DirectDraw

[This is preliminary documentation and subject to change.]

This section provides information about the DirectDraw® component of the DirectX® application programming interface (API). Information is divided into the following groups:

- About DirectDraw
- Why Use DirectDraw?
- Getting Started: Basic Graphics Concepts
- DirectDraw Architecture
- DirectDraw Essentials
- DirectDraw Tutorials
- DirectDraw Reference
- DirectDraw Samples

About DirectDraw

[This is preliminary documentation and subject to change.]

DirectDraw® is the component of the DirectX® application programming interface (API) that allows you to directly manipulate display memory, the hardware blitter, hardware overlay support, and flipping surface support. DirectDraw provides this functionality while maintaining compatibility with existing Microsoft® Windows®-based applications and device drivers.

DirectDraw is a software interface that provides direct access to display devices while maintaining compatibility with the Windows graphics device interface (GDI). It is not a high-level application programming interface (API) for graphics. DirectDraw provides a device-independent way for games and Windows subsystem software, such as three-dimensional (3-D) graphics packages and digital video codecs, to gain access to the features of specific display devices.

DirectDraw works with a wide variety of display hardware, ranging from simple SVGA monitors to advanced hardware implementations that provide clipping, stretching, and non-RGB color format support. The interface is designed so that your applications can enumerate the capabilities of the underlying hardware and then use any supported hardware-accelerated features. Features that are not implemented in hardware are emulated by DirectX.

DirectDraw provides device-dependent access to display memory in a device-independent way. Essentially, DirectDraw manages display memory. Your application need only recognize some basic device dependencies that are standard

across hardware implementations, such as RGB and YUV color formats and the pitch between raster lines. You need not call specific procedures to use the blitter or manipulate palette registers. Using DirectDraw, you can manipulate display memory with ease, taking full advantage of the blitting and color decompression capabilities of different types of display hardware without becoming dependent on a particular piece of hardware.

DirectDraw provides world-class game graphics on computers running Windows 95 and later and Windows NT® version 4.0 or Windows 2000.

Why Use DirectDraw?

[This is preliminary documentation and subject to change.]

The DirectDraw component brings many powerful features to you, the Windows graphics programmer:

- The hardware abstraction layer (HAL) of DirectDraw provides a consistent interface through which to work directly with the display hardware, getting maximum performance.
- DirectDraw assesses the video hardware's capabilities, making use of special hardware features whenever possible. For example, if your video card supports hardware blitting, DirectDraw delegates blits to the video card, greatly increasing performance. Additionally, DirectDraw provides a hardware emulation layer (HEL) to support features when the hardware does not.
- DirectDraw exists under Windows, gaining the advantage of 32-bit memory addressing and a flat memory model that the operating system provides. DirectDraw presents video and system memory as large blocks of storage, not as small segments. If you've ever used segment:offset addressing, you will quickly begin to appreciate this "flat" memory model.
- DirectDraw makes it easy for you to implement page flipping with multiple back buffers in full-screen applications. For more information, see Page Flipping and Back Buffering.
- Support for clipping in windowed or full-screen applications.
- Support for 3-D z-buffers.
- Support for hardware-assisted overlays with z-ordering.
- Access to image-stretching hardware.
- Simultaneous access to standard and enhanced display-device memory areas.
- Other features include custom and dynamic palettes, exclusive hardware access, and resolution switching.

These features combine to make it possible for you to write applications that easily outperform standard Windows GDI-based applications and even MS-DOS applications.

Getting Started: Basic Graphics Concepts

[This is preliminary documentation and subject to change.]

This section provides an overview of graphics programming with DirectDraw. Each concept discussed here begins with a non-technical overview, followed by some specific information about how DirectDraw supports it.

You don't need to be a graphics guru to benefit from this overview—in fact, if you are one you might want to skip this section entirely and move on to the more detailed information in the DirectDraw Essentials section. If you're familiar with Windows programming in C and C++, you won't have difficulty digesting this information. When you finish reading these topics, you will have a solid understanding of basic DirectDraw graphics programming concepts.

The following topics are discussed:

- Device-Independent Bitmaps
- Drawing Surfaces
- Blitting
- Page Flipping and Back Buffering
- Introduction to Rectangles

Device-Independent Bitmaps

[This is preliminary documentation and subject to change.]

Windows, and therefore DirectX, uses the device-independent bitmap (DIB) as its native graphics file format. Essentially, a DIB is a file that contains information describing an image's dimensions, the number of colors it uses, values describing those colors, and data that describes each pixel. Additionally, a DIB contains some lesser-used parameters, like information about file compression, significant colors (if all are not used), and physical dimensions of the image (in case it will end up in print). DIB files usually have the .bmp file extension, although they might occasionally have a .dib extension.

[C++]
Because the DIB is so pervasive in Windows programming, the Platform SDK already contains many functions that you can use with DirectX. For example, the following application-defined function, taken from the Ddutil.cpp file that comes with the DirectX APIs in the Platform SDK, combines Win32® and DirectX functions to load a DIB onto a DirectX surface.

```
extern "C" IDirectDrawSurface * DDLoadBitmap(IDirectDraw *pdd,
    LPCSTR szBitmap, int dx, int dy)
{
```

```
HBITMAP            hbm;
BITMAP             bm;
DDSURFACEDESC      ddsd;
IDirectDrawSurface *pdds;

//
//  This is the Win32 part.
//  Try to load the bitmap as a resource.
//  If that fails, try it as a file.
//
hbm = (HBITMAP)LoadImage(
       GetModuleHandle(NULL), szBitmap,
       IMAGE_BITMAP, dx, dy, LR_CREATEDIBSECTION);

if (hbm == NULL)
   hbm = (HBITMAP)LoadImage(
          NULL, szBitmap, IMAGE_BITMAP, dx, dy,
          LR_LOADFROMFILE|LR_CREATEDIBSECTION);

if (hbm == NULL)
   return NULL;

//
// Get the size of the bitmap.
//
GetObject(hbm, sizeof(bm), &bm);

//
// Now, return to DirectX function calls.
// Create a DirectDrawSurface for this bitmap.
//
ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT |DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwWidth = bm.bmWidth;
ddsd.dwHeight = bm.bmHeight;

if (pdd->CreateSurface(&ddsd, &pdds, NULL) != DD_OK)
   return NULL;

DDCopyBitmap(pdds, hbm, 0, 0, 0, 0);

DeleteObject(hbm);

return pdds;
```

```
}
```

For more detailed information about DIB files, see the Platform SDK.

[C++,Visual Basic]

Drawing Surfaces

[This is preliminary documentation and subject to change.]

Drawing surfaces receive video data to eventually be displayed on the screen as images (bitmaps, to be exact). In most Windows programs, you get access to the drawing surface using a Win32 function such as **GetDC**, which stands for get the device context (DC). After you have the device context, you can start painting the screen. However, Win32 graphics functions are provided by an entirely different part of the system, the graphics device interface (GDI). The GDI is a system component that provides an abstraction layer that enables standard Windows applications to draw to the screen.

The drawback of GDI is that it wasn't designed for high-performance multimedia software, it was made to be used by business applications like word processors and spreadsheet applications. GDI provides access to a video buffer in system memory, not video memory, and doesn't take advantage of special features that some video cards provide. In short, GDI is great for most business applications, but its performance is too slow for multimedia or game software.

On the other hand, DirectDraw can give you drawing surfaces that represent actual video memory. This means that when you use DirectDraw, you can write directly to the memory on the video card, making your graphics routines extremely fast. These surfaces are represented as contiguous blocks of memory, making it easy to perform addressing within them.

For more detailed information, see Surfaces.

Blitting

[This is preliminary documentation and subject to change.]

The term *blit* is shorthand for "bit block transfer," which is the process of transferring blocks of data from one place in memory to another. Graphics programmers use blitting to transfer graphics from one place in memory to another. Blits are often used to perform sprite animation, which is discussed later.

For more information on blitting in DirectDraw, see Blitting to Surfaces.

Page Flipping and Back Buffering

[This is preliminary documentation and subject to change.]

Page flipping is key in multimedia, animation, and game software. Software page flipping is analogous to the way animation can be done with a pad of paper. On each page the artist changes the figure slightly, so that when you flip between sheets rapidly the drawing appears animated.

Page flipping in software is very similar to this process. Initially, you set up a series of DirectDraw surfaces that are designed to "flip" to the screen the way artist's paper flips to the next page. The first surface is referred to as the primary surface, and the surfaces behind it are called back buffers . Your application writes to a back buffer, then flips the primary surface so that the back buffer appears on screen. While the system is displaying the image, your software is again writing to a back buffer. The process continues as long as you're animating, allowing you to animate images quickly and efficiently.

DirectDraw makes it easy for you to set up page flipping schemes, from a relatively simple double-buffered scheme (a primary surface with one back buffer) to more sophisticated schemes that add additional back buffers. For more information see DirectDraw Tutorials and Flipping Surfaces.

Introduction to Rectangles

[This is preliminary documentation and subject to change.]

Throughout DirectDraw and Windows programming, objects on the screen are referred to in terms of bounding rectangles. The sides of a bounding rectangle are always parallel to the sides of the screen, so the rectangle can be described by two points, the top-left corner and bottom-right corner. Most applications use the **RECT** structure to carry information about a bounding rectangle to use when blitting to the screen or performing hit detection.

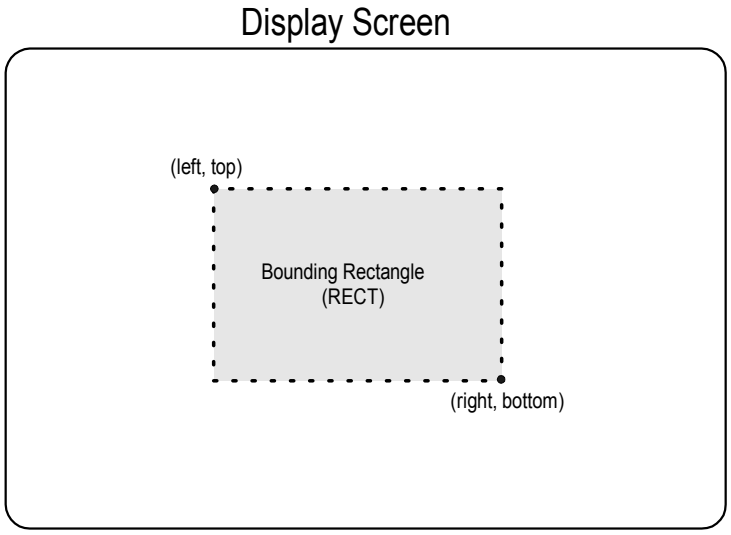
[C++]
In C++, the **RECT** structure has the following definition:

```
typedef struct tagRECT {
    LONG    left;    // This is the top-left corner's x-coordinate.
    LONG    top;     // The top-left corner's y-coordinate.
    LONG    right;   // The bottom-right corner's x-coordinate.
    LONG    bottom;  // The bottom-right corner's y-coordinate.
} RECT, *PRECT, NEAR *NPRECT, FAR *LPRECT;
```

[Visual Basic]
In Visual Basic, the **RECT** type has the following definition:

```
Type RECT
   Left As Long     // This is the top-left corner's x-coordinate.
   Top As Long      // The top-left corner's y-coordinate.
   Right As Long    // The bottom-right corner's x-coordinate.
   Bottom As Long   // The bottom-right corner's y-coordinate.
End Type
```

In the preceding example, the **left** and **top** members are the x- and y-coordinates of a bounding rectangle's top-left corner. Similarly, the **right** and **bottom** members make up the coordinates of the bottom-right corner. The following diagram illustrates how you can visualize these values.



In the interest of efficiency, consistency, and ease of use, all DirectDraw blitting functions work with rectangles. However, you can create the illusion of nonrectangular blit operations by using transparent blitting. For more information, see Transparent Blitting.

DirectDraw Architecture

[This is preliminary documentation and subject to change.]

This section contains general information about the relationship between the DirectDraw component and the rest of DirectX, the operating system, and the system hardware. The following topics are discussed:

- Architectural Overview for DirectDraw
- DirectDraw Object Types

- Hardware Abstraction Layer (HAL)
- Software Emulation
- System Integration

Architectural Overview for DirectDraw

[This is preliminary documentation and subject to change.]

Multimedia software requires high-performance graphics. Through DirectDraw, Microsoft enables a much higher level of efficiency and speed in graphics-intensive applications for Windows than is possible with GDI, while maintaining device independence. DirectDraw provides tools to perform such key tasks as:

- Manipulating multiple display surfaces
- Accessing the video memory directly
- Page flipping
- Back buffering
- Managing the palette
- Clipping

Additionally, DirectDraw enables you to query the display hardware's capabilities at run time, then provide the best performance possible given the host computer's hardware capabilities.

As with other DirectX components, DirectDraw uses the hardware to its greatest possible advantage, and provides software emulation for most features when hardware support is unavailable. Device independence is possible through use of the hardware abstraction layer, or HAL. For more information about the HAL, see the hardware abstraction layer .

The DirectDraw component provides services through COM-based interfaces. In the most recent iteration, these interfaces are **IDirectDraw4**, **IDirectDrawSurface4**, **IDirectDrawPalette**, **IDirectDrawClipper**, and **IDirectDrawVideoPort**. Note that, in addition to these interfaces, DirectDraw continues to support all previous versions. The DirectDraw component doesn't expose an **IDirectDraw3** interface, the interface versions skipped from **IDirectDraw2** to **IDirectDraw4**.

For more information about COM concepts that you should understand to create applications with the DirectX APIs in the Platform SDK, see DirectX and the Component Object Model.

The DirectDraw object represents the display adapter and exposes its methods through the **IDirectDraw**, **IDirectDraw2**, and **IDirectDraw4** interfaces. In most cases you will use the **DirectDrawCreate** function to create a DirectDraw object, but you can also create one with the **CoCreateInstance** COM function. For more information, see Creating DirectDraw Objects by Using CoCreateInstance.

After creating a DirectDraw object, you can create surfaces for it by calling the **IDirectDraw4::CreateSurface** method. Surfaces represent the memory on the display hardware, but can exist on either video memory or system memory. DirectDraw extends support for palettes, clipping (useful for windowed applications), and video ports through its other interfaces.

DirectDraw Object Types

[This is preliminary documentation and subject to change.]

You can think of DirectDraw as being composed of several objects that work together. This section briefly describes the objects you use when working with the DirectDraw component, organized by object type. For detailed information, see DirectDraw Essentials.

The DirectDraw component uses the following objects:

DirectDraw object
The DirectDraw object is the heart of all DirectDraw applications. It's the first object you create, and you use it to make all other related objects. You create a DirectDraw object by calling the **DirectDrawCreate** function. DirectDraw objects expose their functionality through the **IDirectDraw**, **IDirectDraw2**, and **IDirectDraw4** interfaces. For more information, see The DirectDraw Object.

DirectDrawSurface object
The DirectDrawSurface object (casually referred to as a "surface") represents an area in memory that holds data to be displayed on the monitor as images or moved to other surfaces. You usually create a surface by calling the **IDirectDraw4::CreateSurface** method of the DirectDraw object with which it will be associated. DirectDrawSurface objects expose their functionality through the **IDirectDrawSurface**, **IDirectDrawSurface2**, **IDirectDrawSurface3**, and **IDirectDrawSurface4** interfaces. For more information, see Surfaces.

DirectDrawPalette object
The DirectDrawPalette object (casually referred to as a "palette") represents a 16- or 256-color indexed palette to be used with a surface. It contains a series of indexed RGB triplets that describe colors associated with values within a surface. You do not use palettes with surfaces that use a pixel format depth greater than 8 bits. You can create a DirectDrawPalette object by calling the **IDirectDraw4::CreatePalette** method. DirectDrawPalette objects expose their functionality through the **IDirectDrawPalette** interface. For more information, see Palettes.

DirectDrawClipper object
The DirectDrawClipper object (casually referred to as a "clipper") helps you prevent blitting to certain portions of a surface or beyond the bounds of a surface. You can create a clipper by calling the **IDirectDraw4::CreateClipper** method. DirectDrawClipper objects expose their functionality through the **IDirectDrawClipper** interface. For more information, see Clippers.

DirectDrawVideoPort object

The DirectDrawVideoPort object represents video-port hardware present in some systems. This hardware allows direct access to the frame buffer without accessing the CPU or using the PCI bus. You can create a DirectDrawVideoPort object by calling a **QueryInterface** method for the DirectDraw object, specifying the IID_IDDVideoPortContainer reference identifier. DirectDrawVideoPort objects expose their functionality through the **IDDVideoPortContainer** and **IDirectDrawVideoPort** interfaces. For more information, see Video Ports.

Hardware Abstraction Layer (HAL)

[This is preliminary documentation and subject to change.]

DirectDraw provides device independence through the hardware abstraction layer (HAL). The HAL is a device-specific interface, provided by the device manufacturer, that DirectDraw uses to work directly with the display hardware. Applications never interact with the HAL. Rather, with the infrastructure that the HAL provides, DirectDraw exposes a consistent set of interfaces and methods that an application uses to display graphics. The device manufacturer implements the HAL in a combination of 16-bit and 32-bit code under Windows. Under Windows NT/Windows 2000, the HAL is always implemented in 32-bit code. The HAL can be part of the display driver or a separate DLL that communicates with the display driver through a private interface that driver's creator defines.

The DirectDraw HAL is implemented by the chip manufacturer, board producer, or original equipment manufacturer (OEM). The HAL implements only device-dependent code and performs no emulation. If a function is not performed by the hardware, the HAL does not report it as a hardware capability. Additionally, the HAL does not validate parameters; DirectDraw does this before the HAL is invoked.

Software Emulation

[This is preliminary documentation and subject to change.]

When the hardware does not support a feature through the hardware abstraction layer (HAL), DirectDraw attempts to emulate it. This emulated functionality is provided through the hardware emulation layer (HEL). The HEL presents its capabilities to DirectDraw just as the HAL would. And, as with the HAL, applications never work directly with the HEL. The result is transparent support for almost all major features, regardless of whether a given feature is supported by hardware or through the HEL.

Obviously, software emulation cannot equal the performance that hardware features provide. You can query for the features the hardware supports by using the **IDirectDraw4::GetCaps** method. By examining these capabilities during application initialization, you can adjust application parameters to provide optimum performance over varying levels of hardware performance.

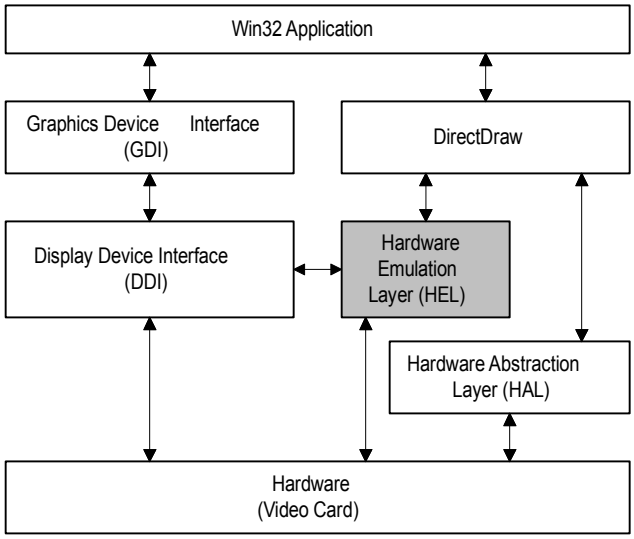
In some cases, certain combinations of hardware supported features and emulation can result in slower performance than emulation alone. For example, if the display device driver supports DirectDraw but not stretch blitting, noticeable performance losses will occur when stretch blitting from video memory surfaces. This happens because video memory is often slower than system memory, forcing the CPU to wait when accessing video memory surfaces. If your application uses a feature that isn't supported by the hardware, it is sometimes best to create surfaces in system memory, thereby avoiding performance losses created when the CPU accesses video memory.

For more information, see Hardware Abstraction Layer (HAL).

System Integration

[This is preliminary documentation and subject to change.]

The following diagram shows the relationships between DirectDraw, the graphics device interface (GDI), the hardware abstraction layer (HAL), the hardware emulation layer (HEL) and the hardware.



As the preceding diagram shows, a DirectDraw object exists alongside GDI, and both have direct access to the hardware through a device-dependent abstraction layer. Unlike GDI, DirectDraw makes use of special hardware features whenever possible. If the hardware does not support a feature, DirectDraw attempts to emulate it by using the HEL. DirectDraw can provide surface memory in the form of a device context, making it possible for you to use GDI functions to work with surface objects.

DirectDraw Essentials

[This is preliminary documentation and subject to change.]

This section contains general information about the DirectDraw® component of DirectX®. Information is organized into the following groups:

- Cooperative Levels
- Display Modes
- The DirectDraw Object
- Surfaces
- Palettes
- Clippers
- Multiple Monitor Systems
- Advanced DirectDraw Topics

Cooperative Levels

[This is preliminary documentation and subject to change.]

In the following topics, this section introduces the concept of cooperative levels and describes some common usage situations:

- About Cooperative Levels
- Testing Cooperative Levels

About Cooperative Levels

[This is preliminary documentation and subject to change.]

Cooperative levels describe how DirectDraw interacts with the display and how it reacts to events that might affect the display. Use the **IDirectDraw4::SetCooperativeLevel** method to set cooperative level of DirectDraw. For the most part, you use DirectDraw cooperative levels to determine whether your application runs as a full-screen program with exclusive access to the display or as a windowed application. However, DirectDraw cooperative levels can also have the following effects:

- Enable DirectDraw to use Mode X resolutions. For more information, see Mode X and Mode 13 Display Modes.
- Prevent DirectDraw from releasing exclusive control of the display or rebooting if the user presses CTRL + ALT + DEL (exclusive mode only).
- Enable DirectDraw to minimize or maximize the application in response to activation events.

The normal cooperative level indicates that your DirectDraw application will operate as a windowed application. At this cooperative level you won't be able to change the primary surface's palette or perform page flipping.

Because applications can use DirectDraw with multiple windows, **IDirectDraw4::SetCooperativeLevel** does not require a window handle to be specified if the application is requesting the DDSCL_NORMAL mode. By passing a NULL to the window handle, all of the windows can be used simultaneously in normal Windows mode.

At the full-screen and exclusive cooperative level, you can use the hardware to its fullest. In this mode, you can set custom and dynamic palettes, change display resolutions, and implement page flipping. The exclusive (full-screen) mode does not prevent other applications from allocating surfaces, nor does it exclude them from using DirectDraw or GDI. However, it does prevent applications other than the one currently with exclusive access from changing the display mode or palette.

DirectDraw takes control of window activation events for full-screen, exclusive mode applications, sending WM_ACTIVATEAPP messages to the window handle registered through the **SetCooperativeLevel** method as needed. DirectDraw only sends activation events to the top-level window. If your application creates child windows that require activation event messages, it is your responsibility to subclass the child windows.

SetCooperativeLevel maintains a binding between a process and a window handle. If **SetCooperativeLevel** is called once in a process, a binding is established between the process and the window. If it is called again in the same process with a different non-null window handle, it returns the DDERR_HWNDALREADYSET error value. Some applications may receive this error value when DirectSound® specifies a different window handle than DirectDraw—they should specify the same, top-level application window handle.

Note

Developers using Microsoft Foundation Classes (MFC) should keep in mind that the window handle given to the **SetCooperativeLevel** method should identify the application's top-level window, not a derived child window. To retrieve your MFC application's top level window handle, you could use the following code:

```
HWND hwndTop = AfxGetMainWnd()->GetSafeHwnd();
```

See also, Multiple Monitor Systems.

Testing Cooperative Levels

[This is preliminary documentation and subject to change.]

Developers often use messages such as WM_ACTIVATEAPP and WM_DISPLAYCHANGE as notifications that their applications should restore or re-create the surfaces being used. In some cases, applications take action when they don't need to, or don't take action when they should. The

IDirectDraw4::TestCooperativeLevel method makes it possible for your application to retrieve more information about the DirectDraw object's cooperative level and take appropriate steps to continue execution without mishap.

The **TestCooperativeLevel** method succeeds, returning DD_OK, if your application can restore its surfaces (if it has not already done so) and continue to execute. Failure codes, on the other hand, are interpreted differently depending on the cooperative-level your application uses:

Full-screen applications

Full-screen applications receive the DDERR_NOEXCLUSIVEMODE return value if they lose exclusive device access–for example, if the user pressed ALT+TAB to switch away from the current application. In this case, applications might call **TestCooperativeLevel** in a loop, exiting only when the method returns DD_OK (meaning that exclusive mode was returned). In the body of the loop, the application should relinquish control of the CPU to prevent using cycles unnecessarily. Windows supports functions such as the **WaitMessage** or **Sleep** Win32 functions for this purpose.

Any existing surfaces should be restored by calling the **IDirectDrawSurface4::Restore** or **IDirectDraw4::RestoreAllSurfaces** methods, and their contents reloaded before displaying them.

Windowed applications

Windowed applications (those that use the normal cooperative level) receive DDERR_EXCLUSIVEMODEALREADYSET if another application has taken exclusive device access. In this case, no action should be taken until the application with exclusive access loses it. This situation is similar to the case for a full-screen application; a windowed application might loop until **TestCooperativeLevel** returns DD_OK before restoring and reloading its surfaces. As mentioned previously, in a loop like this applications should avoid unnecessarily using CPU cycles by relinquishing CPU control periodically during the loop.

The **TestCooperativeLevel** method returns DDERR_WRONGMODE to windowed applications when the display mode has changed. In this case, the application should destroy and re-create any surfaces before continuing execution.

Display Modes

[This is preliminary documentation and subject to change.]

This section contains general information about DirectDraw display modes. The following topics are discussed:

- About Display Modes
- Determining Supported Display Modes
- Setting Display Modes
- Restoring Display Modes

- Mode X and Mode 13 Display Modes
- Support for High Resolutions and True-Color Bit Depths

About Display Modes

[This is preliminary documentation and subject to change.]

A display mode is a hardware setting that describes the dimensions and bit-depth of graphics that the display hardware sends to the monitor from the primary surface. Display modes are described by their defining characteristics: width, height, and bit-depth. For instance, most display adapters can display graphics 640 pixels wide and 480 pixels tall, where each pixel is 8 bits of data. In shorthand, this display mode is called 640×480×8. As the dimensions of a display mode get larger or as the bit-depth increases, more display memory is required.

There are two types of display modes: palettized and non-palettized. For palettized display modes, each pixel is a value representing an index into an associated palette. The bit depth of the display mode determines the number of colors that can be in the palette. For instance, in an 8-bit palettized display mode, each pixel is a value from 0 to 255. In such a display mode, the palette can contain 256 entries.

Non-palettized display modes, as their name states, do not use palettes. The bit depth of a non-palettized display mode indicates the total number of bits that are used to describe a pixel.

The primary surface and any surfaces in the primary flipping chain match the display mode's dimensions, bit depth and pixel format. For more information, see Pixel Formats.

Determining Supported Display Modes

[This is preliminary documentation and subject to change.]

Because display hardware varies, not all devices will support all display modes. To determine the display modes supported on a given system, call the **IDirectDraw4::EnumDisplayModes** method. By setting the appropriate values and flags, the **EnumDisplayModes** method can list all supported display modes or confirm that a single display mode that you specify is supported. The method's first parameter, *dwFlags*, controls extra options for the method; in most cases, you will set *dwFlags* to 0 to ignore extra options. The second parameter, *lpDDSurfaceDesc*, is the address of a **DDSURFACEDESC2** structure that describes a given display mode to be confirmed; you'll usually set this parameter to NULL to request that all modes be listed. The third parameter, *lpContext*, is a pointer that you want DirectDraw to pass to your callback function; if you don't need any extra data in the callback function, use NULL here. Last, you set the *lpEnumModesCallback* parameter to the address of the callback function that DirectDraw will call for each supported mode.

The callback function you supply when calling **EnumDisplayModes** must match the prototype for the **EnumModesCallback** function. For each display mode that the

hardware supports, DirectDraw calls your callback function passing two parameters. The first parameter is the address of a **DDSURFACEDESC2** structure that describes one supported display mode, and the second parameter is the address of the application-defined data you specified when calling **EnumDisplayModes**, if any.

Examine the values in the **DDSURFACEDESC2** structure to determine the display mode it describes. The key structure members are the **dwWidth**, **dwHeight**, and **ddpfPixelFormat** members. The **dwWidth** and **dwHeight** members describe the display mode's dimensions, and the **ddpfPixelFormat** member is a **DDPIXELFORMAT** structure that contains information about the mode's bit depth.

The **DDPIXELFORMAT** structure carries information describing the mode's bit depth and tells you whether or not the display mode uses a palette. If the **dwFlags** member contains the DDPF_PALETTEINDEXED1, DDPF_PALETTEINDEXED2, DDPF_PALETTEINDEXED4, or DDPF_PALETTEINDEXED8 flag, the display mode's bit depth is 1, 2, 4 or 8 bits, and each pixel is an index into an associated palette. If **dwFlags** contains DDPF_RGB, then the display mode is non-palettized and its bit depth is provided in the **dwRGBBitCount** member of the **DDPIXELFORMAT** structure.

Setting Display Modes

[This is preliminary documentation and subject to change.]

You can set the display mode by using the **IDirectDraw4::SetDisplayMode** method. The **SetDisplayMode** method accepts four parameters that describe the dimensions, bit depth, and refresh rate of the mode to be set. The method uses a fifth parameter to indicate special options for the given mode; this is currently only used to differentiate between Mode 13 and the Mode X 320×200×8 display mode.

Although you can specify the desired display mode's bit depth, you cannot specify the pixel format that the display hardware will use for that bit depth. To determine the RGB bit masks that the display hardware uses for the current bit depth, call **IDirectDraw4::GetDisplayMode** after setting the display mode. If the current display mode is not palettized, you can examine the mask values in the **dwRBitMask**, **dwGBitMask**, and **dwBBitMask** members to determine the correct red, green, and blue bits. For more information, see Pixel Format Masks.

Modes can be changed by more than one application as long as they are all sharing a display card. You can change the bit depth of the display mode only if your application has exclusive access to the DirectDraw object. All DirectDrawSurface objects lose surface memory and become inoperative when the mode is changed. A surface's memory must be reallocated by using the **IDirectDrawSurface4::Restore** method.

The DirectDraw exclusive (full-screen) mode does not bar other applications from allocating DirectDrawSurface objects, nor does it exclude them from using DirectDraw or GDI functionality. However, it does prevent applications other than the one that obtained exclusive access from changing the display mode or palette.

Note

You can only call the **IDirectDraw4::SetDisplayMode** method from the thread that created the application window. For single threaded applications (the vast majority), this restriction isn't an issue.

Restoring Display Modes

[This is preliminary documentation and subject to change.]

You can explicitly restore the display hardware to its original mode by calling the **IDirectDraw4::RestoreDisplayMode** method. If the display mode was set by calling **IDirectDraw4::SetDisplayMode** and your application takes the exclusive cooperative level, the original display mode is reset automatically when you set the application's cooperative level back to normal. (This behavior was first offered in the **IDirectDraw2** interface, and is offered by all newer versions of the interface.)

If you're using the **IDirectDraw** interface, you must always explicitly restore the display mode by using the **RestoreDisplayMode** method.

Mode X and Mode 13 Display Modes

[This is preliminary documentation and subject to change.]

DirectDraw supports both Mode 13 and Mode X display modes. Mode 13 is the linear unflippable 320×200×8 bits per pixel palettized mode known widely by its hexadecimal BIOS mode number: 13. For more information, see Mode 13 Support. Mode X is a hybrid display mode derived from the standard VGA Mode 13. This mode allows the use of up to 256 kilobytes (KB) of display memory (rather than the 64 KB allowed by Mode 13) by using the VGA display adapter's EGA multiple video plane system.

DirectDraw provides two Mode X modes (320×200×8 and 320×240×8) for all display cards. Some cards also support linear low-resolution modes. In linear low-resolution modes, the primary surface can be locked and directly accessed. This is not possible in Mode X modes.

Mode X modes are available only if an application uses the DDSCL_ALLOWMODEX, DDSCL_FULLSCREEN, and DDSCL_EXCLUSIVE flags when calling the **IDirectDraw4::SetCooperativeLevel** method. If DDSCL_ALLOWMODEX is not specified, the **IDirectDraw4::EnumDisplayModes** method will not enumerate Mode X modes, and the **IDirectDraw4::SetDisplayMode** method will fail if a Mode X mode is requested.

Windows 95 and Windows NT/Windows 2000 do not natively support Mode X modes; therefore, when your application is in a Mode X mode, you cannot use the **IDirectDrawSurface4::Lock** or **IDirectDrawSurface4::Blt** methods to lock or blit to the primary surface. You also cannot use either the **IDirectDrawSurface4::GetDC** method on the primary surface, or GDI with a

screen DC. Mode X modes are indicated by the DDSCAPS_MODEX flag in the **DDSCAPS2** structure, which is part of the **DDSURFACEDESC2** structure returned by the **IDirectDrawSurface4::GetCaps** and **IDirectDraw4::EnumDisplayModes** methods.

Support for High Resolutions and True-Color Bit Depths

[This is preliminary documentation and subject to change.]

DirectDraw supports all of the screen resolutions and depths supported by the display device driver. DirectDraw allows an application to change the mode to any one supported by the computer's display driver, including all supported 24- and 32-bpp (true-color) modes.

DirectDraw also supports HEL blitting in true-color surfaces. If the display device driver supports blitting at these resolutions, the hardware blitter will be used for display-memory-to-display-memory blits. Otherwise, the HEL will be used to perform the blits.

DirectDraw checks a list of known display modes against the display restrictions of the installed monitor. If DirectDraw determines that the requested mode is not compatible with the monitor, the call to the **IDirectDraw4::SetDisplayMode** method fails. Only modes that are supported on the installed monitor will be enumerated when you call the **IDirectDraw4::EnumDisplayModes** method.

The DirectDraw Object

[This is preliminary documentation and subject to change.]

This section contains information about DirectDraw objects and how you can manipulate them through their **IDirectDraw**, **IDirectDraw2**, or **IDirectDraw4** interfaces. The following topics are discussed:

- What Are DirectDraw Objects?
- What's New in IDirectDraw4?
- Parent and Child Object Lifetimes
- Multiple DirectDraw Objects per Process
- Creating DirectDraw Objects by Using CoCreateInstance

What Are DirectDraw Objects?

[This is preliminary documentation and subject to change.]

The DirectDraw object is the heart of all DirectDraw applications and is an integral part of Direct3D® applications as well. It is the first object you create and, through it, you create all other related objects. Typically, you create a DirectDraw object by

calling the **DirectDrawCreate** function, which returns an **IDirectDraw** interface. If you want to work with a different iteration of the interface (such as **IDirectDraw4**) to take advantage of new features it provides, you can query for it. (See Getting an IDirectDraw4 Interface.) Note that you can create multiple DirectDraw objects, one for each display device installed in a system.

The DirectDraw object represents the display device and makes use of hardware acceleration if the display device for which it was created supports hardware acceleration. Each unique DirectDraw object can manipulate the display device and create surfaces, palettes, and clipper objects that are dependent on (or are, "connected to") the object that created them. For example, to create surfaces, you call the **IDirectDraw4::CreateSurface** method. Or, if you need a palette object to apply to a surface, call the **IDirectDraw4::CreatePalette** method. Additionally, the **IDirectDraw4** interface exposes similar methods to create clipper objects.

You can create more than one instance of a DirectDraw object at a time. The simplest example of this is using two monitors on a Windows 95 or Windows NT 4.0 and earlier system. Although these operating systems don't support dual monitors on their own, it is possible to write a DirectDraw HAL for each display device. The display device Windows and GDI recognizes is the one that will be used when you create the instance of the default DirectDraw object. The display device that Windows and GDI do not recognize can be addressed by another, independent DirectDraw object that must be created by using the second display device's globally unique identifier (GUID). This GUID can be obtained by using the **DirectDrawEnumerate** function.

The DirectDraw object manages all of the objects it creates. It controls the default palette (if the primary surface is in 8-bits-per-pixel mode), the default color key, and the hardware display mode. It tracks what resources have been allocated and what resources remain to be allocated.

What's New in IDirectDraw4?

[This is preliminary documentation and subject to change.]

This section details new features provided by the **IDirectDraw4** interface and describes it's new features or how it behaves differently than its predecessor, **IDirectDraw2** (there is no **IDirectDraw3** interface). The following topics are discussed:

- New Features in IDirectDraw4
- Getting an IDirectDraw4 Interface

The most obvious difference between the **IDirectDraw4** interface and its predecessors is how it works with surfaces—how surfaces are described and which interfaces it automatically provides to access them. All of the surface-related methods in the new interface accept slightly different parameters than their counterparts in former interface versions. Wherever an **IDirectDraw2** interface method might accept a **DDSURFACEDESC** structure or retrieve an

IDirectDrawSurface3 interface, the methods of **IDirectDraw4** accept a **DDSURFACEDESC2** structure and retrieve an **IDirectDrawSurface4** interface instead.

Another behavioral change that **IDirectDraw4** introduces affects the lifetimes of child objects with respect to their parent DirectDraw object. For more information, see Parent and Child Object Lifetimes.

New Features in IDirectDraw4

[This is preliminary documentation and subject to change.]

The **IDirectDraw4** interface extends previous iterations by adding several methods that provide improved surface management and ease of use

The **IDirectDraw4** interface exposes the new **IDirectDraw4::RestoreAllSurfaces** method, which restores all of the surfaces created by a DirectDraw with a single call.

Additionally, you can now retrieve a surface's **IDirectDrawSurface4** interface from a Windows device context by using the **IDirectDraw4::GetSurfaceFromDC** method.

Getting an IDirectDraw4 Interface

[This is preliminary documentation and subject to change.]

The Component Object Model on which DirectX is built specifies that an object can provide new functionality through new interfaces, without affecting backward compatibility. To this end, the **IDirectDraw4** interface supersedes the **IDirectDraw2** interface. This new interface can be obtained by using the **IUnknown::QueryInterface** method, as the following C++ example shows:

```
// Create an IDirectDraw4 interface.
LPDIRECTDRAW  lpDD;
LPDIRECTDRAW4 lpDD4;

ddrval = DirectDrawCreate(NULL, &lpDD, NULL);
if(ddrval != DD_OK)
   return;

ddrval = lpDD->SetCooperativeLevel(hwnd,
   DDSCL_NORMAL);
if(ddrval != DD_OK)
   return;

ddrval = lpDD->QueryInterface(IID_IDirectDraw4,
   (LPVOID *)&lpDD4);
if(ddrval != DD_OK)
   return;
```

The preceding example creates a DirectDraw object, then calls the **IUnknown::QueryInterface** method of the **IDirectDraw** interface it received to create an **IDirectDraw4** interface.

After getting an **IDirectDraw4** interface, you can begin calling its methods to take advantage of new features, performance improvements, and behavioral differences. Because some methods might change with the release of a new interface, mixing methods from an interface and its replacement (between **IDirectDraw2** and **IDirectDraw4**, for example) can cause unpredictable results.

Parent and Child Object Lifetimes

[This is preliminary documentation and subject to change.]

All objects you'll use in DirectDraw programming—the DirectDraw object, surfaces, palettes, clippers, and such—only exist in memory for as long as another object, such as an application, needs them. The time that passes from the moment when an object is created and placed in memory to when it is released and subsequently removed from memory is known as the object's lifetime. The Component Object Model (COM) followed by all DirectX components dictates that an object must keep track of how many other objects require its services. This number, known as a reference count, determines the object's lifetime. COM also dictates that an object expose the **IUnknown::AddRef** and **IUnknown::Release** methods to enable applications to explicitly manage its reference count; make sure you use these methods in accordance to COM rules.

You aren't the only one who is using the **IUnknown** methods to manage reference counts for objects—DirectDraw objects use them internally, too. When you use the **IDirectDraw4** interface (in contrast to **IDirectDraw2** or **IDirectDraw**) to create a "child" object like a surface, the child uses the **IUnknown::AddRef** method of the "parent" DirectDraw object to increment the parent's reference count.

When your application no longer needs an object, call the **Release** method to decrement its reference count. When the count reaches zero, the object is removed from memory. When a child object's reference count reaches zero, it calls the parent's **IUnknown::Release** method to indicate that there is one less object who will be needing the parent's services.

Implicitly allocated objects, such as the back-buffer surfaces in a flipping chain that you create with a single **IDirectDraw4::CreateSurface** call, are automatically deallocated when their parent DirectDrawSurface object is released. Also, you can only release a DirectDraw object from the thread that created the application window. For single-threaded applications, this restriction obviously doesn't apply, as there is only one thread. If your application created a primary flipping chain of two surfaces (created by a single **CreateSurface** call) that used an attached DirectDrawClipper object, the code to release these objects safely might look like:

```
// For this example, the g_lpDDraw, g_lpDDSurface, and
// g_lpDDClip are valid pointers to objects.
void ReleaseDDrawObjects(void)
```

```
{
   // If the DirectDraw object pointer is valid,
   // it should be safe to release it and the objects it owns.
   if(g_lpDDraw)
   {
      // Release the DirectDraw object. (This call wouldn't
      // be safe if the children were created through IDirectDraw2
      // or IDirectDraw. See the following note for
      // more information)
      g_lpDDraw->Release(), g_lpDDraw = NULL;

      // Now, release the clipper that is attached to the surfaces.
      if(g_lpDDClip)
         g_lpDDClip->Release(), g_lpDDClip = NULL;

      // Now, release the primary flipping chain. Note
      // that this is only valid because the flipping
      // chain surfaces were created with a single
      // CreateSurface call. If they were explicitly
      // created and attached, then they must also be
      // explicitly released.
      if(g_lpDDSurface)
         g_lpDDSurface->Release(), g_lpDDSurface = NULL;
   }
}
```

Note

Earlier versions of the DirectDraw interface (**IDirectDraw2** and **IDirectDraw,** to be exact) behave differently than the most recent interface. When using these early interfaces, DirectDraw automatically releases all child objects when the parent itself is released. As a result, if you use these older interfaces, the order in which you release objects is critical. In this case, you should release the children of a DirectDraw object before releasing the DirectDraw object itself (or not release them at all, counting on the parent to do cleanup for you). Because the DirectDraw object releases the child objects, if you release the parent before the children, you are very likely to incur a memory fault for attempting to dereference a pointer that was invalidated when the parent object released its children.

Some older applications relied on the automatic release of child objects and neglected to properly release some objects when no longer needed. At the time, this practice didn't cause any negative side effects, however doing so when using the **IDirectDraw4** interface might result in memory leaks.

Multiple DirectDraw Objects per Process

[This is preliminary documentation and subject to change.]

DirectDraw allows a process to call the **DirectDrawCreate** function as many times as necessary. A unique and independent interface to a unique and independent DirectDraw object is returned after each call. Each DirectDraw object can be used as desired; there are no dependencies between the objects. Each object behaves exactly as if it had been created by a unique process.

DirectDraw objects are independent of one another and the DirectDrawSurface, DirectDrawPalette, and DirectDrawClipper objects they create should not be used with other DirectDraw objects because they are automatically released when the parent DirectDraw object is destroyed. If they are used with another DirectDraw object, they might stop functioning if their parent object is destroyed, causing the remaining DirectDraw object to malfunction.

The exception is DirectDrawClipper objects created by using the **DirectDrawCreateClipper** function. These objects are independent of any particular DirectDraw object and can be used with one or more DirectDraw objects.

Creating DirectDraw Objects by Using CoCreateInstance

[This is preliminary documentation and subject to change.]

