer. External in channels should expect this message at any time. DVM_STREAM_FINI might not have a corresponding DVM_STREAM_INIT message.

For video in channels, DVM_STREAM_INIT finishes data streaming process. Your driver can free any resources that it used to capture data.

For external out channels, DVM_STREAM_INIT disables overlay display. External out channels should expect this message at any time. DVM_STREAM_FINI might not have a corresponding DVM_STREAM_INIT message.

All channels return DV_ERR_OK if the message was processed successfully. The video in channels should return DV_ERR_STILLPLAYING if there are still buffers in its queue.

Additional Stream Messages

The following messages are used in support of data streaming:

DVM_STREAM_RESET
  Stops video input streaming and returns all data buffers to the client application.

DVM_STREAM_GETERROR
  Returns the error encountered while streaming data.

DVM_STREAM_GETPOSITION
  Requests the current position in the video stream.

The client application uses DVM_STREAM_RESET to stop data streaming and release all buffers. When your driver gets this message it should return to the state set with DVM_STREAM_INIT.

The client application uses DVM_STREAM_GETERROR to obtain the error status of a channel. The *lParam1* and *lParam2* parameters point to two DWORDS your driver should use to return error information. Fill the DWORD specified by *lParam1* with the value of the most recent error. Typically, the error encountered is DV_ERR_NO_BUFFERS. If your driver has not encountered an error or if it receives this message when a stream is not initialized, set the DWORD to DV_ERR_OK. Fill the DWORD specified by *lParam2* with the number of frames dropped because of the error.

After processing this message your driver should reset its error value and count of frames dropped. Drivers that do not have access to interrupts might use this message to trigger buffer processing.

Return DV_ERR_OK if your driver processes the message without an error. If your driver does not support this message, return DV_ERR_NOTSUPPORTED.

Applications use the DVM_STREAM_GETPOSITION message to retrieve the current position of the video in stream. The *lParam1* parameter specifies a far pointer to a **MMTIME** data structure and the *lParam2* parameter specifies its size. The **MMTIME** structure has the following fields:

```
typedef struct mmtime_tag {
    UINT  wType;
    union {
        DWORD  ms;
        DWORD  sample;
        DWORD  cb;
```

```
        struct {
            BYTE  hour;
            BYTE  min;
            BYTE  sec;
            BYTE  frame;
            BYTE  fps;
            BYTE  dummy;
        } smpte;
        struct {
            DWORD  songptrpos;
        } midi;
    } u;
} MMTIME;
```

When your device gets DVM_STREAM_POSITION, it should check the **wtype** field. If your driver does not support the format specified, it specifies its current time format in the field. The application checks the format specified in this field when the message returns.

Video capture drivers typically return time in the millisecond format. Normally, your driver sets the position to zero when streaming starts with DVM_STREAM_START.

Your driver should returns DV_ERR_OK if it processed the message successfully. It can return DV_ERR_PARM1 if the data structure supplied for the format has invalid data or DV_ERR_SIZEFIELD if the data structure is too small.

Video Capture Device Driver Reference

This section is an alphabetic reference to the messages and data structures provided by Windows for use by video capture device drivers. There are separate sections for messages and data structures. The messages and data structures are defined in MSVIDDRV.H and MSVIDEO.H.

Video Capture Device Driver Message Reference

Windows communicates with video capture device drivers through messages sent to the driver. The driver processes these messages with its **DriverProc** entry-point function.

This section contains an alphabetical list of the video capture messages that can be received and sent by video capture device drivers. Each message name contains a prefix, identifying the type of the message.

A message consists of three parts: a message number and two DWORD parameters. Message numbers are identified by predefined message names. The two DWORD parameters contain message-dependent values.

Message Summary

The following messages are used for error handling:

DVM_GETERRORTEXT
This message retrieves a string which contains the description of an error.

DVM_STREAM_GETERROR
This message returns the last error encountered by a channel.

The following messages are used for configuring the device driver and obtaining information from it:

DVM_DIALOG
This message displays a dialog box which controls video parameters for a channel.

DVM_DST_RECT
This message sets and retrieves the destination rectangle used by a video device channel.

DVM_FORMAT
This message is for configuring the format of the video device channel.

DVM_GET_CHANNEL_CAPS
This message is used to return the capabilities of a channel to the application.

DVM_GETVIDEOAPIVER
This message returns the version of the video API used by the driver.

DVM_PALETTE
This message sets and retrieves a logical palette used by a video device channel.

DVM_PALETTERGB555
This message associates an RGB555 palette with a video device channel.

DVM_SRC_RECT
This message sets and retrieves the source rectangle used by a video device channel.

The following messages are used for capturing data:

DVM_FRAME
This message processes a single frame from the video device.

DVM_STREAM_ADDBUFFER
This message sends an input buffer to a video device.

DVM_STREAM_FINI
This message terminates streaming on a video channel.

DVM_STREAM_GETPOSITION
This message retrieves the current position of the stream.

DVM_STREAM_INIT
This message initializes a video device for streaming.

DVM_STREAM_PREPAREHEADER
This message prepares an input buffer for video streaming.

DVM_STREAM_RESET
This message stops input of a video stream and resets the current position to 0.

DVM_STREAM_START
This message starts a video stream.

DVM_STREAM_STOP
This message stops a video stream.