You can create a DirectDraw object by using the **CoCreateInstance** function and the **IDirectDraw4::Initialize** method rather than the **DirectDrawCreate** function. The following steps describe how to create the DirectDraw object:

1 Initialize COM at the start of your application by calling **CoInitialize** and specifying NULL.

```
if (FAILED(CoInitialize(NULL)))
   return FALSE;
```

2 Create the DirectDraw object by using **CoCreateInstance** and the **IDirectDraw4::Initialize** method.

```
ddrval = CoCreateInstance(&CLSID_DirectDraw,
   NULL, CLSCTX_ALL, &IID_IDirectDraw4, &lpdd);
if(!FAILED(ddrval))
   ddrval = IDirectDraw4_Initialize(lpdd, NULL);
```

In this call to **CoCreateInstance**, the first parameter, *CLSID_DirectDraw*, is the class identifier of the DirectDraw driver object class, the *IID_IDirectDraw4* parameter identifies the particular DirectDraw interface to be created, and the *lpdd* parameter points to the DirectDraw object that is retrieved. If the call is successful, this function returns an uninitialized object.

3 Before you use the DirectDraw object, you must call **IDirectDraw4::Initialize**. This method takes the driver GUID parameter that the **DirectDrawCreate** function typically uses (NULL in this case). After the DirectDraw object is initialized, you can use and release it as if it had been created by using the **DirectDrawCreate** function. If you do not call the **Initialize** method before using one of the methods associated with the DirectDraw object, a DDERR_NOTINITIALIZED error will occur.

Before you close the application, close the COM library by using the **CoUninitialize** function.

```
CoUninitialize();
```

Surfaces

[This is preliminary documentation and subject to change.]

This section contains information about DirectDrawSurface objects. The following topics are discussed:

- Basic Concepts of Surfaces
- Creating Surfaces
- Flipping Surfaces
- Blitting to Surfaces
- Losing and Restoring Surfaces
- COM Reference Count Semantics for Surfaces
- Enumerating Surfaces
- Updating Surface Characteristics
- Accessing Surface Memory Directly
- Gamma and Color Controls
- Overlay Surfaces
- Compressed Texture Surfaces
- Private Surface Data
- Surface Uniqueness Values
- Using Non-local Video Memory Surfaces
- Converting Color and Format
- Surfaces and Device Contexts

Basic Concepts of Surfaces

[This is preliminary documentation and subject to change.]

This section contains information about the basic concepts associated with DirectDrawSurface objects. The following topics are discussed:

- What Are Surfaces?
- Surface Interfaces
- Width vs. Pitch
- Color Keying
- Pixel Formats

What Are Surfaces?

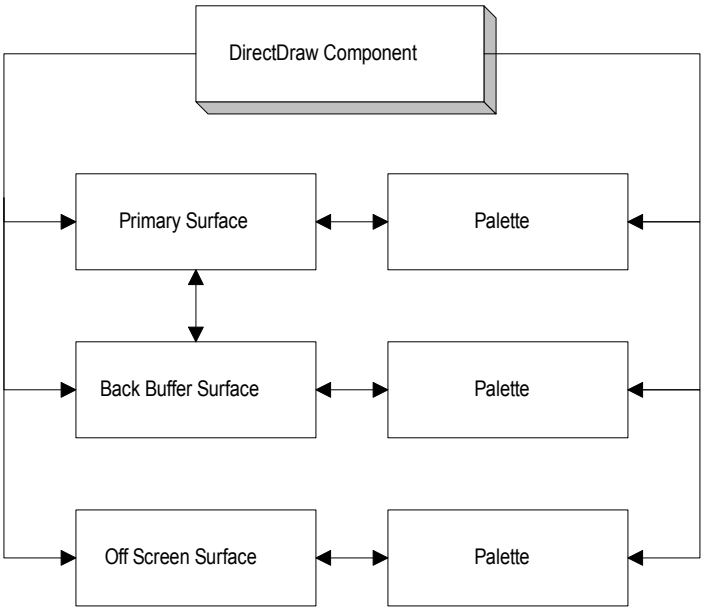
[This is preliminary documentation and subject to change.]

A surface, or DirectDrawSurface object, represents a linear area of display memory. A surface usually resides in the display memory of the display card, although surfaces can exist in system memory. Unless specifically instructed otherwise during the creation of the DirectDrawSurface object, DirectDraw object will put the DirectDrawSurface object wherever the best performance can be achieved given the requested capabilities. DirectDrawSurface objects can take advantage of specialized processors on display cards, not only to perform certain tasks faster, but to perform some tasks in parallel with the system CPU.

Using the **IDirectDraw4::CreateSurface** method, you can create a single surface object, complex surface-flipping chains, or three-dimensional surfaces. The **CreateSurface** method creates the requested surface or flipping chain and retrieves a pointer to the primary surface's **IDirectDrawSurface4** interface through which the object exposes its functionality.

The **IDirectDrawSurface4** interface enables you to indirectly access memory through blit methods, such as **IDirectDrawSurface4::BltFast**. The surface object can provide a device context to the display that you can use with GDI functions. Additionally, you can use **IDirectDrawSurface4** methods to directly access display memory. For example, you can use the **IDirectDrawSurface4::Lock** method to lock the display memory and retrieve the address corresponding to that surface. Addresses of display memory might point to visible frame buffer memory (primary surface) or to nonvisible buffers (off-screen or overlay surfaces). Nonvisible buffers usually reside in display memory, but can be created in system memory if required by hardware limitations or if DirectDraw is performing software emulation. In addition, the **IDirectDrawSurface4** interface extends other methods that you can use to set or retrieve palettes, or to work with specific types or surfaces, like flipping chains or overlays.

From this illustration, you can see that all surface are created by a DirectDraw object and are often used closely with palettes. Although each surface object can be assigned a palette, palettes aren't required for anything but primary surfaces that use pixel formats of 8-bits in depth or less.



Surface Interfaces

[This is preliminary documentation and subject to change.]

DirectDrawSurface objects expose their functionality through the **IDirectDrawSurface**, **IDirectDrawSurface2**, **IDirectDrawSurface3**, and **IDirectDrawSurface4** interfaces. Each new interface version provides the same utility as its predecessors, with additional options available through new methods.

When you create a surface by calling the **IDirectDraw4::CreateSurface** method (or another creation method from **IDirectDraw4**), you receive a pointer to the surface's **IDirectDrawSurface4** interface. This behavior is different than previous versions of DirectX. Before the introduction of the **IDirectDraw4** interface, the **CreateSurface** method provided a pointer to a surface's **IDirectDrawSurface** interface. If you wanted to work with a different iteration of the interface, you had to query for it. When using **IDirectDraw4** this isn't the case, although you are free to query a surface for a previous iteration of an interface if you choose.

Width vs. Pitch

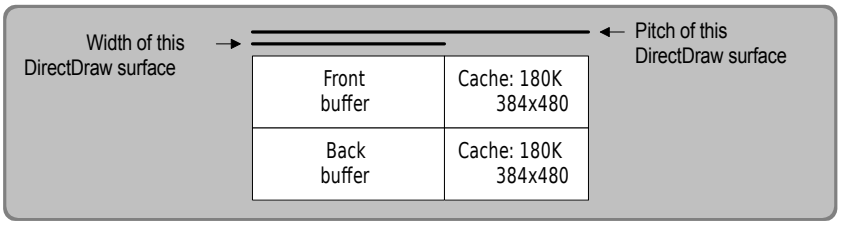
[This is preliminary documentation and subject to change.]

Although the terms *width* and *pitch* are discussed casually, they have very important (and distinctly different) meanings. As a result, you should understand the meanings for each, and how to interpret the values that DirectDraw uses to describe them.

DirectDraw uses the **DDSURFACEDESC2** structure to carry information describing a surface. Among other things, this structure is defined to contain information about a surface's dimensions, as well as how those dimensions are represented in memory.

The structure uses the **dwHeight** and **dwWidth** members to describe the logical dimensions of the surface. Both of these members are measured in pixels. Therefore, the **dwHeight** and **dwWidth** values for a 640×480 surface are the same whether it is an 8-bit palettized surface or a 24-bit RGB surface.

The **DDSURFACEDESC2** structure contains information about how a surface is represented in memory through the **lPitch** member. The value in the **lPitch** member describes the surface's memory pitch (also called *stride*). Pitch is the distance, in bytes, between two memory addresses that represent the beginning of one bitmap line and the beginning of the next bitmap line. Because pitch is measured in bytes rather than pixels, a 640×480×8 surface will have a very different pitch value than a surface with the same dimensions but a different pixel format. Additionally, the pitch value sometimes reflects bytes that DirectDraw has reserved as a cache, so it is not safe to assume that pitch is simply the width multiplied by the number of bytes per pixel. Rather, you could visualize the difference between width and pitch as shown in the following illustration.



In this figure, the front buffer and back buffer are both 640×480×8, and the cache is 384×480×8.

Pitch values are useful when you are directly accessing surface memory. For example, after calling the **IDirectDrawSurface4::Lock** method, the **lpSurface** member of the associated **DDSURFACEDESC2** structure contains the address of the top-left pixel of the locked area of the surface, and the **lPitch** member is the surface pitch. You access pixels horizontally by incrementing or decrementing the surface pointer by the number of bytes per pixel, and you move up or down by adding the pitch value to, or subtracting it from, the current surface pointer.

When accessing surfaces directly, take care to stay within the memory allocated for the dimensions of the surface and stay out of any memory reserved for cache. Additionally, when you lock only a portion of a surface, you must stay within the rectangle you specify when locking the surface. Failing to follow these guidelines will have unpredictable results. When rendering directly into surface memory, always use the pitch returned by the **Lock** method (or the **IDirectDrawSurface4::GetDC** method). Do not assume a pitch based solely on the display mode. If your application works on some display adapters but looks garbled on others, this may be the cause of your problem.

For more information, see Accessing Surface Memory Directly.

Color Keying

[This is preliminary documentation and subject to change.]

DirectDraw supports source and destination *color keying* for blits and overlay surfaces. Color keys enable you to display one image on top of another selectively, so that only certain pixels from the foreground rectangle are displayed, or only certain pixels on the background rectangle are overwritten.

You supply a single color key or a range of colors for source or destination color keying by calling the **IDirectDrawSurface4::SetColorKey** method.

For more information about color keying, see the following topics:

- Overlay Color Keys
- Transparent Blitting

Pixel Formats

[This is preliminary documentation and subject to change.]

Pixel formats dictate how data for each pixel in surface memory is to be interpreted. DirectDraw uses the **DDPIXELFORMAT** structure to describe various pixel formats. The **DDPIXELFORMAT** contains members to describe the following traits of a pixel format:

- Palettized or non-palettized pixel format
- If non-palettized, whether the pixel format is RGB or YUV
- Bit depth
- Bit masks for the pixel format's components

You can retrieve information about an existing surface's pixel format by calling the **IDirectDrawSurface4::GetPixelFormat** method.

Creating Surfaces

[This is preliminary documentation and subject to change.]

The DirectDrawSurface object represents a surface that usually resides in the display memory, but can exist in system memory if display memory is exhausted or if it is explicitly requested.

Use the **IDirectDraw4::CreateSurface** method to create one surface or to simultaneously create multiple surfaces (a complex surface). When calling **CreateSurface**, you specify the dimensions of the surface, whether it is a single surface or a complex surface, and the pixel format (if the surface won't be using an indexed palette). All these characteristics are contained in a **DDSURFACEDESC2** structure, whose address you send with the call. If the hardware can't support the requested capabilities or if it previously allocated those resources to another DirectDrawSurface object, the call will fail.

Creating single surfaces or multiple surfaces is a simple matter that requires only a few lines of code. There are a few common situations (and some less common ones) in which you will need to create surfaces. The following situations are discussed:

- Creating the Primary Surface
- Creating an Off-Screen Surface
- Creating Complex Surfaces and Flipping Chains
- Creating Wide Surfaces
- Creating Client Memory Surfaces

By default, for all surfaces except client memory surfaces, DirectDraw attempts to create a surface in local video memory. If there isn't enough local video memory available to hold the surface, DirectDraw will try to use non-local video memory (on some Accelerated Graphics Port-equipped systems), and fall back on system memory if all other types of memory are unavailable. You can explicitly request that a surface be created in a certain type of memory by including the appropriate flags in the associated **DDSCAPS2** structure when calling **IDirectDraw4::CreateSurface**.

Creating the Primary Surface

[This is preliminary documentation and subject to change.]

The primary surface is the surface currently visible on the monitor and is identified by the DDSCAPS_PRIMARYSURFACE flag. You can only have one primary surface for each DirectDraw object.

When you create a primary surface, remember that the dimensions and pixel format implicitly match the current display mode. Therefore, this is the one time you don't need to declare a surface's dimensions or pixel format. If you do specify them, the call will fail and return DDERR_INVALIDPARAMS—even if the information you used matches the current display mode.

The following example shows how to prepare the **DDSURFACEDESC2** structure members relevant for creating the primary surface.

```
DDSURFACEDESC2 ddsd;
ddsd.dwSize = sizeof(ddsd);

// Tell DirectDraw which members are valid.
ddsd.dwFlags = DDSD_CAPS;

// Request a primary surface.
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
```

After creating the primary surface, you can retrieve information about its dimensions and pixel format by calling its **IDirectDrawSurface4::GetSurfaceDesc** method.

See also, Display Modes.

Creating an Off-Screen Surface

[This is preliminary documentation and subject to change.]

An off-screen surface is often used to cache bitmaps that will later be blitted to the primary surface or a back buffer. You must declare the dimensions of an off-screen surface by including the DDSD_WIDTH and DDSD_HEIGHT flags and the corresponding values in the **dwWidth** and **dwHeight** members. Additionally, you must include the DDSCAPS_OFFSCREENPLAIN flag in the accompanying **DDSCAPS2** structure.

By default, DirectDraw creates a surface in display memory unless it will not fit, in which case it creates the surface in system memory. You can explicitly choose display or system memory by including the DDSCAPS_SYSTEMMEMORY or DDSCAPS_VIDEOMEMORY flags in the **dwCaps** member of the **DDSCAPS2** structure. The method fails, returning an error, if it can't create the surface in the specified location.

The following example shows how to prepare for creating a simple off-screen surface:

```
DDSURFACEDESC2 ddsd;
ddsd.dwSize = sizeof(ddsd);

// Tell DirectDraw which members are valid.
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;

// Request a simple off-screen surface, sized
// 100 by 100 pixels.
//
// (This assumes that the off-screen surface we are about
// to create will match the pixel format of the
// primary surface.)
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 100;
ddsd.dwWidth = 100;
```

Additionally, you can create surfaces whose pixel format differs from the primary surface's pixel format. However, in this case there is one drawback—you are limited to using system memory. The following code fragment shows how to prepare the **DDSURFACEDESC2** structure members in order to create an 8-bit palettized surface (assuming that the current display mode is something other than 8-bits per pixel).

```
ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize  = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH | DDSD_PIXELFORMAT;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN | DDSCAPS_SYSTEMMEMORY;
ddsd.dwHeight = 100;
```

```
ddsd.dwWidth  = 100;
ddsd.ddpfPixelFormat.dwSize  = sizeof(DDPIXELFORMAT);
ddsd.ddpfPixelFormat.dwFlags = DDPF_RGB | DDPF_PALETTEINDEXED8;

// Set the bit depth for an 8-bit surface, but DO NOT
// specify any RGB mask values. The masks must be zero
// for a palettized surface.
ddsd.ddpfPixelFormat.dwRGBBitCount = 8;
```

In previous versions of DirectX, the maximum width of off-screen surfaces was limited to the width of the primary surface. Beginning with DirectX 5.0, you can create surfaces as wide as you need, permitting that the display hardware can support them. Be careful when declaring wide off-screen surfaces; if the video card memory cannot hold a surface as wide as you request, the surface is created in system memory. If you explicitly choose video memory and the hardware can't support it, the call fails. For more information, see Creating Wide Surfaces.

Creating Complex Surfaces and Flipping Chains

[This is preliminary documentation and subject to change.]

You can also create *complex surfaces*. A complex surface is a set of surfaces created with a single call to the **IDirectDraw4::CreateSurface** method. If the DDSCAPS_COMPLEX flag is set when you call **CreateSurface** call, DirectDraw implicitly creates one or more surfaces in addition to the surface explicitly specified. You manage complex surfaces just like a single surface—a single call to the **IDirectDraw::Release** method releases all surfaces, and a single call to the **IDirectDrawSurface4::Restore** method restores them all. However, implicitly created surfaces cannot be detached. For more information, see **IDirectDrawSurface4::DeleteAttachedSurface**.

One of the most useful complex surfaces you can create is a flipping chain. Usually, a flipping chain is made of a primary surface and one or more back buffers. The DDSCAPS_FLIP flag indicates that a surface is part of a flipping chain. Creating a flipping chain this way requires that you also include the DDSCAPS_COMPLEX flag.

The following example shows how to prepare for creating a primary surface flipping chain.

```
DDSURFACEDESC2 ddsd2;
ddsd2.dwSize = sizeof(ddsd2);

// Tell DirectDraw which members are valid.
ddsd2.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;

// Request a primary surface with a single
// back buffer
ddsd2.ddsCaps.dwCaps = DDSCAPS_COMPLEX | DDSCAPS_FLIP |
```

```
DDSCAPS_PRIMARYSURFACE;
ddsd2.dwBackBufferCount = 1;
```

The previous example constructs a double-buffered flipping environment—a single call to the **IDirectDrawSurface4::Flip** method exchanges the surface memory of the primary surface and the back buffer. If you specify 2 for the value of the **dwBackBufferCount** member of the **DDSURFACEDESC2** structure, two back buffers are created, and each call to **Flip** rotates the surfaces in a circular pattern, providing a triple-buffered flipping environment. For more information, see Flipping Surfaces.

Note

To create a flipping chain that comprises surfaces that will be used as 3-D render targets, be sure to include the DDSCAPS_3DDEVICE capability flag in the surface description, as well as the DDSCAPS_COMPLEX and DDSCAPS_FLIP flags.
Unlike the **CreateSurface** method exposed by the **IDirectDraw3** and earlier interfaces, you cannot use **IDirectDraw4::CreateSurface** to implicitly create a flipping chain of render target surfaces with an attached depth-buffer. The **DDSURFACEDESC2** structure that the **IDirectDraw4::CreateSurface** method accepts doesn't contain a field to specify a depth-buffer bit depth. As a result, applications must create a depth-buffer surface explicitly, then attach it to the back-buffer render target surface. For more information, see Depth Buffers.

Creating Wide Surfaces

[This is preliminary documentation and subject to change.]

DirectDraw allows you to create off-screen surfaces in video memory that are wider that the primary surface. This is only possible when display device support for wide surfaces is present.

To check for wide surface support, call **IDirectDraw4::GetCaps** and look for the DDCAPS2_WIDESURFACES flag in the **dwCaps2** member of the first **DDCAPS** structure you send with the call. If the flag is present, you can create video memory off-screen surfaces that are wider that the primary surface.

If you attempt to create a wide surface in video memory when the DDCAPS2_WIDESURFACES flag isn't present, the attempt will fail and return DDERR_INVALIDPARAMS. Note that attempting to create extremely large surfaces might still fail, even if the driver exposes the DDCAPS2_WIDESURFACES flag.

Wide surfaces are always supported for system memory surfaces, video port surfaces, and execute buffers.

Creating Client Memory Surfaces

[This is preliminary documentation and subject to change.]

Client memory surfaces are simply DirectDrawSurface objects that use system memory that your application has previously allocated to hold image data. Creating such a surface isn't common, but it isn't difficult to do and it can be useful for applications that need to use DirectDraw surface features on existing memory buffers.

Like creating all surfaces, DirectDraw needs information about the dimensions of the surface (measured in pixels) and the surface pitch (measured in bytes), as well as the surface's pixel format. However, unlike creating other types of surfaces, this information doesn't tell DirectDraw how you want the surface to be created, it tells DirectDraw how you've already created it. You set these characteristics, plus the memory address of the buffer you've allocated, in the **DDSURFACEDESC2** structure you pass to the **IDirectDraw4::CreateSurface** method.

A client memory surfaces works just like a normal system-memory surface, with the exception that DirectDraw does not attempt to free the surface memory when it's no longer needed; freeing client allocated memory is the application's responsibility.

The following example shows how you might allocate memory and create a DirectDrawSurface object for a 64×64 pixel 24-bit RGB surface:

```
// For this example, g_lpDD4 is a valid IDirectDraw4 interface pointer.

#define WIDTH  64 // in pixels
#define HEIGHT 64
#define DEPTH  3  // in bytes (3bytes == 24 bits)

   HRESULT hr;
   LPVOID  lpSurface  = NULL;
   HLOCAL  hMemHandle = NULL;
   DDSURFACEDESC2 ddsd2;
   LPDIRECTDRAWSURFACE4 lpDDS4;

   // Allocate memory for a 64 by 64, 24-bit per pixel buffer.
   // REMEMBER: The application is responsible for freeing this
   //           buffer when it is no longer needed.
   if (lpSurface = malloc((size_t)WIDTH*HEIGHT*DEPTH))
      ZeroMemory(lpSurface, (DWORD)WIDTH*HEIGHT*DEPTH);
   else
      return DDERR_OUTOFMEMORY;

   // Initialize the surface description.
   ZeroMemory(&ddsd2, sizeof(DDSURFACEDESC2));
   ZeroMemory(&ddsd2.ddpfPixelFormat, sizeof(DDPIXELFORMAT));
   ddsd2.dwSize = sizeof(ddsd2);
   ddsd2.dwFlags = DDSD_WIDTH | DDSD_HEIGHT | DDSD_LPSURFACE |
                   DDSD_PITCH | DDSD_PIXELFORMAT;
   ddsd2.dwWidth = WIDTH;
```

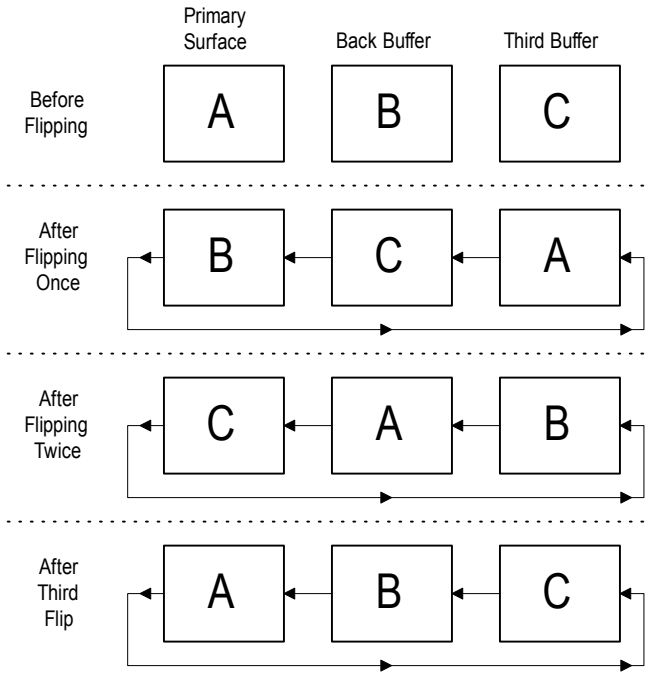
```
ddsd2.dwHeight= HEIGHT;
ddsd2.lPitch  = (LONG)DEPTH * WIDTH;
ddsd2.lpSurface = lpSurface;

// Set up the pixel format for 24-bit RGB (8-8-8).
ddsd2.ddpfPixelFormat.dwSize = sizeof(DDPIXELFORMAT);
ddsd2.ddpfPixelFormat.dwFlags= DDPF_RGB;
ddsd2.ddpfPixelFormat.dwRGBBitCount = (DWORD)DEPTH*8;
ddsd2.ddpfPixelFormat.dwRBitMask    = 0x00FF0000;
ddsd2.ddpfPixelFormat.dwGBitMask    = 0x0000FF00;
ddsd2.ddpfPixelFormat.dwBBitMask    = 0x000000FF;

// Create the surface
hr = g_lpDD4->CreateSurface(&ddsd2, &lpDDS4, NULL);
return hr;
```

Flipping Surfaces

[This is preliminary documentation and subject to change.]

Any surface in DirectDraw can be constructed as a *flipping surface*. A flipping surface is any piece of memory that can be swapped between a *front buffer* and a *back buffer*. (This construct is commonly referred to as a *flipping chain*). Often, the front buffer is the primary surface, but it doesn't have to be.

Typically, when you use the **IDirectDrawSurface4::Flip** method to request a surface flip operation, the pointers to surface memory for the primary surface and back buffers are swapped. Flipping is performed by switching pointers that the display device uses for referencing memory, not by copying surface memory. (The exception to this is when DirectDraw is emulating the flip, in which case it simply copies the surfaces. DirectDraw emulates flip operations if a back buffer cannot fit into display memory or if the hardware doesn't support DirectDraw.) When a flipping chain contains a primary surface and more than one back buffer, the pointers are switched in a circular pattern, as shown in the following illustration.



Other surfaces that are attached to a DirectDraw object, but not part of the flipping chain, are unaffected when the **Flip** method is called.

Remember, DirectDraw flips surfaces by swapping surface memory pointers within DirectDrawSurface objects, not by swapping the objects themselves. This means that, to blit to the back buffer in any type of flipping scheme, you always use the same DirectDrawSurface object — the one that was the back buffer when you created the flipping chain. Conversely, you always perform a flip operation by calling the front surface's **Flip** method.

When working with visible surfaces, such as a primary surface flipping chain or a visible overlay surface flipping chain, the **Flip** method is asynchronous unless you include the DDFLIP_WAIT flag. On these visible surfaces, the **Flip** method can return before the actual flip operation occurs in the hardware (because the hardware doesn't flip until the next vertical refresh occurs). While the actual flip operation is pending, the back buffer behind the currently visible surface can't be locked or blitted by calling the **IDirectDrawSurface4::Lock**, **IDirectDrawSurface4::Blt**, **IDirectDrawSurface4::BltFast**, or **IDirectDrawSurface4::GetDC** methods. If you attempt to call these methods while a flip operation is pending, they will fail and return DDERR_WASSTILLDRAWING. However, if you are using a triple buffered scheme, the rearmost buffer is still available.

Blitting to Surfaces

[This is preliminary documentation and subject to change.]

This section is a guide to copying pixels from one DirectDraw surface to another, or from one part of a surface to another.

The following topics are covered:

- Blitting Basics
- Blitting with BltFast
- Blitting with Blt
- Blit Timing
- Transparent Blitting
- Color Fills
- Blitting to Multiple Windows

Blitting Basics

[This is preliminary documentation and subject to change.]

Two methods are available for copying images to a DirectDraw surface: **IDirectDrawSurface4::Blt** and **IDirectDrawSurface4::BltFast**. (A third method, **IDirectDrawSurface4::BltBatch**, is not implemented in this version of DirectX.) These methods are called on the destination surface and receive the source surface as a parameter. The destination and source surfaces can be one and the same, and you don't have to worry about overlap—DirectDraw takes care to preserve all source pixels before overwriting them.

Of the two implemented methods, **Blt** is the more flexible and **BltFast** is the faster—but only if there is no hardware blitter. You can determine the blitting capabilities of the hardware from the **DDCAPS** structure obtained in the *lpDDDriverCaps* parameter of the **IDirectDraw4::GetCaps** method. If the **dwCaps** member contains DDCAPS_BLT, the hardware has at least minimal blitting capabilities.

Blitting with BltFast

[This is preliminary documentation and subject to change.]

When using **IDirectDrawSurface4::BltFast**, you supply a valid rectangle in the source surface from which the pixels are to be copied (or NULL to specify the entire surface), and an x-coordinate and y-coordinate in the destination surface. The source rectangle must be able to fit in the destination surface with its top left corner at that point, or the call will fail with a return value of DDERR_INVALIDRECT. **BltFast** cannot be used on surfaces that have an attached clipper.

No stretching, mirroring, or other effects can be performed when using **BltFast**.

BltFast Example

The following example copies pixels from an offscreen surface, *lpDDSOffOne*, to the primary surface, *lpDDSPrimary*. The flags ensure that the operation will take place as soon as the blitter is free, and that transparent pixels in the source image will not

be copied. (For more information on the meaning of these flags, see Blit Timing and Transparent Blitting .)

```
lpDDSPrimary->BltFast(
      100, 200,    // Upper left xy of destination
      lpDDSOffOne, // Source surface
      NULL,        // Source rectangle = entire surface
      DDBLTFAST_WAIT | DDBLTFAST_SRCCOLORKEY );
```

Blitting with Blt

[This is preliminary documentation and subject to change.]

When using the **IDirectDrawSurface4::Blt** method, you supply a valid rectangle in the source surface (or NULL to specify the entire surface), and a rectangle in the destination surface to which the source image will be copied (again, NULL means the rectangle covers the entire surface). If a clipper is attached to the destination surface, the bounds of the destination rectangle can fall outside the surface and clipping will be performed. If there is no clipper, the destination rectangle must fall entirely within the surface or else the method will fail with DDERR_INVALIDRECT. (For more information on clipping, see Clippers.)

Scaling

The **Blt** method automatically rescales the source image to fit the destination rectangle. If resizing is not your intention, for best performance you should make sure that your source and destination rectangles are exactly the same size, or else use **IDirectDrawSurface4:BltFast**. (See Blitting with BltFast.)

Hardware acceleration for scaling depends on the DDFXCAPS_BLT* flags in the **dwFXCaps** member of the **DDCAPS** structure for the device. If, for example, a device has the DDFXCAPS_BLTSTRETCHXN capability but not DDFXCAPS_BLTSTRETCHX, it can assist when the x-axis of the source rectangle is being multiplied by a whole number but not when non-integral (arbitrary) scaling is being done.

Devices might also support arithmetic scaling, which is scaling by interpolation rather than simple multiplication or deletion of pixels. For instance, if an axis was being increased by one-third, the pixels would be recolored to provide a closer approximation to the original image than would be produced by the doubling of every third pixel on that axis.

Applications cannot control the type of scaling done by the driver, except by setting the DDBLTFX_ARITHSTRETCHY flag in the **dwDDFX** member of the **DDBLTFX** structure passed to **Blt**. This flag requests that arithmetic stretching be done on the y-axis. Arithmetic stretching on the x-axis and arithmetic shrinking are not currently supported in the DirectDraw API, but a driver may perform them by default.

Other Effects

If you do not require any special effects other than scaling when using **Blt**, you can pass NULL as the *lpDDBltFx* parameter. Otherwise you can choose among a variety of effects specified in a **DDBLTFX** structure. Among these, color fills and mirroring are supported by the HEL, so they are always available. Most other effects depend on hardware support.

For a complete view of the effects capabilities of the HEL, run the DDraw Caps utility supplied with the DirectX Programmer's Reference and select HEL FX Caps from the HEL menu. For an explanation of the various flags, see **DDCAPS**. You can also check HEL capabilities within your own application by using the **IDirectDraw4::GetCaps** method.

When you specify an effect that requires a value in one of the members of the **DDBLTFX** structure passed to the **IDirectDrawSurface4::Blt** method, you must also include the appropriate flags in the *dwFlags* parameter to show which members of the structure are valid.

Some effects require only the setting of a flag in the **dwFlags** member of **DDBLTFX.** One of these is DDBLTFX_NOTEARING. You can use this flag when you are blitting animated images directly to the front buffer, so that the blit is timed to coincide with the screen refresh and the possibility of tearing is minimized. Mirroring and rotation are also set by using flags.

Blitting effects include the standard raster operations (ROPs) used by GDI functions such as **BitBlt**. The only ROPs supported by the HEL are SRCCOPY (the default), BLACKNESS, and WHITENESS. Hardware support for other ROPs can be examined in the **DDCAPS** structure for the driver. If you wish to use any of the standard ROPS with the **Blt** method, you flag them in the **dwROP** member of the **DDBLTFX** structure.

The **dwDDROP** member of the **DDBLTFX** structure is for specifying ROPs specific to DirectDraw. However, no such ROPs are currently defined.

Alpha and Z Values

Opacity and depth values are not currently supported in DirectDraw blits. If alpha values are stored in the pixel format, they simply overwrite any alpha values in the destination rectangle. Values from alpha buffers and z-buffers are ignored. The members of the **DDBLTFX** structure that have to do with alpha channels and z-buffers (members whose names begin with "dwAlpha" and "dwZ"), and the corresponding flags for **Blt**, are not used. The same applies to the DDBLTFX_ZBUFFERBASEDEST and DDBLTFX_ZBUFFERRANGE flags in the **dwDDFX** member of the **DDBLTFX** structure.

Although z-buffers are currently used only in Direct3D applications, you can use **IDirectDrawSurface4::Blt** to set the depth value for a z-buffer surface, by setting the DDBLT_DEPTHFILL flag. For more information, see Clearing Depth Buffers.

For an overview of the use of alpha channels and z-buffers in Direct3D, see the following topics:

- Alpha States

- What Are Depth Buffers?

Blt Example

The following example, in which it is assumed that *lpDDS* is a valid **IDirectDrawSurface4** pointer, creates a symmetrical image within the surface by mirroring a rectangle from left to right:

```
RECT     rcSource, rcDest;
DDBLTFX  ddbltfx;

ZeroMemory(&ddbltfx, sizeof(ddbltfx));
ddbltfx.dwSize = sizeof(ddbltfx);
ddbltfx.dwDDFX = DDBLTFX_MIRRORLEFTRIGHT;

rcSource.top = 0; rcSource.left = 0;
rcSource.bottom = 100; rcSource.right = 200;
rcDest.top = 0; rcDest.left = 201;
rcDest.bottom = 100; rcDest.right = 401;

HRESULT hr = lpDDS->Blt(&rcDest,
                 lpDDS,
                 &rcSource,
                 DDBLT_WAIT | DDBLT_DDFX,
                 &ddbltfx);
```

Blit Timing

[This is preliminary documentation and subject to change.]

When you copy pixels to a surface using either **IDirectDrawSurface4::Blt** or **IDirectDrawSurface4::BltFast**, the method might fail with DDERR_WASSTILLDRAWING because the hardware blitter was not ready to accept the command.

If your application has no urgent business to perform while waiting for the blitter to come back into a state of readiness, you can specify the DDBLT_WAIT flag in the *dwFlags* parameter of **Blt**, or the equivalent DDBLTFAST_WAIT flag for **BltFast**. The flag causes the method to wait until the blit can be handed off to the blitter (or until an error other than DDERR_WASSTILLDRAWING occurs).

Blt accepts another flag, DDBLT_ASYNC, that takes advantage of any hardware FIFO (first in, first out) queuing capabilities.

Transparent Blitting

[This is preliminary documentation and subject to change.]

This section discusses the theory and practice of using transparent blitting to copy parts of a rectangular image selectively, using source and destination color keys.

The concepts are introduced in the following topic:

- What Is Transparent Blitting?

Information about the implementation of transparent blitting in DirectDraw is contained in the following topics:

- Color Key Format
- Setting Color Keys
- Blitting with Color Keys

What Is Transparent Blitting?

[This is preliminary documentation and subject to change.]

Transparent blitting enables you to create the illusion of nonrectangular blits when animating sprites. A sprite image is usually nonrectangular, but blits are always rectangular, so every pixel within the sprite's bounding rectangle becomes part of the data transfer. With transparent blitting, each pixel that is not part of the sprite image is treated as transparent when the blitter is moving the image to its destination, so that it does not overwrite the color in that pixel on the background image.

The artist creating the sprite chooses an arbitrary color or range of colors to be used as the transparency color key. This is typically a single uncommon color that the artist doesn't use for anything but transparency, and it is used to fill in all parts of the sprite rectangle that are not part of the desired image. At run time you set the color key for the surface containing the sprite. (If you wish, you can automatically set it to the color of the pixel in the upper left corner of the image.) Subsequent blits can take advantage of that color key, ignoring the pixels that match it. This type of color key is known as a source color key.

You can also use a color key on the destination surface, provided the hardware supports destination color keying. This destination color key is used for pixels that can be overwritten by a sprite. For example, the artist might be working on a foreground image that sprites are supposed to pass behind, such as the wall of a room with a window to the outside. The artist chooses an arbitrary color—one that isn't used elsewhere in the image—to represent the sky outside the window. When you set this color key for the destination surface and then blit a sprite to that surface, the sprite's pixels will overwrite only pixels that are using the destination color key. In the example, the sprite appears only in the window, but not on the wall or window frame. As a result, the sprite seems to be outside the room.

Source and destination color keys can be combined. In the example, the sprite could use a source color key so that its entire bounding rectangle does not block out the sky background.

Color Key Format

[This is preliminary documentation and subject to change.]

A color key is described in a **DDCOLORKEY** structure. If the color key is a single color, both members of this structure should be assigned the same value. Otherwise the color key is a range of colors.

Color keys are specified using the pixel format of a surface. If a surface is in a palettized format, the color key is given as an index or a range of indices. If the surface's pixel format is specified by a FOURCC code that describes a YUV format, the YUV color key is specified by the three low-order bytes in both the **dwColorSpaceLowValue** and **dwColorSpaceHighValue** members of the **DDCOLORKEY** structure. The lowest order byte contains the V data, the second lowest order byte contains the U data, and the highest order byte contains the Y data.

Some examples of valid color keys follow:

8-bit palettized mode

```
// Palette entry 26 is the color key.
dwColorSpaceLowValue = 26;
dwColorSpaceHighValue = 26;
```

24-bit true-color mode

```
// Color 255,128,128 is the color key.
dwColorSpaceLowValue = RGBQUAD(255,128,128);
dwColorSpaceHighValue = RGBQUAD(255,128,128);
```

FourCC YUV mode

```
// Any YUV color where Y is between 100 and 110
// and U or V is between 50 and 55 is transparent.
dwColorSpaceLowValue = YUVQUAD(100,50,50);
dwColorSpaceHighValue = YUVQUAD(110,55,55);
```

Support for a range of colors rather than a single color is hardware-dependent. Check the **dwCKeyCaps** member of the **DDCAPS** structure for the hardware. The HEL does not support color ranges.

Some hardware supports color ranges only for YUV pixel data, which is usually video. The transparent background in video footage (the "blue screen" against which the subject was photographed) might not be a single pure color, so a range of colors in the color key is desirable.

Setting Color Keys

[This is preliminary documentation and subject to change.]

You can set the source or destination color key for a surface either when creating it or afterwards.

To set a color key or keys when creating a surface, you assign the appropriate color values to one or both of the **ddckCKSrcBlt** and **ddckCKDestBlt** members of the **DDSURFACEDESC2** structure that is passed to **IDirectDraw4::CreateSurface**. To enable the color key for blitting, you must also include one or both of DDSD_CKSRCBLT or DDSD_CKDESTBLT in the **dwFlags** member.

To set a color key for an existing surface you use the **IDirectDrawSurface4::SetColorKey** method. You specify a key in the *lpDDColorKey* parameter and set either DDCKEY_SRCBLT or DDCKEY_DESTBLT in the *dwFlags* parameter to indicated whether you are setting a source or destination key. If the **DDCOLORKEY** structure contains a range of colors, you must also set the DDCKEY_COLORSPACE flag. If this flag is not set, only the **dwColorSpaceLowValue** member of the structure is used.

Blitting with Color Keys

[This is preliminary documentation and subject to change.]

If you want to use color keys for surfaces when calling the **IDirectDrawSurface4::BltFast** method, you must set one or both of the DDBLTFAST_SRCCOLORKEY or DDBLTFAST_DESTCOLORKEY flags in the *dwTrans* parameter.

In order to use colors keys when calling **IDirectDrawSurface4:Blt**, you pass one or both of the DDBLT_KEYSRC or DDBLT_KEYDEST flags in the *dwFlags* parameter. Alternatively, you can put the appropriate color values in the **ddckDestColorkey** and **ddckSrcColorkey** members of the **DDBLTFX** structure that is passed to the method through the *lpDDBltFx* parameter. In this case you must also set the DBLT_KEYSRCOVERRIDE or DDBLT_KEYDESTOVERRIDE flag, or both, in the *dwFlags* parameter, so that the selected keys are taken from the **DDBLTFX** structure rather than from the surface properties.

Color Fills

[This is preliminary documentation and subject to change.]

In order to fill all or part of a surface with a single color, you can use the **IDirectDrawSurface4::Blt** method with the DDBLT_COLORFILL flag. This technique allows you to quickly erase an area or draw a solid-colored background.

The following example fills an entire surface with the color blue, after obtaining the numerical value for blue from the pixel format:

```
/* It is assumed that lpDDS is a valid pointer to
   an IDirectDrawSurface4 interface. */

HRESULT ddrval;
DDPIXELFORMAT ddpf;

ddpf.dwSize = sizeof(ddpf);
if (SUCCEEDED(lpDSS->GetPixelFormat(&ddpf))
```

```
{
   DDBLTFX ddbltfx;

   ddbltfx.dwSize = sizeof(ddbltfx);
   ddbltfx.dwFillColor = ddpf.dwBBitMask; // Pure blue

   ddrval = lpDDS->Blt(
      NULL,       // Destination is entire surface
      NULL,       // No source surface
      NULL,       // No source rectangle
      DDBLT_COLORFILL, &ddbltfx);

   switch(ddrval)
   {
      case DDERR_WASSTILLDRAWING:
         .
         .
         .
      case DDERR_SURFACELOST:
         .
         .
         .
      case DD_OK:
         .
         .
         .
      default:
   }
}
```

Blitting to Multiple Windows

[This is preliminary documentation and subject to change.]

You can use a DirectDraw object and a DirectDrawClipper object to blit to multiple windows created by an application running at the normal cooperative level. For more information, see Using a Clipper with Multiple Windows.

Creating multiple DirectDraw objects that blit to each others' primary surface is not recommended.

Losing and Restoring Surfaces

[This is preliminary documentation and subject to change.]

The surface memory associated with a DirectDrawSurface object may be freed, while the DirectDrawSurface objects representing these pieces of surface memory

are not necessarily released. When a DirectDrawSurface object loses its surface memory, many methods return DDERR_SURFACELOST and perform no other action.

Surfaces can be lost because the display mode was changed or because another application received exclusive access to the display card and freed all of the surface memory currently allocated on the card. The **IDirectDrawSurface4::Restore** method re-creates these lost surfaces and reconnects them to their DirectDrawSurface object. If your application uses more than one surface, you can call the **IDirectDraw4::RestoreAllSurfaces** method to restore all of your surfaces at once.

Restoring a surface doesn't reload any bitmaps that may have existed in the surface prior to being lost. You must completely reconstitute the graphics they once held.

COM Reference Count Semantics for Surfaces

[This is preliminary documentation and subject to change.]

Being built upon COM means that DirectDraw follows certain rules that employ reference counts to manage object lifetimes. For a conceptual overview, see the COM documentation; a DirectDraw-centered discussion of the topic is found in Parent and Child Object Lifetimes.

By COM rules, when an interface pointer is copied by setting it to another variable or passing to another object, that copy represents another reference to the object, and therefore the **IUnknown::AddRef** method of the interface must be called to reflect the change. Not only should you follow COM reference counting rules when working with DirectDraw objects, but you should become familiar with the situations in which DirectDraw internally updates reference counts. Some DirectDraw methods—mostly those involving complex surface flipping chains—affect the reference counts of the surfaces involved, while methods involving clippers or palettes affect the reference counts of those objects. Knowing about these situations can make the difference in your application's stability and can prevent memory leaks. This section presents information divided into the following topics:

- When Reference Counts Will Change
- Reference Counts for Complex Surfaces
- Releasing Surfaces

Note:

There are some things to remember about the reference count of the DirectDraw object, in addition to the relationships discussed in this section. For more information, see Parent and Child Object Lifetimes in The DirectDraw Object.

When Reference Counts will Change

[This is preliminary documentation and subject to change.]

There are several DirectDraw methods that affect the reference count of a surface, and a few that affect other objects you can associate with a surface. You can think of these situations as "surface-only changes" and "cross-object changes":

Surface-only changes

Surface-only changes, as the name states, only affect the reference count of a surface object. For example, you might use the **IDirectDraw4::EnumSurfaces** to enumerate the current surfaces that fit a particular description. When the method invokes the callback function that you provide, it passes a pointer to an **IDirectDrawSurface4** interface, but it increments the reference count for the object before your application receives the pointer. It's your responsibility to release the object when you are finished with it. This will typically be at the end of your callback routine, or later if you choose to keep the object.

Most other surface-only changes affect the reference counts of complex surfaces, such as a flipping chain. Reference counts are a little more tricky for complex surfaces, because (in most cases) DirectDraw treats a complex surface as if it was a single object, even though it is a set of surfaces. In short, the **IDirectDrawSurface4::GetAttachedSurface** and **IDirectDrawSurface4::AddAttachedSurface** methods increment reference counts of surfaces, and **IDirectDrawSurface4::DeleteAttachedSurface** decrements the reference count. These methods don't affect the counts of any surfaces attached to the current surface. See the references for these methods and Reference Counts for Complex Surfaces for a additional details.

Cross-object changes

Cross-object reference count changes occur when you create an association between a surface and another object that performs a task for the surface, such as a clipper or a palette.

The **IDirectDrawSurface4::SetClipper** and **IDirectDrawSurface4::SetPalette** methods increment the reference count of the object being attached. After they are attached, the surface manages them; if the surface is released, it automatically releases any objects it is using. (For this reason, some applications release the interface for the object after these calls succeed. This is a perfectly valid practice.)

Once a clipper or palette is attached to a surface, you can call the **IDirectDrawSurface4::GetClipper** and **IDirectDrawSurface4::GetPalette** methods to retrieve them again. Because these methods return a copy of an interface pointer, they implicitly increment the reference count for the object being retrieved. When you're done with the interfaces, don't forget to release them—the objects that the interfaces represent won't disappear so long as the surface they are attached to still holds a reference to them.

Reference Counts for Complex Surfaces

[This is preliminary documentation and subject to change.]

The methods you use to manipulate a complex surface like a flipping chain all use surface interface pointers, and therefore they all affect the reference counts of the surfaces. Because a complex surface is really a series of single surfaces, the reference count relationships require a little more consideration. As you might expect, the **IDirectDrawSurface4::GetAttachedSurface** method returns the surface interface for a surface attached to the current surface. It does this after incrementing the reference count of the interface being retrieved; it's up to you to release the interface when you no longer need it. The **IDirectDrawSurface4::AddAttachedSurface** method attaches a new surface to the current one. Similarly, **AddAttachedSurface** increments the count for the surface being attached. You would use the **IDirectDrawSurface4::DeleteAttachedSurface** method to remove the surface from the chain and implicitly decrease its reference count.

What isn't immediately clear about these methods is that they don't affect the reference counts of the other objects that make up the complex surface. The **GetAttachedSurface** method simply increments the reference count of the surface it's retrieving, it doesn't affect the counts of the surfaces on which it depends. (The same situation applies to an explicit call to **IUnknown::AddRef**.) This means that the reference count for primary surface in a complex surface can reach zero before its subordinate surfaces reach zero. When the primary surface reference count reaches zero, all other surfaces attached to it are released regardless of their current reference counts. (It's like a tree: if you cut the base, the whole thing falls. In this case, the primary surface is the base.) Attempts to access subordinate surfaces after the primary surface has been deallocated will result in memory faults.

To avoid problems, make sure that your application has released all subordinate surface references before attempting to release the primary surface. It might be helpful to track the references you application holds, only accessing subordinate surface interfaces when you're sure that you also hold a reference the primary surface.

Releasing Surfaces

[This is preliminary documentation and subject to change.]

Like all COM interfaces, you must release surfaces by calling their **IDirectDrawSurface4::Release** method when you no longer need them.

Each surface you individually create must be explicitly released. However, if you implicitly created multiple surfaces with a single call to **IDirectDraw4::CreateSurface**, such as a flipping chain, you need only release the front buffer. In this case, any pointers you might have to back buffer surfaces are implicitly released and can no longer be used.

Explicitly releasing a back buffer surface doesn't affect the reference count of the other surfaces in the chain.

Enumerating Surfaces

[This is preliminary documentation and subject to change.]

By calling the **IDirectDraw4::EnumSurfaces** method you can request that DirectDraw enumerate surfaces in various ways. The **EnumSurfaces** method enables you to look for surfaces that fit, or don't fit, a provided surface description. DirectDraw calls a **EnumSurfacesCallback** that you include with the call for each enumerated surface.

There are two general ways to search—you can search for surfaces that the DirectDraw object has already created, or for surfaces that the DirectDraw object is capable of creating at the time (given the surface description and available memory). You specify what type of search you want by combining flags in the method's *dwFlags* parameter.

Enumerating existing surfaces

This is the most common type of enumeration. You enumerate existing surfaces by calling **EnumSurfaces**, specifying a combination of the DDENUMSURFACES_DOESEXIST search-type flag and one of the matching flags (DDENUMSURFACES_MATCH, DDENUMSURFACES_NOMATCH, or DDENUMSURFACES_ALL) in the *dwFlags* parameter. If you're enumerating all existing surfaces, you can set the *lpDDSD* parameter to NULL, otherwise set it to the address of an initialized **DDSURFACEDESC2** structure that describes the surface for which you're looking. You can set the third parameter, *lpContext*, to an address that will be passed to the enumeration function you specify in the fourth parameter, *lpEnumSurfacesCallback*.

The following code fragment shows what this call might look like to enumerate all of a DirectDraw object's existing surfaces.

```
HRESULT ddrval;
ddrval = lpDD->EnumSurfaces(DDENUMSURFACES_DOESEXIST |
                    DDENUMSURFACES_ALL, NULL, NULL,
                    EnumCallback);
if (FAILED(ddrval))
   return FALSE;
```

When searching for existing surfaces that fit a specific description, DirectDraw determines a match by comparing each member of the provided surface description to those of the existing surfaces. Only exact matches are enumerated. DirectDraw increments the reference counts of the enumerated surfaces, so make sure to release a surface if you don't plan to use it (or when you're done with it).

Enumerating possible surfaces

This type of enumeration is less common than enumerating existing surfaces, but it can be helpful to determine if a surface is supported before you attempt to create it. To perform this search, combine the DDENUMSURFACES_CANBECREATED and DDENUMSURFACES_MATCH flags when you call **IDirectDraw4::EnumSurfaces** (no other flag combinations are valid). The

DDSURFACEDESC2 structure you use with the call must be initialized to contain information about the surface characteristics that DirectDraw will use.

To enumerate surfaces that use a particular pixel format, include the DDSD_PIXELFORMAT flag in the **dwFlags** member of the **DDSURFACEDESC2** structure. Additionally, initialize the **DDPIXELFORMAT** structure in the surface description and set its **dwFlags** member to contain the desired pixel format flags—DDPF_RGB, DDPF_YUV, or both. You need not set any other pixel format values.

If you include the DDSD_HEIGHT and DDSD_WIDTH flags in the **DDSURFACEDESC2** structure, you can specify the desired dimensions in the **dwHeight** and **dwWidth** members. If you exclude these flags, DirectDraw uses the dimensions of the primary surface.

The following code fragment shows what this call could look like to enumerate all valid surface characteristics for 96×96 RGB or YUV surfaces:

```
DDSURFACEDESC2 ddsd;
HRESULT       ddrval;
ZeroMemory(&ddsd, sizeof(ddsd));

ddsd.dwSize  = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_PIXELFORMAT |
           DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddpfPixelFormat.dwFlags = DDPF_YUV | DDPF_RGB;
ddsd.dwHeight = 96;
ddsd.dwWidth  = 96;

ddrval = lpDD->EnumSurfaces(
             DDENUMSURFACES_CANBECREATED | DDENUMSURFACES_MATCH,
             &ddsd, NULL, EnumCallback);
if (ddrval != DD_OK)
   return FALSE;
```

When DirectDraw enumerates possible surfaces, it actually attempts to create a temporary surface that has the desired characteristics. If the attempt succeeds, then DirectDraw calls the provided **EnumSurfacesCallback** function with only the characteristics that worked; it does not provide the callback function with pointer to the temporary surface. Do not assume that a surface isn't supported if it isn't enumerated. DirectDraw's attempt to create a temporary surface could fail due to memory constraints that exist at the time of the call, resulting in those characteristics not being enumerated, even if the driver actually supports them.

Updating Surface Characteristics

[This is preliminary documentation and subject to change.]

You can update the characteristics of an existing surface by using the **IDirectDrawSurface4::SetSurfaceDesc** method. With this method, you can change

the pixel format and location of a DirectDrawSurface object's surface memory to system memory that your application has explicitly allocated. This is useful as it allows a surface to use data from a previously allocated buffer without copying. The new surface memory is allocated by the client application and, as such, the client application must also deallocate it.

When calling the **IDirectDrawSurface4::SetSurfaceDesc** method, the *lpddsd* parameter must be the address of a **DDSURFACEDESC2** structure that describes the new surface memory as well as a pointer to that memory. Within the structure, you can only set the **dwFlags** member to reflect valid members for the location of the surface memory, dimensions, pitch, and pixel format. Therefore, **dwFlags** can only contain combinations of the DDSD_WIDTH, DDSD_HEIGHT, DDSD_PITCH, DDSD_LPSURFACE, and DDSD_PIXELFORMAT flags, which you set to indicate valid structure members.

Before you set the values in the structure, you must allocate memory to hold the surface. The size of the memory you allocate is important. Not only do you need to allocate enough memory to accommodate the surface's width and height, but you need to have enough to make room for the surface pitch, which must be a **QWORD** (8 byte) multiple. Remember, pitch is measured in bytes, not pixels.

When setting surface values in the structure, the **lpSurface** member is a pointer to the memory you allocated and the **dwHeight** and **dwWidth** members describe the surface dimensions in pixels. If you specify surface dimensions, you must fill the **lPitch** member to reflect the surface pitch as well. Pitch must be a **DWORD** multiple. Likewise, if you specify pitch, you must also specify a width value. Lastly, the **ddpfPixelFormat** member describes the pixel format for the surface. With the exception of the **lpSurface** member, if you don't specify a value for these members, the method defaults to using the value from the current surface.

There are some restrictions you must be aware of when using **IDirectDrawSurface4::SetSurfaceDesc**, some of which are common sense. For example, the **lpSurface** member of the **DDSURFACEDESC2** structure must be a valid pointer to a system memory (the method doesn't support video memory pointers at this time). Also, the **dwWidth** and **dwHeight** members must be nonzero values. Lastly, you cannot reassign the primary surface or any surfaces within the primary's flipping chain.

You can set the same memory for multiple DirectDrawSurface objects, but you must take care that the memory is not deallocated while it is assigned to any surface object.

Using the **SetSurfaceDesc** method incorrectly will cause unpredictable behavior. The DirectDrawSurface object will not deallocate surface memory that it didn't allocate. Therefore, when the surface memory is no longer needed, it is your responsibility to deallocate it. However, when **SetSurfaceDesc** is called, DirectDraw frees the original surface memory that it implicitly allocated when creating the surface.

Accessing Surface Memory Directly

[This is preliminary documentation and subject to change.]

You can directly access the frame buffer or off-screen surface memory by using the **IDirectDrawSurface4::Lock** method. When you call this method, the *lpDestRect* parameter is a pointer to a **RECT** structure that describes the rectangle on the surface you want to access directly. To request that the entire surface be locked, set *lpDestRect* to NULL. Also, you can specify a **RECT** that covers only a portion of the surface. Providing that no two rectangles overlap, two threads or processes can simultaneously lock multiple rectangles in a surface.

The **Lock** method fills a **DDSURFACEDESC2** structure with all the information you need to properly access the surface memory. The structure includes information about the pitch (or stride) and the pixel format of the surface, if different from the pixel format of the primary surface. When you finish accessing the surface memory, call the **IDirectDrawSurface4::Unlock** method to unlock it.

While you have a surface locked, you can directly manipulate the contents. The following list describes some tips for avoiding common problems with directly rendering surface memory:

- Never assume a constant display pitch. Always examine the pitch information returned by the **IDirectDrawSurface4::Lock** method. This pitch can vary for a number of reasons, including the location of the surface memory, the type of display card, or even the version of the DirectDraw driver. For more information, see Width vs. Pitch.
- Make certain you blit to unlocked surfaces. DirectDraw blit methods will fail, returning DDERR_SURFACEBUSY or DDERR_LOCKEDSURFACES, if called on a locked surface. Similarly, GDI blit functions fail without returning error values if called on a locked surface that exists in display memory.
- Limit your application's activity while a surface is locked. While a surface is locked, DirectDraw often holds the Win16Mutex (also known as the Win16Lock) so that gaining access to surface memory can occur safely. The Win16Mutex serializes access to GDI and USER dynamic-link libraries, shutting down Windows for the duration between the **IDirectDrawSurface4::Lock** and **IDirectDrawSurface4::Unlock** calls. The **IDirectDrawSurface4::GetDC** method implicitly calls **Lock**, and the **IDirectDrawSurface4::ReleaseDC** implicitly calls **Unlock**.
- Always copy data aligned to display memory. (Windows 95 and Windows 98 use a page fault handler, Vflatd.386, to implement a virtual flat-frame buffer for display cards with bank-switched memory. The handler allows these display devices to present a linear frame buffer to DirectDraw. Copying data unaligned to display memory can cause the system to suspend operations if the copy spans memory banks.)

Unless you include the DDLOCK_NOSYSLOCK flag when you call the **Lock** method, locking the surface typically causes DirectDraw to take the Win16Mutex.

During the Win16Mutex all other applications, including Windows, cease execution. Since the Win16Mutex stops applications from executing, standard debuggers cannot be used while the lock is held. Kernel debuggers can be used during this period. DirectDraw always takes the Win16Mutex when locking the primary surface.

If a blit is in progress when you call **IDirectDrawSurface4::Lock**, the method will return immediately with an error, as a lock cannot be obtained. To prevent the error, use the DDLOCK_WAIT flag to cause the method to wait until a lock can be successfully obtained.

Locking portions of the primary surface can interfere with the display of a software cursor. If the cursor intersects the locked rectangle, it is hidden for the duration of the lock. If it doesn't intersect the rectangle, it is frozen for the duration of the lock. Neither of these effects occurs if the entire surface is locked.

Gamma and Color Controls

[This is preliminary documentation and subject to change.]

This section contains information about the gamma and color control interfaces used with DirectDrawSurface objects. Information is organized into the following topics:

- What Are Gamma and Color Controls?
- Using Gamma Controls
- Using Color Controls

Note

You should not attempt to use both the **IDirectDrawGammaControl** and **IDirectDrawColorControl** interfaces on a single surface. Their effects are undefined when used together.

What Are Gamma and Color Controls?

[This is preliminary documentation and subject to change.]

Through the gamma and color control interfaces, DirectDrawSurface objects enable you to change how the system displays the contents of the surface, without affecting the contents of the surface itself. You can think of these controls as very simple filters that DirectDraw applies to the data as it leaves a surface before being rendered on the screen. Surface objects implement the **IDirectDrawGammaControl** and **IDirectDrawColorControl** interfaces which expose methods to adjust how the surface's contents are filtered. You can retrieve a pointer to either interface by using the **IUnknown::QueryInterface** method of the target surface, specifying the *IID_IDirectDrawGammaControl* or *IID_IDirectDrawColorControl* reference identifiers.

Gamma controls, represented by the **IDirectDrawGammaControl** interface, make it possible for you to dynamically change how a surface's individual red, green, and blue levels map to the actual levels that the system displays. By setting gamma

levels, you can cause the user's screen to flash colors—red when the user's character is shot, green when they pick up a new item, and so on—without blitting new images to the frame buffer to achieve the effect. Or, you might adjust color levels to apply a color bias to the images in the frame buffer. Although this interface is similar to the color control interface, this one is the easiest to use, making it the best choice for game applications. For details, see Using Gamma Controls.

The **IDirectDrawColorControl** interface allows you to control color in a surface much like the color controls you might find on a television. The similarity between **IDirectDrawColorControl** and the actual controls on a TV is no mistake—this interface is most appropriate for adjusting how broadcast video looks in an overlay surface, so it makes sense that it should provide similar control over colors. You can use color controls to allow a user to change video characteristics such as hue, saturation, contrast, and several others. For more information, see Using Color Controls.

Using Gamma Controls

[This is preliminary documentation and subject to change.]

The **IDirectDrawGammaControl** interface, which you retrieve by querying the surface with the *IID_IDirectDrawGammaControl* reference identifier, allows you to manipulate ramp levels that affect the red, green, and blue color components of pixels from the surface before they are sent to the digital-to-analog converter (DAC) for display. Although all surface types support the **IDirectDrawGammaControl** interface, you are only allowed to adjust gamma on the primary surface. Attempts to call **IDirectDrawGammaControl::GetGammaRamp** or **IDirectDrawGammaControl::SetGammaRamp** on a surface other than the primary surface will fail.

In the following topics, this section describes the general concept of ramp levels, and provides information about working with those levels through the methods of **IDirectDrawGammaControl**:

- About Gamma Ramp Levels
- Detecting Gamma Ramp Support
- Setting and Retrieving Gamma Ramp Levels

About Gamma Ramp Levels

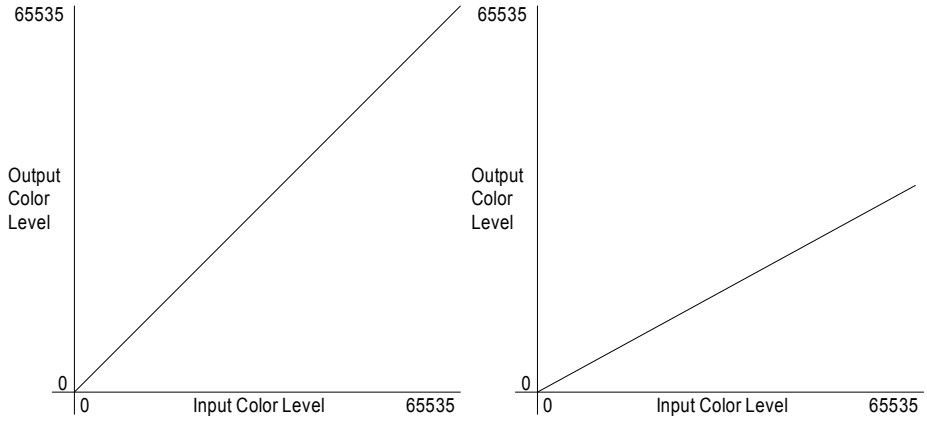
[This is preliminary documentation and subject to change.]

A *gamma ramp* in DirectDraw is a term used to describe a set of values that map the level of a particular color component (red, green, blue) for all pixels in the frame buffer to new levels that are received by the digital-to-analog converter (DAC) for display on the monitor. The remapping is performed by way of three simple look-up tables, one for each color component.

Here's how it works: DirectDraw takes a pixel from the frame buffer, and looks at it in terms of its individual red, green, and blue color components. Each component is

represented by a value from 0 to 65535. DirectDraw takes the original value, and uses it to index into an 256-element array (the ramp), where each element contains a value that replaces the original one. DirectDraw performs this "look-up and replace" process for each color component of each pixel within the frame buffer, thereby changing the final colors for all of the on-screen pixels.

It's handy to visualize the ramp values by graphing them. The left graph of the two following graphs shows a ramp that doesn't modify colors at all, and the right graph shows a ramp that imposes a negative bias to the color component to which it is applied.



The array elements for the graph on the left would contain values identical to their index (0 in the element at index 0, and 65535 at index 255). This type of ramp is the default, as it doesn't change the input values before they're displayed. The right graph is a little more interesting; its ramp contains values that range from 0 in the first element to 32768 in the last element, with values ranging relatively uniformly in between. The effect is that the color component that uses this ramp appears muted on the display. You are not limited to using linear graphs; if your application needs to assign arbitrary mapping, it's free to do so. You can even set the entries to all zeroes to leech a particular color component completely from the display.

Detecting Gamma Ramp Support

[This is preliminary documentation and subject to change.]

You can determine whether the hardware supports dynamic gamma ramp adjustment by calling the **IDirectDraw4::GetCaps** method. After the call, if the DDCAPS2_PRIMARYGAMMA flag is present in the **dwFlags2** member of the associate **DDCAPS** structure, the hardware supports dynamic gamma ramps. DirectDraw does not attempt to emulate this feature, so if the hardware doesn't support it, you can't use it.

Setting and Retrieving Gamma Ramp Levels

[This is preliminary documentation and subject to change.]

Gamma ramp levels are effectively look-up tables that DirectDraw uses to map the frame buffer color components to new levels that will be displayed. For more information, see About Gamma Ramp Levels. You set and retrieve ramp levels for the primary surface by calling the **IDirectDrawGammaControl::SetGammaRamp** and **IDirectDrawGammaControl::GetGammaRamp** methods. Both methods accept two parameters, but the first parameter is reserved for future use, and should be set to zero. The second parameter, *lpRampData*, is the address of a **DDGAMMARAMP** structure. The **DDGAMMARAMP** structure contains three 256-element arrays of **WORD**s, one array each to contain the red, green, and blue gamma ramps.

You can include the DDSGR_CALIBRATE value when calling the **IDirectDrawGammaControl::SetGammaRamp** to invoke the calibrator when setting new gamma levels. Calibrating gamma ramps incurs some processing overhead, and should not be used frequently. Setting a calibrated gamma ramp will provide a consistent and absolute gamma value for the viewer, regardless of the display adapter and monitor.

Not all systems support gamma calibration. To determine if gamma calibration is supported, call **IDirectDraw4::GetCaps**, and examine the **dwCaps2** member of the associated **DDCAPS** structure after the method returns. If the DDCAPS2_CANCALIBRATEGAMMA capability flag is present, then gamma calibration is supported.

When setting new ramp levels, keep in mind that that the levels you set in the arrays are only used when your application is in full-screen, exclusive mode. Whenever your application changes to normal mode, the ramp levels are set aside, taking effect again when the application reinstates full-screen mode. In addition, remember that you cannot set ramp levels for any surface other than the primary.

Note

Those very familiar with the Win32® API might wonder why DirectDraw exposes an interface like **IDirectDrawGammaControl**, when Win32 offers the **GetDeviceGammaRamp** and **SetDeviceGammaRamp** functions for the same surfaces. Although the Win32 API includes these functions, they do not always succeed on all Windows platforms like the methods of the **IDirectDrawGammaControl** interface.

Using Color Controls

[This is preliminary documentation and subject to change.]

You set and retrieve surface color controls through the **IDirectDrawColorControl** interface, which can be retrieved by querying the DirectDrawSurface object using the *IID_IDirectDrawColorControl* reference identifier.

Color control information is represented by a **DDCOLORCONTROL** structure, which is used with both methods of the interface, **IDirectDrawColorControl::SetColorControls** and **IDirectDrawColorControl::GetColorControls**. The first structure member,

dwSize, should be set to the size of the structure, in bytes, before you use it. How you use the next member, **dwFlags**, depends on whether you are setting or retrieving color controls. If you are setting new color controls, set **dwFlags** to a combination of the appropriate flags to indicate which of the other structure members contain valid data that you've set. However, when retrieving color controls, you don't need to set the **dwFlags** before using it—it will contain flags telling you which members are valid after the **IDirectDrawColorControl::GetColorControls** method returns.

The remaining **DDCOLORCONTROL** structure members can contain values that describe the brightness, contrast, hue, saturation, sharpness, gamma, and whether color is used. Note that the structure contains information about gamma correction. This is a single gamma value that affects overall brightness, and it should not be confused with the gamma adjustment features provided through the **IDirectDrawGammaControl** interface.

Overlay Surfaces

[This is preliminary documentation and subject to change.]

This section contains information about DirectDraw overlay surface support. The following topics are discussed:

- Overlay Surface Overview
- Significant DDCAPS Members and Flags
- Source and Destination Rectangles
- Boundary and Size Alignment
- Minimum and Maximum Stretch Factors
- Overlay Color Keys
- Positioning Overlay Surfaces
- Creating Overlay Surfaces
- Overlay Z-Orders
- Flipping Overlay Surfaces

For information about implementing overlay surfaces, see Tutorial 6: Using Overlay Surfaces.

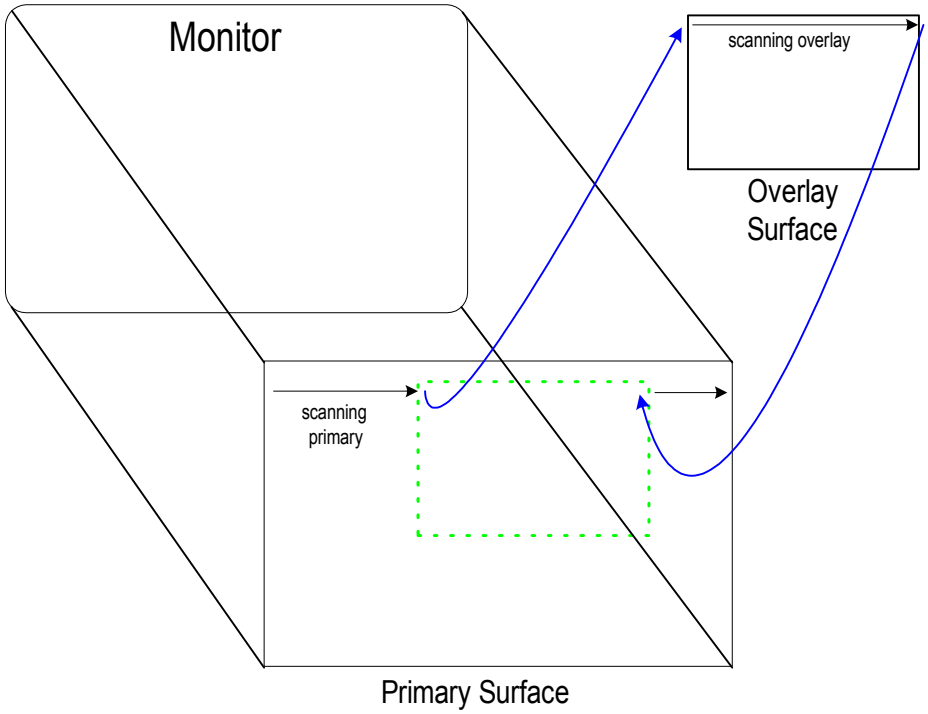
Overlay Surface Overview

[This is preliminary documentation and subject to change.]

Overlay surfaces, casually referred to as overlays, are surfaces with special hardware-supported capabilities. Overlay surfaces are frequently used to display live video, recorded video, or still bitmaps over the primary surface without blitting to the primary surface or changing the primary surface's contents in any way. Overlay surface support is provided entirely by the hardware; DirectDraw supports any

capabilities as reported by the display device driver. DirectDraw does not emulate overlay surfaces.

An overlay surface is analogous to a clear piece of plastic that you draw on and place in front of the monitor. When the overlay is in front of the monitor, you can see both the overlay and the contents of the primary surface together, but when you remove it, the primary surface's contents are unchanged. In fact, the mechanics of overlays work much like the clear plastic analogy. When you display an overlay surface, you're telling the device driver where and how you want it to be visible. While the display device paints scan lines to the monitor, it checks the location of each pixel in the primary surface to see if an overlay should be visible there instead. If so, the display device substitutes data from the overlay surface for the corresponding pixel, as shown in the following illustration.



By using this method, the display adapter produces a composite of the primary surface and the overlay on the monitor, providing transparency and stretching effects, without modifying the contents of either surface. The composited surfaces are injected into the video stream and sent directly to the monitor. Because this on-the-fly processing and pixel substitution is handled at the hardware level, no noticeable performance loss occurs when displaying overlays. Additionally, this method makes it possible to seamlessly composite primary and overlay surfaces with different pixel formats.

You create overlay surfaces by calling the **IDirectDraw4::CreateSurface** method, specifying the DDSCAPS_OVERLAY flag in the associated **DDSCAPS2** structure.

Overlay surfaces can only be created in video memory, so you must also include the DDSCAPS_VIDEOMEMORY flag. As with other types of surfaces, by including the appropriate flags you can create either a single overlay or a flipping chain made up of multiple overlay surfaces.

Significant DDCAPS Members and Flags

[This is preliminary documentation and subject to change.]

You can retrieve information about the supported overlay features by calling the **IDirectDraw4::GetCaps** method. The method fills a **DDCAPS** structure with information describing all features.

When reporting hardware features, the device driver sets flags in the **dwCaps** structure member to indicate when a given type of restriction is enforced by the hardware. After retrieving the driver capabilities, examine the flags in the **dwCaps** member for information about which restrictions apply. The **DDCAPS** structure contains nine members that carry information describing hardware restrictions for overlay surfaces. The following table lists the overlay related members and their corresponding flags:

Member	Flag
dwMaxVisibleOverlays	This member is always valid
dwCurrVisibleOverlays	This member is always valid
dwAlignBoundarySrc	DDCAPS_ALIGNBOUNDARYSRC
dwAlignSizeSrc	DDCAPS_ALIGNSIZESRC
dwAlignBoundaryDest	DDCAPS_ALIGNBOUNDARYDEST
dwAlignSizeDest	DDCAPS_ALIGNSIZEDEST
dwAlignStrideAlign	DDCAPS_ALIGNSTRIDE
dwMinOverlayStretch	DDCAPS_OVERLAYSTRETCH
dwMaxOverlayStretch	DDCAPS_OVERLAYSTRETCH

The **dwMaxVisibleOverlays** and **dwCurrVisibleOverlays** members carry information about the maximum number of overlays the hardware can display, and how many of them are currently visible.

Additionally, the hardware reports rectangle position and size alignment restrictions in the **dwAlignBoundarySrc**, **dwAlignSizeSrc**, **dwAlignBoundaryDest**, **dwAlignSizeDest**, and **dwAlignStrideAlign** members. The values in these members dictate how you must size and position source and destination rectangles when displaying overlay surfaces. For more information, see Source and Destination Rectangles and Boundary and Size Alignment.

Also, the hardware reports information about stretch factors in the **dwMinOverlayStretch** and **dwMaxOverlayStretch** members. For more information, see Minimum and Maximum Stretch Factors.

Source and Destination Rectangles

[This is preliminary documentation and subject to change.]

To display an overlay surface, you call the overlay surface's **IDirectDrawSurface4::UpdateOverlay** method, specifying the DDOVER_SHOW flag in the *dwFlags* parameter. The method requires you to specify a source and destination rectangle in the *lpSrcRect* and *lpDestRect* parameters. The source rectangle describes a rectangle on the overlay surface that will be visible on the primary surface. To request that the method use the entire surface, set the *lpSrcRect* parameter to NULL. The destination rectangle describes a portion of the primary surface on which the overlay surface will be displayed.

Source and destination rectangles do not need to be the same size. You can often specify a destination rectangle smaller or larger than the source rectangle, and the hardware will shrink or stretch the overlay appropriately when it is displayed.

To successfully display an overlay surface, you might need to adjust the size and position of both rectangles. Whether this is necessary depends on the restrictions imposed by the device driver. For more information, see Boundary and Size Alignment and Minimum and Maximum Stretch Factors.

Boundary and Size Alignment

[This is preliminary documentation and subject to change.]

Due to various hardware limitations, some device drivers impose restrictions on the position and size of the source and destination rectangles used to display overlay surfaces. To find out which restrictions apply for a device, call the **IDirectDraw4::GetCaps** method and then examine the overlay-related flags in the **dwCaps** member of the **DDCAPS** structure. The following table shows the members and flags specific to boundary and size alignment restrictions:

Category	Flag	Member
Boundary (position) restrictions	DDCAPS_ALIGNBOUNDARYSRC	**dwAlignBoundarySrc**
	DDCAPS_ALIGNBOUNDARYDEST	**dwAlignBoundaryDest**
Size restrictions	DDCAPS_ALIGNSIZESRC	**dwAlignSizeSrc**
	DDCAPS_ALIGNSIZEDEST	**dwAlignSizeDest**

There are two types of restrictions, boundary restrictions and size restrictions. Both types of restrictions are expressed in terms of pixels (not bytes) and can apply to the source and destination rectangles. Also, these restrictions can vary depending on the pixel formats of the overlay and primary surface.

Boundary restrictions affect where you can position a source or destination rectangle. The values in the **dwAlignBoundarySrc** and **dwAlignBoundaryDest** members tell you how to align the top left corner of the corresponding rectangle. The x-coordinate

of the top left corner of the rectangle (the **left** member of the **RECT** structure), must be a multiple of the reported value.

Size restrictions affect the valid widths for source and destination rectangles. The values in the **dwAlignSizeSrc** and **dwAlignSizeDest** members tell you how to align the width, in pixels, of the corresponding rectangle. Your rectangles must have a pixel width that is a multiple of the reported value. If you stretch the rectangle to comply with a minimum required stretch factor, be sure that the stretched rectangle is still size aligned. After stretching the rectangle, align its width by rounding up, not down, so you preserve the minimum stretch factor. For more information, see Minimum and Maximum Stretch Factors.

Minimum and Maximum Stretch Factors

[This is preliminary documentation and subject to change.]

Due to hardware limitations, some devices restrict how wide a destination rectangle can be compared with the corresponding source rectangle. DirectDraw communicates these restrictions as stretch factors. A stretch factor is the ratio between the widths of the source and destination rectangles. If the driver provides information about stretch factors, it sets the DDCAPS_OVERLAYSTRETCH flag in the **DDCAPS** structure after you call the **IDirectDraw4::GetCaps** method. Note that stretch factors are reported multiplied by 1000, so a value of 1300 actually means 1.3 (and 750 would be 0.75).

Devices that do not impose limits on stretching or shrinking an overlay destination rectangle often report a minimum and maximum stretch factor of 0.

The minimum stretch factor tells you how much wider or narrower than the source rectangle the destination rectangle needs to be. If the minimum stretch factor is greater than 1000, then you must increase the destination rectangle's width by that ratio. For instance, if the driver reports 1300, you must make sure that the destination rectangle's width is at least 1.3 times the width of the source rectangle. Similarly, a minimum stretch factor less than 1000 indicates that the destination rectangle can be smaller than the source rectangle by that ratio.

The maximum stretch factor tells the maximum amount you can stretch the width of the destination rectangle. For example, if the maximum stretch factor is 2000, you can specify destination rectangles that are up to, but not wider than, twice the width of the source rectangle. If the maximum stretch factor is less than 1000, then you must shrink the width of the destination rectangle by that ratio to be able to display the overlay.

After stretching, the destination rectangle must conform to any size alignment restrictions the device might require. Therefore, it's a good idea to stretch the destination rectangle before adjusting it to be size aligned. For more information, see Boundary and Size Alignment.

Hardware does not require that you adjust the height of destination rectangles. You can increase a destination rectangle's height to preserve aspect ratio without negative effects.

Overlay Color Keys

[This is preliminary documentation and subject to change.]

Like other types of surfaces, overlay surfaces use source and destination color keys for controlling transparent blit operations between surfaces. Because overlay surfaces are not displayed by blitting, there needs to be a different way to control how an overlay surface is displayed over the primary surface when you call the **IDirectDrawSurface4::UpdateOverlay** method. This need is filled by overlay color keys. Overlay color keys, like their blit-related counterparts, have a source version and a destination version that you set by calling the **IDirectDrawSurface4::SetColorKey** method. (For more information, see Setting Color Keys.) You use the DDCKEY_SRCOVERLAY or DDCKEY_DESTOVERLAY flags to set a source or destination overlay color key. Overlay surfaces can employ blit and overlay color keys together to control blit operations and overlay display operations appropriately; the two types of color keys do not conflict with one another.

The **IDirectDrawSurface4::UpdateOverlay** method uses the source overlay color key to determine which pixels in the overlay surface should be considered transparent, allowing the contents of the primary surface to show through. Likewise, the method uses the destination overlay color key to determine the parts of the primary surface that will be covered up by the overlay surface when it is displayed. The resulting visual effect is the same as that created by blit-related color keys.

Positioning Overlay Surfaces

[This is preliminary documentation and subject to change.]

After initially displaying an overlay by calling the **IDirectDrawSurface4::UpdateOverlay** method, you can update the destination rectangle's by calling the **IDirectDrawSurface4::SetOverlayPosition** method.

Make sure that the positions you specify comply with any boundary alignment restrictions enforced by the hardware. For more information, see Boundary and Size Alignment. Also remember that **SetOverlayPosition** doesn't perform clipping for you; using coordinates that would potentially make the overlay run off the edge of the target surface will cause the method to fail, returning DDERR_INVALIDPOSITION.

Creating Overlay Surfaces

[This is preliminary documentation and subject to change.]

Like all surfaces, you create an overlay surface by calling the **IDirectDraw4::CreateSurface** method. To create an overlay, include the DDSCAPS_OVERLAY flag in the associated **DDSCAPS2** structure.

Overlay support varies widely across display devices. As a result, you cannot be sure that a given pixel format will be supported by most drivers and must therefore be prepared to work with a variety of pixel formats. You can request information about

the non-RGB formats that a driver supports by calling the **IDirectDraw4::GetFourCCCodes** method.

When you attempt to create an overlay surface, it is advantageous to try creating a surface with the most desirable pixel format, falling back on other pixel formats if a given pixel format isn't supported.

You can create overlay surface flipping chains. For more information, see Creating Complex Surfaces and Flipping Chains.

Overlay Z-Orders

[This is preliminary documentation and subject to change.]

Overlay surfaces are assumed to be on top of all other screen components, but when you display multiple overlay surfaces, you need some way to visually organize them. DirectDraw supports *overlay z-ordering* to manage the order in which overlays clip each other. Z-order values represent conceptual distances from the primary surface toward the viewer. They range from 0, which is just on top of the primary surface, to 4 billion, which is as close to the viewer as possible, and no two overlays can share the same z-order. You set z-order values by calling the **IDirectDrawSurface4::UpdateOverlayZOrder** method.

Destination color keys are affected only by the bits on the primary surface, not by overlays occluded by other overlays. Source color keys work on an overlay whether or not a z-order was specified for the overlay.

Overlays without a specified z-order are assumed to have a z-order of 0. Overlays that do not have a specified z-order behave in unpredictable ways when overlaying the same area on the primary surface.

A DirectDraw object does not track the z-orders of overlays displayed by other applications.

Note

You can ensure proper clipping of multiple overlay surfaces by calling **UpdateOverlayZOrder** in response to WM_KILLFOCUS messages. When you receive this message, set your overlay surface to the rearmost z-order position by calling the **UpdateOverlayZOrder** method with the *dwFlags* parameter set to DDOVERZ_SENDTOBACK.

Flipping Overlay Surfaces

[This is preliminary documentation and subject to change.]

Like other types of surfaces, you can create overlay flipping chains. After creating a flipping chain of overlays, call the **IDirectDrawSurface4::Flip** method to flip between them. For more information, see Flipping Surfaces.

Software decoders displaying video with overlay surfaces can use the DDFLIP_ODD and DDFLIP_EVEN flags when calling the **Flip** method to use features that reduce

motion artifacts. If the driver supports odd-even flipping, the DDCAPS2_CANFLIPODDEVEN flag will be set in the **DDCAPS** structure after retrieving driver capabilities. If DDCAPS2_CANFLIPODDEVEN is set, you can include the DDOVER_BOB flag when calling the **IDirectDrawSurface4::UpdateOverlay** method to inform the driver that you want it to use the "Bob" algorithm to minimize motion artifacts. Later, when you call **Flip** with the DDFLIP_ODD or DDFLIP_EVEN flag, the driver will automatically adjust the overlay source rectangle to compensate for jittering artifacts.

If the driver doesn't set the DDCAPS2_CANFLIPODDEVEN flag when you retrieve hardware capabilities, **UpdateOverlay** will fail if you specify the DDOVER_BOB flag.

For more information about the Bob algorithm, see Solutions to Common Video Artifacts.

Compressed Texture Surfaces

[This is preliminary documentation and subject to change.]

A surface can contain a bitmap to be used for texturing 3-D objects. When creating the surface you must specify the DDSCAPS_TEXTURE flag in the **dwFlags** member of the **DDSCAPS** structure.

For more information on the use of textures in Direct3D Immediate Mode, see Textures.

In order to reduce the amount of memory consumed by textures, DirectDraw supports the compression of texture surfaces.

Some Direct3D devices support compressed texture surfaces natively. On such devices, once you have created a compressed surface and loaded the data into it, the surface can be used in Direct3D just like any other texture surface. Direct3D handles decompression when the texture is mapped to a 3-D object.

Other devices do not support compressed texture surfaces natively. When using such devices, you may still find it useful to use compressed surfaces to represent textures on disk or for textures that are loaded into memory but not currently being used. You can use DirectDraw to convert the compressed textures to an uncompressed format before giving the texture to Direct3D.

For more information on texture compression in DirectDraw, see the following topics:

- Creating Compressed Textures
- Decompressing Compressed Textures
- Transparency in Blits to Compressed Textures
- Compressed Texture Formats

For information on using compressed textures in Direct3D Immediate Mode, see Texture Compression.

Creating Compressed Textures

[This is preliminary documentation and subject to change.]

To describe a compressed texture surface in the **DDSURFACEDESC2** structure when creating the surface, you must include the following steps:

- Specify the DDSCAPS_TEXTURE flag in the **dwFlags** member of the **DDSCAPS** structure, just as you would for any texture.
- Set the **dwFourCC** member of the **DDPIXELFORMAT** structure to one of the DXT codes described later.
- Include DDPF_FOURCC in the **dwFlags** member of **DDPIXELFORMAT**. Do not set the DDPF_RGB flag.
- Specify a width and height that are a multiple of 4 pixels.

There are two ways to load image data into a compressed texture surface:

- Create a regular RGB or ARGB surface and load a normal bitmap into it, then use **IDirectDrawSurface4::Blt** or **IDirectDrawSurface4::BltFast** to blit from the uncompressed surface to the compressed surface. DirectDraw does the compression for you.
- Load the compressed data from a file and copy it directly into the surface memory. (See Accessing Surface Memory Directly.) You can create and convert compressed texture (DDS) files using the DirectX Texture Tool (Dxtex.exe) supplied with the Programmer's Reference. You can also create your own DDS files and either copy the data from compressed surfaces or else use your own routines to convert regular bitmap data to one of the compressed formats.

Note

When you call **IDirectDrawSurface4::Lock** or **IDirectDrawSurface4::GetSurfaceDesc** on a compressed surface, the DDSD_LINEARSIZE flag is set in the **dwFlags** member of the **DDSURFACEDESC** structure, and the **dwLinearSize** member contains the number of bytes allocated to contain the compressed surface data. The **dwLinearSize** parameter resides in a union with the **lPitch** parameter, so these parameters are mutually exclusive, as are the flags DDSD_LINEARSIZE and DDSD_PITCH.
The advantage of this behavior is that an application can copy the contents of a compressed surface to a file without having to calculate for itself how much storage is required for a surface of a particular width and height in the specific format.

The following table shows the five types of compressed textures. For more information on how the data is stored (you need to know this only if you are writing your own compression routines) see Compressed Texture Formats.

FOURCC	Description	Alpha

		premultiplied?
DXT1	Opaque / one-bit alpha	n/a
DXT2	Explicit alpha	Yes
DXT3	Explicit alpha	No
DXT4	Interpolated alpha	Yes
DXT5	Interpolated alpha	No

Note

When you blit from a non-premultiplied format to a premultiplied format, DirectDraw scales the colors based on the alpha values. Blitting from a premultiplied format to a non-premultiplied format is not supported. If you try to blit from a premultiplied-alpha source to a non-premultiplied-alpha destination, the method will return DDERR_INVALIDPARAMS. If you blit from a premultiplied-alpha source to a destination that has no alpha, the source color components, which have been scaled by alpha, will be copied as is.

Decompressing Compressed Textures

[This is preliminary documentation and subject to change.]

As with compressing a texture surface, decompressing a compressed texture is performed through DirectDraw blitting services. The HEL performs decompressing blits between system memory surfaces, so these always supported. Likewise, the HEL always performs blits for compressed managed textures (the DDSCAPS2_TEXTUREMANAGE capability). For other situations, the restrictions discussed in the following paragraphs apply.