DVM_STREAM_UNPREPAREHEADER
This message cleans up the preparation performed by **DVM_STREAM_PREPAREHEADER**.

DVM_UPDATE
This message is used with a **EXTERNAL_OUT** channel to indicate that the display needs to be updated.

The following messages are used with video callback functions:

MM_DRVM_CLOSE
This message is sent to a video callback function or window when a video channel is closed.

MM_DRVM_DATA
This message is sent to a video callback function or window when the specified buffer is being returned to the application.

MM_DRVM_ERROR
This message is sent to a video callback function or window when an error has occurred.

MM_DRVM_OPEN
This message is sent to a video callback function or window when a video channel is opened.

Video Capture Device Driver Messages

This section contains an alphabetical list of the video capture messages that can be received and sent by video capture device drivers. Each message name contains a prefix, identifying the type of the message.

A message consists of three parts: a message number and two DWORD parameters. Message numbers are identified by predefined message names. The two DWORD parameters contain message-dependent values.

DVM_DIALOG

This message displays a dialog box for setting the video parameters of a channel.

Parameters

DWORD *dwParam1*
Specifies the handle to the parent window.

DWORD *dwFlags*
Specifies flags for the dialog box. The following flag is defined:

VIDEO_DLG_QUERY
If this flag is set, the driver immediately returns DV_ERR_OK if it supplies a dialog box for the channel, or DV_ERR_NOTSUPPORTED if it does not.

Return Value

Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_INVALHANDLE
Specified device handle is invalid.

DV_ERR_NOTSUPPORTED
Message is not supported.

Comments

Typically, this dialog box lets the user configure a video channel. For example, a VIDEO_IN channel might supply a dialog box to let the user select image dimensions and bit depth. Each channel type (VIDEO_IN, VIDEO_OUT, VIDEO_EXTERNALIN, and VIDEO_EXTERNALOUT) can have a unique configuration dialog box.

DVM_DST_RECT

This message sets and retrieves the destination rectangle used by a video device channel.

Parameters

LPRECT lpDstRect
A far pointer to a **RECT** structure.

DWORD *dwFlags*
Specifies flags that indicate the type of transfer requested. Either the VIDEO_CONFIGURE_SET or the VIDEO_CONFIGURE_GET flag must be set, specifying the direction of the transfer. The following flags are defined:

VIDEO_CONFIGURE_SET
Send a rectangle to the device driver.

VIDEO_CONFIGURE_GET
Get the current rectangle from the device driver.

VIDEO_CONFIGURE_MIN
Get the minimum destination rectangle from the device driver.

VIDEO_CONFIGURE_MAX
Get the maximum destination rectangle from the device driver.

VIDEO_CONFIGURE_CURRENT
Get or set the current destination rectangle.

VIDEO_CONFIGURE_QUERY
This flag is used to query the device driver to determine if it supports the message.

Return Value

Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_NOTSUPPORTED
: Message is not supported.

Comments

The use of the destination rectangle for a channel depends on the channel type. For the VIDEO_EXTERNALIN channel, the destination rectangle specifies the location in the frame buffer used to digitize the image. This rectangle is specified in pixel coordinates.

For the VIDEO_EXTERNALOUT channel, the destination rectangle specifies the location used to display the overlay image. This rectangle is given in Windows screen coordinates.

For the VIDEO_IN and VIDEO_OUT channels, the destination rectangle is currently undefined.

DVM_FORMAT

This message is used for configuring the format of the VIDEO_IN channel.

Parameters

DWORD *dwFlags*

Specifies flags to indicate the type of format transfer requested. Either the VIDEO_CONFIGURE_SET or the VIDEO_CONFIGURE_GET flag must be set, specifying the direction of the transfer. The following flags are defined:

VIDEO_CONFIGURE_SET
: Set the current format.

VIDEO_CONFIGURE_GET
: Get the current format.

VIDEO_CONFIGURE_QUERY
: Queries the driver whether it supports the message.

VIDEO_CONFIGURE_QUERYSIZE
: Returns the size in bytes of the format in **lpdwReturn**. This flag must be used with VIDEO_CONFIGURE_GET.

LPVIDEOCONFIGPARMS *lpVConfigParms*

Specifies a far pointer to a **VIDEOCONFIGPARMS** structure. This structure has the following fields:

lpdwReturn
: Specifies a far pointer to a **DWORD**. If the **VIDEO_CONFIGURE_QUERYSIZE** flag is used, the driver fills this field with the size (in bytes) of the **BITMAPINFOHEADER** data structure.

lpData1
: Specifies a far pointer to a **BITMAPINFOHEADER** data structure.

dwSize1
: Specifies the size in bytes of the **BITMAPINFOHEADER** data structure.

lpData2
: Not used.

dwSize2
Not used.

Return Value

Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_NOTSUPPORTED
Message is not supported.

Comments

The **DVM_FORMAT** message globally defines the attributes of the frame buffer. This includes dimensions, color depth, and compression of images transferred with **DVM_FRAME** and buffers transferred during streaming capture. Changing the format may affect overall dimensions of the active frame buffer as well as bit depth and color space representation. Since changing between NTSC and PAL video standards can also affect image dimensions, applications should request the current format following display of the VIDEO_EXTERNALIN channel dialog box.