If the driver supports the creation of compressed video-memory surfaces, then the driver can also perform decompressing blits from a compressed video-memory surface to an uncompressed video or system memory surface, so long as the destination surface has the DDSCAPS_OFFSCREENPLAIN capability.

Blits from compressed system-memory surfaces to uncompressed video-memory surfaces are largely unsupported and should not be attempted, even when the driver supports compressed textures. This does not mean that it is impossible to decompress a compressed system-memory surface and move its contents into a video memory surface; it merely requires an additional step:

To decompress a system memory surface into video memory:

1. Create an uncompressed, offscreen-plain, surface in system memory of the desired dimensions and pixel format.
2. Blit from the compressed system-memory surface to the uncompressed system-memory surface. (The DirectDraw HEL performs decompression in this case.)
3. Blit the uncompressed surface to the uncompressed video-memory surface.

Transparency in Blits to Compressed Textures

[This is preliminary documentation and subject to change.]

DirectDraw provides a special trick for creating compressed textures with alpha from plain RGB surfaces. If a source color key is provided on the source RGB surface, DirectDraw assigns an alpha value of 0 to all pixels of that color in the destination. This technique is especially useful for creating DXT1 textures, since they effectively have only 1 bit of alpha information per pixel.

Note

There are no flags that control this behavior. If you do not want any transparency in your compressed texture, do not set a source color key on the source surface.

Compressed Texture Formats

[This is preliminary documentation and subject to change.]

This section contains information on the internal organization of compressed texture formats. You don't need these details in order to use compressed textures, because DirectDraw handles conversion to and from compressed formats. However, you might find this information useful if you want to operate on compressed surface data directly.

DirectDraw uses a compression format that divides texture maps into 4x4 texel blocks. If the texture contains no transparency (is opaque), or if the transparency is specified by a one-bit alpha, an 8-byte block represents the texture map block. If the texture map does contain transparent texels, using an alpha channel, a 16-byte block represents it.

These two types of format are discussed in the following sections:

- Opaque and One-bit Alpha Textures
- Textures with Alpha Channels

Note

Any single texture must specify that its data is stored as 64 or 128 bits per group of 16 texels. If 64-bit blocks—that is, format DXT1—are used for the texture, it is possible to mix the opaque and one-bit alpha formats on a per-block basis within the same texture. In other words, the comparison of the unsigned integer magnitude of color_0 and color_1 is performed uniquely for each block of 16 texels.
When 128-bit blocks are used, then the alpha channel must be specified in either explicit (format DXT2 or DXT3) or interpolated mode (format DXT4 or DXT5) for the entire texture. Note that as with color, once interpolated mode is selected then either 8 interpolated alphas or 6 interpolated alphas mode can be used on a block-by-block basis. Again the magnitude comparison of alpha_0 and alpha_1 is done uniquely on a block-by-block basis.

Opaque and One-bit Alpha Textures

[This is preliminary documentation and subject to change.]

Texture format DXT1 is for textures that are opaque or have a single transparent color.

For each opaque or one-bit alpha block, two 16-bit values (RGB 5:6:5 format) and a 4x4 bitmap with 2-bits-per-pixel are stored. This totals 64 bits for 16 texels, or 4-bits-per-texel. In the block bitmap, there are two bits per texel to select between the four colors, two of which are stored in the encoded data. The other two colors are derived from these stored colors by linear interpolation.

The one-bit alpha format is distinguished from the opaque format by comparing the two 16-bit color values stored in the block. They are treated as unsigned integers. If the first color is greater than the second, it implies that only opaque texels are defined. This means four colors will be used to represent the texels. In four-color encoding, there are two derived colors and all four colors are equally distributed in RGB color space. This format is analogous to RGB 5:6:5 format. Otherwise, for one-bit alpha transparency, three colors are used and the fourth is reserved to represent transparent texels.

In three-color encoding, there is one derived color and the fourth two-bit code is reserved to indicate a transparent texel (alpha information). This format is analogous to RGBA 5:5:5:1, where the final bit is used for encoding the alpha mask.

The following piece of pseudo-code illustrates the algorithm for deciding whether three- or four-color encoding is selected:

```
if (color_0 > color_1)
{
   // Four-color block: derive the other two colors.
   // 00 = color_0, 01 = color_1, 10 = color_2, 11 = color_3
   // These two bit codes correspond to the 2-bit fields
   // stored in the 64-bit block.
    color_2 = (2 * color_0 + color_1) / 3;
    color_3 = (color 0 + 2 * color_1) / 3;
}
else
{
   // Three-color block: derive the other color.
   // 00 = color_0,  01 = color_1,  10 = color_2,
   // 11 = transparent.
   // These two bit codes correspond to the 2-bit fields
   // stored in the 64-bit block.
    color_2 = (color_0 + color_1) / 2;
    color_3 = transparent;
}
```

The following tables show the memory layout for the 8-byte block. It is assumed that the first index corresponds to the y-coordinate and the second corresponds to the x-coordinate. For example, Texel[1][2] refers to the texture map pixel at (x,y) = (2,1).

Here is the memory layout for the 8-byte (64-bit) block:

Word address	**16-bit word**
0	Color_0
1	Color_1
2	Bitmap Word_0
3	Bitmap Word_1

Color_0 and Color_1 (colors at the two extremes) are laid out as follows:

Bits	**Color**
4:0 (LSB)	Blue color component
10:5	Green color component
15:11	Red color component

Bitmap Word_0 is laid out as follows:

Bits	**Texel**
1:0 (LSB)	Texel[0][0]
3:2	Texel[0][1]
5:4	Texel[0][2]
7:6	Texel[0][3]
9:8	Texel[1][0]
11:10	Texel[1][1]
13:12	Texel[1][2]
15:14 (MSB)	Texel[1][3]

Bitmap Word_1 is laid out as follows:

Bits	**Texel**
1:0 (LSB)	Texel[2][0]
3:2	Texel[2][1]
5:4	Texel[2][2]
7:6	Texel[2][3]
9:8	Texel[3][0]
11:10	Texel[3][1]
13:12	Texel[3][2]
15:14 (MSB)	Texel[3][3]

Example of Opaque Color Encoding

As an example of opaque encoding, we will assume that the colors red and black are at the extremes. We will call red color_0 and black color_1. There will be four interpolated colors that form the uniformly distributed gradient between them. To determine the values for the 4x4 bitmap, the following calculations are used:

```
00 ? color_0
01 ? color_1
10 ? 2/3 color_0 + 1/3 color_1
11 ? 1/3 color_0 + 2/3 color_1
```

Example of One-bit Alpha Encoding

This format is selected when the unsigned 16-bit integer, color_0, is less than the unsigned 16-bit integer, color_1. An example of where this format could be used is leaves on a tree to be shown against a blue sky. Some texels could be marked as transparent while three shades of green are still available for the leaves. Two of these colors fix the extremes, and the third color is an interpolated color.

The bitmap encoding for the colors and the transparency is determined using the following calculations:

```
00 ? color_0
01 ? color_1
10 ? 1/2 color_0 + 1/2 color_1
11  ?  Transparent
```

Textures with Alpha Channels

[This is preliminary documentation and subject to change.]

There are two ways to encode texture maps that exhibit more complex transparency. In each case, a block that describes the transparency precedes the 64-bit block already described. The transparency is either represented as a 4x4 bitmap with four bits per pixel (explicit encoding), or with fewer bits and linear interpolation analogous to what is used for color encoding.

The transparency block and the color block are laid out as follows:

Word Address	64-bit Block
3:0	Transparency block
7:4	Previously described 64-bit block

Explicit Texture Encoding

For explicit texture encoding (DXT2 and DXT3 formats), the alpha components of the texels that describe transparency are encoded in a 4x4 bitmap with 4 bits per texel. These 4 bits can be achieved through a variety of means such as dithering or

by simply using the 4 most significant bits of the alpha data. However they are produced, they are used just as they are, without any form of interpolation.

Note

DirectDraw's compression method uses the 4 most significant bits.

The following tables illustrate how the alpha information is laid out in memory, for each 16-bit word.

This is the layout for Word 0:

Bits	Alpha
3:0 (LSB)	[0][0]
7:4	[0][1]
11:8	[0][2]
15:12 (MSB)	[0][3]

This is the layout for Word 1:

Bits	Alpha
3:0 (LSB)	[1][0]
7:4	[1][1]
11:8	[1][2]
15:12 (MSB)	[1][3]

This is the layout for Word 2:

Bits	Alpha
3:0 (LSB)	[2][0]
7:4	[2][1]
11:8	[2][2]
15:12 (MSB)	[2][3]

This is the layout for Word 3:

Bits	Alpha
3:0 (LSB)	[3][0]
7:4	[3][1]
11:8	[3][2]
15:12 (MSB)	[3][3]

Three-Bit Linear Alpha Interpolation

The encoding of transparency for the DXT4 and DXT5 formats is based on a concept similar to the linear encoding used for color. Two 8-bit alpha values and a 4x4

bitmap with three bits per pixel are stored in the first eight bytes of the block. The representative alpha values are used to interpolate intermediate alpha values. Additional information is available in the way the two alpha values are stored. If alpha_0 is greater than alpha_1, then six intermediate alpha values are created by the interpolation. Otherwise, four intermediate alpha values are interpolated between the specified alpha extremes. The two additional implicit alpha values are 0 (fully transparent) and 255 (fully opaque).

The following pseudo-code illustrates this algorithm:

```
// 8-alpha or 6-alpha block?
if (alpha_0 > alpha_1) {
    // 8-alpha block:  derive the other 6 alphas.
    // 000 = alpha_0, 001 = alpha_1, others are interpolated
    alpha_2 = (6 * alpha_0 + alpha_1) / 7;     // bit code 010
    alpha_3 = (5 * alpha_0 + 2 * alpha_1) / 7;  // Bit code 011
    alpha_4 = (4 * alpha_0 + 3 * alpha_1) / 7;  // Bit code 100
    alpha_5 = (3 * alpha_0 + 4 * alpha_1) / 7;  // Bit code 101
    alpha_6 = (2 * alpha_0 + 5 * alpha_1) / 7;  // Bit code 110
    alpha_7 = (alpha_0 + 6 * alpha_1) / 7;     // Bit code 111
  }
else {  // 6-alpha block:  derive the other alphas.
    // 000 = alpha_0, 001 = alpha_1, others are interpolated
    alpha_2 = (4 * alpha_0 + alpha_1) / 5;     // Bit code 010
    alpha_3 = (3 * alpha_0 + 2 * alpha_1) / 5;  // Bit code 011
    alpha_4 = (2 * alpha_0 + 3 * alpha_1) / 5;  // Bit code 100
    alpha_5 = (alpha_0 + 4 * alpha_1) / 5;     // Bit code 101
    alpha_6 = 0;                         // Bit code 110
    alpha_7 = 255;                         // Bit code 111
}
```

The memory layout of the alpha block is as follows:

Byte	**Alpha**
0	Alpha_0
1	Alpha_1
2	[0][2] (2 LSBs), [0][1], [0][0]
3	[1][1] (1 LSB), [1][0], [0][3], [0][2] (1 MSB)
4	[1][3], [1][2], [1][1] (2 MSBs)
5	[2][2] (2 LSBs), [2][1], [2][0]
6	[3][1] (1 LSB), [3][0], [2][3], [2][2] (1 MSB)
7	[3][3], [3][2], [3][1] (2 MSBs)

Private Surface Data

[This is preliminary documentation and subject to change.]

You can store any kind of application-specific data with a surface. For example, a surface representing a map in a game might contain information about terrain.

A surface can have more than one private data buffer. Each buffer is identified by a GUID which you supply when attaching the data to the surface.

To store private surface data, you use the **IDirectDrawSurface4::SetPrivateData** method, passing in a pointer to the source buffer, the size of the data, and an application-defined GUID for the data. Optionally, the source data can exist in the form of a COM object; in this case, you pass a pointer to the object's **IUnknown** interface pointer and you set the DDSPD_IUNKNOWNPOINTER flag. Another flag, DDSPD_VOLATILE, indicates that the data being attached to the surface is valid only as long as the contents of the surface do not change. (See Surface Uniqueness Values.)

SetPrivateData allocates an internal buffer for the data and copies it. You can then safely free the source buffer or object. The internal buffer or interface reference is released when **IDirectDrawSurface4::FreePrivateData** is called. This happens automatically when the surface is freed.

To retrieve private data for a surface, you must allocate a buffer of the correct size and then call the **IDirectDrawSurface4::GetPrivateData** method, passing the GUID that was assigned to the data by **SetPrivateData**. You are responsible for freeing any dynamic memory you use for this buffer. If the data is a COM object, this method retrieves the **IUnknown** pointer.

If you don't know how big a buffer to allocate, first call **GetPrivateData** with zero in **lpcbBufferSize*. If the method fails with DDERR_MOREDATA, it returns the necessary number of bytes in **lpcbBufferSize*.

Surface Uniqueness Values

[This is preliminary documentation and subject to change.]

The uniqueness value of a surface allows you to determine whether the surface has changed. When DirectDraw creates a surface, it assigns a uniqueness value, which you can retrieve and store by using the **IDirectDrawSurface4::GetUniquenessValue** method. Then, whenever you need to determine whether the surface has changed, you call the method again and compare the new value against the old one. If it's different, the surface has changed.

The actual value returned by **GetUniquenessValue** is irrelevant, unless it is 0. DirectDraw assigns this value to a surface when it knows that the surface might be changed by some process beyond its control. When **GetUniquenessValue** returns 0, you know only that the state of the surface is indeterminate.

To force the uniqueness value for a surface to change, an application can use the **IDirectDrawSurface4::ChangeUniquenessValue** method. This method could be called, for example, by an application or component that changed the private data for a surface without changing the surface itself, and wished to notify some other process of the change. Most applications, however, never need to change the uniqueness value.

Using Non-local Video Memory Surfaces

[This is preliminary documentation and subject to change.]

DirectDraw supports the Accelerated Graphics Port (AGP) architecture for creating surfaces in non-local video memory. On AGP-equipped systems, DirectDraw will use non-local video memory if local video memory is exhausted or if non-local video memory is explicitly requested, depending on the type of AGP implementation that is in place.

Currently, there are two implementations of the AGP architecture, known as the "execute model" and the "DMA model." In the execute model implementation, the display device supports the same features for non-local video memory surfaces and local video memory surfaces. As a result, when you retrieve hardware capabilities by calling the **IDirectDraw4::GetCaps** method, the blit-related flags in the **dwNLVBCaps**, **dwNLVBCaps2**, **dwNLVBCKeyCaps**, **dwNLVBFXCaps**, and **dwNLVBRops** members of the **DDCAPS** structure will be identical to those for local video memory. Under the execute model, if local video memory is exhausted, DirectDraw will automatically fall back on non-local video memory unless the caller specifically requests otherwise.

In the DMA model implementation, support for blitting and texturing from non-local video memory surfaces is limited. When the display device uses the DMA model, the DDCAPS2_NONLOCALVIDMEMCAPS flag will be set in the **dwCaps2** member when you retrieve device capabilities. In the DMA model, the blit-related flags included in the **dwNLVBCaps**, **dwNLVBCaps2**, **dwNLVBCKeyCaps**, **dwNLVBFXCaps**, and **dwNLVBRops** members of the **DDCAPS** structure describe the features that are supported; these features will often be a smaller subset of those supported for local video memory surfaces. Under the DMA model, when local video memory is exhausted, DirectDraw will automatically fall back on non-local video memory for texture surfaces only, unless the caller had explicitly requested local video memory. Texture surfaces are the only types of surfaces that will be treated this way; all other types of surfaces cannot be created in non-local video memory unless the caller explicitly requests it.

DMA model implementations vary in support for texturing from non-local video memory surfaces. If the driver supports texturing from non-local video memory surfaces, the D3DDEVCAPS_TEXTURENONLOCALVIDMEM flag will be set when you retrieve the 3-D device's capabilities by calling the **IDirect3DDevice3::GetCaps** method.

Converting Color and Format

[This is preliminary documentation and subject to change.]

Non-RGB surface formats are described by four-character codes (FOURCC). If an application calls the **IDirectDrawSurface4::GetPixelFormat** method to request the pixel format, and the surface is a non-RGB surface, the DDPF_FOURCC flag will be set and the **dwFourCC** member of the **DDPIXELFORMAT** structure will be valid. If the FOURCC code represents a YUV format, the DDPF_YUV flag will also be set and the **dwYUVBitCoun**t, **dwYBitMask**, **dwUBitMask**, **dwVBitMask**, and **dwYUVAlphaBitMask** members will be valid masks that can be used to extract information from the pixels.

If an RGB format is present, the DDPF_RGB flag will be set and the **dwRGBBitCount**, **dwRBitMask**, **dwGBitMask**, **dwBBitMask**, and **dwRGBAlphaBitMask** members will be valid masks that can be used to extract information from the pixels. The DDPF_RGB flag can be set in conjunction with the DDPF_FOURCC flag if a nonstandard RGB format is being described.

During color and format conversion, two sets of FOURCC codes are exposed to the application. One set of FOURCC codes represents the capabilities of the blitting hardware; the other represents the capabilities of the overlay hardware.

For more information, see Four Character Codes (FOURCC).

Surfaces and Device Contexts

[This is preliminary documentation and subject to change.]

It is often convenient to mix-and-match DirectDraw and GDI services to manipulate the contents of DirectDraw surfaces. DirectDraw offers methods to enable GDI to access DirectDraw surfaces through device contexts, and to retrieve a surface given the surface's device context. This section contains the follows topics that describe these features in detail:

- Retrieving the Device Context for a Surface
- Finding a Surface with a Device Context

Retrieving the Device Context for a Surface

[This is preliminary documentation and subject to change.]

If you want to modify the contents of a DirectDraw surface object by using GDI functions, you must retrieve a GDI-compatible device context handle. This could be useful if you wanted to display text in a DirectDraw surface by calling the **DrawText** Win32 function, which accepts a handle to a device context as a parameter. It is possible to retrieve a GDI-compatible device context for a surface by calling the **IDirectDrawSurface4::GetDC** method for that surface. The following example shows how this might be done:

```
// For this example the lpDDS4 variable is a valid pointer
// to an IDirectDrawSurface4 interface.

   HDC     hdc;
   HRESULT HR;

   hr = lpDDS4->GetDC(&hdc);
   if(FAILED(hr))
      return hr;

   // Call DrawText, or some other GDI
   // function here.

   lpDDS4->ReleaseDC(hdc);
```

Note that the code calls the **IDirectDrawSurface4::ReleaseDC** method when the surface's device context is no longer needed. This step is required, because the **IDirectDrawSurface4::GetDC** method uses an internal version of the **IDirectDrawSurface4::Lock** method to lock the surface. The surface remains locked until the **IDirectDrawSurface4::ReleaseDC** method is called.

Finding a Surface with a Device Context

[This is preliminary documentation and subject to change.]

You can retrieve a pointer to a surface's **IDirectDrawSurface4** interface from the device context for the surface by calling the **IDirectDraw4::GetSurfaceFromDC** method. This feature might be very useful for component applications or ActiveX® controls, that are commonly given a device context to draw into at run-time, but could benefit by exploiting the features exposed by the **IDirectDrawSurface4** interface.

A device context might identify memory that isn't associated with a DirectDraw object, or the device context might identify a surface for another DirectDraw object entirely. The latter case is most likely to occur on a system with multiple monitors. If the device context doesn't identify a surface that wasn't created by that DirectDraw object, the method fails, returning DDERR_NOTFOUND.

The following sample code shows what a very simple scenario might look like:

```
   // For this example, the hdc variable is a valid
   // handle to a video memory device context, and the
   // lpDD4 variable is a valid IDirectDraw4 interface pointer.

   LPDIRECTDRAWSURFACE4 lpDDS4;
   HRESULT hr;

   hr = lpDD4->GetSurfaceFromDC(hdc, &lpDDS4);
   if(SUCCEEDED(hr)) {
```

```
    // Use the surface interface.
}
else if(DDERR_NOTFOUND == hr) {
    OutputDebugString("HDC not from this DirectDraw surface\n");
}
```

Palettes

[This is preliminary documentation and subject to change.]

This section contains information about DirectDrawPalette objects. The following topics are discussed:

- What Are Palettes?
- Palette Types
- Setting Palettes on Nonprimary Surfaces
- Sharing Palettes
- Palette Animation

What Are Palettes?

[This is preliminary documentation and subject to change.]

Palettized surfaces need palettes to be meaningfully displayed. A palettized surface, also known as a color-indexed surface, is simply a collection of numbers where each number represents a pixel. The value of the number is an index into a color table that tells DirectDraw what color to use when displaying that pixel. DirectDrawPalette objects, casually referred to as *palettes*, provide you with an easy way to manage a color table. Surfaces that use a 16-bit or greater pixel format do not use palettes.

A DirectDrawPalette object represents an indexed color table that has 2, 4, 16 or 256 entries to be used with a color indexed surface. Each entry in the palette is an RGB triplet that describes the color to be used when displaying pixels within the surface. The color table can contain 16- or 24-bit RGB triplets representing the colors to be used. For 16-color palettes, the table can also contain indexes to another 256-color palette. Palettes are supported for textures, off-screen surfaces, and overlay surfaces, none of which is required to have the same palette as the primary surface.

You can create a palette by calling the **IDirectDraw4::CreatePalette** method. This method retrieves a pointer to the palette object's **IDirectDrawPalette** interface. You can use the methods of this interface to manipulate palette entries, retrieve information about the object's capabilities, or initialize the object (if you used the **CoCreateInstance** COM function to create it).

You apply a palette to a surface by calling the surface's **IDirectDrawSurface4::SetPalette** method. A single palette can be applied to multiple surfaces.

DirectDrawPalette objects reserve entry 0 and entry 255 for 8-bit palettes, unless you specify the DDPCAPS_ALLOW256 flag to request that these entries be made available to you.

You can retrieve palette entries by using the **IDirectDrawPalette::GetEntries** method, and you can change entries by using the **IDirectDrawPalette::SetEntries** method.

The Ddutil.cpp source file included with the SDK contains some handy application-defined functions for working with palettes. For more information, see the DDLoadPalette functions in that source file.

Palette Types

[This is preliminary documentation and subject to change.]

DirectDraw supports 1-bit (2 entry), 2-bit (4 entry), 4-bit (16 entry), and 8-bit (256 entry) palettes. A palette can only be attached to a surface that has a matching pixel format. For example, a 2-entry palette created with the DDPCAPS_1BIT flag can be attached only to a 1-bit surface created with the DDPF_PALETTEINDEXED1 flag.

Additionally, you can create palettes that don't contain a color table at all, known as *index palettes*. Instead of a color table, an index palette contains index values that represent locations in another palette's color table.

To create an indexed palette, specify the DDPCAPS_8BITENTRIES flag when calling the **IDirectDraw4::CreatePalette** method. For example, to create a 4-bit indexed palette, specify both the DDPCAPS_4BIT and DDPCAPS_8BITENTRIES flags. When you create an indexed palette, you pass a pointer to an array of bytes rather than a pointer to an array of **PALETTEENTRY** structures. You must cast the pointer to the array of bytes to an **LPPALETTEENTRY** type when you use the **IDirectDraw4::CreatePalette** method.

Note that DirectDraw does not dereference index palette entries during blit operations.

Setting Palettes on Nonprimary Surfaces

[This is preliminary documentation and subject to change.]

Palettes can be attached to any palettized surface (primary, back buffer, off-screen plain, or texture map). Only those palettes attached to primary surfaces will have any effect on the system palette. It is important to note that DirectDraw blits never perform color conversion; any palettes attached to the source or destination surface of a blit are ignored.

Nonprimary surface palettes are intended for use by Direct3D applications.

Sharing Palettes

[This is preliminary documentation and subject to change.]

Palettes can be shared among multiple surfaces. The same palette can be set on the front buffer and the back buffer of a flipping chain or shared among multiple texture surfaces. When an application attaches a palette to a surface by using the **IDirectDrawSurface4::SetPalette** method, the surface increments the reference count of that palette. When the reference count of the surface reaches 0, the surface will decrement the reference count of the attached palette. In addition, if a palette is detached from a surface by using **IDirectDrawSurface4::SetPalette** with a NULL palette interface pointer, the reference count of the surface's palette will be decremented.

Note

If **IDirectDrawSurface4::SetPalette** is called several times consecutively on the same surface with the same palette, the reference count for the palette is incremented only once. Subsequent calls do not affect the palette's reference count.

Palette Animation

[This is preliminary documentation and subject to change.]

Palette animation refers to the process of modifying a surface's palette to change how the surface itself looks when displayed. By repeatedly changing the palette, the surface appears to change without actually modifying the contents of the surface. To this end, palette animation gives you a way to modify the appearance of a surface without changing its contents and with very little overhead.

There are two methods for providing straightforward palette animation:

- Modifying palette entries within a single palette
- Switching between multiple palettes

Using the first method, you change individual palette entries that correspond to the colors you want to animate, then reset the entries with a single call to the **IDirectDrawPalette::SetEntries** method.

The second method requires two or more DirectDrawPalette objects. When using this method, you perform the animation by attaching one palette object after another to the surface object by calling the **IDirectDrawSurface4::SetPalette** method.

Neither method is hardware intensive, so use whichever technique works best for your application.

For specific information and an example of how to implement palette animation, see Tutorial 5: Dynamically Modifying Palettes.

Clippers

[This is preliminary documentation and subject to change.]

This section contains information about DirectDrawClipper objects. The following topics are discussed:

- What Are Clippers?
- Clip Lists
- Sharing DirectDrawClipper Objects
- Independent DirectDrawClipper Objects
- Creating DirectDrawClipper Objects with CoCreateInstance
- Using a Clipper with the System Cursor
- Using a Clipper with Multiple Windows

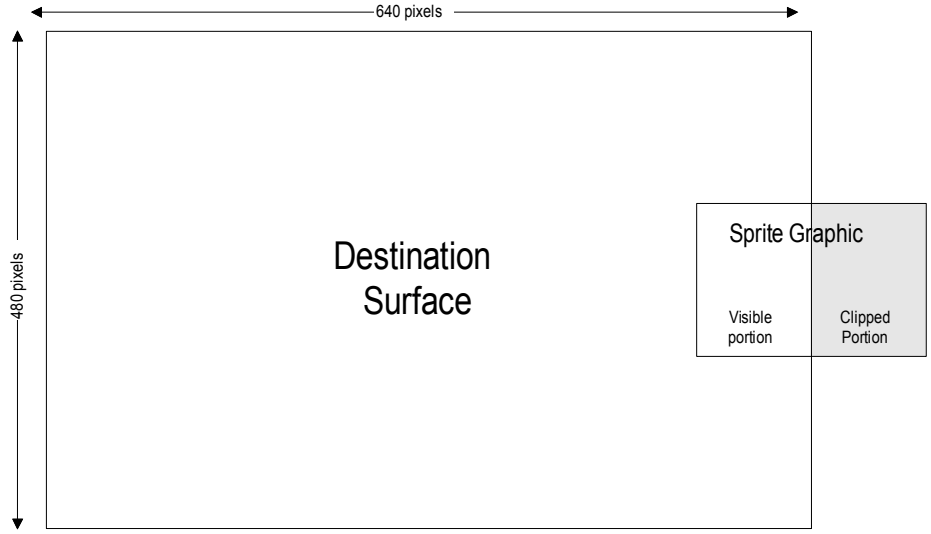
What Are Clippers?

[This is preliminary documentation and subject to change.]

Clippers, or DirectDrawClipper objects, allow you to blit to selected parts of a surface represented by a bounding rectangle or a list of several bounding rectangles. (See Clip Lists.)

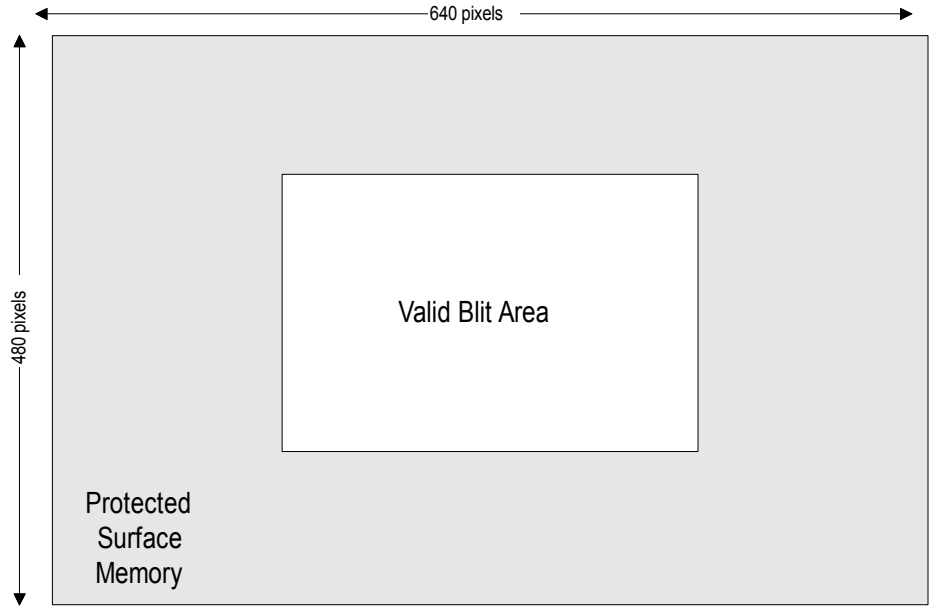
One common use for a clipper is to define the boundaries of the screen or window. For example, imagine that you want to display a sprite as it enters the screen from an edge. You don't want to make the sprite pop suddenly onto the screen; you want it to appear as though it is smoothly moving into view. Without a clipper object, DirectDraw does not allow you to blit the entire sprite, because part of it would fall outside the destination surface. A straight copy of the pixel values in the sprite to the destination surface buffer would result in an incorrect display and even memory access violations. With a clipper that has the screen rectangle as its clip list, DirectDraw knows how to trim the sprite as it performs the blit so that only the visible portion is copied.

The following illustration shows this type of clipping.



You can also use clipper objects to designate certain areas within a destination surface as writable. DirectDraw clips blit operations in these areas, protecting the pixels outside the specified clipping rectangle.

The following illustration shows this use of a clipper.

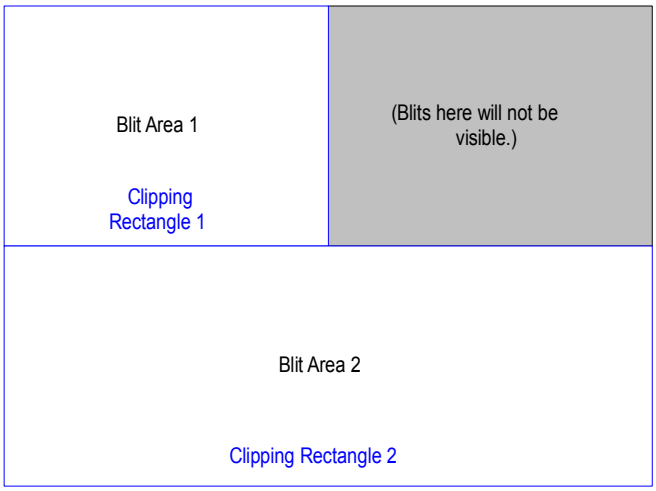


Clip Lists

[This is preliminary documentation and subject to change.]

A clip list consists of one or more **RECT** structures, in pixel coordinates. DirectDraw manages clip lists by using a DirectDrawClipper object, which can be attached to any surface.

The **IDirectDrawSurface4::Blt** method copies data only to rectangles in the clip list. For instance, if the upper-right quarter of a surface was excluded by the rectangles in the clip list, and an application blitted to the entire area of the clipped surface, DirectDraw would effectively perform two blits, the first being to the upper-left corner of the surface, and the second being to the bottom half of the surface, as shown in the following diagram.



You can manage a surface's clip list manually or, for a primary surface, have it done automatically by DirectDraw.

To manage the clip list yourself, create a list of rectangles in the form of a **RGNDATA** structure and pass this to the **IDirectDrawClipper::SetClipList** method.

To have DirectDraw manage the clip list for a primary surface, you attach the clipper to a window (even a full-screen window) by calling the **IDirectDrawClipper::SetHWnd** method, specifying the target window's handle. This has the effect of setting the clipping region to the client area of the window and ensuring that the clip list is automatically updated as the window is resized, covered, or uncovered.

If you set a clipper using a window handle, you cannot set additional rectangles.

Clipping for overlay surfaces is supported only if the overlay hardware can support clipping and if destination color keying is not active.

Sharing DirectDrawClipper Objects

[This is preliminary documentation and subject to change.]

DirectDrawClipper objects can be shared between multiple surfaces. For example, the same DirectDrawClipper object can be set on both the front buffer and the back buffer of a flipping chain. When an application attaches a DirectDrawClipper object to a surface by using the **IDirectDrawSurface4::SetClipper** method, the surface increments the reference count of that object. When the reference count of the surface reaches 0, the surface will decrement the reference count of the attached DirectDrawClipper object. In addition, if a DirectDrawClipper object is detached from a surface by calling **IDirectDrawSurface4::SetClipper** with a NULL clipper interface pointer, the reference count of the surface's DirectDrawClipper object will be decremented.

Note

If **IDirectDrawSurface4::SetClipper** is called several times consecutively on the same surface for the same DirectDrawClipper object, the reference count for the object is incremented only once. Subsequent calls do not affect the object's reference count.

Independent DirectDrawClipper Objects

[This is preliminary documentation and subject to change.]

You can create DirectDrawClipper objects that are not directly owned by any particular DirectDraw object. These DirectDrawClipper objects can be shared across multiple DirectDraw objects. Driver-independent DirectDrawClipper objects are created by using the new **DirectDrawCreateClipper** DirectDraw function. An application can call this function before any DirectDraw objects are created.

Because DirectDraw objects do not own these DirectDrawClipper objects, they are not automatically released when your application's objects are released. If the application does not explicitly release these DirectDrawClipper objects, DirectDraw will release them when the application closes.

You can still create DirectDrawClipper objects by using the **IDirectDraw4::CreateClipper** method. These DirectDrawClipper objects are automatically released when the DirectDraw object from which they were created is released.

Creating DirectDrawClipper Objects with CoCreateInstance

[This is preliminary documentation and subject to change.]

DirectDrawClipper objects have full class-factory support for COM compliance. In addition to using the standard **DirectDrawCreateClipper** function and **IDirectDraw4::CreateClipper** method, you can also create a DirectDrawClipper object either by using the **CoGetClassObject** function to obtain a class factory and

then calling the **CoCreateInstance** function, or by calling **CoCreateInstance** directly. The following example shows how to create a DirectDrawClipper object by using **CoCreateInstance** and the **IDirectDrawClipper::Initialize** method.

```
ddrval = CoCreateInstance(&CLSID_DirectDrawClipper,
   NULL, CLSCTX_ALL, &IID_IDirectDrawClipper, &lpClipper);
if (!FAILED(ddrval))
   ddrval = IDirectDrawClipper_Initialize(lpClipper,
      lpDD, 0UL);
```

In this call to **CoCreateInstance**, the first parameter, *CLSID_DirectDrawClipper*, is the class identifier of the DirectDrawClipper object class, the *IID_IDirectDrawClipper* parameter identifies the currently supported interface, and the *lpClipper* parameter points to the DirectDrawClipper object that is retrieved.

An application must use the **IDirectDrawClipper::Initialize** method to initialize DirectDrawClipper objects that were created by the class-factory mechanism before it can use the object. The value 0UL is the *dwFlags* parameter, which in this case has a value of 0 because no flags are currently supported. In the example shown here, *lpDD* is the DirectDraw object that owns the DirectDrawClipper object. However, you could supply a NULL value instead, which would create an independent DirectDrawClipper object. (This is equivalent to creating a DirectDrawClipper object by using the **DirectDrawCreateClipper** function.)

Before you close the application, close the COM library by using the **CoUninitialize** function.

Using a Clipper with the System Cursor

[This is preliminary documentation and subject to change.]

DirectDraw applications often need to provide a way for users to navigate using the mouse. For full-screen exclusive mode applications that use page-flipping, the only option is to implement a mouse cursor manually with a sprite, moving the sprite based on data retrieved from the device by DirectInput® or by responding to Windows mouse messages. However, any application that doesn't use page-flipping can still use the system's mouse cursor support.

When you use the system mouse cursor, you will sometimes fall victim to graphic artifacts that occur when you blit to parts of the primary surface. These artifacts appear as portions of the mouse cursor seemingly left behind by the system.

A DirectDrawClipper object can prevent these artifacts from appearing by preventing the mouse cursor image from "being in the way" during a blit operation. It's a relatively simple matter to implement, as well. To do so, create a DirectDrawClipper object by calling the **IDirectDraw4::CreateClipper** method. Then, assign your application's window handle to the clipper with the **IDirectDrawClipper::SetHWnd** method. Once a clipper is attached, any subsequent blits you perform on the primary surface with the **IDirectDrawSurface4::Blt** method will not exhibit the artifact.

Note that the **IDirectDrawSurface4::BltFast** method, and its counterparts in the **IDirectDrawSurface**, **IDirectDrawSurface2**, and **IDirectDrawSurface3** interfaces, will not work on surfaces with attached clippers.

Using a Clipper with Multiple Windows

[This is preliminary documentation and subject to change.]

You can use a DirectDrawClipper object to blit to multiple windows created by an application running at the normal cooperative level.

To do this, create a single DirectDraw object with a primary surface. Then, create a DirectDrawClipper object and assign it to your primary surface by calling the **IDirectDrawSurface4::SetClipper** method. To blit only to the client area of a window, set the clipper to that window's client area by calling the **IDirectDrawClipper::SetHWnd** method before blitting to the primary surface. Whenever you need to blit to another window's client area, call the **IDirectDrawClipper::SetHWnd** method again with the new target window handle.

Creating multiple DirectDraw objects that blit to each others' primary surface is not recommended. The technique just described provides an efficient and reliable way to blit to multiple client areas with a single DirectDraw object.

Multiple Monitor Systems

[This is preliminary documentation and subject to change.]

Windows 98 and Windows 2000 support multiple display devices and monitors on a single system. The multiple monitor architecture (sometimes referred to as "MultiMon") enables the operating system to use the display area from two or more display devices and monitors to create a single logical desktop. For example, in a MultiMon system with two monitors, the user could display applications on either monitor, or even drag windows from one monitor to another. DirectDraw supports this architecture, allowing applications to directly access hardware on multiple display devices in a MultiMon system.

Note

As long as it is created on the null device and is not rendering directly to the primary surface, a non-full-screen DirectDraw application will work automatically with MultiMon, and the user will be able to drag the window from one monitor to another. However, DirectDraw will take advantage of hardware acceleration only when the window is entirely within the primary display. It is recommended that windowed DirectDraw applications be specifically designed for MultiMon by maintaining separate DirectDraw objects and surfaces for each monitor. For more information, see Devices and Acceleration in MultiMon Systems.

This section contains information about using DirectDraw on systems with multiple monitor support. The following topics are discussed:

- Enumerating Devices on MultiMon Systems
- DirectDraw Objects on Multiple Monitors
- Focus and Device Windows
- Devices and Acceleration in MultiMon Systems
- Debugging Full-Screen DirectDraw Applications with MultiMon

The Multimon.h header file included with the DirectX Programmer's Reference makes it possible for code written around Windows 98 multiple monitor functions to compile and run successfully on operating systems that do not support MultiMon.

The following sample applications demonstrate the implementation of MultiMon in DirectDraw:

- Stretch2 Sample
- Stretch3 Sample
- Multimonitor Space Donuts Sample

Enumerating Devices on MultiMon Systems

[This is preliminary documentation and subject to change.]

Use the **DirectDrawEnumerateEx** function to enumerate devices on systems with multiple monitors, specifying flags to determine what types of DirectDraw devices should be enumerated. The function calls an application-defined **DDEnumCallbackEx** function for each enumerated device.

The **DirectDrawEnumerateEx** function is supported on Windows 98 and Windows 2000 operating systems. It is available in Ddraw.lib for applications compiled under DirectX 6.0 and later versions. Applications that statically link to the function will always run under DirectX 6.0 and later, and will always run under any version of DirectX on Windows 98 and Windows 2000. Such applications will fail if run on previous versions of DirectX under Windows 95.

If your application needs to run on versions of DirectX older than DirectX 5.0, it should use **GetProcAddress** to see if **DirectDrawEnumerateEx** is available. The following example shows one way you can do this:

```
HINSTANCE h = LoadLibrary("ddraw.dll");

// If ddraw.dll doesn't exist in the search path,
// then DirectX probably isn't installed, so fail.
if (!h)
   return FALSE;
```

```
    // Note that you must know which version of the
    // function to retrieve (see the following text).
    // For this example, we use the ANSI version.
    LPDIRECTDRAWENUMERATEEX lpDDEnumEx;
    lpDDEnumEx = (LPDIRECTDRAWENUMERATEEX)
GetProcAddress(h,"DirectDrawEnumerateExA");

    // If the function is there, call it to enumerate all display
    // devices attached to the desktop, and any non-display DirectDraw
    // devices.
    if (lpDDEnumEx)
        lpDDEnumEx(Callback, NULL,
            DDENUM_ATTACHEDSECONDARYDEVICES |
            DDENUM_NONDISPLAYDEVICES
            );
    else
    {
        /*
         * We must be running on an old version of DirectDraw.
         * Therefore MultiMon isn't supported. Fall back on
         * DirectDrawEnumerate to enumerate standard devices on a
         * single-monitor system.
         */
        DirectDrawEnumerate(OldCallback,NULL);

        /* Note that it could be handy to let the OldCallback function
         * be a wrapper for a DDEnumCallbackEx.
         *
         * Such a function would look like:
         *    BOOL FAR PASCAL OldCallback(GUID FAR *lpGUID,
         *                          LPSTR pDesc,
         *                          LPSTR pName,
         *                          LPVOID pContext)
         *    {
         *        return Callback(lpGUID,pDesc,pName,pContext,NULL);
         *    }
         */
    }

    // If the library was loaded by calling LoadLibrary(),
    // then you must use FreeLibrary() to let go of it.
    FreeLibrary(h);
```

The previous example will work for applications that link to Ddraw.dll at run-time or load-time.

Note that you must retrieve the address of either the ANSI or Unicode version of the **DirectDrawEnumerateEx** function, depending of the type of strings your application uses. When declaring the corresponding callback function, use the **LPTSTR** data type for the string parameters. The **LPTSTR** data type compiles to use Unicode strings if you declare the _UNICODE symbol, and ANSI strings otherwise. By using the **LPTSTR** data type, the function should compile properly regardless of the string type you use in your application.

DirectDraw Objects on Multiple Monitors

[This is preliminary documentation and subject to change.]

Windowed DirectDraw applications written for the null or default display driver will work on MultiMon systems, but in applications optimized for MultiMon you will want to create a separate DirectDraw object for each device, using the GUID returned in the enumeration callback. (See Enumerating Devices on MultiMon Systems.)

Avoid setting the cooperative level multiple times on a MultiMon system. If you need to switch from full-screen to normal mode, it is best to create a new DirectDraw object.

It is good practice to release all DirectDraw objects at the same time. If you release only the secondary device or devices, the primary device goes back to its original desktop mode, but only the taskbar is redrawn and the DirectDraw primary surface is still present. You cannot draw to this surface without first releasing the DirectDraw object and then re-creating it.

Focus and Device Windows

[This is preliminary documentation and subject to change.]

Each DirectDraw application that uses one or more monitors in full-screen exclusive mode must have a single focus window, which is the window that receives keyboard input.

Each device that is to hold a full-screen DirectDraw surface must be represented by a DirectDraw object and a device window. The device window is the one that is sized to fit the window and is put on top of all other windows.