DVM_FRAME

This message transfers a single frame from the video device.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a **VIDEOHDR** structure identifying the buffer.

DWORD *dwParam2*
Contains the size of the **VIDEOHDR** structure.

Return Value

Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_SIZEFIELD
Specified field size is too small.

Comments

This message returns immediately after transferring the frame. For a VIDEO_IN channel, this message transfers an image from the hardware frame buffer to the buffer specified in the **VIDEOHDR**. For a VIDEO_OUT channel, this message transfers an image from the buffer specified in the **VIDEOHDR** to the hardware frame buffer.

DVM_GET_CHANNEL_CAPS

This message is used to return the capabilities of a channel to the application.

Parameters

LPCHANNEL_CAPS *lpChannelCaps*
Specifies a far pointer to a **CHANNEL_CAPS** data structure.

DWORD *dwSize*
Specifies the size of the **CHANNEL_CAPS** data structure.

Return Value

Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_NOTSUPPORTED
Message is not supported.

DVM_GETERRORTEXT

This message retrieves a string describing an error.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a **VIDEO_GETERRORTEXT_PARMS** structure. The structure identifies the error number and return buffer.

DWORD *dwParam2*
Not used.

Return Value

Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_BADERRNUM
Indicates the specified error number is out of range.

Comments

If the error description is longer than the specified buffer, the description is truncated. The returned error string is always null-terminated. If the size of the return buffer is zero, a string description is not returned and DV_ERR_OK is used as the return value.

DVM_GETVIDEOAPIVER

This message returns the version of the video capture command set used by the driver.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a DWORD which will be filled with the version.

DWORD *dwParam2*
Not used.

Return Value

Returns DV_ERR_OK if the message is successful.

DVM_PALETTE

This message sets and retrieves a logical palette used by a video device channel. This message applies only to VIDEO_IN and VIDEO_OUT channels.

Parameters

DWORD *dwFlags*
Specifies any flags that indicate the type of palette transfer requested. Either the VIDEO_CONFIGURE_SET or the VIDEO_CONFIGURE_GET flag must be set, specifying the direction of the transfer. The following flags are defined:

VIDEO_CONFIGURE_SET
Send a palette to the driver.

VIDEO_CONFIGURE_GET
Get the current palette from the driver.

VIDEO_CONFIGURE_QUERY
This flag is used to query the driver to determine if it supports the message.

VIDEO_CONFIGURE_QUERYSIZE
Returns the size in bytes of the palette in **lpdwReturn**. This flag is only valid if the VIDEO_CONFIGURE_GET flag is also set.

LPVIDEOCONFIGPARMS *lpVConfigParms*
A far pointer to a **VIDEOCONFIGPARMS** structure. The **VIDEOCONFIGPARMS** structure has the following fields:

lpdwReturn is a far pointer to a **DWORD**. If the VIDEO_CONFIGURE_QUERYSIZE flag is used, the driver fills this field with the size of the logical palette (in bytes).

lpData1 is a far pointer to a **LOGPALETTE** structure.

dwSize1 is the size in bytes of the **LOGPALETTE**.

lpData2 is not used.

dwSize2 is not used.

Return Value

Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_NOTSUPPORTED
Message is not supported.

Comments

A palette is used when converting between frame buffer internal data formats and 8-bit palettized DIBs.

See Also

DVM_PALETTERGB

DVM_PALETTERGB555

This message associates an RGB555 palette with a video device channel. Applications can provide an RGB555 translation table to a driver for fast conversions between RGB formats and 8 bit palettized formats. This message applies only to VIDEO_IN and VIDEO_OUT channels.

Parameters

DWORD *dwFlags*
Specifies the flags indicating the type of palette transfer requested. The following flags are defined:

VIDEO_CONFIGURE_SET
Indicates values are being sent to the driver.

VIDEO_CONFIGURE_QUERY
This flag, when combined with VIDEO_CONFIGURE_SET is used to query the driver to determine if it supports the message.

LPVIDEOCONFIGPARMS *lpVConfigParms*
Specifies a far pointer to a **VIDEOCONFIGPARMS** data structure. This data structure has the following fields:

lpdwReturn
Not used.

lpData1
Specifies a far pointer to a **LOGPALETTE** data structure.

dwSize1
Specifies the size (in bytes) of the **LOGPALETTE** data structure.

lpData2
Specifies a far pointer to a 32 kilobyte RGB555 translation table. This table is used by the device driver to translate from RGB555 triplets into palette colors when capturing in 8 bit palette mode.

dwSize2
Specifies the size of the translate table in bytes. This value must be 32,768.

Return Value

Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_NOTSUPPORTED
Message is not supported.

DV_ERR_CREATEPALETTE
The device driver was not able to associate the palette with the video device channel.

DV_ERR_PARM1
The information supplied for *dwParam1* is invalid.

DV_ERR_PARM2
The information supplied for *dwParam2* is invalid.

DV_ERR_SIZEFIELD
The data structure supplied for the format is too small.

Comments

A translation table provides a fast method of converting between RGB and palettized color spaces. The palette index corresponding to an RGB color is found by indexing the translation table at xRRRRRGGGGGBBBBB (the five most significant bits of each color component is used to create the index).

DVM_SRC_RECT

This message sets and retrieves the source rectangle used by a video device channel.

Parameters

LPRECT lpSrcRect
A far pointer to a **RECT** structure.

DWORD *dwFlags*
Specifies flags that indicate the type of transfer requested. Either the VIDEO_CONFIGURE_SET or the VIDEO_CONFIGURE_GET flag must be set, specifying the direction of the transfer. The following flags are defined:

VIDEO_CONFIGURE_SET
Send a source rectangle to the device driver.

VIDEO_CONFIGURE_GET
Get the current source rectangle from the device driver.

VIDEO_CONFIGURE_MIN
Get the minimum source rectangle from the device driver.

VIDEO_CONFIGURE_MAX
Get the maximum source rectangle from the device driver.

VIDEO_CONFIGURE_CURRENT
Get or set the current source rectangle.

VIDEO_CONFIGURE_QUERY
This flag is used to query the driver to determine if it supports the message.

Return Value
Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_NOTSUPPORTED
Message is not supported.

Comments
The use of the source rectangle for a channel depends on the channel type. For the VIDEO_EXTERNALOUT channel, the source rectangle specifies the portion of the frame buffer displayed in the overlay window, in pixel coordinates. For the VIDEO_IN, VIDEO_EXTERNALIN, and VIDEO_OUT channels, the source rectangle is currently undefined.

DVM_STREAM_ADDBUFFER

This message sends an input buffer to a video device. When the buffer is filled, the device sends it back to the application.

Parameters
DWORD *dwParam1*
Specifies a far pointer to a **VIDEOHDR** structure identifying the buffer.

DWORD *dwParam2*
Specifies the size of the **VIDEOHDR** structure.

Return Value
Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_NONSPECIFIC
A buffer is not specified.

DV_ERR_UNPREPARED
The buffer was not prepared.

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

DVM_STREAM_FINI

This message terminates streaming on a video channel. This should always be the last streaming message received by a channel.

Parameters
DWORD *dwParam1*
Not used.

DWORD *dwParam2*
Not used.

Return Value
Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_STILLPLAYING
There are still buffers in the queue.

Comments

If all the input buffers sent with **DVM_STREAM_ADDBUFFER** haven't been returned to the application, your driver should fail the message. Client applications should send **DVM_STREAM_RESET** to mark all pending buffers as done prior to sending **DVM_STREAM_FINI**.

For VIDEO_EXTERNALIN channels, this message halts capturing of data to the frame buffer.

For VIDEO_EXTERNALOUT channels that support overlay, this message disables the overlay video.

See Also

videoStreamInit

DVM_STREAM_GETERROR

This message returns the error status of a channel.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a DWORD that the device will fill with the value of the most recent error.

DWORD *dwParam2*
Specifies a far pointer to a DWORD that the device will fill with the number of frames dropped.

Return Value

Returns DV_ERR_OK if there is no error. Otherwise, it returns an error number. The following error is defined:

DV_ERR_NOTSUPPORTED
Message is not supported.

Comments

A device driver should reset its internal error values and count of frames dropped to zero after it processes this message.

Client applications should send this message frequently during capture since some device drivers that do not have access to interrupts use this message to trigger buffer processing.

DVM_STREAM_GETPOSITION

This message retrieves the current position of the VIDEO_IN stream.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a **MMTIME** structure.

DWORD *dwParam2*
Specifies the size in bytes of the **MMTIME** structure.

Return Value

Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_PARM1
The data structure supplied for the format has invalid data.

DV_ERR_SIZEFIELD
The data structure supplied for the format is too small.

Comments

If a device does not support the format specified in the **wtype** field of the **MMTIME** data structure it specifies the current time format in the field. The application checks the format specified in this field when the message returns. Video capture drivers typically return time in the milliseconds format.

The device sets the position to zero when it receives the **DVM_STREAM_START** message.

DVM_STREAM_INIT

This message initializes a video device for streaming. This message must precede all other streaming messages for a channel.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a **VIDEO_STREAM_INIT_PARMS** structure. This structure has the following fields:

dwMicroSecPerFrame
Contains the number of microseconds between successive capture frames.

dwCallback
Specifies the address of a callback function or the handle to a window called during video streaming to process messages related to the progress of recording. This parameter can be NULL.

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

dwFlags
Specifies flags for opening the device. The following flags are defined:

CALLBACK_WINDOW

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

CALLBACK_FUNCTION

If this flag is specified, **dwCallback** is a callback function address.

DWORD *dwParam2*
Specifies the size, in bytes, of the data structure.

Return Value

Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_ALLOCATED
Specified resource is already allocated.

DV_ERR_NOMEM
Unable to allocate or lock memory.

Comments

If a window or callback function will receive callback messages, the device driver uses the following messages to indicate the progress of video input: **MM_DRVM_OPEN**, **MM_DRVM_CLOSE**, **MM_DRVM_DATA**, and **MM_DRVM_ERROR**.

If a callback function is used, it must reside in a DLL. You do not have to use **MakeProcInstance** to get a procedure-instance address for the callback function.

For VIDEO_EXTERNALIN channels, DVM_STREAM_INIT triggers capturing of data to the frame buffer.

For VIDEO_EXTERNALOUT channels with overlay capabilities, DVM_STREAM_INIT enables the overlay.

DVM_STREAM_PREPAREHEADER

This message prepares an input buffer for video streaming.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a **VIDEOHDR** structure identifying the buffer.