For single-monitor applications, there is no distinction between the device and focus window. They are one and the same.For multiple-monitor applications, however, you need to set a device window for each monitor, and you have to let each DirectDraw object know about the application's focus window. The focus window can also serve as the device window for one of the monitors. Other device windows should be children of the focus window so that the application does not minimize when the user clicks on one of them.

See also:

- Setting the Focus Window
- Setting Device Windows

Setting the Focus Window

[This is preliminary documentation and subject to change.]

To set the focus window, you call the **IDirectDraw4::SetCooperativeLevel** method for each of the DirectDraw objects. You pass in a window handle (normally the application window handle) and set the DDSCL_SETFOCUSWINDOW flag, as in the following example:

```
/* It is presumed that lpDD is a valid IDirectDraw interface pointer,
   and that hWnd is a valid window handle. */

HRESULT ddrval = lpDD->SetCooperativeLevel( hWnd,
      DDSCL_SETFOCUSWINDOW );
```

The focus window must be the same for all devices.

Setting Device Windows

[This is preliminary documentation and subject to change.]

There are two ways to set a device window:

- Create a window yourself and pass its handle to the **IDirectDraw4::SetCooperativeLevel** method of the DirectDraw object representing the monitor, setting the DDSCL_SETDEVICEWINDOW, DDSCL_FULLSCREEN, and DDSCL_EXCLUSIVE flags. This creates a full-screen window and sets it as the device window for the monitor. Your application will receive mouse messages for the window, and you are responsible for destroying the window at the appropriate time. The window you pass to **SetCooperativeLevel** should be either the focus window (possible only if it is on the same device) or a child of the focus window.
- Let DirectDraw create the window. You pass the focus window handle to **SetCooperativeLevel** and set the DDSCL_CREATEDEVICEWINDOW, DDSCL_FULLSCREEN, and DDSCL_EXCLUSIVE flags. DirectDraw creates a default device window that is a child of the focus window. It manages this window and will destroy it at the appropriate time. Your application will not receive any mouse messages for the window.

The following example sets an existing device window for the DirectDraw object represented by *lpDD*.

```
/* It is presumed that lpDD is a valid IDirectDraw interface pointer,
   and that hWnd is the handle to an appropriate device window. */
```

```
HRESULT hr = lpDD->SetCooperativeLevel( hWnd,
    DDSCL_SETDEVICEWINDOW | DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
```

The following example sets a default device window created by DirectDraw. In this case, *hWnd* is the handle to the existing focus window.

```
HRESULT hr = lpDD->SetCooperativeLevel( hWnd,
    DDSCL_CREATEDEVICEWINDOW | DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
```

Although a focus window can be a device window, you cannot set a window as both the focus window and a device window with a single call to **SetCooperativeLevel**. You must first set it as the focus window and then set it as a device window. However, it is possible to set a focus window and a default device window on the same device with a single call to **SetCooperativeLevel**. The following example shows how this can be done:

```
HRESULT hr = lpDD->SetCooperativeLevel(
    hwndFocus,
    DDSCL_SETFOCUSWINDOW | DDSCL_FULLSCREEN |
    DDSCL_EXCLUSIVE | DDSCL_CREATEDEVICEWINDOW);
```

In this example, an existing window (probably the application window) is set as the focus window, and DirectDraw creates a default device window.

Devices and Acceleration in MultiMon Systems

[This is preliminary documentation and subject to change.]

Full-screen exclusive mode DirectDraw objects will take advantage of hardware acceleration regardless of whether they are running on the primary device or on a secondary device. However, they cannot use built-in support for spanning graphics operations across display devices. It is the application's responsibility to perform operations on the appropriate device.

When the normal cooperative level is set, DirectDraw uses hardware acceleration only when the window is wholly within the display area of the primary device. When a window straddles two or more monitors, all blits are done in emulation and performance can be significantly slower. This is necessarily the case, because hardware buffers cannot blit to a display surface controlled by another piece of hardware.

As long as you create the DirectDraw object for the null device—that is, pass NULL to **DirectDrawCreate** as the *lpGUID* parameter—DirectDraw will blit to the entire window regardless of where it is located. However, if the device is created by its actual GUID, this is not the case, and blit operations that cross an edge of the primary surface will be clipped (if you are using a clipper) or will fail, returning DDERR_INVALIDRECT.

Note

When you are blitting to a window in a MultiMon application, negative coordinates are valid when the logical location of the secondary monitor is to the left of the primary monitor.

To get the best performance in a windowed MultiMon application, you need to create a DirectDraw object for each device, maintain off-screen surfaces in parallel on each device, keep track of which part of the window resides on each device, and perform separate blits to each device.

Debugging Full-Screen DirectDraw Applications with MultiMon

[This is preliminary documentation and subject to change.]

It is possible to use a multimonitor system rather than remote debugging in order to step through code while debugging a full-screen DirectDraw application.

You should use the primary monitor for your development environment and the secondary monitor for the DirectDraw output. Also, you need to change a registry setting through the DirectX property sheet in Control Panel. On the **DirectDraw** page, click **Advanced Settings** and select the **Enable Multi-Monitor Debugging** checkbox. This setting will prevent DirectDraw from minimizing your application when it loses focus.

Under Windows 98, you cannot step through code when a surface is locked. For more information, see Accessing Surface Memory Directly.

Advanced DirectDraw Topics

[This is preliminary documentation and subject to change.]

This section supplements the DirectDraw overview, providing information about advanced DirectDraw issues. The following topics are discussed:

- Mode 13 Support
- Taking Advantage of DMA Support
- Using DirectDraw Palettes in Windowed Mode
- Video Ports
- Getting the Flip and Blit Status
- Determining the Capabilities of the Display Hardware
- Storing Bitmaps in Display Memory
- Triple Buffering
- DirectDraw Applications and Window Styles
- Matching True RGB Colors to the Frame Buffer's Color Space
- Displaying a Window in Full-Screen Mode

Mode 13 Support

[This is preliminary documentation and subject to change.]

This section contains information about DirectDraw Mode 13 graphics mode support. The following topics are discussed:

- About Mode 13
- Setting Mode 13
- Mode 13 and Surface Capabilities
- Using Mode 13

About Mode 13

[This is preliminary documentation and subject to change.]

DirectDraw supports access to the linear unflippable 320x200 8 bits per pixel palettized mode known widely by the name Mode 13, its hexadecimal BIOS mode number. DirectDraw treats this mode like a Mode X mode, but with some important differences imposed by the physical nature of Mode 13.

Setting Mode 13

[This is preliminary documentation and subject to change.]

Mode 13 has similar enumeration and mode-setting behavior as Mode X. DirectDraw will only enumerate Mode 13 if the DDSCL_ALLOWMODEX flag was passed to the **IDirectDraw4::SetCooperativeLevel** method.

You enumerate the Mode 13 display mode like all other modes, but you make a surface capabilities check before calling **IDirectDraw4::EnumDisplayModes**. To do this, call **IDirectDraw4::GetCaps** and check for the DDSCAPS_STANDARDVGAMODE flag in the **DDSCAPS2** structure after the method returns. If this flag is not present, then Mode 13 is not supported, and attempts to enumerate with the DDEDM_STANDARDVGAMODES flag will fail, returning DDERR_INVALIDPARAMS.

The **EnumDisplayModes** method now supports a new enumeration flag, DDEDM_STANDARDVGAMODES, which causes DirectDraw to enumerate Mode 13 in addition to the 320x200x8 Mode X mode. There is also a new **IDirectDraw4::SetDisplayMode** flag, DDSDM_STANDARDVGAMODE, which you must pass in order to distinguish Mode 13 from 320x200x8 Mode X.

Note that some video cards offer linear accelerated 320x200x8 modes. On such cards DirectDraw will not enumerate Mode 13, enumerating the linear mode instead. In this case, if you attempt to set Mode 13 by passing the DDSDM_STANDARDVGAMODE flag to **SetDisplayMode**, the method will succeed, but the linear mode will be used. This is analogous to the way that linear low resolution modes override Mode X modes.

Mode 13 and Surface Capabilities

[This is preliminary documentation and subject to change.]

When DirectDraw calls an application's **EnumModesCallback** callback function, the **ddsCaps** member of the associated **DDSURFACEDESC** or **DDSURFACEDESC2** structure contains flags that reflect the mode being enumerated. You can expect DDSCAPS_MODEX for a Mode X mode or DDSCAPS_STANDARDVGAMODE for Mode 13. These flags are mutually exclusive. If neither of these bits is set, then the mode is a linear accelerated mode. This behavior also applies to the flags retrieved by the **IDirectDraw4::GetDisplayMode** method.

Using Mode 13

[This is preliminary documentation and subject to change.]

Because Mode 13 is a linear mode, unlike the Mode X modes, DirectDraw can give an application direct access to the frame buffer.You can call the **IDirectDrawSurface4::Lock**, **IDirectDrawSurface4::Blt**, and **IDirectDrawSurface4::BltFast** methods to gain direct access to the primary surface.

When using Mode 13, DirectDraw supports an emulated **IDirectDrawSurface4::Flip** that is implemented as a straight copy of the contents of a back buffer to the primary surface. You can emulate this yourself by copying all or part of the back-buffer's contents to the primary surface using **Blt** or **BltFast**.

There is one warning concerning **Lock** and Mode 13. Although DirectDraw allows direct linear access to the Mode 13 VGA frame buffer, do not assume that the buffer is always located at address 0xA0000, since DirectDraw can return an aliased virtual-memory pointer to the frame buffer which will not be 0xA0000. Similarly, do not assume that the pitch of a Mode 13 surface is 320, because display cards that support an accelerated 320x200x8 mode will very likely use a different pitch.

Taking Advantage of DMA Support

[This is preliminary documentation and subject to change.]

This section contains information about how you can take advantage of device support for Direct Memory Access (DMA) to increase performance in completing certain tasks. The following topics are discussed:

- About DMA Device Support
- Testing for DMA Support
- Typical Scenarios for DMA
- Using DMA

About DMA Device Support

[This is preliminary documentation and subject to change.]

Some display devices can perform blit operations (or other operations) on system memory surfaces. These operations are commonly referred to as Direct Memory Access (DMA) operations. You can exploit DMA support to accelerate certain combinations of operations. For example, on such a device, you could perform a blit from system memory to video memory while using the processor to prepare the next frame. In order to use such facilities, you must assume certain responsibilities. This section details these tasks.

Testing for DMA Support

[This is preliminary documentation and subject to change.]

Before using DMA operations, you must test the device for DMA support and, if it does support DMA, how much support it provides. Begin by retrieving the driver capabilities by calling the **IDirectDraw4::GetCaps** method, then look for the DDCAPS_CANBLTSYSMEM flag in the **dwCaps** member of the associated **DDCAPS** structure. If the flag is set, the device supports DMA.

If you know that DMA is generally supported, you also need to find out how well the driver supports it. You do so by looking at some other structure members that provide information about system-to-video, video-to-system, and system-to-system blit operations. These capabilities are provided in 12 **DDCAPS** structure members that are named according to blit and capability type. The following table shows these new members.

System-to-video	Video-to-system	System-to-system
dwSVBCaps	**dwVSBCaps**	**dwSSBCaps**
dwSVBCKeyCaps	**dwVSBCKeyCaps**	**dwSSBCKeyCaps**
dwSVBFXCaps	**dwVSBFXCaps**	**dwSSBFXCaps**
dwSVBRops	**dwVSBRops**	**dwSSBRops**

For example, the system-to-video blit capability flags are provided in the **dwSVBCaps**, **dwSVBCKeyCaps**, **dwSVBFXCaps** and **dwSVBRops** members. Similarly, video-to-system blit capabilities are in the members whose names begin with "**dwVSB**," and system-to system capabilities are in the "**dwSSB**" members. Examine the flags present in these members to determine the level of hardware support for that blit category.

The flags in these members are parallel with the blit-related flags included in the **dwCaps**, **dwCKeyCaps**, and **dwFXCaps** members, with respect to that member's blit type. For example, the **dwSVBCaps** member contains general blit capabilities as specified by the same flags you might find in the **dwCaps** member. Likewise, the raster operation values in the **dwSVBRops**, **dwVSBRops**, and **dwSSBRops** members provide information about the raster operations supported for a given type of blit operation.

One of the key features to look for in these members is support for asynchronous DMA blit operations. If the driver supports asynchronous DMA blits between surfaces, the DDCAPS_BLTQUEUE flag will be set in the **dwSVBCaps**, **dwVSBCaps**, or **dwSSBCaps** member. (Generally, you'll see the best support for system-memory-to-video-memory surfaces.) If the flag isn't present, the driver isn't reporting support for asynchronous DMA blit operations.

Typical Scenarios for DMA

[This is preliminary documentation and subject to change.]

System memory to video memory transfers that use the SRCCOPY raster operation are the most common type of hardware-supported blit operation. (The SRCCOPY raster operation, which is documented in the Platform SDK, causes the data within the source rectangle to be copied directly to the destination rectangle.) The most typical use for such an operation is to move textures from a large collection of system memory surfaces to a surface in video memory in preparation for subsequent operations. System-to-video DMA transfers are about as fast as processor-controlled transfers (for example, HEL blits), but are of great utility since they can operate in parallel with the host processor.

Using DMA

[This is preliminary documentation and subject to change.]

Hardware transfers use physical memory addresses, not the virtual addresses which are home to applications. Some device drivers require that you provide the surface's physical memory address. This mechanism is implemented by the **IDirectDrawSurface4::PageLock** method. If the device driver does not require page locking, the DDCAPS2_NOPAGELOCKREQUIRED flag will be set when you retrieve the hardware capabilities by calling the **IDirectDraw4::GetCaps** method.

Page locking a surface prevents the system from committing a surface's physical memory to other uses, and guarantees that the surface's physical address will remain constant until a corresponding **IDirectDrawSurface4::PageUnlock** call is made. If the device driver requires page locking, DirectDraw will allow asynchronous DMA operations only on system memory surfaces that the application has page locked. If you do not call **IDirectDrawSurface4::PageLock** in such a situation, DirectDraw will perform the transfers by using software emulation. Note that locking a large amount of system memory will make Windows run poorly. Therefore, it is highly recommended that only full-screen exclusive mode applications use **IDirectDrawSurface4::PageLock** for large amounts of system memory, and that such applications take care to unlock these surfaces when the application is minimized. Of course, when the application is restored, you should page lock the system memory surface again.

Responsibility for managing page locking is entirely in the hands of the application developer. DirectDraw will never page lock or page unlock a surface. Additionally, it is up to you to determine how much memory you can safely page lock without adversely affecting system performance.

Using DirectDraw Palettes in Windowed Mode

[This is preliminary documentation and subject to change.]

IDirectDrawPalette interface methods write directly to the hardware when the display is in exclusive (full-screen) mode. However, when the display is in nonexclusive (windowed) mode, the **IDirectDrawPalette** interface methods call the GDIs palette handling functions to work cooperatively with other windowed applications.

The discussion in the following topics assumes that the desktop is in an 8-bit palettized mode and that you have created a primary surface and a typical window.

- Types of Palette Entries in Windowed Mode
- Creating a Palette in Windowed Mode
- Setting Palette Entries in Windowed Mode

Types of Palette Entries in Windowed Mode

[This is preliminary documentation and subject to change.]

Unlike full-screen exclusive mode applications, windowed applications must share the desktop palette with other applications. This imposes several restrictions on which palette entries you can safely modify and how you can modify them. The **PALETTEENTRY** structure you use when working with DirectDrawPalette objects and GDI contains a **peFlags** member to carry information that describes how the system should interpret the **PALETTEENTRY** structure.

The **peFlags** member describes three types of palette entries, discussed in this topic:

- Windows static entries
- Animated entries
- Nonanimated entries

Windows static entries

In normal mode, Windows reserves palette entries 0 through 9 and 246 through 255 for system colors that it uses to display menu bars, menu text, window borders, and so on. In order to maintain a consistent look for your application and avoid damaging the appearance of other applications, you need to protect these entries in the palette you set to the primary surface. Often, developers retrieve the system palette entries by calling the **GetSystemPaletteEntries** Win32® function, then explicitly set the identical entries in a custom palette to match before assigning it to the primary surface. Duplicating the system palette entries in a custom palette will work initially, but it becomes invalid if the user changes the desktop color scheme.

To avoid having your palette look bad when the user changes color schemes, you can protect the appropriate entries by providing a reference into the system palette

instead specifying a color value. This way, no matter what color the system is using for a given entry, your palette will always match and you won't need to do any updating. The PC_EXPLICIT flag, used in the **peFlags** member, makes it possible for you to directly refer to a system palette entry. When you use this flag, the system no longer assumes that the other structure members include color information. Rather, when you use PC_EXPLICIT, you set the value in the **peRed** member to the desired system palette index and set the other colors to zero.

For instance, if you want to ensure that the proper entries in your palette always match the system's color scheme, you could use the following code:

```
// Set the first and last 10 entries to match the system palette.
PALETTEENTRY pe[256];
ZeroMemory(pe, sizeof(pe));
for(int i=0;i<10;i++){
   pe[i].peFlags  = pe[i+246].peFlags = PC_EXPLICIT;
   pe[i].peRed = i;
   pe[i+246].peRed = i+246;
}
```

You can force Windows to use only the first and last palette entry (0 and 255) by calling the **SetSystemPaletteUse** Win32 function. In this case, you should set only entries 0 and 255 of your **PALETTEENTRY** structure to PC_EXPLICIT.

Animated entries

You specify palette entries that you will be animating by using the PC_RESERVED flag in the corresponding **PALETTEENTRY** structure. Windows will not allow any other application to map its logical palette entry to that physical entry, thereby preventing other applications from cycling their colors when your application animates the palette.

Nonanimated entries

You specify normal, nonanimated palette entries by using the PC_NOCOLLAPSE flag in the corresponding **PALETTEENTRY** structure. The PC_NOCOLLAPSE flag informs Windows not to substitute some other already-allocated physical palette entry for that entry.

Creating a Palette in Windowed Mode

[This is preliminary documentation and subject to change.]

The following example illustrates how to create a DirectDraw palette in nonexclusive (windowed) mode. In order for your palette to work correctly, it is vital that you set up every one of the 256 entries in the **PALETTEENTRY** structure that you submit to the **IDirectDraw4::CreatePalette** method.

```
LPDIRECTDRAW4       lpDD; // Assumed to be initialized previously
PALETTEENTRY        pPaletteEntry[256];
int                 index;
```

```
HRESULT             ddrval;
LPDIRECTDRAWPALETTE2 lpDDPal;

// First set up the Windows static entries.
for (index = 0; index < 10 ; index++)
{
   // The first 10 static entries:
   pPaletteEntry[index].peFlags = PC_EXPLICIT;
   pPaletteEntry[index].peRed = index;
   pPaletteEntry[index].peGreen = 0;
   pPaletteEntry[index].peBlue = 0;

   // The last 10 static entries:
   pPaletteEntry[index+246].peFlags = PC_EXPLICIT;
   pPaletteEntry[index+246].peRed = index+246;
   pPaletteEntry[index+246].peGreen = 0;
   pPaletteEntry[index+246].peBlue = 0;
}

// Now set up private entries. In this example, the first 16
// available entries are animated.
for (index = 10; index < 26; index ++)
{
   pPaletteEntry[index].peFlags = PC_NOCOLLAPSE|PC_RESERVED;
   pPaletteEntry[index].peRed = 255;
   pPaletteEntry[index].peGreen = 64;
   pPaletteEntry[index].peBlue = 32;
}

// Now set up the rest, the nonanimated entries.
for (; index < 246; index ++) // Index is set up by previous for loop
{
   pPaletteEntry[index].peFlags = PC_NOCOLLAPSE;
   pPaletteEntry[index].peRed = 25;
   pPaletteEntry[index].peGreen = 6;
   pPaletteEntry[index].peBlue = 63;
}

// All 256 entries are filled. Create the palette.
ddrval = lpDD->CreatePalette(DDPCAPS_8BIT, pPaletteEntry,
   &lpDDPal,NULL);
```

Setting Palette Entries in Windowed Mode

[This is preliminary documentation and subject to change.]

The rules that apply to the **PALETTEENTRY** structure used with the **IDirectDraw4::CreatePalette** method also apply to the **IDirectDrawPalette::SetEntries** method. Typically, you maintain your own array of **PALETTEENTRY** structures, so you do not need to rebuild it. When necessary, you can modify the array, and then call **IDirectDrawPalette::SetEntries** when it is time to update the palette.

In most circumstances, you should not attempt to set any of the Windows static entries when in nonexclusive (windowed) mode or you will get unpredictable results. The only exception is when you reset the 256 entries.

For palette animation, you typically change only a small subset of entries in your **PALETTEENTRY** array. You submit only those entries to **IDirectDrawPalette::SetEntries**. If you are resetting such a small subset, you must reset only those entries marked with the PC_NOCOLLAPSE and PC_RESERVED flags. Attempting to animate other entries can have unpredictable results.

The following example illustrates palette animation in nonexclusive mode:

```
LPDIRECTDRAW       lpDD;       // Already initialized
PALETTEENTRY pPaletteEntry[256]; // Already initialized
LPDIRECTDRAWPALETTE lpDDPal;    // Already initialized
int                index;
HRESULT            ddrval;
PALETTEENTRY       temp;

// Animate some entries. Cycle the first 16 available entries.
// They were already animated.
temp = pPaletteEntry[10];
for (index = 10; index < 25; index ++)
{
   pPaletteEntry[index] = pPaletteEntry[index+1];
}
pPaletteEntry[25] = temp;

// Set the values. Do not pass a pointer to the entire palette entry
// structure, but only to the changed entries.
ddrval = lpDDPal->SetEntries(
   0,                  // Flags must be zero
   10,                  // First entry
   16,                  // Number of entries
   & (pPaletteEntry[10])); // Where to get the data
```

Video Ports

[This is preliminary documentation and subject to change.]

DirectDraw video-port extensions are a low-level programming interface, not intended for mainstream multimedia programmers. The target customer is the video-streaming software industry, which creates products like DirectShow™. Developers who want to include video playback in their software can make use of video-port extensions. However, for most software, a high-level programming interface like the one provided by DirectShow is recommended for greater ease of use.

This section contains information about DirectDrawVideoPort objects. The following topics are discussed:

- What Are Video Ports?
- Video-Port Technology Overview
- About DirectDraw Video-Port Extensions
- Video Frames and Fields
- HREF, VREF, and Connections
- Vertical Blanking Interval Data
- Auto-Flipping
- Solutions to Common Video Artifacts
- Solving Problems Caused by Half-Lines
- Exploiting Hardware Features

What Are Video Ports?

[This is preliminary documentation and subject to change.]

A DirectDrawVideoPort object represents the video-port hardware found on some display adapters. Generally, a video-port object controls how the video-port hardware applies a video signal it receives from a video decoder directly to the frame buffer.

More than one channel of video can be controlled by creating as many DirectDrawVideoPort objects as is required. Because each channel can be separately enumerated and configured, the video hardware for each channel does not need to be identical.

For more information, see Video-Port Technology Overview.

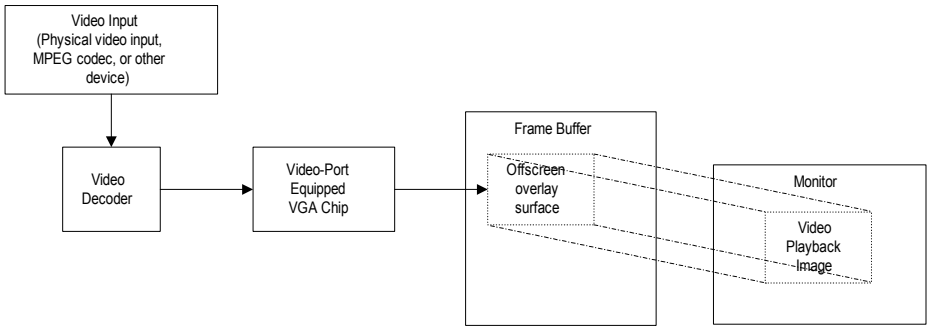
Video-Port Technology Overview

[This is preliminary documentation and subject to change.]

A *video port* is hardware on a display device that enables direct access to a surface within the frame buffer, bypassing the CPU and PCI bus. Direct frame buffer access makes it possible to efficiently play live or recorded video without creating noticeable load on the CPU. Once in a surface, an image can be displayed on the screen as an overlay, used as a Direct3D texture, or accessed by the CPU for capture or other processing. The following paragraphs provide general information about the components that make up the technology and how they work.

Data Flow

In a machine equipped with a video port, data in a video stream can flow directly from a video source through a video decoder and the video port to the frame buffer. These components often exist together on a display adapter, but can be on separate hardware components that are physically connected to one another. An example of this data flow is provided in the following illustration.



Video source

In the scope of video-port technology, a video source is strictly a hardware video input device, such as a Zoom Video port, MPEG codec, or other hardware source. These sources broadcast signals in a variety of formats, including NTSC, PAL, and SECAM through a physical connection to a video decoder.

Video Decoder

A video decoder is also a hardware component. The video decoder's job is to decipher the information provided by the video source and send it to the video port in an agreed upon connection format. The decoder possesses a physical connection to the video port, and exposes its services through a stream class minidriver. The decoder is responsible for sending video data and clock and sync information to the video port.

Video port

Like the other components in the data flow path, the video port is a piece of hardware. The video port exists on the display adapter's VGA chip and has direct access to the frame buffer. It receives information sent from the decoder, processes it, and places it in the frame buffer to be displayed. During processing, the video port can manipulate image data to provide scaling, shrinking, color control, or cropping services.

Frame Buffer

The frame buffer accepts video data as provided by the video port. Once received, applications can programmatically manipulate the image data, blit it to other locations, or show it on the display using an overlay (the most common use).

About DirectDraw Video-Port Extensions

[This is preliminary documentation and subject to change.]

DirectDraw has been extended to include the DirectDrawVideoPort object, which takes advantage of video-port technology and provides its services through the **IDDVideoPortContainer** and **IDirectDrawVideoPort** interfaces.

DirectDrawVideoPort objects do not control the video decoder, because it provides services of its own, nor does DirectDraw control the video source; it is beyond the scope of the video port. Rather, a DirectDrawVideoPort object represents the video port itself. It monitors the incoming signal and passes image data to the frame buffer, using parameters set though its interface methods to modify the image, perform flipping, or carry out other services.

The **IDDVideoPortContainer** interface, which you can retrieve by calling the **IDirectDraw4::QueryInterface** method, provides methods to query the hardware for its capabilities and create video-port objects. You create a video-port object by calling the **IDDVideoPortContainer::CreateVideoPort** method. Video-port objects expose their functionality through the **IDirectDrawVideoPort** interface, enabling you to manipulate the video-port hardware itself. Using these interfaces, you can examine the video-port's capabilities, assign an overlay surface to receive image data, start and stop video playback, and set hardware parameters to manipulate image data for cropping, color control, scaling, or shrinking effects.

DirectDraw video-port extensions provide for multiple video ports on the same machine by allowing you to create multiple DirectDrawVideoPort objects. There is no requirement that multiple video ports on a machine be identical—each port is separately enumerated and configured separately, regardless of any hardware differences that might exist.

In keeping with the general philosophy of DirectX, this technology gives programmers low-level access to hardware features while insulating them from specific hardware implementation details. It is not a high-level API.

Video Frames and Fields

[This is preliminary documentation and subject to change.]

Video can be interlaced or non-interlaced. When a video signal is interlaced, each video frame is made of two fields of image data. Each field is a collection of every other scan line in an image, starting with the first or second scan line. The first field, referred to as the odd field (or field 1), contains the data for the first scan line and skips every other scan line to the end of the image. Similarly, the even field (or field 2), carries every other scan line starting with the second. The "even-ness" or "odd-ness" of a field is referred to as its field polarity.

When video is not interlaced, each field contains all of a frame's scan lines. Typically, video signals are sent at a rate of 30 frames per second; in the case of interleaved video, this means the rate is 60 fields per second.

The fields that make up a frame do not always reflect the same moment in time. For example, if the frames are separated by 1/30 of a second then the two fields of a frame may be separated by 1/60 of a second. Because a television displays each field individually, no two fields are simultaneously visible, and the difference between fields adds to the illusion of movement.

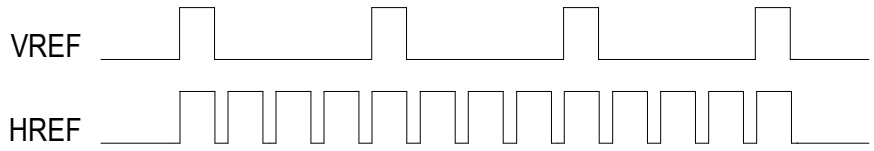
HREF, VREF, and Connections

[This is preliminary documentation and subject to change.]

When a monitor or other display device is displaying an image, it typically scans down the screen, creating an image from left to right, top to bottom. (Sometimes, the device makes two passes down the screen to create a single image; this type of display is called an interlaced display.) The video stream contains signals that instruct the display device when a new line or new screen is to be drawn.

The terms HREF and VREF, also known as hsync and vsync, are the signals within the video stream that tell a display device what to do and when to do it. The HREF signals that a new line is to be drawn and the VREF signals a new screen.

For instance, imagine you're working with a video signal intended for the world's smallest monitor. The monitor only has 4 scan lines. (This is not at all realistic, of course, but it's simple.) On an oscilloscope, the HREF and VREF signals would look somewhat like the following illustration.



In the preceding illustration, both HREF and VREF signals are "active high," meaning that they are considered active when in a heightened state (when the waves go up). There is no standard for these signals. In some cases, places where the waves go down ("low" states) might signal an active HREF or VREF, or sometimes one will be active high and the other active low. Although the preceding illustration is only an imaginary example, note that there are lots of HREF signals for each VREF. This is because for each new screen, there are several scan lines. Of course, in a real video signal for a real broadcast, you would see hundreds of HREFs for a single VREF.

HREF signals, VREF signals, and video data are carried across physical data lines from the decoder to the video port. In many cases, a number of lines are reserved for video data, and others are dedicated to carrying HREF and VREF signals. However, there is no standard for how these data lines are used.

A connection is a protocol that a video port or decoder uses to define how it uses these data lines. Video ports and video decoders will support a variety of connections. DirectDraw video-port extensions use globally-unique identifiers (GUIDs) to identify each type of connection. You can query for the connections that

the video port supports by calling the **IDDVideoPortContainer::GetVideoPortConnectInfo** method. You create a DirectDrawVideoPort object that supports a given connection by calling the **IDDVideoPortContainer::CreateVideoPort** method.

Keep in mind that the video decoder is outside the scope of DirectDraw video-port extensions, and exposes its supported connections through an interface of its own. By enumerating the connections that the video-port supports and comparing the results with the connections supported by the decoder, you can negotiate a common connection (or "language") that both components understand.

Vertical Blanking Interval Data

[This is preliminary documentation and subject to change.]

In broadcast video, a small period of time elapses between video frames, during which a display device refreshes its display for the next frame. This period of time is called the Vertical Blanking Interval (VBI). Instead of sitting idle during the VBI, broadcast video encodes data in the first twenty-one scan lines of a video frame and sends these lines during the VBI. This data is often used for closed captioning or time-stamping, but can be used for other purposes.

DirectDraw video-port extensions enable you to divert data contained with the VBI to a surface, bypass scaling of VBI data, and automatically flip between VBI surfaces in a flipping chain. Once data is in a surface, you can directly access the surface's memory as needed.

For more information, see Auto-Flipping.

Auto-Flipping

[This is preliminary documentation and subject to change.]

To avoid tearing images when refreshing the screen between frames, DirectDrawVideoPort objects can automatically flip their target overlay surfaces in response to VREF signals. To use this service, the target surface you set to the video-port object with the **IDirectDrawVideoPort::SetTargetSurface** method must be the first surface in a flipping chain of overlay surfaces. Then, to begin playing the video sequence, call the **IDirectDrawVideoPort::StartVideo** method, specifying the DDVP_AUTOFLIP flag in the **dwVPFlags** member of the associated **DDVIDEOPORTINFO** structure. The video-port object will flip to the next surface in the flipping chain for each VREF signal it receives. If the video port is interleaving fields, it will flip once for every two VREF signals it receives.

If you are using auto-flipping and want to direct VBI data to separate auto-flipped surfaces, you must have the same number of VBI surfaces as you do standard video surfaces.

Solutions to Common Video Artifacts

[This is preliminary documentation and subject to change.]

Several problems are inherent in displaying broadcast video on display devices other than televisions. This section briefly discusses some common problems, then describes how DirectDraw video-port extensions tries to solve them.

NTSC Interlaced Display and Interleaved Memory

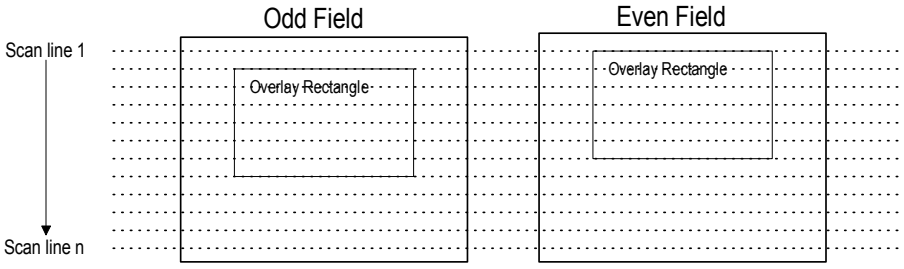
An NTSC signal broadcasts video at an approximate rate of 30 frames, or 60 fields, per second. Like a frame, a field in an NTSC signal is independent of the other field in a frame and can contain different image data. For more information on this behavior, see Video Frames and Fields.

The problems caused by the independence of fields within a frame become apparent when two fields are interleaved for display. In video with a lot of movement, the two fields of a single frame will contain images that don't match each other, resulting in motion artifacts.

One way that developers have tried to work around this behavior is by discarding one of the fields. This solution causes a loss in image quality by roughly one-half, but provides acceptable results for some purposes. Another method frequently used is to display fields individually, stretching each vertically by a factor of two when it is displayed. This provides better image quality, but because fields are offset by one pixel in the y-direction, the result is an animation that "jitters" up and down as it plays.

DirectDraw video-port extensions can employ two, more advanced, techniques for improving image quality, known as "Bob" and "Weave." Both are supported by the DirectDraw overlay surfaces that are used with video-port extensions.

The first algorithm, Bob, is very similar to the method of displaying each field in a frame individually. However, for each field, the overlay's source rectangle is adjusted to accommodate for any jittering effects. Effectively, the source rectangle bounces up and down in time with the fields, negating the jittering on the screen. The following illustration depicts this process.



The Weave algorithm provides the best image quality for material that originates from film by exploiting a common technique used in the video industry for converting motion pictures to television. Unlike Bob, a video-port object does not Weave by itself; you must combine the default overlay behavior of displaying both fields simultaneously with kernel mode video transport (provided with Windows 2000 and Windows 98) to implement the algorithm.

Here is a synopsis of the algorithm, provided for completeness. Motion pictures capture video at a rate of 24 frames per second. When converting a motion picture for television, technicians use a technique called *3:2 pulldown* to convert the frame rate to the 30 frames per second required for television broadcasts. This technique involves inserting a redundant field for every four true fields in the video stream to come up with the required number of fields.

When you weave, you are reversing this process. You detect when 3:2 pulldown is being used, removing any redundant fields to restore the original motion-picture frames. The fields that make up the restored frames can then be interleaved in memory without risk of motion artifacts. Occasionally, the pattern of redundant frames will change due to edits within the original film or reel breaks. You must monitor when these changes occur and update the behavior to adjust for the new pattern.

By default, an overlay surface displays both fields simultaneously. This works well if you're implementing the Weave algorithm, but prevents the video port from using the Bob algorithm. You can programmatically change how the overlay treats video data by calling the **IDirectDrawSurface4::UpdateOverlay** method. The flags you include in the **dwFlags** parameter determine the overlay's behavior: if you include the DDOVER_BOB flag, the video port will use the Bob algorithm; if you don't, it displays both fields. Note that by simply displaying both fields simultaneously, the resulting video will show motion artifacts.

Solving Problems Caused by Half-Lines

[This is preliminary documentation and subject to change.]

Some video decoders output a half line of meaningless data at the beginning of the even field. If this extra line is written to the frame buffer, the resulting image will appear garbled. In some cases, the video-port hardware is capable of sensing and discarding this data before writing it to the frame buffer.

You can determine if a video port is capable of discarding this data when retrieving connection information with the **IDDVideoPortContainer::GetVideoPortConnectInfo** method. If the video port cannot discard half-lines, the DDVPCONNECT_HALFLINE flag will be specified in the **dwFlags** member of the associated **DDVIDEOPORTCONNECT** structure for each supported connection.

If the video port is unable to discard half-lines, you have two options: you can discard one of the fields, or you can work around the hardware's limitations by making some adjustments in how you create the video-port object and display images with the target overlay surface

Here's how to work around the problem. When creating the video-port object by calling the **IDDVideoPortContainer::CreateVideoPort** method, include the DDVPCONNECT_INVERTPOLARITY flag in the **dwFlags** member of the associated **DDVIDEOPORTCONNECT** structure. This causes the video port to invert the polarity of the fields in the video stream, treating even fields like odd

fields and vice versa. Once reversed, the half-line preceding even fields will be written to the frame buffer as the first scan line of each frame. To remove the unwanted data, adjust the source rectangle of the overlay surface used to display the image down one pixel by calling the **IDirectDrawVideoPort::StartVideo** method with the necessary coordinates. Note that this technique requires that you allocate one extra line in the surface containing the even field.

Exploiting Hardware Features

[This is preliminary documentation and subject to change.]

Video-port hardware often supports special features for adjusting color, shrinking or zooming images, handling VBI data, or skipping fields. The HAL provides information about these features by using flags in the **DDVIDEOPORTCAPS** structure. You retrieve the capabilities of a machine's video-port hardware by calling the **IDDVideoPortContainer::EnumVideoPorts** method.

To exploit these features for playback, you use the **IDirectDrawVideoPort::StartVideo** method, which uses a **DDVIDEOPORTINFO** structure to request that hardware features be used to modify image data before placing it in the frame buffer or for display. By setting values and flags in this structure, you can specify the source rectangle used with the overlay surface, indicate cropping regions, request hardware scaling, and set pixel formats.

DirectDrawVideoPort objects do not emulate video-port hardware services.

Getting the Flip and Blit Status

[This is preliminary documentation and subject to change.]

When the **IDirectDrawSurface4::Flip** method is called, the primary surface and back buffer are exchanged. However, the exchange may not occur immediately. For example, if a previous flip has not finished, or if it did not succeed, this method returns DDERR_WASSTILLDRAWING. In the samples included with the SDK, the **IDirectDrawSurface4::Flip** call continues to loop until it returns DD_OK. Also, a **IDirectDrawSurface4::Flip** call does not complete immediately. It schedules a flip for the next time a vertical blank occurs on the system.

An application that waits until the DDERR_WASSTILLDRAWING value is not returned is very inefficient. Instead, you could create a function in your application that calls the **IDirectDrawSurface4::GetFlipStatus** method on the back buffer to determine if the previous flip has finished.

If the previous flip has not finished and the call returns DDERR_WASSTILLDRAWING, your application can use the time to perform another task before it checks the status again. Otherwise, you can perform the next flip. The following example demonstrates this concept:

```
while(lpDDSBack->GetFlipStatus(DDGFS_ISFLIPDONE) ==
   DDERR_WASSTILLDRAWING);
```

```
    // Waiting for the previous flip to finish. The application can
    // perform another task here.

ddrval = lpDDSPrimary->Flip(NULL, 0);
```

You can use the **IDirectDrawSurface4::GetBltStatus** method in much the same way to determine whether a blit has finished. Because **IDirectDrawSurface4::GetFlipStatus** and **IDirectDrawSurface4::GetBltStatus** return immediately, you can use them periodically in your application with little loss in speed.

Determining the Capabilities of the Display Hardware

[This is preliminary documentation and subject to change.]

DirectDraw uses software emulation to perform the DirectDraw functions not supported by the user's hardware. To accelerate performance of your DirectDraw applications, you should determine the capabilities of the user's display hardware after you have created a DirectDraw object, then structure your program to take advantage of these capabilities when possible.

You can determine these capabilities by using the **IDirectDraw4::GetCaps** method. Not all hardware features are supported in emulation. If you want to use a feature only supported by some hardware, you must also be prepared to supply some alternative for systems with hardware that lacks that feature.

Storing Bitmaps in Display Memory

[This is preliminary documentation and subject to change.]

Blitting from display memory to display memory is usually much more efficient than blitting from system memory to display memory. As a result, you should store as many of the sprites your application uses as possible in display memory.

Most display adapter hardware contains enough extra memory to store more than the primary surface and the back buffer. Call the **IDirectDraw4::GetAvailableVidMem** method to determine the amount of total and available memory for storing bitmaps in the display adapter's memory. After the call, the *lpdwTotal* parameter contains the total amount of display memory, minus the primary surface and any private caches held by driver, and *lpdwFree* contains the amount of display memory currently free that can be allocated for a surface that matches the capabilities specified by the structure at *lpDDSCaps2*.

Triple Buffering

[This is preliminary documentation and subject to change.]

In some cases, that is, when the display adapter has enough memory, it may be possible to speed up the process of displaying your application by using triple buffering. Triple buffering uses one primary surface and two back buffers. The following example shows how to initialize a triple-buffering scheme:

```
// The lpDDSPrimary and lpDDSBack variables are globally
// declared, uninitialized LPDIRECTDRAWSURFACE4 variables.
//
// The lpDD variable is a pointer to an IDirectDraw4 interface

DDSURFACEDESC2 ddsd;
ZeroMemory (&ddsd, sizeof(ddsd));

// Create the primary surface with two back buffers.
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
   DDSCAPS_FLIP | DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 2;
ddrval = lpDD->CreateSurface(&ddsd, &lpDDSPrimary, NULL);

// If we successfully created the flipping chain,
// retrieve pointers to the surfaces we need for
// flipping and blitting.
if(ddrval == DD_OK)
{
   // Get the surface directly attached to the primary (the back buffer).
   ddsd.ddsCaps.dwCaps = DDSCAPS_BACKBUFFER;
   ddrval = lpDDSPrimary->GetAttachedSurface(&ddsd.ddsCaps,
      &lpDDSBack);
   if(ddrval != DD_OK) ;
      // Display an error message here.
}
```

You do not need to keep track of all surfaces in a triple buffered flipping chain. The only surfaces you must keep pointers to are the primary surface and the back-buffer surface. You need a pointer to the primary surface in order to flip the surfaces in the flipping chain, and you need a pointer to the back buffer for blitting. For more information, see Flipping Surfaces.

Triple buffering allows your application to continue blitting to the back buffer even if a flip has not completed and the back buffer's blit has already finished. Performing a flip is not a synchronous event; one flip can take longer than another. Therefore, if your application uses only one back buffer, it may spend some time idling while waiting for the **IDirectDrawSurface4::Flip** method to return with DD_OK.

DirectDraw Applications and Window Styles

[This is preliminary documentation and subject to change.]

If your application uses DirectDraw in windowed mode, you can create windows with any window style. However, full-screen exclusive mode applications cannot be created with the WS_EX_TOOLWINDOW style without risk of unpredictable behavior. The WS_EX_TOOLWINDOW style prevents a window from being the top most window, which is required for a DirectDraw full-screen, exclusive mode application.

Full-screen exclusive mode applications should use the WS_EX_TOPMOST extended window style and the WS_VISIBLE window style to display properly. These styles keep the application at the front of the window z-order and prevent GDI from drawing on the primary surface.