DWORD *dwParam2*
Specifies the size of the **VIDEOHDR** structure.

Return Value

Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_NOMEM
Unable to allocate or lock memory.

DV_ERR_NOTSUPPORTED
Unable to prepare data block. (This return lets MSVIDEO prepare the data block.)

Comments

The **VIDEOHDR** data structure and the data block pointed to by its **lpData** field must be allocated with **GlobalAlloc** using the GMEM_MOVEABLE and GMEM_SHARE flags, and locked with **GlobalLock**. Preparing a header previously prepared will have no effect, and the message will return zero. Typically, this operation is used to ensure that the buffer will be available for use at interrupt time.

DVM_STREAM_RESET

This message stops input of a video stream and resets the current position to 0. All pending buffers are marked as done and returned to the application.

Parameters

DWORD *dwParam1*
Not used.

DWORD *dwParam2*
Not used.

Return Value

Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_NOTSUPPORTED
Message is not supported.

Comments

When a device driver receives this message, it should return to the state established for **DVM_STREAM_INIT**.

DVM_STREAM_START

This message starts a video stream.

Parameters

DWORD *dwParam1*
Not used.

DWORD *dwParam2*
Not used.

Return Value

Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_NOTSUPPORTED
Message is not supported.

Comments

For VIDEO_IN channels, this message begins transferring the contents of the frame buffer to the system supplied buffers.

DVM_STREAM_STOP

This message stops a video stream.

Parameters

DWORD *dwParam1*
Not used.

DWORD *dwParam2*
Not used.

Return Value

Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_NOTSUPPORTED
Message is not supported.

Comments

When a device receives this message, it marks the current buffer as done and retains any empty buffers remaining in the queue. For the buffer marked as done, the device places the actual length of the data in the **dwBytesUsed** field of the **VIDEOHDR** structure.

If the input is not started, this message has no effect and the device driver returns DV_ERR_OK.

DVM_STREAM_UNPREPAREHEADER

This message cleans up the preparation performed by **DVM_STREAM_PREPAREHEADER**.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a **VIDEOHDR** structure identifying the buffer.

DWORD *dwParam2*
Specifies the size of the **VIDEOHDR** structure.

Return Value

Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following errors are defined:

DV_ERR_STILLPLAYING
The data buffer is still in the device driver's available queue.

DV_ERR_NOTSUPPORTED
Unable to handle data block preparation. (This return lets MSVIDEO unprepare the data block.)

Comments This message is the complementary message to **DVM_STREAM_PREPAREHEADER**. This message unlocks the data buffer. Unpreparing a buffer not previously prepared has no effect, and the device driver returns DRV_ERR_OK.

DVM_UPDATE

This message is used with a VIDEO_EXTERNALOUT channel to indicate the display needs updating. This is typically sent to an overlay device whenever its client window is moved, sized, or requires painting.

Parameters HWND hWndClient
Specifies a window handle to the client window in which the VIDEO_EXTERNALOUT channel is displayed.

HDC *hDC*
Specifies the device context to be repainted.

Return Value Returns DV_ERR_OK if the message was successful. Otherwise, it returns an error number. The following error is defined:

DV_ERR_NOTSUPPORTED
Message is not supported.

Comments This message is sent to a driver when video overlay is enabled and the overlay key color might need updating. Painting the key color is the responsibility of the driver. An application initiates this message whenever it receives a **WM_PAINT**, **WM_MOVE**, **WM_POSITIONCHANGED**, or **WM_SIZE** message.

This message always follows **DVM_SRC_RECT** and **DVM_DST_RECT** messages for the VIDEO_EXTERNALOUT channel.

The **DVM_STREAM_INIT** and **DVM_STREAM_FINI** messages are used to enable and disable the overlay.

MM_DRVM_CLOSE

This message is sent by a driver to a video callback function or window when a DVM_STREAM_FINI message is received.

Parameters DWORD *dwParam1*
Not used.

DWORD *dwParam2*
Not used.

Comments This message is used by video capture drivers, installable compressors, and other types of installable drivers whenever a device is closed.

MM_DRVM_DATA

This message is sent by a driver to a video callback function or window when the specified buffer is returned to the application.

Parameters

DWORD *dwParam1*
Specifies a far pointer to a **VIDEOHDR** structure identifying the buffer.

DWORD *dwParam2*
Not used.

Comments

For VIDEO_IN channels, buffers are returned when they have been filled. For VIDEO_OUT channels, buffers are returned after they are displayed. All buffers are returned for the DVM_STREAM_RESET message.

This message is used by video capture drivers, installable compressors, and other types of installable drivers to signal an application that new data is available.

MM_DRVM_ERROR

This message is sent by a device driver to a video callback function or window when an error has occurred.

Parameters

DWORD *dwParam1*
Specifies the error ID.

DWORD *dwParam2*
Not used.

Comments

Although a device driver can send this message for any reason, it most often indicates that no more buffers are available for video streaming.

The **MM_DRVM_ERROR** message is used by video capture drivers, installable compressors, and other types of installable drivers to signal an application that an error occurred.

MM_DRVM_OPEN

This message is sent by a driver to a video callback function or window when a DVM_STREAM_INIT message is received.

Parameters

DWORD *dwParam1*
Not used.

DWORD *dwParam2*
Not used.