The following example shows one way to safely prepare a window to be used in a full-screen, exclusive mode application.

```
//////////////////////////////////////////////////////
// Register the window class, display the window, and init
// all DirectX and graphic objects.
//////////////////////////////////////////////////////
BOOL WINAPI InitApp(INT nWinMode)
{
   WNDCLASSEX wcex;

   wcex.cbSize         =   sizeof(WNDCLASSEX);
   wcex.hInstance      =   g_hinst;
   wcex.lpszClassName  =   g_szWinName;
   wcex.lpfnWndProc    =   WndProc;
   wcex.style          =   CS_VREDRAW|CS_HREDRAW|CS_DBLCLKS;
   wcex.hIcon          =   LoadIcon (NULL, IDI_APPLICATION);
   wcex.hIconSm        =   LoadIcon (NULL, IDI_WINLOGO);
   wcex.hCursor        =   LoadCursor (NULL, IDC_ARROW);
   wcex.lpszMenuName   =   MAKEINTRESOURCE(IDR_APPMENU);
   wcex.cbClsExtra     =   0 ;
   wcex.cbWndExtra     =   0 ;
   wcex.hbrBackground  =   GetStockObject (NULL_BRUSH);

   RegisterClassEx(&wcex);

   g_hwndMain = CreateWindowEx(
                WS_EX_TOPMOST,
                g_szWinName,
                g_szWinCaption,
                WS_VISIBLE|WS_POPUP,
```

```
                0,0,CX_SCREEN,CY_SCREEN,
                NULL,
                NULL,
                g_hinst,
                NULL);

   if(!g_hwndMain)
      return(FALSE);

   SetFocus(g_hwndMain);
   ShowWindow(g_hwndMain, nWinMode);
   UpdateWindow(g_hwndMain);

   return TRUE;
}
```

Matching True RGB Colors to the Frame Buffer's Color Space

[This is preliminary documentation and subject to change.]

Applications often need to find out how a true RGB color (RGB 888) will be mapped into a frame buffer's color space when the display device is not in RGB 888 mode. For example, imagine you're working on an application that will run in 16- and 24-bit RGB display modes. You know that when the art was created, a color was reserved for use as a transparent blitting color key; for the sake of argument, it is a 24-bit color such as RGB(128,64,255). Because your application will also run in a 16-bit RGB mode, you need a way to find out how this 24-bit color key maps into the color space that the frame buffer uses when it's running in a 16-bit RGB mode.

Although DirectDraw does not perform color matching services for you, there are ways to calculate how your color key will be mapped in the frame buffer. These methods can be pretty complicated. For most purposes, you can use the GDI built-in color matching services, combined with the DirectDraw direct frame buffer access, to determine how a color value maps into a different color space. In fact, the Ddutil.cpp source file included in the DirectX examples of the Platform SDK includes a sample function called DDColorMatch that performs this task. The DDColorMatch sample function performs the following main tasks:

1. Retrieves the color value of a pixel in a surface at 0,0.
2. Calls the Win32 **SetPixel** function, using a **COLORREF** structure that describes your 24-bit RGB color.
3. Uses DirectDraw to lock the surface, getting a pointer to the frame buffer memory.
4. Retrieves the actual color value from the frame buffer (set by GDI in Step 2) and unlocks the surface

5. Resets the pixel at 0,0 to its original color using **SetPixel**.

The process used by the DDColorMatch sample function is not fast; it isn't intended to be. However, it provides a reliable way to determine how a color will be mapped across different RGB color spaces. For more information, see the source code for DDColorMatch in the Ddutil.cpp source file.

Note

Because the **SetPixel** GDI function only accepts a **COLORREF** structure on input, this technique only works for matching RGB 888 colors to the frame buffer's pixel format. If your application needs to match colors of another pixel format, you should translate them to RGB 888 before using this technique or query the primary surface for its pixel format and match colors manually.

Displaying a Window in Full-Screen Mode

[This is preliminary documentation and subject to change.]

In full-screen mode, DirectDraw has exclusive control over the display. As a result, dialog boxes and other windows created through GDI are not normally visible. However, by using special techniques you can incorporate a Windows dialog box, HTML Help, or any other kind of window in your application.

The FSWindow Sample illustrates how a dialog box can be displayed and updated in a full-screen application, and how mouse clicks and keystrokes work just as if the dialog box were being displayed by GDI.

In FSWindow, the dialog box is created and "shown" as an ordinary dialog window:

```
hWndDlg = CreateDialog(g_hInstance,
      MAKEINTRESOURCE(IDD_DIALOG_SAMPLE),
      hWnd, (DLGPROC) SampleDlgProc);
ShowWindow(hWndDlg, SW_SHOWNORMAL);
```

Of course, at this point the dialog box is shown only on the hidden GDI surface. It does not appear on the primary surface, which is controlled by DirectDraw.

If the hardware capabilities include DDCAPS2_CANRENDERWINDOWED (see DDCAPS), displaying and updating the dialog box is easy. The application simply calls the **IDirectDraw4::FlipToGDISurface** method, which makes the GDI surface the primary surface. From now on, all updates to the dialog box will be displayed automatically, because GDI is now rendering directly to the front buffer. The application continues rendering to the back buffer, and on each pass through the rendering loop the contents of the back buffer are blitted to the front buffer by DirectDraw. The dialog box is not overwritten because the front buffer is clipped to the application window, and the dialog box is obscuring part of that window.

The following code, from the FSWindow_Init function, creates the clipper, associates it with the application window, and brings the GDI surface to the front:

```
if (ddObject->CreateClipper(0, &ddClipper, NULL) == DD_OK)
   ddClipper->SetHWnd(0, hwndAppWindow);
ddObject->FlipToGDISurface();
```

Then, in the FSWindow_Update function, the following code blits the rendered contents of the back buffer to the clipping region:

```
ddFrontBuffer->SetClipper(ddClipper);
ddFrontBuffer->Blt(NULL, ddBackBuffer, NULL, DDBLT_WAIT, NULL);
```

Note that because the GDI surface is the primary surface, Windows continues displaying the mouse cursor. (This would not be the case, however, if the application were using DirectInput with the mouse device at the exclusive cooperative level.)

For hardware that does not have the DDCAPS2_CANRENDERWINDOWED capability, the process of displaying and updating a window in full-screen mode is somewhat more complicated. In this case, the application is responsible for obtaining the image of the window created by GDI and blitting it to the back buffer after the full-screen rendering has been done. The entire back buffer is then flipped to the front in the usual way.

The FSWindow sample provides two different methods for accessing the display memory of the window, depending on whether the content is static or dynamic. The method for static content is faster because it involves blitting from a memory device context rather than a screen device context. This method should be used for windows that do not change, such as informational dialog boxes. (Remember, though, that unless you manually update the bitmap in response to events, even basic animations such as a button press will not be visible to the user.)

If the content is static, FSWindow calls the CreateBMPFromWindow function when the window is initialized. This function creates a bitmap and blits the contents of the window into it. The bitmap handle is stored in the global variable *hwndFSWindowBMP*. Whenever the primary surface is about to be updated, this bitmap is blitted to the back buffer, as follows:

```
if (FSWindow_IsStatic)
{
   hdcMemory = CreateCompatibleDC(NULL);
   SelectObject(hdcMemory, hwndFSWindowBMP);
   BitBlt(hdcBackBuffer, x, y, cx, cy, hdcMemory, 0, 0, SRCCOPY);
   DeleteDC(hdcMemory);
}
```

If, on the other hand, the content of the window is dynamic, the following code is executed. It blits the image directly from the GDI surface (represented by the *hdcScreen* device context) to the back buffer.

```
BitBlt(hdcBackBuffer, x, y, cx, cy, hdcScreen, x, y, SRCCOPY);
```

The coordinates represent the position and dimensions of the window on the GDI surface, as retrieved through a call to **GetWindowRect**.

When the FSWindow application is running on hardware that does not have the DDCAPS2_CANRENDERWINDOWED capability, it does not use the GDI surface, so Windows cannot display the mouse cursor. The application takes over this task by obtaining information about the cursor and displaying it on the back buffer just before the flip.

DirectDraw Tutorials

[This is preliminary documentation and subject to change.]

This section contains a series of tutorials, each providing step-by-step instructions for implementing the basics of DirectDraw in a C/C++ or Visual Basic application. The tutorials are written parallel to a set of sample files that are provided with this SDK in the \Samples\Multimedia\DDraw\Tutorials directory, following their code path and providing explanations along the way. Readers are encouraged to follow along in the sample code as they move through these tutorials.

- DirectDraw C/C++ Tutorials
- DirectDraw Visual Basic Tutorials

DirectDraw C/C++ Tutorials

[This is preliminary documentation and subject to change.]

This section contains a series of tutorials, each of which provides step-by-step instructions for implementing a simple DirectDraw application. These tutorials use many of the DirectDraw sample files that are provided with this SDK. These samples demonstrate how to set up DirectDraw, and how to use the DirectDraw methods to perform common tasks:

- Tutorial 1: The Basics of DirectDraw
- Tutorial 2: Loading Bitmaps on the Back Buffer
- Tutorial 3: Blitting from an Off-Screen Surface
- Tutorial 4: Color Keys and Bitmap Animation
- Tutorial 5: Dynamically Modifying Palettes
- Tutorial 6: Using Overlay Surfaces

Some samples in these tutorials use older versions **IDirectDraw** and **IDirectDrawSurface** interfaces. If you want to update these examples so they use the DirectX 5.0 interfaces query for the new versions of the interfaces before using them. In addition, you must change the appropriate parameters of any methods that have been updated for new versions of the interfaces.

Note

The sample files in these tutorials are written in C++. If you are using a C compiler, you must make the appropriate changes to the files for them to successfully compile. At the very least, you need to add the *vtable* and *this* pointers to the interface methods.

Tutorial 1: The Basics of DirectDraw

[This is preliminary documentation and subject to change.]

To use DirectDraw, you first create an instance of the DirectDraw object, which represents the display adapter on the computer. You then use the interface methods to manipulate the object. In addition, you need to create one or more instances of a DirectDrawSurface object to be able to display your application on a graphics surface.

To demonstrate this, the DDEx1 sample included with this SDK performs the following steps:

- Step 1: Creating a DirectDraw Object
- Step 2: Determining the Application's Behavior
- Step 3: Changing the Display Mode
- Step 4: Creating Flipping Surfaces
- Step 5: Rendering to the Surfaces
- Step 6: Writing to the Surface
- Step 7: Flipping the Surfaces
- Step 8: Deallocating the DirectDraw Objects

Note

To use GUIDs successfully in your applications, you must either define INITGUID prior to all other include and define statements, or you must link to the Dxguid.lib library. You should define INITGUID in only one of your source modules.

Step 1: Creating a DirectDraw Object

[This is preliminary documentation and subject to change.]

To create an instance of a DirectDraw object, your application should use the **DirectDrawCreate** function as shown in the doInit sample function of the DDEx1

program. **DirectDrawCreate** contains three parameters. The first parameter takes a globally unique identifier (GUID) that represents the display device. The GUID, in most cases, is set to NULL, which means DirectDraw uses the default display driver for the system. The second parameter contains the address of a pointer that identifies the location of the DirectDraw object if it is created. The third parameter is always set to NULL and is included for future expansion.

The following example shows how to create the DirectDraw object and how to determine if the creation was successful or not:

```
ddrval = DirectDrawCreate(NULL, &lpDD, NULL);
if(ddrval == DD_OK)
{
   // lpDD is a valid DirectDraw object.
}
else
{
   // The DirectDraw object could not be created.
}
```

Step 2: Determining the Application's Behavior

[This is preliminary documentation and subject to change.]

Before you can change the resolution of the display, you must at a minimum specify the DDSCL_EXCLUSIVE and DDSCL_FULLSCREEN flags in the *dwFlags* parameter of the **IDirectDraw::SetCooperativeLevel** method. This gives your application complete control over the display device, and no other application will be able to share it. In addition, the DDSCL_FULLSCREEN flag sets the application in exclusive (full-screen) mode. Your application covers the entire desktop, and only your application can write to the screen. The desktop is still available, however. (To see the desktop in an application running in exclusive mode, start DDEx1 and press ALT+ TAB.)

The following example demonstrates the use of the **SetCooperativeLevel** method:

```
HRESULT     ddrval;
LPDIRECTDRAW lpDD;    // Already created by DirectDrawCreate

ddrval = lpDD->SetCooperativeLevel(hwnd, DDSCL_EXCLUSIVE |
   DDSCL_FULLSCREEN);
if(ddrval == DD_OK)
{
   // Exclusive mode was successful.
}
else
{
   // Exclusive mode was not successful.
   // The application can still run, however.
```

```
}
```

If **SetCooperativeLevel** does not return DD_OK, you can still run your application. The application will not be in exclusive mode, however, and it might not be capable of the performance your application requires. In this case, you might want to display a message that allows the user to decide whether or not to continue.

If you are setting the full-screen, exclusive cooperative level, you must pass your application's window handle to **SetCooperativeLevel** to allow Windows to determine if your application terminates abnormally. For example, if a general protection (GP) fault occurs and GDI is flipped to the back buffer, the user will not be able to return to the Windows screen. To prevent this from occurring, DirectDraw provides a process running in the background that traps messages that are sent to that window. DirectDraw uses these messages to determine when the application terminates. This feature imposes some restrictions, however. You have to specify the window handle that is retrieving messages for your application—that is, if you create another window, you must ensure that you specify the window that is active. Otherwise, you might experience problems, including unpredictable behavior from GDI, or no response when you press ALT + TAB.

Step 3: Changing the Display Mode

[This is preliminary documentation and subject to change.]

After you have set the application's behavior, you can use the **IDirectDraw::SetDisplayMode** method to change the resolution of the display. The following example shows how to set the display mode to 640×480×8 bpp:

```
HRESULT     ddrval;
LPDIRECTDRAW lpDD;  // Already created

ddrval = lpDD->SetDisplayMode(640, 480, 8);
if(ddrval == DD_OK)
{
   // The display mode changed successfully.
}
else
{
   // The display mode cannot be changed.
   // The mode is either not supported or
   // another application has exclusive mode.
}
```

When you set the display mode, you should ensure that if the user's hardware cannot support higher resolutions, your application reverts to a standard mode that is supported by a majority of display adapters. For example, your application could be designed to run on all systems that support 640×480×8 as a standard backup resolution.

Note

IDirectDraw::SetDisplayMode returns a DDERR_INVALIDMODE error value if the display adapter could not be set to the desired resolution. Therefore, you should use the **IDirectDraw::EnumDisplayModes** method to determine the capabilities of the user's display adapter before trying to set the display mode.

Step 4: Creating Flipping Surfaces

[This is preliminary documentation and subject to change.]

After you have set the display mode, you must create the surfaces on which to place your application. Because the DDEx1 example is using the **IDirectDraw::SetCooperativeLevel** method to set the mode to exclusive (full-screen) mode, you can create surfaces that flip between the surfaces. If you were using **SetCooperativeLevel** to set the mode to DDSCL_NORMAL, you could create only surfaces that blit between the surfaces. Creating flipping surfaces requires the following steps, also discussed in this topic:

- Defining the surface requirements
- Creating the surfaces

Defining the Surface Requirements

[This is preliminary documentation and subject to change.]

The first step in creating flipping surfaces is to define the surface requirements in a **DDSURFACEDESC** structure. The following example shows the structure definitions and flags needed to create a flipping surface.

```
// Create the primary surface with one back buffer.
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
   DDSCAPS_FLIP | DDSCAPS_COMPLEX;

ddsd.dwBackBufferCount = 1;
```

In this example, the **dwSize** member is set to the size of the **DDSURFACEDESC** structure. This is to prevent any DirectDraw method call you use from returning with an invalid member error. (The **dwSize** member was provided for future expansion of the **DDSURFACEDESC** structure.)

The **dwFlags** member determines which members in the **DDSURFACEDESC** structure will be filled with valid information. For the DDEx1 example, **dwFlags** is set to specify that you want to use the **DDSCAPS** structure (DDSD_CAPS) and that you want to create a back buffer (DDSD_BACKBUFFERCOUNT).

The **dwCaps** member in the example indicates the flags that will be used in the **DDSCAPS** structure. In this case, it specifies a primary surface

(DDSCAPS_PRIMARYSURFACE), a flipping surface (DDSCAPS_FLIP), and a complex surface (DDSCAPS_COMPLEX).

Finally, the example specifies one back buffer. The back buffer is where the backgrounds and sprites will actually be written. The back buffer is then flipped to the primary surface. In the DDEx1 example, the number of back buffers is set to 1. You can, however, create as many back buffers as the amount of display memory allows. For more information on creating more than one back buffer, see Triple Buffering.

Surface memory can be either display memory or system memory. DirectDraw uses system memory if the application runs out of display memory (for example, if you specify more than one back buffer on a display adapter with only 1 MB of RAM). You can also specify whether to use only system memory or only display memory by setting the **dwCaps** member in the **DDSCAPS** structure to DDSCAPS_SYSTEMMEMORY or DDSCAPS_VIDEOMEMORY. (If you specify DDSCAPS_VIDEOMEMORY, but not enough memory is available to create the surface, **IDirectDraw::CreateSurface** returns with a DDERR_OUTOFVIDEOMEMORY error.)

Creating the Surfaces

[This is preliminary documentation and subject to change.]

After the **DDSURFACEDESC** structure is filled, you can use it and *lpDD*, the pointer to the DirectDraw object that was created by the **DirectDrawCreate** function, to call the **IDirectDraw::CreateSurface** method, as shown in the following example:

```
ddrval = lpDD->CreateSurface(&ddsd, &lpDDSPrimary, NULL);
if(ddrval == DD_OK)
{
   // lpDDSPrimary points to the new surface.
}
else
{
   // The surface was not created.
   return FALSE;
}
```

The *lpDDSPrimary* parameter will point to the primary surface returned by **CreateSurface** if the call succeeds.

After the pointer to the primary surface is available, you can use the **IDirectDrawSurface3::GetAttachedSurface** method to retrieve a pointer to the back buffer, as shown in the following example:

```
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
ddrval = lpDDSPrimary->GetAttachedSurface(&ddcaps, &lpDDSBack);
if(ddrval == DD_OK)
```

```
{
   // lpDDSBack points to the back buffer.
}
else
{
   return FALSE;
}
```

By supplying the address of the surface's primary surface and by setting the capabilities value with the DDSCAPS_BACKBUFFER flag, the *lpDDSBack* parameter will point to the back buffer if the **IDirectDrawSurface3::GetAttachedSurface** call succeeds.

Step 5: Rendering to the Surfaces

[This is preliminary documentation and subject to change.]

After the primary surface and a back buffer have been created, the DDEx1 example renders some text on the primary surface and back buffer surface by using standard Windows GDI functions, as shown in the following example:

```
if (lpDDSPrimary->GetDC(&hdc) == DD_OK)
{
   SetBkColor(hdc, RGB(0, 0, 255));
   SetTextColor(hdc, RGB(255, 255, 0));
   TextOut(hdc, 0, 0, szFrontMsg, lstrlen(szFrontMsg));
   lpDDSPrimary->ReleaseDC(hdc);
}

if (lpDDSBack->GetDC(&hdc) == DD_OK)
{
   SetBkColor(hdc, RGB(0, 0, 255));
   SetTextColor(hdc, RGB(255, 255, 0));
   TextOut(hdc, 0, 0, szBackMsg, lstrlen(szBackMsg));
   lpDDSBack->ReleaseDC(hdc);
}
```

The example uses the **IDirectDrawSurface3::GetDC** method to retrieve the handle of the device context, and it internally locks the surface. If you are not going to use Windows functions that require a handle of a device context, you could use the **IDirectDrawSurface3::Lock** and **IDirectDrawSurface3::Unlock** methods to lock and unlock the back buffer.

Locking the surface memory (whether the whole surface or part of a surface) ensures that your application and the system blitter cannot obtain access to the surface memory at the same time. This prevents errors from occurring while your application is writing to surface memory. In addition, your application cannot page flip until the surface memory is unlocked.

After the surface is locked, the example uses standard Windows GDI functions: **SetBkColor** to set the background color, **SetTextColor** to select the color of the text to be placed on the background, and **TextOut** to print the text and background color on the surfaces.

After the text has been written to the buffer, the example uses the **IDirectDrawSurface3::ReleaseDC** method to unlock the surface and release the handle. Whenever your application finishes writing to the back buffer, you must call either **IDirectDrawSurface3::ReleaseDC** or **IDirectDrawSurface3::Unlock**, depending on your application. Your application cannot flip the surface until the surface is unlocked.

Typically, you write to a back buffer, which you then flip to the primary surface to be displayed. In the case of DDEx1, there is a significant delay before the first flip, so DDEx1 writes to the primary buffer in the initialization function to prevent a delay before displaying the surface. As you will see in a subsequent step of this tutorial, the DDEx1 example writes only to the back buffer during WM_TIMER. An initialization function or title page may be the only place where you might want to write to the primary surface.

Note

After the surface is unlocked by using **IDirectDrawSurface3::Unlock**, the pointer to the surface memory is invalid. You must use **IDirectDrawSurface3::Lock** again to obtain a valid pointer to the surface memory.

Step 6: Writing to the Surface

[This is preliminary documentation and subject to change.]

The first half of the WM_TIMER message in DDEx1 is devoted to writing to the back buffer, as shown in the following example:

```
case WM_TIMER:
   // Flip surfaces.
   if(bActive)
   {
      if (lpDDSBack->GetDC(&hdc) == DD_OK)
      {
         SetBkColor(hdc, RGB(0, 0, 255));
         SetTextColor(hdc, RGB(255, 255, 0));
         if(phase)
         {
            TextOut(hdc, 0, 0, szFrontMsg, lstrlen(szFrontMsg));
            phase = 0;
         }
         else
         {
            TextOut(hdc, 0, 0, szBackMsg, lstrlen(szBackMsg));
```

```
            phase = 1;
        }
        lpDDSBack->ReleaseDC(hdc);
    }
```

The line of code that calls the **IDirectDrawSurface3::GetDC** method locks the back buffer in preparation for writing. The **SetBkColor** and **SetTextColor** functions set the colors of the background and text.

Next, the *phase* variable determines whether the primary buffer message or the back buffer message should be written. If *phase* equals 1, the primary surface message is written, and *phase* is set to 0. If *phase* equals 0, the back buffer message is written, and *phase* is set to 1. Note, however, that in both cases the messages are written to the back buffer.

After the message is written to the back buffer, the back buffer is unlocked by using the **IDirectDrawSurface3::ReleaseDC** method.

Step 7: Flipping the Surfaces

[This is preliminary documentation and subject to change.]

After the surface memory is unlocked, you can use the **IDirectDrawSurface3::Flip** method to flip the back buffer to the primary surface, as shown in the following example:

```
while(1)
{
    HRESULT ddrval;
    ddrval = lpDDSPrimary->Flip(NULL, 0);
    if(ddrval == DD_OK)
    {
        break;
    }
    if(ddrval == DDERR_SURFACELOST)
    {
        ddrval = lpDDSPrimary->Restore();
        if(ddrval != DD_OK)
        {
            break;
        }
    }
    if(ddrval != DDERR_WASSTILLDRAWING)
    {
        break;
    }
}
```

In the example, *lpDDSPrimary* parameter designates the primary surface and its associated back buffer. When **IDirectDrawSurface3::Flip** is called, the front and back surfaces are exchanged (only the pointers to the surfaces are changed; no data is actually moved). If the flip is successful and returns DD_OK, the application breaks from the while loop.

If the flip returns with a DDERR_SURFACELOST value, an attempt is made to restore the surface by using the **IDirectDrawSurface3::Restore** method. If the restore is successful, the application loops back to the **IDirectDrawSurface3::Flip** call and tries again. If the restore is unsuccessful, the application breaks from the while loop, and returns with an error.

Note

When you call **IDirectDrawSurface3::Flip**, the flip does not complete immediately. Rather, a flip is scheduled for the next time a vertical blank occurs on the system. If, for example, the previous flip has not occurred, **IDirectDrawSurface3::Flip** returns DDERR_WASSTILLDRAWING. In the example, the **IDirectDrawSurface3::Flip** call continues to loop until it returns DD_OK.

Step 8: Deallocating the DirectDraw Objects

[This is preliminary documentation and subject to change.]

When you press the F12 key, the DDEx1 application processes the WM_DESTROY message before exiting the application. This message calls the finiObjects sample function, which contains all of the **IUnknown::Release** calls, as shown in the following example:

```
static void finiObjects(void)
{
   if(lpDD != NULL)
   {
      if(lpDDSPrimary != NULL)
      {
         lpDDSPrimary->Release();
         lpDDSPrimary = NULL;
      }
      lpDD->Release();
      lpDD = NULL;
   }
} // finiObjects
```

The application checks if the pointers to the DirectDraw object (*lpDD*) and the DirectDrawSurface object (*lpDDSPrimary*) are not equal to NULL. Then DDEx1 calls the **IDirectDrawSurface3::Release** method to decrease the reference count of the DirectDrawSurface object by 1. Because this brings the reference count to 0, the DirectDrawSurface object is deallocated. The DirectDrawSurface pointer is then

destroyed by setting its value to NULL. Next, the application calls **IDirectDraw::Release** to decrease the reference count of the DirectDraw object to 0, deallocating the DirectDraw object. This pointer is then also destroyed by setting its value to NULL.

Tutorial 2: Loading Bitmaps on the Back Buffer

[This is preliminary documentation and subject to change.]

The sample discussed in this tutorial (DDEx2) expands on the DDEx1 sample that was discussed in Tutorial 1. DDEx2 includes functionality to load a bitmap file on the back buffer. This new functionality is demonstrated in the following steps:

- Step 1: Creating the Palette
- Step 2: Setting the Palette
- Step 3: Loading a Bitmap on the Back Buffer
- Step 4: Flipping the Surfaces

As in DDEx1, doInit is the initialization function for the DDEx2 application. Although the code for the DirectDraw initialization does not look quite the same in DDEx2 as it did in DDEx1, it is essentially the same, except for the following section:

```
lpDDPal = DDLoadPalette(lpDD, szBackground);

if (lpDDPal == NULL)
   goto error;

ddrval = lpDDSPrimary->SetPalette(lpDDPal);

if(ddrval != DD_OK)
   goto error;

// Load a bitmap into the back buffer.
ddrval = DDReLoadBitmap(lpDDSBack, szBackground);

if(ddrval != DD_OK)
   goto error;
```

Step 1: Creating the Palette

[This is preliminary documentation and subject to change.]

The DDEx2 sample first loads the palette into a structure by using the following code:

```
lpDDPal = DDLoadPalette(lpDD, szBackground);

if (lpDDPal == NULL)
   goto error;
```

The sample function DDLoadPalette is part of the common DirectDraw functions found in the Ddutil.cpp file located in the \Dxsdk\Sdk\Samples\Misc directory. Most of the DirectDraw sample files in this SDK use this file. Essentially, it contains the functions for loading bitmaps and palettes from either files or resources. To avoid having to repeat code in the example files, these functions were placed in a file that could be reused. Make sure you include Ddutil.cpp in the list of files to be compiled with the rest of the DDEx*n* samples.

For DDEx2, the DDLoadPalette sample function creates a DirectDrawPalette object from the Back.bmp file. The DDLoadPalette sample function determines if a file or resource for creating a palette exists. If one does not, it creates a default palette. For DDEx2, it extracts the palette information from the bitmap file and stores it in a structure pointed to by *ape*.

DDEx2 then creates the DirectDrawPalette object, as shown in the following example:

```
pdd->CreatePalette(DDPCAPS_8BIT, ape, &ddpal, NULL);
return ddpal;
```

When the **IDirectDraw2::CreatePalette** method returns, the *ddpal* parameter points to the DirectDrawPalette object, which is then returned from the DDLoadPalette call.

The *ape* parameter is a pointer to a structure that can contain either 2, 4, 16, or 256 entries, organized linearly. The number of entries depends on the *dwFlags* parameter in the **CreatePalette** method. In this case, the *dwFlags* parameter is set to DDPCAPS_8BIT, which indicates that there are 256 entries in this structure. Each entry contains 4 bytes (a red channel, a green channel, a blue channel, and a flags byte).

Step 2: Setting the Palette

[This is preliminary documentation and subject to change.]

After you create the palette, you pass the pointer to the DirectDrawPalette object (*ddpal*) to the primary surface by calling the **IDirectDrawSurface3::SetPalette** method, as shown in the following example:

```
ddrval = lpDDSPrimary->SetPalette(lpDDPal);

if(ddrval != DD_OK)
   // SetPalette failed.
```

After you have called **IDirectDrawSurface3::SetPalette**, the DirectDrawPalette object is associated with the DirectDrawSurface object. Any time you need to change the palette, you simply create a new palette and set the palette again. (Although this tutorial uses these steps, there are other ways of changing the palette, as will be shown in later examples.)

Step 3: Loading a Bitmap on the Back Buffer

[This is preliminary documentation and subject to change.]

After the DirectDrawPalette object is associated with the DirectDrawSurface object, DDEx2 loads the Back.bmp bitmap on the back buffer by using the following code:

```
// Load a bitmap into the back buffer.
ddrval = DDReLoadBitmap(lpDDSBack, szBackground);

if(ddrval != DD_OK)
   // Load failed.
```

DDReLoadBitmap is another sample function found in Ddutil.cpp. It loads a bitmap from a file or resource into an already existing DirectDraw surface. (You could also use DDLoadBitmap to create a surface and load the bitmap into that surface. For more information, see Tutorial 5: Dynamically Modifying Palettes.) For DDEx2, it loads the Back.bmp file pointed to by *szBackground* onto the back buffer pointed to by *lpDDSBack*. The DDReLoadBitmap function calls the DDCopyBitmap function to copy the file onto the back buffer and stretch it to the proper size.

The DDCopyBitmap function copies the bitmap into memory, and it uses the GetObject function to retrieve the size of the bitmap. It then uses the following code to retrieve the size of the back buffer onto which it will place the bitmap:

```
// Get the size of the surface.
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_HEIGHT | DDSD_WIDTH;
pdds->GetSurfaceDesc(&ddsd);
```

The *ddsd* value is a pointer to the **DDSURFACEDESC** structure. This structure stores the current description of the DirectDraw surface. In this case, the **DDSURFACEDESC** members describe the height and width of the surface, which are indicated by DDSD_HEIGHT and DDSD_WIDTH. The call to the **IDirectDrawSurface3::GetSurfaceDesc** method then loads the structure with the proper values. For DDEx2, the values will be 480 for the height and 640 for the width.

The DDCopyBitmap sample function locks the surface and copies the bitmap to the back buffer, stretching or compressing it as applicable by using the StretchBlt function, as shown in the following example:

```
if ((hr = pdds->GetDC(&hdc)) == DD_OK)
{
```

```
    StretchBlt(hdc, 0, 0, ddsd.dwWidth, ddsd.dwHeight, hdcImage, x, y,
      dx, dy, SRCCOPY);
    pdds->ReleaseDC(hdc);
}
```

Step 4: Flipping the Surfaces

[This is preliminary documentation and subject to change.]

Flipping surfaces in the DDEx2 sample is essentially the same process as that in the DDEx1 tutorial (see Tutorial 1: The Basics of DirectDraw) except that if the surface is lost (**DDERR_SURFACELOST**), the bitmap must be reloaded on the back buffer by using the DDReLoadBitmap function after the surface is restored.

Tutorial 3: Blitting from an Off-Screen Surface

[This is preliminary documentation and subject to change.]

The sample in Tutorial 2 (DDEx2) takes a bitmap and puts it in the back buffer, and then it flips between the back buffer and the primary buffer. This is not a very realistic approach to displaying bitmaps. The sample in this tutorial (DDEx3) expands on the capabilities of DDEx2 by including two off-screen buffers in which the two bitmaps—one for the even screen and one for the odd screen—are stored. It uses the **IDirectDrawSurface3::BltFast** method to copy the contents of an off-screen surface to the back buffer, and then it flips the buffers and copies the next off-screen surface to the back buffer.

The new functionality demonstrated in DDEx3 is shown in the following steps:

- Step 1: Creating the Off-Screen Surfaces
- Step 2: Loading the Bitmaps to the Off-Screen Surfaces
- Step 3: Blitting the Off-Screen Surfaces to the Back Buffer

Step 1: Creating the Off-Screen Surfaces

[This is preliminary documentation and subject to change.]

The following code is added to the doInit sample function in DDEx3 to create the two off-screen buffers:

```
// Create an offscreen bitmap.
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 480;
ddsd.dwWidth = 640;
ddrval = lpDD->CreateSurface(&ddsd, &lpDDSOne, NULL);
if(ddrval != DD_OK)
```

```
{
   return initFail(hwnd);
}

// Create another offscreen bitmap.
ddrval = lpDD->CreateSurface(&ddsd, &lpDDSTwo, NULL);
if(ddrval != DD_OK)
{
   return initFail(hwnd);
}
```

The **dwFlags** member specifies that the application will use the **DDSCAPS** structure, and it will set the height and width of the buffer. The surface will be an off-screen plain buffer, as indicated by the DDSCAPS_OFFSCREEN flag set in the **DDSCAPS** structure. The height and the width are set as 480 and 640, respectively, in the **DDSURFACEDESC** structure. The surface is then created by using the **IDirectDraw::CreateSurface** method.

Because both of the off-screen plain buffers are the same size, the only requirement for creating the second buffer is to call **IDirectDraw::CreateSurface** again with a different pointer name.

You can also specifically request that the off-screen buffer be placed in system memory or display memory by setting either the DDSCAPS_SYSTEMMEMORY or DDSCAPS_VIDEOMEMORY capability in the **DDSCAPS** structure. By saving the bitmaps in display memory, you can increase the speed of the transfers between the off-screen surfaces and the back buffer. This will become more important when using bitmap animation. However, if you specify DDSCAPS_VIDEOMEMORY for the off-screen buffer and not enough display memory is available to hold the entire bitmap, a DDERR_OUTOFVIDEOMEMORY error value will be returned when you attempt to create the surface.

Step 2: Loading the Bitmaps to the Off-Screen Surfaces

[This is preliminary documentation and subject to change.]

After the two off-screen surfaces are created, DDEx3 uses the InitSurfaces sample function to load the bitmaps from the Frntback.bmp file onto the surfaces. The InitSurfaces function uses the DDCopyBitmap sample function located in Ddutil.cpp to load both of the bitmaps, as shown in the following example:

```
// Load the bitmap resource.
hbm = (HBITMAP)LoadImage(GetModuleHandle(NULL), szBitmap,
   IMAGE_BITMAP, 0, 0, LR_CREATEDIBSECTION);

if (hbm == NULL)
   return FALSE;
```

```
DDCopyBitmap(lpDDSOne, hbm, 0, 0,   640, 480);
DDCopyBitmap(lpDDSTwo, hbm, 0, 480, 640, 480);
DeleteObject(hbm);

return TRUE;
```

If you look at the Frntback.bmp file in Microsoft® Paint or another drawing application, you can see that the bitmap consists of two screens, one on top of the other. The DDCopyBitmap function breaks the bitmap in two at the point where the screens meet. In addition, it loads the first bitmap into the first off-screen surface (*lpDDSOne*) and the second bitmap into the second off-screen surface (*lpDDSTwo*).

Step 3: Blitting the Off-Screen Surfaces to the Back Buffer

[This is preliminary documentation and subject to change.]

The WM_TIMER message contains the code for writing to surfaces and flipping surfaces. In the case of DDEx3, it contains the following code to select the proper off-screen surface and to blit it to the back buffer:

```
rcRect.left = 0;
rcRect.top = 0;
rcRect.right = 640;
rcRect.bottom = 480;
if(phase)
{
   pdds = lpDDSTwo;
   phase = 0;
}
else
{
   pdds = lpDDSOne;
   phase = 1;
}
while(1)
{
   ddrval = lpDDSBack->BltFast(0, 0, pdds, &rcRect, FALSE);
   if(ddrval == DD_OK)
   {
      break;
   }
}
```

The phase variable determines which off-screen surface will be blitted to the back buffer. The **IDirectDrawSurface3::BltFast** method is then called to blit the selected off-screen surface onto the back buffer, starting at position (0, 0), the upper-left corner. The *rcRect* parameter points to the **RECT** structure that defines the upper-

left and lower-right corners of the off-screen surface that will be blitted from. The last parameter is set to FALSE (or 0), indicating that no specific transfer flags are used.

Depending on the requirements of your application, you could use either the **IDirectDrawSurface3::Blt** method or the **IDirectDrawSurface::BltFast** method to blit from the off-screen buffer. If you are performing a blit from an off-screen plain buffer that is in display memory, you should use **IDirectDrawSurface3::BltFast**. Although you will not gain speed on systems that use hardware blitter on their display adapters, the blit will take about 10 percent less time on systems that use hardware emulation to perform the blit. Because of this, you should use **IDirectDrawSurface3::BltFast** for all display operations that blit from display memory to display memory. If you are blitting from system memory or require special hardware flags, however, you have to use **IDirectDrawSurface3::Blt**.

After the off-screen surface is loaded in the back buffer, the back buffer and the primary surface are flipped in much the same way as shown in the previous tutorials.

Tutorial 4: Color Keys and Bitmap Animation

[This is preliminary documentation and subject to change.]

The sample in Tutorial 3 (DDEx3) shows one simple method of placing bitmaps into an off-screen buffer before they are blitted to the back buffer. The sample in this tutorial (DDEx4) uses the techniques described in the previous tutorials to load a background and a series of sprites into an off-screen surface. Then it uses the **IDirectDrawSurface3::BltFast** method to copy portions of the off-screen surface to the back buffer, thereby generating a simple bitmap animation.

The bitmap file that DDEx4 uses, All.bmp, contains the background and 60 iterations of a rotating red donut with a black background. The DDEx4 sample contains new functions that set the color key for the rotating donut sprites. Then, the sample copies the appropriate sprite to the back buffer from the off-screen surface.

The new functionality demonstrated in DDEx4 is shown in the following steps:

- Step 1: Setting the Color Key
- Step 2: Creating a Simple Animation

Step 1: Setting the Color Key

[This is preliminary documentation and subject to change.]

In addition to the other functions found in the doInit sample function of some of the other DirectDraw samples, the DDEx4 sample contains the code to set the color key for the sprites. Color keys are used for setting a color value that will be used for transparency. When the system contains a hardware blitter, all the pixels of a rectangle are blitted except the value that was set as the color key, thereby creating

nonrectangular sprites on a surface. The following code shows how to set the color key in DDEx4:

```
// Set the color key for this bitmap (black).
DDSetColorKey(lpDDSOne, RGB(0,0,0));

return TRUE;
```

You can select the color key by setting the RGB values for the color you want in the call to the DDSetColorKey sample function. The RGB value for black is (0, 0, 0). The DDSetColorKey function calls the DDColorMatch function. (Both functions are in Ddutil.cpp.) The DDColorMatch function stores the current color value of the pixel at location (0, 0) on the bitmap located in the *lpDDSOne* surface. Then it takes the RGB values you supplied and sets the pixel at location (0, 0) to that color. Finally, it masks the value of the color with the number of bits per pixel that are available. After that is done, the original color is put back in location (0, 0), and the call returns to DDSetColorKey with the actual color key value. After it is returned, the color key value is placed in the **dwColorSpaceLowValue** member of the **DDCOLORKEY** structure. It is also copied to the **dwColorSpaceHighValue** member. The call to **IDirectDrawSurface3::SetColorKey** then sets the color key.

You may have noticed the reference to CLR_INVALID in DDSetColorKey and DDColorMatch. If you pass CLR_INVALID as the color key in the DDSetColorKey call in DDEx4, the pixel in the upper-left corner (0, 0) of the bitmap will be used as the color key. As the DDEx4 bitmap is delivered, that does not mean much because the color of the pixel at (0, 0) is a shade of gray. If, however, you would like to see how to use the pixel at (0, 0) as the color key for the DDEx4 sample, open the All.bmp bitmap file in a drawing application and then change the single pixel at (0, 0) to black. Be sure to save the change (it's hard to see). Then change the DDEx4 line that calls DDSetColorKey to the following:

```
DDSetColorKey(lpDDSOne, CLR_INVALID);
```

Recompile the DDEx4 sample, and ensure that the resource definition file is also recompiled so that the new bitmap is included. (To do this, you can simply add and then delete a space in the Ddex4.rc file.) The DDEx4 sample will then use the pixel at (0, 0), which is now set to black, as the color key.

Step 2: Creating a Simple Animation

[This is preliminary documentation and subject to change.]

The DDEx4 sample uses the updateFrame sample function to create a simple animation using the red donuts included in the All.bmp file. The animation consists of three red donuts positioned in a triangle and rotating at various speeds. This sample compares the Win32 **GetTickCount** function with the number of milliseconds since the last call to **GetTickCount** to determine whether to redraw any of the sprites. It subsequently uses the **IDirectDrawSurface3::BltFast** method first to blit the background from the off-screen surface (*lpDDSOne*) to the back buffer,

and then to blit the sprites to the back buffer using the color key that you set earlier to determine which pixels are transparent. After the sprites are blitted to the back buffer, DDEx4 calls the **IDirectDrawSurface3::Flip** method to flip the back buffer and the primary surface.

Note that when you use **IDirectDrawSurface3::BltFast** to blit the background from the off-screen surface, the *dwTrans* parameter that specifies the type of transfer is set to DDBLTFAST_NOCOLORKEY. This indicates that a normal blit will occur with no transparency bits. Later, when the red donuts are blitted to the back buffer, the *dwTrans* parameter is set to DDBLTFAST_SRCCOLORKEY. This indicates that a blit will occur with the color key for transparency as it is defined, in this case, in the *lpDDSOne* buffer.

In this sample, the entire background is redrawn each time through the updateFrame function. One way of optimizing this sample would be to redraw only that portion of the background that changes while rotating the red donuts. Because the location and size of the rectangles that make up the donut sprites never change, you should be able to easily modify the DDEx4 sample with this optimization.

Tutorial 5: Dynamically Modifying Palettes

[This is preliminary documentation and subject to change.]

The sample described in this tutorial (DDEx5) is a modification of the sample described in Tutorial 4 (DDEx4) example. DDEx5 demonstrates how to dynamically change the palette entries while an application is running. The new functionality demonstrated in DDEx5 is shown in the following steps:

- Step 1: Loading the Palette Entries
- Step 2: Rotating the Palettes

Step 1: Loading the Palette Entries

[This is preliminary documentation and subject to change.]

The following code in DDEx5 loads the palette entries with the values in the lower half of the All.bmp file (the part of the bitmap that contains the red donuts):

```
// First, set all colors as unused.
for(i=0; i<256; i++)
{
   torusColors[i] = 0;
}

// Lock the surface and scan the lower part (the torus area),
// and keep track of all the indexes found.
ddsd.dwSize = sizeof(ddsd);
while (lpDDSOne->Lock(NULL, &ddsd, 0, NULL) == DDERR_WASSTILLDRAWING)
```

```
    ;

  // Search through the torus frames and mark used colors.
  for(y=480; y<480+384; y++)
  {
    for(x=0; x<640; x++)
    {
      torusColors[((BYTE *)ddsd.lpSurface)[y*ddsd.lPitch+x]] = 1;
    }
  }

  lpDDSOne->Unlock(NULL);
```

The *torusColors* array is used as an indicator of the color index of the palette used in the lower half of the All.bmp file. Before it is used, all of the values in the *torusColors* array are reset to 0. The off-screen buffer is then locked in preparation for determining if a color index value is used.

The *torusColors* array is set to start at row 480 and column 0 of the bitmap. The color index value in the array is determined by the byte of data at the location in memory where the bitmap surface is located. This location is determined by the **lpSurface** member of the **DDSURFACEDESC** structure, which is pointing to the memory location corresponding to row 480 and column 0 of the bitmap (y × **lPitch** + x). The location of the specific color index value is then set to 1. The y-value (row) is multiplied by the **lPitch** value (found in the **DDSURFACEDESC** structure) to get the actual location of the pixel in linear memory.