Comments

This message is used by video capture drivers, installable video compressors, and other types of installable drivers whenever a device is opened.

Video Capture Device Driver Data Structure Reference

This section lists data structures used by video capture device drivers for Windows. The data structures are presented in alphabetical order. The structure definition is given, followed by a description of each field.

CHANNEL_CAPS

The **CHANNEL_CAPS** structure is used with the DVM_GET_CHANNEL_CAPS message to return the capabilities of a channel to an application.

```
typedef struct channel_caps_tag {
    DWORD  dwFlags;
    DWORD  dwSrcRectXMod;
    DWORD  dwSrcRectYMod;
    DWORD  dwSrcRectWidthMod;
    DWORD  dwSrcRectHeightMod;
    DWORD  dwDstRectXMod;
    DWORD  dwDstRectYMod;
    DWORD  dwDstRectWidthMod;
    DWORD  dwDstRectHeightMod;
} CHANNEL_CAPS;
```

Fields

The **CHANNEL_CAPS** structure has the following fields:

dwFlags
Returns flags giving information about the channel. The following flags are defined:

VCAPS_OVERLAY
Indicates the channel is capable of overlay. This flag is used only for VIDEO_EXTERNALOUT channels.

VCAPS_SRC_CAN_CLIP
Indicates that the source rectangle can be set smaller than the maximum dimensions.

VCAPS_DST_CAN_CLIP
Indicates that the destination rectangle can be set smaller than the maximum dimensions.

VCAPS_CAN_SCALE
Indicates that 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

Some channels can only use source and destination rectangles which fall on 2, 4, or 8 pixel boundaries. Similarly, some channels only accept capture rectangles widths and heights that are multiples of a fixed value. Rectangle dimensions indicated by modulus operators are considered advisory. When requesting a particular rectangle, the application must always check the return value to insure the request was accepted by the driver. For example, if **dwDstRectWidthMod** is set to 64, the application might try to set destination rectangles with widths of 64, 128, 192, 256, ..., and 640 pixels. The driver might actually support a subset of these sizes and indicates the supported sizes with the return value of the **DVM_DST_RECT** message. If a channel supports arbitrarily positioned rectangles, with arbitrary sizes, the values above should all be set to 1.

VIDEO_GETERRORTEXT_PARMS

The **VIDEO_GETERRORTEXT_PARMS** structure specifies a return buffer for the error text.

```
typedef struct tag_video_geterrortext_parms {
    DWORD  dwError;
    LPSTR  lpText;
    DWORD  dwLength;
} VIDEO_GETERRORTEXT_PARMS;
```

Fields

The **VIDEO_GETERRORTEXT_PARMS** structure has the following fields:

dwError
Specifies the error number.

lpText
Specifies a far pointer to the error return buffer.

dwLength
Specifies the length of the error return buffer.

VIDEO_OPEN_PARMS

The **VIDEO_OPEN_PARMS** structure defines the type of channel to open on a video capture device.

```
typedef struct {
    DWORD   dwSize;
    FOURCC  fccType;
    FOURCC  fccComp;
    DWORD   dwVersion;
    DWORD   dwFlags;
    DWORD   dwError;
} VIDEO_OPEN_PARMS;
```

Fields

The **VIDEO_OPEN_PARMS** structure has the following fields:

dwSize
Specifies the size of the **VIDEO_OPEN_PARMS** structure.

fccType
Specifies a four-character code identifying the type of channel being opened. For capture devices, this is set to "vcap".

fccComp
Unused.

dwVersion
Specifies the current version number of the video capture command set in MSVIDEO.DLL.

dwFlags
Specifies flags used to indicate the type of channel. The following flags are defined:

VIDEO_EXTERNALIN
Specifies a channel that loads data from an external source into a frame buffer. This can also be called the capture channel.

VIDEO_IN
Specifies a channel that transfers data from the frame buffer to system memory.

VIDEO_OUT
Specifies a channel that transfers data from system memory to the frame buffer.

VIDEO_EXTERNALOUT
Specifies a channel that controls display of frame buffer images. Display might be either on a second monitor, or via overlay.

dwError
Specifies an error value the driver should return to the client application if it fails the open.

Comments

This structure is identical to the **IC_OPEN** structure used by installable compressors. This lets a driver handle both video capture and decompressor messages with a single **DriverProc** entry point.

VIDEO_STREAM_INIT_PARMS

The **VIDEO_STREAM_INIT_PARMS** structure contains the fields used to initialize a video stream for video capture.

```
typedef struct tag_video_stream_init_parms {
    DWORD  dwMicroSecPerFrame;
    DWORD  dwCallback;
    DWORD  dwCallbackInst;
    DWORD  dwFlags;
    DWORD  hVideo;
} VIDEO_STREAM_INIT_PARMS;
```

Fields

The **VIDEO_STREAM_INIT_PARMS** structure has the following fields:

dwMicroSecPerFrame
Specifies the number of microseconds between the start of one frame capture and the start of the next.

dwCallback
An optional parameter which specifies an address to a callback function or a handle to a window called during video recording. The callback function or window processes messages related to the progress of recording.

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

dwFlags
Specifies flags for the data capture. The following flags are defined:

VIDEO_ALLOWSYNC
If this flag is not specified, the device will fail to open if it is a synchronous device.

CALLBACK_WINDOW
If this flag is specified, *dwCallback* contains a window handle.

CALLBACK_FUNCTION
If this flag is specified, *dwCallback* contains a callback function address.

hVideo
Specifies a handle to the video channel.

VIDEOCONFIGPARMS

The **VIDEOCONFIGPARMS** structure is used to send or return message specific configuration parameters.

```
typedef struct {
    LPDWORD  lpdwReturn;
    LPVOID   lpData1;
    DWORD    dwSize1;
    LPVOID   lpData2;
    DWORD    dwSize2;
} VIDEOCONFIGPARMS;
```

Fields

The **VIDEOCONFIGPARMS** structure has the following fields:

lpdwReturn
Specifies a far pointer to a **DWORD** to be filled with a message specific return value.

lpData1
Specifies a far pointer to message-specific data.

dwSize1
Specifies the size in bytes of data passed in **lpData1**.

lpData2
Specifies a far pointer to message specific data.

dwSize2
Specifies the size in bytes of data passed in **lpData2**.

See Also

DVM_FORMAT, DVM_PALETTE, DVM_PALETTERGB555

VIDEOHDR

The **VIDEOHDR** structure defines the header used to identify a video data buffer.

```
typedef struct videohdr_tag {
    LPSTR  lpData;
    DWORD  dwBufferLength;
    DWORD  dwBytesUsed;
    DWORD  dwTimeCaptured;
    DWORD  dwUser;
    DWORD  dwFlags;
    DWORD  dwReserved[4];
} VIDEOHDR;
```

Fields

The **VIDEOHDR** structure has the following fields:

lpData
Specifies a far pointer to the video data buffer.

dwBufferLength
Specifies the length of the data buffer.

dwBytesUsed
Specifies the number of bytes used in the data buffer.

dwTimeCaptured
Specifies the time (in milliseconds) when the frame was captured relative to the first frame in the stream.

dwUser
Specifies 32 bits of user data.

dwFlags
Specifies flags giving information about the data buffer. The following flags are defined for this field:

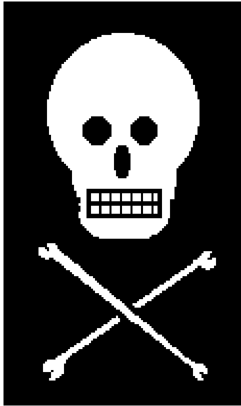
VHDR_DONE
Set by the device driver to indicate it is finished with the data buffer and it is returning the buffer to the application.

VHDR_PREPARED
Set by Windows to indicate the data buffer has been prepared with **videoStreamPrepareHeader**.

VHDR_INQUEUE
Set by Windows to indicate the data buffer is queued for playback.

VHDR_KEYFRAME
Set by the device driver to indicate a key frame.

dwReserved[4]
Reserved for use by the device driver. Typically, these maintain a linked list of buffers in the queue.



NOTE: MCIWnd is an unsupported API for the Video for Windows 1.0 Developer's Kit and is included as a sample only. Your application is free to use mciwnd.lib and the APIs for mciwnd but you do so at your own risk and PSS will not answer questions about this component if you ask.

Using MCIWND to Make Developing MCI Applications Easier

Overview

Using the MCI interface to allow your application to play MCI device files can sometimes be confusing and complicated, especially with all of the different commands that a device might support. Also, there is no user interface built in to MCI to let the user control the playback of files - an application must provide its own scrollbar and buttons for the user to play, pause, rewind, or seek through a file, or not provide this service at all.

MCIWND is a library that your application can link to that will create a new class of window. Your application needs only to create a window of this class, and then send it a message to open an MCI file. It can then send other messages to control the playback of the file, or give the user this control with the built in toolbar, scrollbar and menus.

MCIPLAY.C is sample code of an Multiple Document Interface (MDI) application that uses this window class to allow the playback and control of multiple MCI devices/files.

Services of the MCIWND window class

- A Toolbar with a PLAY, PAUSE and STOP button
- A Trackbar (scrollbar) to allow seeking within a file
- A Pop-Up Track Menu with some common commands when the right mouse button is clicked over the window
- Simple single-command macros for many of the common MCI commands which eliminates the need for longer, multi-line mciSendCommand or mciSendString calls.

Using MCIWND

- Include the file **VFW.H** in your application's source files to give you access to function prototypes and Macros and defines you will need.

- Link your application with **MCIWND.LIB** to get the new functionality.

- Call **MCIWndRegisterClass(HINSTANCE hInst)** to register the new Window Class **"MCIWND_WINDOW_CLASS"**. This function returns TRUE if successful.

- Use the standard windows function **CreateWindow()** to create a window to play an MCI file inside of.

  or...

  Use the **MCIWndCreate(HWND hwndParent, HINSTANCE hInst, DWORD dwStyle, LPSTR szFile)** function. This takes the place of CreateWindow() and has the advantage of being able to open an MCI device in the same call as the window is created in. NOTE: You do not need to call MCIWndRegisterClass() if you use this function, only if you use CreateWindow().

  For non-windowed devices, you will need a window to hold the toolbar and trackbar. If there will be no controls, and you are playing a non-windowed device (EG playing a Sound file) you may want to leave the window invisible.