The color index values that are set in *torusColors* will be used later to determine which colors in the palette are rotated. Because there are no common colors between the background and the red donuts, only those colors associated with the red donuts are rotated. If you want to check whether this is true or not, just remove the "*ddsd.lPitch" from the array and see what happens when you recompile and run the program. (Without multiplying y × **lPitch**, the red donuts are never reached and only the colors found in the background are indexed and later rotated.) For more information about width and pitch, see Width vs. Pitch.

Step 2: Rotating the Palettes

[This is preliminary documentation and subject to change.]

The updateFrame sample function in DDEx5 works in much the same way as it did in Tutorial 4 (DDEx4). It first blits the background into the back buffer, and then it blits the three donuts in the foreground. However, before it flips the surfaces, updateFrame changes the palette of the primary surface from the palette index that was created in the doInit function, as shown in the following code:

```
  // Change the palette.
  if(lpDDPal->GetEntries(0, 0, 256, pe) != DD_OK)
  {
```

```
    return;
}

for(i=1; i<256; i++)
{
    if(!torusColors[i])
    {
      continue;
    }
    pe[i].peRed = (pe[i].peRed+2) % 256;
    pe[i].peGreen = (pe[i].peGreen+1) % 256;
    pe[i].peBlue = (pe[i].peBlue+3) % 256;
}

if(lpDDPal->SetEntries(0, 0, 256, pe) != DD_OK)
{
    return;
}
```

The **IDirectDrawPalette::GetEntries** method in the first line queries palette values from a DirectDrawPalette object. Because the palette entry values pointed to by *pe* should be valid, the method will return DD_OK and continue. The loop that follows checks *torusColors* to determine if the color index was set to 1 during its initialization. If so, the red, green, and blue values in the palette entry pointed to by *pe* are rotated.

After all of the marked palette entries are rotated, the **IDirectDrawPalette::SetEntries** method is called to change the entries in the DirectDrawPalette object. This change takes place immediately if you are working with a palette set to the primary surface.

With this done, the surfaces are subsequently flipped.

Tutorial 6: Using Overlay Surfaces

[This is preliminary documentation and subject to change.]

This tutorial shows you, step by step, how to use DirectDraw and hardware supported overlay surfaces in your applications. The tutorial is written around the Mosquito sample application included with the DirectX SDK samples. Mosquito is a simple application that uses a flipping chain of overlay surfaces to display an animated bitmap on the desktop without blitting to the primary surface. The sample adjusts the characteristics of the overlay surface as needed to accommodate for hardware limitations.

The Mosquito sample application performs the following steps (complex tasks are divided into smaller sub-steps):

- Step 1: Creating a Primary Surface

- Step 2: Testing for Hardware Overlay Support
- Step 3: Creating an Overlay Surface
- Step 4: Displaying the Overlay Surface
- Step 5: Updating the Overlay Display Position
- Step 6: Hiding the Overlay Surface

Step 1: Creating a Primary Surface

[This is preliminary documentation and subject to change.]

To prepare for using overlay surfaces, you must first initialize DirectDraw and create a primary surface over which the overlay surface will be displayed. Mosquito creates a primary surface with the following code:

```
// Zero-out the structure and set the dwSize member.
ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);

// Set flags and create a primary surface.
ddsd.dwFlags = DDSD_CAPS;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
ddrval = g_lpdd->CreateSurface(&ddsd, &g_lpddsPrimary, NULL );
```

The preceding example begins by initializing the **DDSURFACEDESC** structure it will use. It then sets the flags appropriate to create a primary surface and creates it by calling the **IDirectDraw2::CreateSurface** method. For the call, the first parameter is a pointer to a **DDSURFACEDESC** structure that describes the surface to be created. The second parameter is a pointer to a variable that will receive an **IDirectDrawSurface** interface pointer if the call succeeds. The last parameter is set to NULL to indicate that no COM aggregation is taking place.

Step 2: Testing for Hardware Overlay Support

[This is preliminary documentation and subject to change.]

After initializing DirectDraw, you need to verify that the device supports overlay surfaces. Because DirectDraw doesn't emulate overlays, if the hardware device driver doesn't support them, you can't continue. You can test for overlay support by retrieving the device driver capabilities with the **IDirectDraw2::GetCaps** method. After the call, look for the presence of the DDCAPS_OVERLAY flag in the **dwFlags** member of the associated **DDCAPS** structure. If the flag is present, then the display hardware supports overlays; if not, you can't use overlay surfaces with that device.

The following example, taken from the Mosquito sample application, shows how to test for hardware overlay support:

```
BOOL AreOverlaysSupported()
```

```
{
    DDCAPS  capsDrv;
    HRESULT ddrval;

    // Get driver capabilities to determine Overlay support.
    ZeroMemory(&capsDrv, sizeof(capsDrv));
    capsDrv.dwSize = sizeof(capsDrv);

    ddrval = g_lpdd->GetCaps(&capsDrv, NULL);
    if (FAILED(ddrval))
        return FALSE;

    // Does the driver support overlays in the current mode?
    // (Currently the DirectDraw emulation layer does not support overlays.
    // Overlay related APIs will fail without hardware support).
    if (!(capsDrv.dwCaps & DDCAPS_OVERLAY))
        return FALSE;

    return TRUE;
}
```

The preceding example calls the **IDirectDraw2::GetCaps** method to retrieve device driver capabilities. The first parameter for the call is the address of a **DDCAPS** that will be filled with information describing the device driver's capabilities. Because the application doesn't need information about emulation capabilities, the second parameter is set to NULL.

After retrieving the driver capabilities, the example checks the **dwCaps** member for the presence of the DDCAPS_OVERLAY flag using a logical **AND** operation. If the flag isn't present, the example returns FALSE to indicate failure. Otherwise, the example returns TRUE to indicate that the device driver supports overlay surfaces.

In your code, this might be a good time for you to check the **dwMaxVisibleOverlays** and **dwCurrentVisibleOverlays** members in the **DDCAPS** structure to ensure that no other overlay surfaces are in use by other applications.

Step 3: Creating an Overlay Surface

[This is preliminary documentation and subject to change.]

Now that you know that the driver supports overlay surfaces, you can try to create one. Because there is no standard dictating how devices must support overlay surfaces, you can't count on being able to create overlays of any particular size or pixel format. Additionally, you can't expect to succeed in creating an overlay surface on the first try. Therefore, be prepared to attempt creation multiple times starting with the most desirable characteristics, falling back on less desirable (but possibly less hardware intensive) configurations until one works.

Note

You can call the **IDirectDraw2::GetFourCCCodes** method to retrieve a list of FOURCC codes that describe non-RGB pixel formats that the driver will likely support for overlay surfaces. However, in you want to try using RGB overlay surfaces, it is recommended that you attempt to create surfaces in various common RGB formats, falling back on another format if you fail.

The Mosquito sample follows a "best case to worst case" philosophy when creating an overlay surface. Mosquito first tries to create a triple-buffered page flipping complex overlay surface. If the creation attempt fails, the sample tries the configuration with other common pixel formats. The following code fragment shows how this can be done:

```
ZeroMemory(&ddsdOverlay, sizeof(ddsdOverlay));
ddsdOverlay.dwSize = sizeof(ddsdOverlay);

ddsdOverlay.dwFlags= DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH |
                DDSD_BACKBUFFERCOUNT| DDSD_PIXELFORMAT;
ddsdOverlay.ddsCaps.dwCaps = DDSCAPS_OVERLAY | DDSCAPS_FLIP |
                     DDSCAPS_COMPLEX | DDSCAPS_VIDEOMEMORY;
ddsdOverlay.dwWidth  =320;
ddsdOverlay.dwHeight =240;
ddsdOverlay.dwBackBufferCount=2;

// Try to create an overlay surface using one of the pixel formats in our
// global list.
i=0;
do{
ddsdOverlay.ddpfPixelFormat=g_ddpfOverlayFormats[i];
  // Try to create the overlay surface
  ddrval = g_lpdd->CreateSurface(&ddsdOverlay, &g_lpddsOverlay, NULL);
} while( FAILED(ddrval) && (++i < NUM_OVERLAY_FORMATS) );
```

The preceding example sets the flags and values within a **DDSURFACEDESC** structure to reflect a triple-buffered page flipping complex overlay surface. Then, the sample performs a loop during which it attempts to create the requested surface in a variety of common pixel formats, in order of most desirable to least desirable pixel formats. If the attempt succeeds, the loop ends. If all the attempts fail, it's likely that the display hardware doesn't have enough memory to support a triple-buffered scheme or that it doesn't support flipping overlay surfaces. In this case, the sample falls back on a less desirable configuration using a single non-flipping overlay surface, as shown in the following example:

```
// If we failed to create a triple buffered complex overlay surface, try
// again with a single non-flippable buffer.
if(FAILED(ddrval))
{
   ddsdOverlay.dwBackBufferCount=0;
```

```
    ddsdOverlay.ddsCaps.dwCaps=DDSCAPS_OVERLAY |
DDSCAPS_VIDEOMEMORY;
    ddsdOverlay.dwFlags= DDSD_CAPS|DDSD_HEIGHT|DDSD_WIDTH|
DDSD_PIXELFORMAT;

    // Try to create the overlay surface
    ddrval = g_lpdd->CreateSurface(&ddsdOverlay, &g_lpddsOverlay, NULL);
    i=0;
    do{
       ddsdOverlay.ddpfPixelFormat=g_ddpfOverlayFormats[i];
       ddrval = g_lpdd->CreateSurface(&ddsdOverlay, &g_lpddsOverlay, NULL);
    } while( FAILED(ddrval) && (++i < NUM_OVERLAY_FORMATS) );

    // We couldn't create an overlay surface.  Exit, returning failure.
    if (FAILED(ddrval))
       return FALSE;
  }
```

The previous code resets the flags and values in the **DDSURFACEDESC** structure to reflect a single non-flipping overlay surface. Again, the example loops through pixel formats attempting to create the surfaces, stopping the loop if an attempt succeeded. If the attempts still didn't work, the sample returns FALSE to indicate failure.

After you've successfully created your overlay surface or surfaces, you can load bitmaps onto them in preparation for display.

Step 4: Displaying the Overlay Surface

[This is preliminary documentation and subject to change.]

After creating your overlay surface, you can display it. Often, display hardware imposes alignment restrictions on the position and pixel width of the rectangles you use to display the overlay. Additionally, you will often need to account for a minimum required stretch factor by adjusting the width of the destination rectangle in order to successfully display the overlay surface. The Mosquito sample performs the following tasks to prepare and display the overlay surface:

- Step 4.1: Determining the Minimum Display Requirements
- Step 4.2: Setting Up the Source and Destination Rectangles
- Step 4.3: Displaying the Overlay Surface

Step 4.1: Determining the Minimum Display Requirements

[This is preliminary documentation and subject to change.]

Most display hardware imposes restrictions on displaying overlay surfaces. You must carefully meet these restrictions in order to successfully display an overlay surface. You can retrieve information about these restrictions by calling the **IDirectDraw2::GetCaps** method. The **DDCAPS** structure that the method fills contains information about overlay capabilities and their usage restrictions. Hardware restrictions vary, so always look at the flags included in the **dwFlags** member to determine which restrictions apply to you.

The Mosquito sample starts by retrieving the hardware capabilities, then takes action based upon the minimum stretch factor, as shown in the following code fragment:

```
    // Get driver capabilities
    ddrval = g_lpdd->GetCaps(&capsDrv, NULL);
    if (FAILED(ddrval))
       return FALSE;

    // Check the minimum stretch and set the local variable accordingly.
    if(capsDrv.dwCaps & DDCAPS_OVERLAYSTRETCH)
       uStretchFactor1000 = (capsDrv.dwMinOverlayStretch>1000) ?
 capsDrv.dwMinOverlayStretch : 1000;
    else
       uStretchFactor1000 = 1000;
```

The preceding code calls **GetCaps** to retrieve only the hardware capabilities. For this call, the first parameter is a pointer the **DDCAPS** structure that will be filled with the capability information for the device driver, and the second parameter is NULL to indicate that emulation information is not to be retrieved.

The example retains the minimum stretch factor in a temporary variable for use later. (Keep in mind that stretch factors are reported multiplied by 1000, so 1300 really means 1.3.) If the driver reports a value greater than 1000, it means that the driver requires that all destination rectangles must be stretched along the x-axis by a ratio of the reported value. For example, if the driver reports a stretch factor 1.3 and the source rectangle is 320 pixels wide, the destination rectangle must be at least 416 pixels wide. If the driver reports a stretch factor less than 1000, it means that the driver can display overlays smaller than the source rectangle, but can also stretch the overlay if desired.

Next, the sample examines values describing the driver's size alignment restrictions, as shown in the following example:

```
    // Grab any alignment restrictions and set the local variables acordingly.
    uSrcSizeAlign = (capsDrv.dwCaps & DDCAPS_ALIGNSIZESRC)?
 capsDrv.dwAlignSizeSrc:0;
    uDestSizeAlign= (capsDrv.dwCaps & DDCAPS_ALIGNSIZESRC)?
 capsDrv.dwAlignSizeDest:0;
```

The sample uses more temporary variables to hold the reported size alignment restrictions taken from the **dwAlignSizeSrc** and **dwAlignSizeDest** members. These

values provide information about pixel width alignment restrictions and are needed when setting the dimensions of the source and destination rectangles to reflect these restrictions later. Source and destination rectangles must have a pixel width that is a multiple of the values in these members.

Last, the sample examines the value that describes the destination rectangle boundary alignment:

```
// Set the "destination position alignment" global so we won't have to
// keep calling GetCaps() every time we move the overlay surface.
if (capsDrv.dwCaps & DDCAPS_ALIGNBOUNDARYDEST)
   g_dwOverlayXPositionAlignment = capsDrv.dwAlignBoundaryDest;
else
   g_dwOverlayXPositionAlignment = 0;
```

The preceding code uses a global variable to hold the value for the destination rectangle's boundary alignment, as taken from the **dwAlignBoundaryDest** member. This value will be used when the program repositions the overlay later. (For details, see Step 5: Updating the Overlay Display Position) You must set the x-coordinate of the destination rectangle's top left corner to be aligned with this value, in pixels. That is, if the value specified is 4, you can only specify destination rectangles whose top-left corner has an x-coordinate at pixels 0, 4, 8, 12, and so on. The Mosquito application initially displays the overlay at 0,0, so alignment compliance is assumed and the sample doesn't need to retrieve the restriction information until after displaying the overlay the first time. Your implementation might vary, so you will probably need to check this information and adjust the destination rectangle before displaying the overlay.

Step 4.2: Setting Up the Source and Destination Rectangles

[This is preliminary documentation and subject to change.]

After retrieving the driver's overlay restrictions you should set the values for your source and destination rectangles accordingly, assuring that you will be able to successfully display the overlay. The following sample from the Mosquito sample application starts by setting the characteristics of the source rectangle:

```
// Set initial values in the source RECT.
rs.left=0; rs.top=0;
rs.right = 320;
rs.bottom = 240;

// Apply size alignment restrictions, if necessary.
if (capsDrv.dwCaps & DDCAPS_ALIGNSIZESRC && uSrcSizeAlign)
   rs.right -= rs.right % uSrcSizeAlign;
```

The preceding code sets initial values for the surface to include the dimensions of the entire surface. If the device driver requires size alignment for the source rectangle,

the example adjusts the source rectangle to conform. The example adjusts the width of the source rectangle to be narrower than the original size because the width cannot be expanded without completely recreating the surface. However, your code could just as easily start with a smaller rectangle and widen the rectangle to meet driver restrictions.

After the dimensions of the source rectangle are set and conform with hardware restrictions, you need to set and adjust the dimensions of the destination rectangle. This process requires a little more work because the rectangle might need to be stretched first, then adjusted to meet size alignment restrictions. The following code performs the task of accounting for the minimum stretch factor:

```
// Set up the destination RECT, starting with the source RECT values.
// We use the source RECT dimensions instead of the surface dimensions in
// case they differ.
rd.left=0; rd.top=0;
rd.right  = (rs.right*uStretchFactor1000+999)/1000;
// (Adding 999 avoids integer truncation problems.)

// (This isn't required by DDraw, but we'll stretch the
//  height, too, to maintain aspect ratio).
rd.bottom = rs.bottom*uStretchFactor1000/1000;
```

The preceding code sets the top left corner of the destination rectangle to the top left corner of the screen, then sets the width to account for the minimum stretch factor. While adjusting for the stretch factor, note that the example adds 999 to the product of the width and stretch factor. This is done to prevent integer truncation that could result in a rectangle that isn't as wide as the minimum stretch factor requires. For more information, see Minimum and Maximum Stretch Factors. Also, after the example stretches the width, it stretches the height. Stretching the height isn't required, but was done to preserve the bitmap's aspect ratio and avoid a distorted appearance.

After stretching the destination rectangle, the example continues by adjusting it to conform to size alignment restrictions as follows:

```
// Adjust the destination RECT's width to comply with any imposed
// alignment restrictions.
if (capsDrv.dwCaps & DDCAPS_ALIGNSIZEDEST && uDestSizeAlign)
   rd.right = (int)((rd.right+uDestSizeAlign-1)/uDestSizeAlign)*uDestSizeAlign;
```

The example checks the capabilities flags to see if the driver imposes destination size alignment restrictions. If so, the destination rectangle's width is increased by enough pixels to meet alignment restrictions. Note that the rectangle is adjusted by expanding the width, not by decreasing it. This is done because decreasing the width could cause the destination rectangle to be smaller than is required by the minimum stretch factor, consequently causing attempts to display the overlay surface to fail.

Step 4.3: Displaying the Overlay Surface

[This is preliminary documentation and subject to change.]

After you've set up the source and destination rectangles, you can display the overlay for the first time. If you've prepared correctly, this will be simple. The Mosquito sample uses the following code to initially display the overlay:

```
    // Set the flags we'll send to UpdateOverlay
    dwUpdateFlags = DDOVER_SHOW | DDOVER_DDFX;

    // Does the overlay hardware support source color keying?
    // If so, we can hide the black background around the image.
    // This probably won't work with YUV formats
    if (capsDrv.dwCKeyCaps & DDCKEYCAPS_SRCOVERLAY)
       dwUpdateFlags |= DDOVER_KEYSRCOVERRIDE;

    // Create an overlay FX structure so we can specify a source color key.
    // This information is ignored if the DDOVER_SRCKEYOVERRIDE flag isn't set.
    ZeroMemory(&ovfx, sizeof(ovfx));
    ovfx.dwSize = sizeof(ovfx);

    ovfx.dckSrcColorkey.dwColorSpaceLowValue=0; // Specify black as the color key
    ovfx.dckSrcColorkey.dwColorSpaceHighValue=0;

    // Call UpdateOverlay() to displays the overlay on the screen.
    ddrval = g_lpddsOverlay->UpdateOverlay(&rs, g_lpddsPrimary, &rd,
  dwUpdateFlags, &ovfx);
    if(FAILED(ddrval))
       return FALSE;
```

The preceding example starts by setting the DDOVER_SHOW and DDOVER_DDFX flags in the *dwUpdateFlags* temporary variable, indicating that the overlay is to be displayed for the first time, and that the hardware should use the effects information included in an associated **DDOVERLAYFX** structure to do so. Next, the example checks a previously existing **DDCAPS** structure to determine if overlay source color keying is supported. If it is, the DDOVER_KEYSRCOVERRIDE is included in the *dwUpdateFlags* variable to take advantage of source color keying and the example sets color key values accordingly.

After preparation is complete, the example calls the **IDirectDrawSurface3::UpdateOverlay** method to display the overlay. For the call, the first and third parameters are the addresses of the adjusted source and destination rectangles. The second parameter is the address of the primary surface over which the overlay will be displayed. The fourth parameter consists of the flags placed in the previously prepared *dwUpdateFlags* variable, and the fifth parameter is the address of **DDOVERLAYFX** structure whose members were set to match those flags.

If the hardware only supports one overlay surface and that surface is in use, the **UpdateOverlay** method fails, returning DDERR_OUTOFCAPS. Additionally, if **UpdateOverlay** fails, you might try increasing the width of the destination rectangle to accommodate for the possibility that the hardware incorrectly reported a minimum stretch factor that was too small. However, this rarely occurs and Mosquito simply fails if **UpdateOverlay** doesn't succeed.

Step 5: Updating the Overlay Display Position

[This is preliminary documentation and subject to change.]

After displaying the overlay surface, you might not need to do anything else. However, some software might need to reposition the overlay surface. The Mosquito sample uses the **IDirectDrawSurface3::SetOverlayPosition** method to reposition the overlay, as shown in the following example:

```
    // Set x- and y-coordinates
    .
    .
    .
    // We need to check for any alignment restrictions on the x-position
    // and align it if necessary.
    if (g_dwOverlayXPositionAlignment)
       dwXAligned = g_nOverlayXPos - g_nOverlayXPos %
g_dwOverlayXPositionAlignment;
    else
       dwXAligned = g_nOverlayXPos;

    // Set the overlay to its new position.
    ddrval = g_lpddsOverlay->SetOverlayPosition(dwXAligned, g_nOverlayYPos);
    if (ddrval == DDERR_SURFACELOST)
    {
       if (!RestoreAllSurfaces())
          return;
    }
```

The preceding example starts by aligning the rectangle to meet any destination rectangle boundary alignment restrictions that might exist. The global variable that it checks, *g_dwOverlayXPositionAlignment*, was set earlier to equal the value reported in the **dwAlignBoundaryDest** member of the **DDCAPS** structure when the application previously called the **IDirectDraw2::GetCaps** method. (For details, see Step 4.1: Determining the Minimum Display Requirements). If destination alignment restrictions exist, the example adjusts the new x-coordinate to be pixel-aligned accordingly. Failing to meet this requirement will cause the overlay surface not to be displayed.

After making any requisite adjustments to the new x-coordinate, the example calls **IDirectDrawSurface3::SetOverlayPosition** method to reposition the overlay. For

the call, the first parameter is the aligned x-coordinate, and the second parameter is the new y-coordinate. These values represent the new location of the overlay's top-left corner. Width and height information are not accepted, nor are they needed because DirectDraw already knows the dimensions of the surface from the **IDirectDrawSurface3::UpdateOverlay** method made to initially display the overlay. If the call fails because one or more surfaces were lost, the example calls an application-defined function to restore them and reload their bitmaps.

Note

Take care not to use coordinates too close to the bottom or right edge of the target surface. The **IDirectDraw2::SetOverlayPosition** method does not perform clipping for you; using coordinates that would potentially make the overlay run off the edge of the target surface will cause the method to fail, returning DDERR_INVALIDPOSITION.

Step 6: Hiding the Overlay Surface

[This is preliminary documentation and subject to change.]

When you do not need the overlay surface anymore, or if you simply want to remove it from view, you can hide the surface by calling the **IDirectDrawSurface3::UpdateOverlay** method with the appropriate flags. Mosquito hides the overlay in preparation for closing the application using the following code:

```
void DestroyOverlay()
{
    if (g_lpddsOverlay){
        // Use UpdateOverlay() with the DDOVER_HIDE flag to remove an overlay
        // from the display.
        g_lpddsOverlay->UpdateOverlay(NULL, g_lpddsPrimary, NULL, DDOVER_HIDE,
NULL);
        g_lpddsOverlay->Release();
        g_lpddsOverlay=NULL;
    }
}
```

When the preceding example calls **IDirectDrawSurface3::UpdateOverlay**, it specifies NULL for the source and destination rectangles, because they are irrelevant when hiding the overlay. Similarly, the example uses NULL in the fifth parameter because overlay effects aren't being used. The second parameter is a pointer to the target surface. Lastly, the example uses the DDOVER_HIDE flag in the fourth parameter to indicate that the overlay will be removed from view.

After the example hides the overlay, the example releases its **IDirectDrawSurface3** interface and invalidates its global variable by setting it to NULL. For the purposes of the Mosquito sample application, the overlay surface is no longer needed. If you

still need the overlay surface for later, you could simply hide the overlay without releasing it, then redisplay it whenever you require.

DirectDraw Visual Basic Tutorials

[This is preliminary documentation and subject to change.]

This section contains a series of tutorials, each of which provides step-by-step instructions for implementing a simple DirectDraw application. These tutorials use many of the DirectDraw sample files that are provided with this SDK. These samples demonstrate how to set up DirectDraw, and how to use the DirectDraw methods to perform common tasks:

- Tutorial 1: Blitting to the Screen
- Tutorial 2: Using Transparency
- Tutorial 3: Using Full Screen Features
- Tutorial 4: Blitting to Areas of the Screen
- Tutorial 5: Enumerating DirectDraw Devices

Tutorial 1: Blitting to the Screen

[This is preliminary documentation and subject to change.]

This first tutorial deals with blitting a bitmap to the display adapter. The term blit is shorthand for "bit block transfer," which is the process of transferring blocks of data from one place in memory to another. Graphics programmers use blitting to transfer graphics from one place in memory to another. Blits are often used to perform sprite animation.

To use DirectDraw, you first create an instance of the DirectDraw object, which represents the display adapter on the computer. You then use methods to manipulate the object. In addition, you need to create one or more instances of a DirectDrawSurface object to be able to display your application on a graphics surface.

To demonstrate this, the Tutorial 1 - Blitting to the Screen sample included in this SDK performs the following steps:

- Step 1: Creating the Form
- Step 2: Declaring Module Level Variables
- Step 2: Initializing Variables

Step 1: Creating the Form

[This is preliminary documentation and subject to change.]

The Tutorial 1 - blitting sample is a Standard EXE project with a picture box control placed on the form. The picture box control is placed with the top left corner of the picture box with the top left corner of the form. This picture box is used to display the bitmap image of the application and is named **Picture1**.

Step 2: Declaring Module Level Variables

[This is preliminary documentation and subject to change.]

The first step of coding DirectX application written in Visual Basic is to create the **DirectX7** Class. This object contains the methods necessary to create the starting objects of all the DirectX components including DirectDraw, DirectSound, Direct3D Immediate Mode, Direct3D Retained Mode, DirectInput and DirectPlay. Information on Direct3D Retained Mode can be found in the DirectX Media for Visual Basic node of this documentation.

The **DirectX7** class is the top level class of the DxVBLib type library and an object of this class is created with the statement:

```
Dim objDX As New DirectX7
```

Additional module level variables declarations found in the Tutorial 1 - blitting sample are for a DirectDraw object, DirectDrawSurface objects, DirectDraw surface description types, and a Boolean variable used to hold initialization information.

Note that in this sample we have set the Option Explicit flag and have an external Win32 procedure declaration.

Step 3: Initializing Variables

[This is preliminary documentation and subject to change.]

The first procedure called from the Form_Load event is the init procedure. This procedure creates the DirectDraw object. This is accomplished by invoking the **DirectX7.DirectDrawCreate** method of the DirectX7 object and setting the returned object to DirectDraw which we declared as an object variable of class DirectDraw4. In the Tutorial 1 - blitting sample, this is done with the statement:

```
Set objDD = objDX.DirectDrawCreate("")
```

This method takes only one string argument and passing an empty string specifies the active display driver.

Next you must specify the behavior of the application by calling the **DirectDraw4.SetCooperativeLevel** method of the DirectDraw object. The Tutorial 1 - blitting sample is run as a regular windowed application and this is done with the statement:

```
Call objDD.SetCooperativeLevel (Me.hwnd, DDSCL_NORMAL)
```

You are now ready to start creating surfaces. Before you actually create the surface object, you need to create a surface description by setting the members of the **DDSURFACEDESC2** type. One of the members of this type is **ddscaps**, a nested type, and by setting the **lFlags** member of the **DDSURFACEDESC2** to DDSD_CAPS, you are stating that the **ddscaps** member is valid in this type. This is done with the statement:

```
ddsd1.lFlags = DDSD_CAPS
```

Next you need to specify that this type description is for a primary surface, which is done with the statement:

```
ddsd1.ddsCaps.lCaps = DDSCAPS_PRIMARYSURFACE
```

After creating the surface description, you actually create the surface object by invoking the CreateSurface method from the DirectDraw object with the set surface description as an argument. This is done in the Tutorial 1 - blitting sample with the statement:

```
Set objDDPrimSurf = objDD.CreateSurface(ddsd1)
```

The surface object creation steps are repeated for a second surface which has the DDSCAPS_OFFSCREENPLAIN flag set to specify that this surface is any off-screen surface that is not an overlay, texture, z-buffer, front-buffer, back-buffer, or alpha surface. It is used to identify plain surfaces. Then the CreateSurfaceFromFile method is invoked from the DirectDraw object. This method creates the surface object and loads a bitmap onto the surface. These steps are shown with the statements:

```
ddsd2.lFlags = DDSD_CAPS
ddsd2.ddsCaps.lCaps = DDSCAPS_OFFSCREENPLAIN
Set objDDSurf = objDD.CreateSurfaceFromFile("lake.bmp", ddsd2)
```

After initializing all the variables and objects we set the bInit variable to True and call the blt procedure.

Step 4: Blitting the Surface

[This is preliminary documentation and subject to change.]

So far in the Tutorial 1 - Blitting to the Screen sample you have created a primary surface and a off-screen surface which has a loaded bitmap. To display the bitmap on the screen, you must blit the off-screen surface to the primary surface. The blit method of the DirectDraw surface object takes four arguments and returns a Long indicating the success or failure of the blit. Two of the arguments of are type **RECT** which specify the bounding rectangles of the destination surface and the source surface.

The coordinates of the source rectangle are obtained from the **lHeight** and the **lWidth** members of the off-screen surface description. There are a few extra steps in obtaining the destination rectangle. First of all, since Visual Basic uses twips for screen measurement and DirectX uses pixels, the dimensions of the Visual Basic picture box control must be converted to pixels before setting the destination rectangle. This is accomplished by setting the ScaleMode property of the form to Pixels and then making the width and height of the picture box equal to the **ScaleWidth** and **ScaleHeight** of the form. In the Tutorial 1 - blitting sample, this is done with the statements:

```
Picture1.Width = Me.ScaleWidth
Picture1.Height = Me.ScaleHeight
```

These statements are in the Form_Resize procedure and are executed when the form is initially displayed and resized.

Furthermore, the destination **RECT** type is filled when the Win32 function GetWindowRect is called.

```
Lastly the blit is perform with the blt method of the primary surface object:
ddrval = objDDPrimSurf.blt(r1, objDDSurf, r2, DDBLT_WAIT)
```

The result of the method is a **Long** that is stored in **ddrval**. You can check this variable for success or failure. The last argument of the above method is the DDBLT_WAIT flag which tells DirectDraw to wait if the blitter is busy and blit the surface when it becomes available.

Tutorial 2: Using Transparency

[This is preliminary documentation and subject to change.]

<To be written>

Tutorial 3: Using Full Screen Features

[This is preliminary documentation and subject to change.]

<To be written>

Tutorial 4: Blitting to Areas of the Screen

[This is preliminary documentation and subject to change.]

<To be written>

Tutorial 5: Enumerating DirectDraw Devices

[This is preliminary documentation and subject to change.]

<To be written>

DirectDraw Reference

[This is preliminary documentation and subject to change.]

This section contains reference information for the application programming interface (API) elements provided by DirectDraw® in C/C++ and Visual Basic. Reference material is organized by language:

- DirectDraw C/C++ Reference
- DirectDraw Visual Basic Reference

DirectDraw C/C++ Reference

[This is preliminary documentation and subject to change.]

This section contains reference information for the API elements that DirectDraw provides. Reference material is divided into the following categories:

- Interfaces
- Functions
- Callback Functions
- Structures
- Return Values
- Pixel Format Masks
- Four Character Codes (FOURCC)

Interfaces

[This is preliminary documentation and subject to change.]

This section contains reference information about the interfaces used with the DirectDraw component. The following interfaces are covered:

- **IDDVideoPortContainer**
- **IDirectDraw4**

- **IDirectDrawClipper**
- **IDirectDrawColorControl**
- **IDirectDrawGammaControl**
- **IDirectDrawPalette**
- **IDirectDrawSurface4**
- **IDirectDrawVideoPort**

IDDVideoPortContainer

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDDVideoPortContainer** interface to create and manipulate DirectDrawVideoPort objects. You retrieve a pointer to this interface by calling the **IUnknown::QueryInterface** method of a DirectDraw object, specifying the IID_IDDVideoPortContainer reference identifier.

The methods of the **IDDVideoPortContainer** interface can be organized into the following groups:

Creating objects	**CreateVideoPort**
Video ports	**EnumVideoPorts**
	QueryVideoPortStatus
Connections	**GetVideoPortConnectInfo**

The **IDDVideoPortContainer** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

You can use the **LPDDVIDEOPORTCONTAINER** data type to declare a variable that contains a pointer to an **IDDVideoPortContainer** interface. The Dvp.h header file declares the **LPDDVIDEOPORTCONTAINER** with the following code:

```
typedef struct IDDVideoPortContainer    FAR *LPDDVIDEOPORTCONTAINER;
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

IDDVideoPortContainer::CreateVideoPort

[This is preliminary documentation and subject to change.]

The **IDDVideoPortContainer::CreateVideoPort** method creates a DirectDrawVideoPort object.

```
HRESULT CreateVideoPort(
  DWORD dwFlags,
  LPDDVIDEOPORTDESC lpDDVideoPortDesc,
  LPDIRECTDRAWVIDEOPORT FAR *lplpDDVideoPort,
  IUnknown FAR *pUnkOuter
);
```

Parameters

dwFlags
Flags specifying video-port control options. This parameter can be one of the following flags, or NULL if control options are not needed:

DDVPCREATE_VBIONLY
The process only wants to control the VBI portion of the video stream.

DDVPCREATE_VIDEOONLY
The process only wants to control the non-VBI (video) portion of the video stream.

lpDDVideoPortDesc
Address of a **DDVIDEOPORTDESC** structure that describes the DirectDrawVideoPort object to be created.

lplpDDVideoPort
Address of a variable that will be filled with a pointer to the new DirectDrawVideoPort object's **IDirectDrawVideoPort** interface if the call succeeds.

pUnkOuter
Allows for future compatibility with COM aggregation features. Presently, however, this method will return an error if this parameter is anything but NULL.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_CURRENTLYNOTAVAIL
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCOOPERATIVELEVELSET
DDERR_OUTOFCAPS
DDERR_OUTOFMEMORY

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

IDDVideoPortContainer::EnumVideoPorts

[This is preliminary documentation and subject to change.]

The **IDDVideoPortContainer::EnumVideoPorts** method enumerates all of the video ports that the hardware exposes that are compatible with a provided video port description.

```
HRESULT EnumVideoPorts(
  DWORD dwFlags,
  LPDDVIDEOPORTCAPS lpDDVideoPortCaps,
  LPVOID lpContext,
  LPENUMVIDEOCALLBACK lpEnumVideoCallback
);
```

Parameters

dwFlags
Reserved for future use. This parameter must be zero.

lpDDVideoPortCaps
Pointer to a **DDVIDEOPORTCAPS** structure that will be checked against the available video ports. If this parameter is NULL, all video ports will be enumerated.

lpContext
Address of a caller-defined structure that will be passed to each enumeration member.

lpEnumVideoCallback

Address of the **EnumVideoCallback** function that will be called each time a match is found.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

IDDVideoPortContainer::GetVideoPortConnectInfo

[This is preliminary documentation and subject to change.]

The **IDDVideoPortContainer::GetVideoPortConnectInfo** method retrieves the connection information supported by all video ports.

```
HRESULT GetVideoPortConnectInfo(
  DWORD dwPortId,
  LPDWORD lpNumEntries,
  LPDDVIDEOPORTCONNECT lpConnectInfo
);
```

Parameters

dwPortId
Identifier of the video port for which the connection information will be retrieved.

lpNumEntries
Address of a variable containing the number of entries that the array at *lpConnectInfo* can hold. If this number is less than the total number of connections, the method fills the array with as many entries as will fit, sets the value at *lpNumEntries* to indicate the total number of connections, and returns DDERR_MOREDATA.

lpConnectInfo

Address of an array of **DDVIDEOPORTCONNECT** structures that will be filled with the connection options supported by the specified video port. If this parameter is NULL, the method sets *lpNumEntries* to indicate the total number of connections that the video port supports, then returns DD_OK.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_MOREDATA

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

IDDVideoPortContainer::QueryVideoPortStatus

[This is preliminary documentation and subject to change.]

The **IDDVideoPortContainer::QueryVideoPortStatus** method retrieves the status of a DirectDrawVideoPort object.

```
HRESULT QueryVideoPortStatus(
  DWORD dwPortId,
  LPDDVIDEOPORTSTATUS lpVPStatus
);
```

Parameters

dwPortId
Identifier of the video port for which the status information will be retrieved.

lpVPStatus
Address of a **DDVIDEOPORTSTATUS** structure that will be filled with information about the status of the specified video port.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_CURRENTLYNOTAVAIL
DDERR_EXCEPTION
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

IDirectDraw4

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirectDraw4** interface to create DirectDraw objects and work with system-level variables. This section is a reference to the methods of this interface. For a conceptual overview, see The DirectDraw Object.

The methods of the **IDirectDraw4** interface can be organized into the following groups:

Allocating memory	**Compact**
	Initialize
Cooperative levels	**SetCooperativeLevel**
	TestCooperativeLevel
Creating objects	**CreateClipper**
	CreatePalette
	CreateSurface
Device capabilities	**GetCaps**
Display modes	**EnumDisplayModes**

	GetDisplayMode
	GetMonitorFrequency
	RestoreDisplayMode
	SetDisplayMode
	WaitForVerticalBlank
Display status	**GetScanLine**
	GetVerticalBlankStatus
Miscellaneous	**GetAvailableVidMem**
	GetDeviceIdentifier
	GetFourCCCodes
Surface management	**DuplicateSurface**
	EnumSurfaces
	FlipToGDISurface
	GetGDISurface
	GetSurfaceFromDC
	RestoreAllSurfaces

The **IDirectDraw4** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **IDirectDraw4** interface extends the features of previous versions of the interface by offering methods enabling more flexible surface management than previous versions. Note that all of the surface-related methods in the **IDirectDraw4** interface accept slightly different parameters than their counterparts in the **IDirectDraw2** interface. Wherever an **IDirectDraw2** interface method might accept a **DDSURFACEDESC** structure and retrieve an **IDirectDrawSurface3** interface, the methods in **IDirectDraw4** accept a **DDSURFACEDESC2** structure and retrieve an **IDirectDrawSurface4** interface instead.

IDirectDraw4 introduces improved compliance with COM rules dictating the lifetimes of child objects. For more information, see Parent and Child Object Lifetimes.

You can use the **LPDIRECTDRAW**, **LPDIRECTDRAW2**, or **LPDIRECTDRAW4** data types to declare a variable that contains a pointer to an

IDirectDraw, **IDirectDraw2**, or **IDirectDraw4** interface. The Ddraw.h header file declares these data types with the following code:

```
typedef struct IDirectDraw     FAR *LPDIRECTDRAW;
typedef struct IDirectDraw2    FAR *LPDIRECTDRAW2;
typedef struct IDirectDraw4    FAR *LPDIRECTDRAW4;
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDraw4::Compact

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::Compact** method is not currently implemented.

HRESULT Compact();

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOEXCLUSIVEMODE
DDERR_SURFACEBUSY

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDraw4::CreateClipper

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::CreateClipper** method creates a DirectDrawClipper object.

```
HRESULT CreateClipper(
  DWORD dwFlags,
  LPDIRECTDRAWCLIPPER FAR *lplpDDClipper,
  IUnknown FAR *pUnkOuter
);
```

Parameters

dwFlags
This parameter is currently not used and must be set to 0.

lplpDDClipper
Address of a variable that will be set to a valid **IDirectDrawClipper** interface pointer if the call succeeds.

pUnkOuter
Allows for future compatibility with COM aggregation features. Presently, however, this method will return an error if this parameter is anything but NULL.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCOOPERATIVELEVELSET
DDERR_OUTOFMEMORY

Remarks

The DirectDrawClipper object can be attached to a DirectDrawSurface and used during **IDirectDrawSurface4::Blt**, **IDirectDrawSurface4::BltBatch**, and **IDirectDrawSurface4::UpdateOverlay** operations.

To create a DirectDrawClipper object that is not owned by a specific DirectDraw object, use the **DirectDrawCreateClipper** function.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::GetClipper, **IDirectDrawSurface4::SetClipper**

IDirectDraw4::CreatePalette

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::CreatePalette** method creates a DirectDrawPalette object for this DirectDraw object.

```
HRESULT CreatePalette(
  DWORD dwFlags,
  LPPALETTEENTRY lpDDColorArray,
  LPDIRECTDRAWPALETTE FAR *lplpDDPalette,
  IUnknown FAR *pUnkOuter
);
```

Parameters

dwFlags

One or more of the following flags:

DDPCAPS_1BIT
: Indicates that the index is 1 bit. There are two entries in the color table.

DDPCAPS_2BIT
: Indicates that the index is 2 bits. There are four entries in the color table.

DDPCAPS_4BIT
: Indicates that the index is 4 bits. There are 16 entries in the color table.

DDPCAPS_8BIT
: Indicates that the index is 8 bits. There are 256 entries in the color table.

DDPCAPS_8BITENTRIES
: Indicates that the index refers to an 8-bit color index. This flag is valid only when used with the DDPCAPS_1BIT, DDPCAPS_2BIT, or DDPCAPS_4BIT flag, and when the target surface is in 8 bpp. Each color entry is 1 byte long and is an index to a destination surface's 8-bpp palette.

DDPCAPS_ALPHA
: Indicates that the **peFlags** member of the associated **PALETTEENTRY** structure is to be interpreted as a single 8-bit alpha value (in addition to the **peRed**, **peGreen**, and **peBlue** members). A palette created with this flag can only be attached to a texture—a surface created with the DDSCAPS_TEXTURE capability flag.

DDPCAPS_ALLOW256
: Indicates that this palette can have all 256 entries defined.

DDPCAPS_INITIALIZE
: Initialize this palette with the colors in the color array passed at *lpDDColorArray*.

DDPCAPS_PRIMARYSURFACE
This palette is attached to the primary surface. Changing this palette's color table immediately affects the display unless DDPSETPAL_VSYNC is specified and supported.

DDPCAPS_PRIMARYSURFACELEFT
This palette is the one attached to the left eye primary surface. Changing this palette's color table immediately affects the left eye display unless DDPSETPAL_VSYNC is specified and supported.