  For both of these calls, you have some new styles you can choose from as well as the standard window styles. They are as follows:

  - **MCIWND_NOAUTOSIZE** Let the app size the window. The default is to automatically size the created window to a default size big enough for the window and the playbar (if used).

  - **MCIWND_NOPLAYBAR** Do not put a playbar (Toolbar and Trackbar) in the window. By default, a playbar appears. It provides a PLAY, PAUSE and a STOP button, as well as a Trackbar to seek through the file.

  - **MCIWND_NORESIZETOWINDOW** For MCI devices that can window (have video to display), MCIWND will ordinarily resize the image to any size you make the window. This flag inhibits that action. (The image is a constant size regardless of the size of the window containing it).

  - **MCIWND_NOTRACKMENU** Do not provide a pop-up menu. Normally, pressing the right mouse button over the window will bring up a track menu with commands for Play, Pause, Stop, Rewind, Volume, Speed, and for windowed devices, a Window (zoom) command.

- **MCIWND_NOTIFYSTATE** Whenever the state of the device changes (eg. from Stop to Play) the parent window will receive a MCIWNDM_NOTIFYSTATE msg with an lParam of the new state of the device (eg. MCI_MODE_STOP).

- **MCIWND_NOTIFYPOS** Whenever the position of the device changes (eg. as it's playing) the parent window will receive a MCIWNDM_NOTIFYPOS msg with an lParam of the new position in the media.

- **MCIWND_SHOWNAME** Sets the window text of the window to the filename of any MCI file you open in this window.

- Open an MCI file or device using the macro **MCIWndOpen(HWND hwnd, LPSTR sz, UINT f)**. Hwnd is the window you have created, and sz is the name of the file or device to open. F is currently unused and should be set to 0.

- Send the window a command using one of the following macros. Unless otherwise specified, the return code is the same as you would get from mciSendString() using the same command.

- **MCIWndClose(hwnd)** Close the MCI file. You can then re-open another file in the same window, or just call open again and the currrent file will close automatically.

- **MCIWndPlay(hwnd)** Play the file from the current position.

- **MCIWndStop(hwnd)** Stops the device.

- **MCIWndPause(hwnd)** Pauses the device.

- **MCIWndResume(hwnd)** Resumes playing (after a pause).

- **MCIWndSeek(hwnd, lPos)** Seeks to a specified position in the file. If lpos == MCIWND_SEEKSTART it will seek to the beginning. If lpos == MCIWND_SEEKEND it will seek to the end.

- **MCIWndPlayReverse(hwnd)** Play the file backwards starting at the current position.

- **MCIWndGetMode(hwnd)** Returns the current mode of the device (eg. MCI_MODE_PLAY.

- **MCIWndGetDeviceID(hwnd)** Gets the deviceID of the open file which you will need if you wish to call mciSendCommand or mciSendString to do any commands that are not supported by this interface.

- **MCIWndGetStart(hwnd)** Returns the starting position of the file. Seeking here will place the file at the beginning

- **MCIWndGetLength(hwnd)** Returns the length of the file. The start plus the length will give you the end of the media.

- **MCIWndGetEnd(hwnd)** Returns the end position of the file.

- **MCIWndStep(hwnd, n)** Steps n frames or milliseconds, depending on the time format of the device. A positive value is a step forward. A negative value is a step backwards.

- **MCIWndDestroy(hwnd)** Destroys the window. No return code.

- **MCIWndSetZoom(hwnd, n)** For windowed devices, sets the size of the window to n percent of the original size of the window.

- **MCIWndSetVolume(hwnd, n)** Sets the volume of audio playback (if supported) to n. 1000 is normal volume. Higher numbers are louder. Lower numbers are quieter.

- **MCIWndGetVolume(hwnd)** Gets the current volume.

- **MCIWndSetSpeed(hwnd, n)** Sets the playback speed of the device (if supported) to n. 1000 is normal speed. Higher numbers are faster. Lower numbers are slower.

- **MCIWndGetSpeed(hwnd)** Gets the playback speed of the device.

- **MCIWndRealize(hwnd, f)** Tells MCI to realize the palette of the image it is displaying in the window. f is TRUE if the window is in a background application. You should call this function in your app's WM_PALETTECHANGED and WM_QUERYNEWPALETTE code, instead of using the standard windows function RealizePalette. This MCIWnd function will use the palette of the MCI device and call RealizePalette for you. On the other hand, you could just pass the WM_PALETTECHANGED or WM_QUERYNEWPALETTE msg on to the MCI window and this will happen automatically.

- **MCIWndSendString(hwnd, sz)** Takes the place of mciSendString(sz, NULL, 0, NULL). Simply give the string to send to the window. Leave out the alias after the first word. (eg. sz = "set time format frames" is a valid command).

- **MCIWndUseTime(hwnd)**
- **MCIWndUseFrames(hwnd)**Sets the time format of the device to either milliseconds or frames mode. This determines how to interpret a position in the

file. By default, when you open a file in an MCI Window, the device will be set to frames mode. If that fails, it will try millisecond mode.

- **MCIWndValidateMedia(hwnd)** If you ever do anything to a device that changes the time format of the media (like changing time formats in some other way than by using an MCIWnd macro) the starting and ending position of the media, as well as the trackbar will still be using the old values, and need to be updated. Send this message to update these values. Normally, you should not need to use this macro.