DDPCAPS_VSYNC
This palette can have modifications to it synced with the monitors refresh rate.

lpDDColorArray
Address of an array of 2, 4, 16, or 256 **PALETTEENTRY** structures that will initialize this DirectDrawPalette object.

lplpDDPalette
Address of a variable that will be set to a valid **IDirectDrawPalette** interface pointer if the call succeeds.

pUnkOuter
Allows for future compatibility with COM aggregation features. Presently, however, this method will return an error if this parameter is anything but NULL.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_NOCOOPERATIVELEVELSET

DDERR_OUTOFMEMORY

DDERR_UNSUPPORTED

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDraw4::CreateSurface

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::CreateSurface** method creates a DirectDrawSurface object for this DirectDraw object.

```
HRESULT CreateSurface(
  LPDDSURFACEDESC2 lpDDSurfaceDesc2,
  LPDIRECTDRAWSURFACE4 FAR *lplpDDSurface,
  IUnknown FAR *pUnkOuter
);
```

Parameters

lpDDSurfaceDesc2
Address of a **DDSURFACEDESC2** structure that describes the requested surface. You should set any unused members of the **DDSURFACEDESC2** structure to zero before calling this method. A **DDSCAPS2** structure is a member of **DDSURFACEDESC2**.

lplpDDSurface
Address of a variable that will be set to a valid **IDirectDrawSurface4** interface pointer if the call succeeds.

pUnkOuter
Allows for future compatibility with COM aggregation features. Presently, however, this method will return an error if this parameter is anything but NULL.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INCOMPATIBLEPRIMARY
DDERR_INVALIDCAPS
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDPIXELFORMAT
DDERR_NOALPHAHW
DDERR_NOCOOPERATIVELEVELSET
DDERR_NODIRECTDRAWHW
DDERR_NOEMULATION
DDERR_NOEXCLUSIVEMODE
DDERR_NOFLIPHW
DDERR_NOMIPMAPHW
DDERR_NOOVERLAYHW
DDERR_NOZBUFFERHW

DDERR_OUTOFMEMORY
DDERR_OUTOFVIDEOMEMORY
DDERR_PRIMARYSURFACEALREADYEXISTS
DDERR_UNSUPPORTEDMODE

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDraw4::DuplicateSurface

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::DuplicateSurface** method duplicates a DirectDrawSurface object.

```
HRESULT DuplicateSurface(
  LPDIRECTDRAWSURFACE4 lpDDSurface,
  LPLPDIRECTDRAWSURFACE4 FAR *lplpDupDDSurface
);
```

Parameters

lpDDSurface
Address of the **IDirectDrawSurface4** interface for the surface to be duplicated.

lplpDupDDSurface
Address of a variable that will be filled with an **IDirectDrawSurface4** interface pointer for the newly duplicated DirectDrawSurface object.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_CANTDUPLICATE
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY
DDERR_SURFACELOST

Remarks

This method creates a new DirectDrawSurface object that points to the same surface memory as an existing DirectDrawSurface object. This duplicate can be used just like the original object. The surface memory is released after the last object referencing it is released. A primary surface, 3-D surface, or implicitly created surface cannot be duplicated.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDraw4::EnumDisplayModes

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::EnumDisplayModes** method enumerates all of the display modes the hardware exposes through the DirectDraw object that are compatible with a provided surface description.

```
HRESULT EnumDisplayModes(
  DWORD dwFlags,
  LPDDSURFACEDESC2 lpDDSurfaceDesc2,
  LPVOID lpContext,
  LPDDENUMMODESCALLBACK2 lpEnumModesCallback
);
```

Parameters

dwFlags

DDEDM_REFRESHRATES
Enumerates modes with different refresh rates. **IDirectDraw4::EnumDisplayModes** guarantees that a particular mode will be enumerated only once. This flag specifies whether the refresh rate is taken into account when determining if a mode is unique.

DDEDM_STANDARDVGAMODES
Enumerates Mode 13 in addition to the 320x200x8 Mode X mode.

lpDDSurfaceDesc2

Address of a **DDSURFACEDESC2** structure that will be checked against available modes. If the value of this parameter is NULL, all modes are enumerated.

lpContext

Address of an application-defined structure that will be passed to each enumeration member.

lpEnumModesCallback
Address of the **EnumModesCallback2** function that the enumeration procedure will call every time a match is found.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method enumerates the **dwRefreshRate** member of the **DDSURFACEDESC2** structure; the **IDirectDraw::EnumDisplayModes** method does not contain this capability. If you use the **IDirectDraw4::SetDisplayMode** method to set the refresh rate of a new mode, you must use **IDirectDraw4::EnumDisplayModes** to enumerate the **dwRefreshRate** member.

This method differs from its counterparts in former interfaces in that it accepts the address of an **EnumModesCallback2** function as a parameter rather than an **EnumModesCallback** function.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDraw4::GetDisplayMode, **IDirectDraw4::SetDisplayMode**, **IDirectDraw4::RestoreDisplayMode**

IDirectDraw4::EnumSurfaces

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::EnumSurfaces** method enumerates all of the existing or possible surfaces that meet the specified surface description.

HRESULT EnumSurfaces(

DWORD *dwFlags*,
LPDDSURFACEDESC2 *lpDDSD2*,
LPVOID *lpContext*,
LPDDENUMSURFACESCALLBACK2 *lpEnumSurfacesCallback*
);

Parameters

dwFlags

A combination of one search type flag and one matching flag. The search type flag determines how the method searches for matching surfaces; you can search for surfaces that can be created using the description in the *lpDDSD2* parameter or you can search for existing surfaces that already match that description. The matching flag determines whether the method enumerates all surfaces, only those that match, or only those that don't match the description in the *lpDDSD2* parameter.

Search type flags

DDENUMSURFACES_CANBECREATED
Enumerates the first surface that can be created and meets the search criterion. This flag can only be used with the DDENUMSURFACES_MATCH flag.

DDENUMSURFACES_DOESEXIST
Enumerates the already existing surfaces that meet the search criterion.

Matching flags

DDENUMSURFACES_ALL
Enumerates all of the surfaces that meet the search criterion. This flag can only be used with the DDENUMSURFACES_DOESEXIST search type flag.

DDENUMSURFACES_MATCH
Searches for any surface that matches the surface description.

DDENUMSURFACES_NOMATCH
Searches for any surface that does not match the surface description.

lpDDSD2

Address of a **DDSURFACEDESC2** structure that defines the surface of interest. This parameter can be NULL if *dwFlags* includes the DDENUMSURFACES_ALL flag.

lpContext

Address of an application-defined structure that will be passed to each enumeration member.

lpEnumSurfacesCallback

Address of the **EnumSurfacesCallback2** function the enumeration procedure will call every time a match is found.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

If the DDENUMSURFACES_CANBECREATED flag is set, this method attempts to temporarily create a surface that meets the search criterion.

When using the DDENUMSURFACES_DOESEXIST flag, note that an enumerated surface's reference count is incremented—if you are not going to use the surface, be sure to use **IDirectDrawSurface4::Release** to release it after each enumeration. If you will be using the surface, release it when it is no longer needed.

This method differs from its counterparts in previous interface versions in that it accepts a pointer to an **EnumSurfacesCallback2** function, rather than an **EnumSurfacesCallback** function.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

Enumerating Surfaces

IDirectDraw4::FlipToGDISurface

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::FlipToGDISurface** method makes the surface that GDI writes to the primary surface.

```
HRESULT FlipToGDISurface();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTFOUND

Remarks

This method can be called at the end of a page-flipping application to ensure that the display memory that GDI is writing to is visible to the user.

The method can also be used to make the GDI surface the primary surface, so that normal windows such as dialog boxes can be displayed in full-screen mode. The hardware must have the DDCAPS2_CANRENDERWINDOWED capability. For more information, see Displaying a Window in Full-Screen Mode

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDraw4::GetGDISurface

IDirectDraw4::GetAvailableVidMem

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::GetAvailableVidMem** method retrieves the total amount of display memory available and the amount of display memory currently free for a given type of surface.

```
HRESULT GetAvailableVidMem(
  LPDDSCAPS2 lpDDSCaps2,
  LPDWORD    lpdwTotal,
  LPDWORD    lpdwFree
);
```

Parameters

lpDDSCaps2
Address of a **DDSCAPS2** structure that indicates the hardware capabilities of the proposed surface.

lpdwTotal
Address of a variable that will be filled with the total amount of display memory available, in bytes. The value retrieved reflects the total video memory, less the video memory required for the primary surface and any private caches the display driver reserves.

lpdwFree
Address of a variable that will be filled with the amount of display memory currently free that can be allocated for a surface that matches the capabilities specified by the structure at *lpDDSCaps2*.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDCAPS
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NODIRECTDRAWHW

If NULL is passed to either *lpdwTotal* or *lpdwFree*, the value for that parameter is not returned.

Remarks

The following C++ example demonstrates using **IDirectDraw4::GetAvailableVidMem** to determine both the total and free display memory available for texture-map surfaces:

```
// For this example, the lpDD variable is a valid
// pointer to an IDirectDraw interface.
LPDIRECTDRAW4 lpDD4;
DDSCAPS2     ddsCaps2;
DWORD        dwTotal;
DWORD        dwFree;
HRESULT      hr;

hr = lpDD->QueryInterface(IID_IDirectDraw4, &lpDD4);
if (FAILED(hr))
   return hr;
```

```
// Initialize the structure.
ZeroMemory(&ddsCaps2, sizeof(ddsCaps2));

ddsCaps2.dwCaps = DDSCAPS_TEXTURE;
hr = lpDD4->GetAvailableVidMem(&ddsCaps2, &dwTotal, &dwFree);
if (FAILED(hr))
   return hr;
```

This method provides only a snapshot of the current display-memory state. The amount of free display memory is subject to change as surfaces are created and released. Therefore, you should use the free memory value only as an approximation. In addition, a particular display adapter card may make no distinction between two different memory types. For example, the adapter might use the same portion of display memory to store z-buffers and textures. So, allocating one type of surface (for example, a z-buffer) can affect the amount of display memory available for another type of surface (for example, textures). Therefore, it is best to first allocate an application's fixed resources (such as front and back buffers , and z-buffers) before determining how much memory is available for dynamic use (such as texture mapping).

This method was not implemented in the **IDirectDraw** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDraw4::GetCaps

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::GetCaps** method fills in the capabilities of the device driver for the hardware and the hardware-emulation layer (HEL).

```
HRESULT GetCaps(
  LPDDCAPS lpDDDriverCaps,
  LPDDCAPS lpDDHELCaps
);
```

Parameters

lpDDDriverCaps
Address of a **DDCAPS** structure that will be filled with the capabilities of the hardware, as reported by the device driver. Set this parameter to NULL if device driver capabilities are not to be retrieved.

lpDDHELCaps
Address of a **DDCAPS** structure that will be filled with the capabilities of the HEL. Set this parameter to NULL if HEL capabilities are not to be retrieved.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

You can only set one of the two parameters to NULL to exclude it. If you set both to NULL the method will fail, returning DDERR_INVALIDPARAMS.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDraw4::GetDeviceIdentifier

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::GetDeviceIdentifier** method obtains information about the driver. This method can be used, with caution, to recognize specific hardware installations in order to implement workarounds for poor driver/chipset behavior.

```
HRESULT GetDeviceIdentifier(
  LPDDDEVICEIDENTIFIER lpdddi,
  DWORD dwFlags
);
```

Parameters

lpdddi
Address of a **DDDEVICEIDENTIFIER** structure to receive information about the driver.

dwFlags
Flags specifying options. The following flag is defined:

DDGDI_GETHOSTIDENTIFIER
Causes the method to return information about the host (typically 2-D) adapter in a system equipped with a stacked secondary 3-D adapter. Such an adapter appears to the application as if it were part of the host adapter, but is

typically located on a separate card. When the *dwFlags* parameter is zero, the stacked secondary's information is returned, because this most accurately reflects the qualities of the DirectDraw object involved.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be DDERR_INVALIDPARAMS.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDraw4::GetDisplayMode

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::GetDisplayMode** method retrieves the current display mode.

```
HRESULT GetDisplayMode(
  LPDDSURFACEDESC2 lpDDSurfaceDesc2
);
```

Parameters

lpDDSurfaceDesc2
Address of a **DDSURFACEDESC2** structure that will be filled with a description of the surface.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTEDMODE

Remarks

An application should not save the information returned by this method to restore the display mode on clean-up. The application should use the **IDirectDraw4::RestoreDisplayMode** method to restore the mode on clean-up, thereby avoiding mode-setting conflicts that could arise in a multiprocess environment.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDraw4::SetDisplayMode, **IDirectDraw4::RestoreDisplayMode**, **IDirectDraw4::EnumDisplayModes**

IDirectDraw4::GetFourCCCodes

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::GetFourCCCodes** method retrieves the FOURCC codes supported by the DirectDraw object. This method can also retrieve the number of codes supported.

```
HRESULT GetFourCCCodes(
  LPDWORD lpNumCodes,
  LPDWORD lpCodes
);
```

Parameters

lpNumCodes
Address of a variable that contains the number of entries that the array pointed to by *lpCodes* can hold. If the number of entries is too small to accommodate all the codes, *lpNumCodes* is set to the required number and the array pointed to by *lpCodes* is filled with all that fits.

lpCodes
Address of an array of variables that will be filled with FOURCC codes supported by this DirectDraw object. If you specify NULL, *lpNumCodes* is set to the number of supported FOURCC codes and the method will return.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDraw4::GetGDISurface

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::GetGDISurface** method retrieves the DirectDrawSurface object that currently represents the surface memory that GDI is treating as the primary surface.

```
HRESULT GetGDISurface(
  LPDIRECTDRAWSURFACE4 FAR *lplpGDIDDSSurface4
);
```

Parameters

lplpGDIDDSSurface4
Address of a variable that will be filled with a pointer to the **IDirectDrawSurface4** interface for the surface that currently controls GDI's primary surface memory.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTFOUND

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDraw4::FlipToGDISurface

IDirectDraw4::GetMonitorFrequency

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::GetMonitorFrequency** method retrieves the frequency of the monitor being driven by the DirectDraw object.

```
HRESULT GetMonitorFrequency(
  LPDWORD lpdwFrequency
);
```

Parameters

lpdwFrequency
Address of the variable that will be filled with the monitor frequency, reported in Hz.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDraw4::GetScanLine

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::GetScanLine** method retrieves the scan line that is currently being drawn on the monitor.

```
HRESULT GetScanLine(
  LPDWORD lpdwScanLine
);
```

Parameters

lpdwScanLine
Address of the variable that will contain the scan line the display is currently drawing.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED
DDERR_VERTICALBLANKINPROGRESS

Remarks

Scan lines are reported as zero-based integers. The returned scan line value is between 0 and *n*, where scan line 0 is the first visible scan line on the screen and *n* is the last visible scan line, plus any scan lines that occur during the vertical blank period. So, in a case where an application is running at 640×480, and there are 12 scan lines during vblank, the values returned by this method will range from 0 to 491.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDraw4::GetVerticalBlankStatus, **IDirectDraw4::WaitForVerticalBlank**

IDirectDraw4::GetSurfaceFromDC

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::GetSurfaceFromDC** method retrieves the **IDirectDrawSurface4** interface for a surface based on its GDI device context handle.

```
HRESULT GetSurfaceFromDC(
  HDC hdc,
  LPDIRECTDRAWSURFACE4 * lpDDS4
);
```

Parameters

hdc
Handle to a display device context.

lpDDS4
Address of a variable that will be filled with a valid **IDirectDrawSurface4** interface pointer if the call succeeds.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY
DDERR_NOTFOUND

Remarks

This method will succeed only for device context handles that identify surfaces already associated with the DirectDraw object.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

Surfaces and Device Contexts

IDirectDraw4::GetVerticalBlankStatus

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::GetVerticalBlankStatus** method retrieves the status of the vertical blank.

```
HRESULT GetVerticalBlankStatus(
  LPBOOL lpbIsInVB
);
```

Parameters

lpbIsInVB
Address of the variable that will be filled with the status of the vertical blank. This parameter is TRUE if a vertical blank is occurring, and FALSE otherwise.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

To synchronize with the vertical blank, use the **IDirectDraw4::WaitForVerticalBlank** method.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDraw4::GetScanLine, **IDirectDraw4::WaitForVerticalBlank**

IDirectDraw4::Initialize

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::Initialize** method initializes a DirectDraw object that was created by using the **CoCreateInstance** COM function.

```
HRESULT Initialize(
  GUID FAR *lpGUID
);
```

Parameters

lpGUID
Address of the globally unique identifier (GUID) used as the interface identifier.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_ALREADYINITIALIZED
DDERR_DIRECTDRAWALREADYCREATED
DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NODIRECTDRAWHW
DDERR_NODIRECTDRAWSUPPORT
DDERR_OUTOFMEMORY

This method is provided for compliance with the Component Object Model (COM) protocol. If the **DirectDrawCreate** function was used to create a DirectDraw object, this method returns DDERR_ALREADYINITIALIZED. If **IDirectDraw4::Initialize** is not called when using **CoCreateInstance** to create a DirectDraw object, any method that is called afterward returns DDERR_NOTINITIALIZED.

Remarks

For more information about using **IDirectDraw4::Initialize** with **CoCreateInstance**, see Creating DirectDraw Objects by Using CoCreateInstance.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDraw4::RestoreAllSurfaces

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::RestoreAllSurfaces** method restores all the surfaces created for the DirectDraw object, in the order they were created.

```
HRESULT RestoreAllSurfaces();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is provided for convenience. Effectively, this method calls the **IDirectDrawSurface4::Restore** method for each surface created by this DirectDraw object.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::Restore, Losing and Restoring Surfaces

IDirectDraw4::RestoreDisplayMode

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::RestoreDisplayMode** method resets the mode of the display device hardware for the primary surface to what it was before the **IDirectDraw4::SetDisplayMode** method was called. Exclusive-level access is required to use this method.

```
HRESULT RestoreDisplayMode();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_LOCKEDSURFACES
DDERR_NOEXCLUSIVEMODE

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDraw4::SetDisplayMode, **IDirectDraw4::EnumDisplayModes**, **IDirectDraw4::SetCooperativeLevel**

IDirectDraw4::SetCooperativeLevel

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::SetCooperativeLevel** method determines the top-level behavior of the application.

```
HRESULT SetCooperativeLevel(
  HWND hWnd,
  DWORD dwFlags
);
```

Parameters

hWnd

Window handle used for the application. Set to the calling application's top-level window handle (not a handle for any child windows created by the top-level window). This parameter can be NULL when the DDSCL_NORMAL flag is specified in the *dwFlags* parameter.

dwFlags

One or more of the following flags:

DDSCL_ALLOWMODEX

Allows the use of Mode X display modes. This flag can only be used if the DDSCL_EXCLUSIVE and DDSCL_FULLSCREEN flags are present.

DDSCL_ALLOWREBOOT

Allows CTRL+ALT+DEL to function while in exclusive (full-screen) mode.

DDSCL_CREATEDEVICEWINDOW

This flag is supported in Windows 98 and Windows 2000 only. Indicates that DirectDraw is to create and manage a default device window for this DirectDraw object. For more information, see Focus and Device Windows.

DDSCL_EXCLUSIVE

Requests the exclusive level. This flag must be used with the DDSCL_FULLSCREEN flag.

DDSCL_FPUSETUP

Indicates that the calling application is likely to keep the FPU set up for optimal Direct3D performance (single precision and exceptions disabled) so Direct3D does not need to explicitly set the FPU each time. For more information, see DirectDraw Cooperative Levels and FPU Precision.

DDSCL_FULLSCREEN

Indicates that the exclusive-mode owner will be responsible for the entire primary surface. GDI can be ignored. This flag must be used with the DDSCL_EXCLUSIVE flag.

DDSCL_MULTITHREADED

Requests multithread-safe DirectDraw behavior. This causes Direct3D to take the global critical section more frequently.

DDSCL_NORMAL

Indicates that the application will function as a regular Windows application. This flag cannot be used with the DDSCL_ALLOWMODEX, DDSCL_EXCLUSIVE, or DDSCL_FULLSCREEN flags.

DDSCL_NOWINDOWCHANGES
Indicates that DirectDraw is not allowed to minimize or restore the application window on activation.

DDSCL_SETDEVICEWINDOW
This flag is supported in Windows 98 and Windows 2000 only. Indicates that the *hWnd* parameter is the window handle of the device window for this DirectDraw object. This flag cannot be used with the DDSCL_SETFOCUSWINDOW flag.

DDSCL_SETFOCUSWINDOW
This flag is supported in Windows 98 and Windows 2000 only. Indicates that the *hWnd* parameter is the window handle of the focus window for this DirectDraw object. This flag cannot be used with the DDSCL_SETDEVICEWINDOW flag.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_EXCLUSIVEMODEALREADYSET
DDERR_HWNDALREADYSET
DDERR_HWNDSUBCLASSED
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

Remarks

This method must be called by the same thread that created the application window.

An application must set either the DDSCL_EXCLUSIVE or DDSCL_NORMAL flag.

The DDSCL_EXCLUSIVE flag must be set to call functions that can have drastic performance consequences for other applications. For more information, see Cooperative Levels.

Interaction between this method and the **IDirectDraw4::SetDisplayMode** method differs from their **IDirectDraw** counterparts. For more information, see Restoring Display Modes.

Developers using Microsoft Foundation Classes (MFC) should keep in mind that the window handle passed to this method should identify the application's top-level window, not a derived child window. To retrieve your MFC application's top level window handle, you could use the following code:

```
HWND hwndTop = AfxGetMainWnd()->GetSafeHwnd();
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDraw4::SetDisplayMode, **IDirectDraw4::Compact**, **IDirectDraw4::EnumDisplayModes**, Mode X and Mode 13 Display Modes, Focus and Device Windows

IDirectDraw4::SetDisplayMode

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::SetDisplayMode** method sets the mode of the display-device hardware.

```
HRESULT SetDisplayMode(
  DWORD dwWidth,
  DWORD dwHeight,
  DWORD dwBPP,
  DWORD dwRefreshRate,
  DWORD dwFlags
);
```

Parameters

dwWidth and *dwHeight*
Width and height of the new mode.

dwBPP
Bits per pixel (bpp) of the new mode.

dwRefreshRate
Refresh rate of the new mode. Set this value to 0 to request the default refresh rate for the driver.

dwFlags
Flags describing additional options. Currently, the only valid flag is DDSDM_STANDARDVGAMODE, which causes the method to set Mode 13 instead of Mode X 320x200x8 mode. If you are setting another resolution, bit depth, or a Mode X mode, do not use this flag and set the parameter to 0.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDMODE
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_LOCKEDSURFACES
DDERR_NOEXCLUSIVEMODE
DDERR_SURFACEBUSY
DDERR_UNSUPPORTED
DDERR_UNSUPPORTEDMODE
DDERR_WASSTILLDRAWING

Remarks

This method must be called by the same thread that created the application window.

If another application changes the display mode, the primary surface will be lost and will return DDERR_SURFACELOST until it is recreated to match the new display mode.

As part of the **IDirectDraw** interface, this method did not include the *dwRefreshRate* and *dwFlags* parameters.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDraw4::RestoreDisplayMode, **IDirectDraw4::GetDisplayMode**, **IDirectDraw4::EnumDisplayModes**, **IDirectDraw4::SetCooperativeLevel**, Setting Display Modes, Restoring Display Modes

IDirectDraw4::TestCooperativeLevel

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::TestCooperativeLevel** method reports the current cooperative-level status of the DirectDraw device for a windowed or full-screen application.

HRESULT TestCooperativeLevel(void);

Parameters

None.

Return Values

If the method succeeds, the return value is DD_OK, indicating that the calling application can continue executing.

If the method fails, the return value may be one of the following error values (see remarks):

DDERR_INVALIDOBJECT
DDERR_EXCLUSIVEMODEALREADYSET
DDERR_NOEXCLUSIVEMODE
DDERR_WRONGMODE

Remarks

This method is particularly useful to applications that use the WM_ACTIVATEAPP and WM_DISPLAYCHANGE system messages as a notification to restore surfaces or re-create DirectDraw objects. The DD_OK return value always indicates that the application can continue execution, but the failure codes are interpreted differently depending on the cooperative-level that the application uses. For more information, see Testing Cooperative Levels.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDraw4::WaitForVerticalBlank

[This is preliminary documentation and subject to change.]

The **IDirectDraw4::WaitForVerticalBlank** method helps the application synchronize itself with the vertical-blank interval.

```
HRESULT WaitForVerticalBlank(
  DWORD dwFlags,
  HANDLE hEvent
);
```

Parameters

dwFlags

Determines how long to wait for the vertical blank.

DDWAITVB_BLOCKBEGIN
: Returns when the vertical-blank interval begins.

DDWAITVB_BLOCKBEGINEVENT
: Triggers an event when the vertical blank begins. This value is not currently supported.

DDWAITVB_BLOCKEND
: Returns when the vertical-blank interval ends and the display begins.

hEvent

Handle of the event to be triggered when the vertical blank begins. This parameter is not currently used.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_UNSUPPORTED

DDERR_WASSTILLDRAWING

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDraw4::GetVerticalBlankStatus, **IDirectDraw4::GetScanLine**

IDirectDrawClipper

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirectDrawClipper** interface to manage clip lists. This section is a reference to the methods of this interface. For a conceptual overview, see Clippers.

The methods of the **IDirectDrawClipper** interface can be organized into the following groups:

Allocating memory	**Initialize**
Clip list	**GetClipList**
	IsClipListChanged
	SetClipList
	SetHWnd
Handles	**GetHWnd**

The **IDirectDrawClipper** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

You can use the **LPDIRECTDRAWCLIPPER** data type to declare a variable that contains a pointer to an **IDirectDrawClipper** interface. The Ddraw.h header file declares these data types with the following code:

```
typedef struct IDirectDrawClipper    FAR *LPDIRECTDRAWCLIPPER;
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawClipper::GetClipList

[This is preliminary documentation and subject to change.]

The **IDirectDrawClipper::GetClipList** method retrieves a copy of the clip list associated with a DirectDrawClipper object. A subset of the clip list can be selected by passing a rectangle that clips the clip list.

```
HRESULT GetClipList(
  LPRECT lpRect,
  LPRGNDATA lpClipList,
  LPDWORD lpdwSize
);
```

Parameters

lpRect
Address of a rectangle that will be used to clip the clip list. This parameter can be NULL to retrieve the entire clip list.

lpClipList
Address of an **RGNDATA** structure that will contain the resulting copy of the clip list. If this parameter is NULL, the method fills the variable at *lpdwSize* to the number of bytes necessary to hold the entire clip list.

lpdwSize
Size of the resulting clip list. When retrieving the clip list, this parameter is the size of the buffer at *lpClipList*. When *lpClipList* is NULL, the variable at *lpdwSize* receives the required size of the buffer, in bytes.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDCLIPLIST
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCLIPLIST
DDERR_REGIONTOOSMALL

Remarks

The **RGNDATA** structure used with this method has the following syntax:

```
typedef struct _RGNDATA {
   RGNDATAHEADER rdh;
   char        Buffer[1];
} RGNDATA;
```

The **rdh** member of the **RGNDATA** structure is an **RGNDATAHEADER** structure that has the following syntax:

```
typedef struct _RGNDATAHEADER {
    DWORD dwSize;
    DWORD iType;
    DWORD nCount;
    DWORD nRgnSize;
    RECT  rcBound;
} RGNDATAHEADER;
```

For more information about these structures, see the documentation in the Platform SDK.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawClipper::SetClipList

IDirectDrawClipper::GetHWnd

[This is preliminary documentation and subject to change.]

The **IDirectDrawClipper::GetHWnd** method retrieves the window handle previously associated with this DirectDrawClipper object by the **IDirectDrawClipper::SetHWnd** method.

```
HRESULT GetHWnd(
  HWND FAR *lphWnd
);
```

Parameters

lphWnd
Address of the window handle previously associated with this DirectDrawClipper object by the **IDirectDrawClipper::SetHWnd** method.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawClipper::SetHWnd

IDirectDrawClipper::Initialize

[This is preliminary documentation and subject to change.]

The **IDirectDrawClipper::Initialize** method initializes a DirectDrawClipper object that was created by using the **CoCreateInstance** COM function.

```
HRESULT Initialize(
  LPDIRECTDRAW lpDD,
  DWORD dwFlags
);
```

Parameters

lpDD
Address of the DirectDraw structure that represents the DirectDraw object. If this parameter is set to NULL, an independent DirectDrawClipper object is created (the equivalent of using the **DirectDrawCreateClipper** function).

dwFlags
This parameter is currently not used and must be set to 0.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_ALREADYINITIALIZED
DDERR_INVALIDPARAMS

This method is provided for compliance with the Component Object Model (COM) protocol. If **DirectDrawCreateClipper** or the **IDirectDraw4::CreateClipper** method was used to create the DirectDrawClipper object, this method returns DDERR_ALREADYINITIALIZED.

Remarks

For more information about using **IDirectDrawClipper::Initialize** with **CoCreateInstance**, see Creating DirectDrawClipper Objects with CoCreateInstance.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IUnknown::AddRef, **IUnknown::QueryInterface**, **IUnknown::Release**, **IDirectDraw4::CreateClipper**

IDirectDrawClipper::IsClipListChanged

[This is preliminary documentation and subject to change.]

The **IDirectDrawClipper::IsClipListChanged** method monitors the status of the clip list if a window handle is associated with a DirectDrawClipper object.

```
HRESULT IsClipListChanged(
  BOOL FAR *lpbChanged
);
```

Parameters

lpbChanged
Address of a variable that is set to TRUE if the clip list has changed.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawClipper::SetClipList

[This is preliminary documentation and subject to change.]

The **IDirectDrawClipper::SetClipList** method sets or deletes the clip list used by the **IDirectDrawSurface4::Blt**, **IDirectDrawSurface4::BltBatch**, and **IDirectDrawSurface4::UpdateOverlay** methods on surfaces to which the parent DirectDrawClipper object is attached.

```
HRESULT SetClipList(
  LPRGNDATA lpClipList,
  DWORD dwFlags
);
```

Parameters

lpClipList
Either an address to a valid **RGNDATA** structure or NULL. If there is an existing clip list associated with the DirectDrawClipper object and this value is NULL, the clip list will be deleted.

dwFlags
This parameter is currently not used and must be set to 0.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_CLIPPERISUSINGHWND
DDERR_INVALIDCLIPLIST
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

Remarks

The clip list cannot be set if a window handle is already associated with the DirectDrawClipper object. Note that the **BltFast** method cannot clip.

The **RGNDATA** structure used with this method has the following syntax:

```
typedef struct _RGNDATA {
    RGNDATAHEADER rdh;
    char          Buffer[1];
} RGNDATA;
```

The **rdh** member of the **RGNDATA** structure is an **RGNDATAHEADER** structure that has the following syntax:

```
typedef struct _RGNDATAHEADER {
    DWORD dwSize;
    DWORD iType;
    DWORD nCount;
    DWORD nRgnSize;
    RECT  rcBound;
} RGNDATAHEADER;
```

For more information about these structures, see the documentation in the Platform Software Development Kit.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawClipper::GetClipList, **IDirectDrawSurface4::Blt**, **IDirectDrawSurface4::BltFast**, **IDirectDrawSurface4::BltBatch**, **IDirectDrawSurface4::UpdateOverlay**

IDirectDrawClipper::SetHWnd

[This is preliminary documentation and subject to change.]

The **IDirectDrawClipper::SetHWnd** method sets the window handle that the clipper object uses to obtain clipping information.

```
HRESULT SetHWnd(
  DWORD dwFlags,
```

HWND *hWnd*
);

Parameters

dwFlags
This parameter is currently not used and must be set to 0.

hWnd
Window handle that obtains the clipping information.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDCLIPLIST
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawClipper::GetHWnd

IDirectDrawColorControl

[This is preliminary documentation and subject to change.]

The **IDirectDrawColorControl** interface allows you to get and set color controls:

Color controls	**GetColorControls**
	SetColorControls

The **IDirectDrawColorControl** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

You can use the **LPDIRECTDRAWCOLORCONTROL** data type to declare a variable that contains a pointer to an **IDirectDrawColorControl** interface. The Ddraw.h header file declares these data types with the following code:

```
typedef struct IDirectDrawColorControl    FAR *LPDIRECTDRAWCOLORCONTROL;
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawColorControl::GetColorControls

[This is preliminary documentation and subject to change.]

The **IDirectDrawColorControl::GetColorControls** method returns the current color control settings associated with the specified overlay or primary surface. The **dwFlags** member of the **DDCOLORCONTROL** structure indicates which of the color control options are supported.

```
HRESULT GetColorControls(
  LPDDCOLORCONTROL lpColorControl
);
```

Parameters

lpColorControl
Address of the **DDCOLORCONTROL** structure that will receive the current control settings of the specified surface.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawColorControl::SetColorControls, Using Color Controls, Gamma and Color Controls

IDirectDrawColorControl::SetColorControls

[This is preliminary documentation and subject to change.]

The **IDirectDrawColorControl::SetColorControls** method sets the color control settings associated with the specified overlay or primary surface.

```
HRESULT SetColorControls(
  LPDDCOLORCONTROL lpColorControl
);
```

Parameters

lpColorControl
Address of the **DDCOLORCONTROL** structure containing the new values to be applied to the specified surface.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawColorControl::GetColorControls, Using Color Controls, Gamma and Color Controls

IDirectDrawGammaControl

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirectDrawGammaControl** interface to adjust the red, green, and blue gamma ramp levels of the primary surface. This section is a reference to the methods of this interface. This interface is supported by DirectDrawSurface objects; you can retrieve a pointer to this interface by calling the **IUnknown::QueryInterface** method of a DirectDrawSurface object, specifying the IID_IDirectDrawGammaControl reference identifier.

For a conceptual overview, see Gamma and Color Controls.

Gamma ramps	**GetGammaRamp**
	SetGammaRamp

The **IDirectDrawGammaControl** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

You can use the **LPDIRECTDRAWGAMMACONTROL** data type to declare a variable that contains a pointer to an **IDirectDrawGammaControl** interface. The Ddraw.h header file declares the data type with the following code:

```
typedef struct IDirectDrawGammaControl    FAR *LPDIRECTDRAWGAMMACONTROL;
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawGammaControl::GetGammaRamp

[This is preliminary documentation and subject to change.]

The **IDirectDrawGammaControl::GetGammaRamp** method retrieves the red, green, and blue gamma ramps for the primary surface.

```
HRESULT GetGammaRamp(
  DWORD  dwFlags,
  LPGAMMARAMP lpRampData
);
```

Parameters

dwFlags
Not currently used; set to 0.

lpRampData
Address of a **DDGAMMARAMP** structure that will be filled with the current red, green, and blue gamma ramps. Each array maps color values in the frame buffer to the color values that will be passed to the DAC (Digital-to-Analog Converter).

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_EXCEPTION
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawGammaControl::SetGammaRamp, Gamma and Color Controls

IDirectDrawGammaControl::SetGammaRamp

[This is preliminary documentation and subject to change.]

The **IDirectDrawGammaControl::SetGammaRamp** method retrieves the red, green, and blue gamma ramps for the primary surface.

```
HRESULT SetGammaRamp(
  DWORD  dwFlags,
  LPGAMMARAMP lpRampData
);
```

Parameters

dwFlags

Flag indicating if gamma calibration is desired. Set this parameter DDSGR_CALIBRATE to request that the calibrator adjust the gamma ramp according to the physical properties of the display, making the result identical on all systems. If calibration is not needed, set this parameter to 0.

lpRampData

Address of a **DDGAMMARAMP** structure that contains the new red, green, and blue gamma ramp entries. Each array maps color values in the frame buffer to the color values that will be passed to the DAC (Digital-to-Analog Converter).

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_EXCEPTION

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_OUTOFMEMORY

Remarks

Not all systems support gamma calibration. To determine if gamma calibration is supported, call **IDirectDraw4::GetCaps**, and examine the **dwCaps2** member of the associated **DDCAPS** structure after the method returns. If the

DDCAPS2_CANCALIBRATEGAMMA capability flag is present, then gamma calibration is supported.

Calibrating gamma ramps incurs some processing overhead, and should not be used frequently.

Including the DDSGR_CALIBRATE flag in the *dwFlags* parameter when running on systems that do not support gamma calibration will not cause this method to fail. The method succeeds, setting new gamma ramp values without calibration.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawGammaControl::GetGammaRamp, Gamma and Color Controls

IDirectDrawPalette

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirectDrawPalette** interface to create DirectDrawPalette objects and work with system-level variables. This section is a reference to the methods of this interface. For a conceptual overview, see Palettes.

The methods of the **IDirectDrawPalette** interface can be organized into the following groups:

Allocating memory	**Initialize**
Palette capabilities	**GetCaps**
Palette entries	**GetEntries**
	SetEntries

The **IDirectDrawPalette** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

You can use the **LPDIRECTDRAWPALETTE** data type to declare a variable that contains a pointer to an **IDirectDrawPalette** interface. The Ddraw.h header file declares the data type with the following code:

```
typedef struct IDirectDrawPalette    FAR *LPDIRECTDRAWPALETTE;
```

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawPalette::GetCaps

[This is preliminary documentation and subject to change.]

The **IDirectDrawPalette::GetCaps** method retrieves the capabilities of this palette object.

```
HRESULT GetCaps(
  LPDWORD lpdwCaps
);
```

Parameters

lpdwCaps
Flag from the **dwPalCaps** member of the **DDCAPS** structure that defines palette capabilities:

DDPCAPS_1BIT
DDPCAPS_2BIT
DDPCAPS_4BIT
DDPCAPS_8BIT
DDPCAPS_8BITENTRIES
DDPCAPS_ALPHA
DDPCAPS_ALLOW256
DDPCAPS_PRIMARYSURFACE
DDPCAPS_PRIMARYSURFACELEFT
DDPCAPS_VSYNC

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawPalette::GetEntries

[This is preliminary documentation and subject to change.]

The **IDirectDrawPalette::GetEntries** method queries palette values from a DirectDrawPalette object.

```
HRESULT GetEntries(
  DWORD dwFlags,
  DWORD dwBase,
  DWORD dwNumEntries,
  LPPALETTEENTRY lpEntries
);
```

Parameters

dwFlags
This parameter is currently not used and must be set to 0.

dwBase
Start of the entries that should be retrieved sequentially.

dwNumEntries
Number of palette entries that can fit in the address specified in *lpEntries*. The colors of each palette entry are returned in sequence, from the value of the *dwStartingEntry* parameter through the value of the *dwCount* parameter minus 1. (These parameters are set by **IDirectDrawPalette::SetEntries**.)

lpEntries
Address of the palette entries. The palette entries are 1 byte each if the DDPCAPS_8BITENTRIES flag is set and 4 bytes otherwise. Each field is a color description.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTPALETTIZED

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawPalette::SetEntries

IDirectDrawPalette::Initialize

[This is preliminary documentation and subject to change.]

The **IDirectDrawPalette::Initialize** method initializes the DirectDrawPalette object.

```
HRESULT Initialize(
  LPDIRECTDRAW lpDD,
  DWORD dwFlags,
  LPPALETTEENTRY lpDDColorTable
);
```

Parameters

lpDD
Address of the DirectDraw structure that represents the DirectDraw object.

dwFlags and *lpDDColorTable*
These parameters are currently not used and must be set to 0.

Return Values

This method returns DDERR_ALREADYINITIALIZED.

This method is provided for compliance with the Component Object Model (COM) protocol. Because the DirectDrawPalette object is initialized when it is created, this method always returns DDERR_ALREADYINITIALIZED.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IUnknown::AddRef, **IUnknown::QueryInterface**, **IUnknown::Release**

IDirectDrawPalette::SetEntries

[This is preliminary documentation and subject to change.]

The **IDirectDrawPalette::SetEntries** method changes entries in a DirectDrawPalette object immediately.

```
HRESULT SetEntries(
  DWORD dwFlags,
  DWORD dwStartingEntry,
  DWORD dwCount,
  LPPALETTEENTRY lpEntries
);
```

Parameters

dwFlags
This parameter is currently not used and must be set to 0.

dwStartingEntry
First entry to be set.

dwCount
Number of palette entries to be changed.

lpEntries
Address of the palette entries. The palette entries are 1 byte each if the DDPCAPS_8BITENTRIES flag is set and 4 bytes otherwise. Each field is a color description.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

DDERR_NOPALETTEATTACHED
DDERR_NOTPALETTIZED
DDERR_UNSUPPORTED

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawPalette::GetEntries, **IDirectDrawSurface4::SetPalette**

IDirectDrawSurface4

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirectDrawSurface4** interface to create DirectDrawSurface objects and work with system-level variables. This section is a reference to the methods of this interface. For a conceptual overview, see Surfaces.

The methods of the **IDirectDrawSurface4** interface can be organized into the following groups:

Allocating memory	**Initialize**
	IsLost
	Restore
Attaching surfaces	**AddAttachedSurface**
	DeleteAttachedSurface
	EnumAttachedSurfaces
	GetAttachedSurface
Blitting	**Blt**
	BltBatch
	BltFast
	GetBltStatus
Color keying	**GetColorKey**
	SetColorKey

Device contexts	**GetDC**
	ReleaseDC
Flipping	**Flip**
	GetFlipStatus
Locking surfaces	**Lock**
	PageLock
	PageUnlock
	Unlock
Miscellaneous	**GetDDInterface**
Overlays	**AddOverlayDirtyRect**
	EnumOverlayZOrders
	GetOverlayPosition
	SetOverlayPosition
	UpdateOverlay
	UpdateOverlayDisplay
	UpdateOverlayZOrder
Private surface data	**FreePrivateData**
	GetPrivateData
	SetPrivateData
Surface capabilities	**GetCaps**
Surface clipper	**GetClipper**
	SetClipper
Surface characteristics	**ChangeUniquenessValue**
	GetPixelFormat
	GetSurfaceDesc
	GetUniquenessValue
	SetSurfaceDesc
Surface palettes	**GetPalette**

SetPalette

The **IDirectDrawSurface4** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

The **IDirectDrawSurface4** interface extends the features of previous versions of the interface by offering methods that offer better surface management and ease of use. Note that many methods in this interface accept slightly different parameters than their counterparts in former versions of the interface. Wherever an **IDirectDrawSurface3** interface method might accept a **DDSURFACEDESC** structure or an **IDirectDrawSurface3** interface, the methods in **IDirectDrawSurface4** accept a **DDSURFACEDESC2** structure or an **IDirectDrawSurface4** interface instead.

You can use the **LPDIRECTDRAWSURFACE**, **LPDIRECTDRAWSURFACE2**, **LPDIRECTDRAWSURFACE3**, or **LPDIRECTDRAWSURFACE4** data types to declare variables that point to various DirectDrawSurface object interfaces. The Ddraw.h header file declares these data types with the following code:

```
typedef struct IDirectDrawSurface     FAR *LPDIRECTDRAWSURFACE;
typedef struct IDirectDrawSurface2    FAR *LPDIRECTDRAWSURFACE2;
typedef struct IDirectDrawSurface3    FAR *LPDIRECTDRAWSURFACE3;
typedef struct IDirectDrawSurface4    FAR *LPDIRECTDRAWSURFACE4;
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawSurface4::AddAttachedSurface

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::AddAttachedSurface** method attaches the specified surface to this surface.

```
HRESULT AddAttachedSurface(
  LPDIRECTDRAWSURFACE4 lpDDSAttachedSurface
);
```

Parameters

lpDDSAttachedSurface
Address of an **IDirectDrawSurface4** interface for the surface to be attached.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_CANNOTATTACHSURFACE
DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACEALREADYATTACHED
DDERR_SURFACELOST
DDERR_WASSTILLDRAWING

Remarks

This method increments the reference count of the surface being attached. You can explicitly unattach the surface and decrement its reference count by using the **IDirectDrawSurface4::DeleteAttachedSurface** method. Unlike complex surfaces that you create with a single call to **IDirectDraw4::CreateSurface**, surfaces attached with this method are not automatically released. It is the application's responsibility to release such surfaces.

Possible attachments include z-buffers, alpha channels, and back buffers. Some attachments automatically break other attachments. For example, the 3-D z-buffer can be attached only to one back buffer at a time. Attachment is not bidirectional, and a surface cannot be attached to itself. Emulated surfaces (in system memory) cannot be attached to nonemulated surfaces. Unless one surface is a texture map, the two attached surfaces must be the same size. A flipping surface cannot be attached to another flipping surface of the same type; however, attaching two surfaces of different types is allowed. For example, a flipping z-buffer can be attached to a regular flipping surface. If a nonflipping surface is attached to another nonflipping surface of the same type, the two surfaces will become a flipping chain. If a nonflipping surface is attached to a flipping surface, it becomes part of the existing flipping chain. Additional surfaces can be added to this chain, and each call of the **IDirectDrawSurface4::Flip** method will advance one step through the surfaces.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::DeleteAttachedSurface,
IDirectDrawSurface4::EnumAttachedSurfaces

IDirectDrawSurface4::AddOverlayDirtyRect

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::AddOverlayDirtyRect** method is not currently implemented.

```
HRESULT AddOverlayDirtyRect(
  LPRECT lpRect
);
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::UpdateOverlayDisplay

IDirectDrawSurface4::Blt

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::Blt** method performs a bit block transfer. This method does not support z-buffering or alpha blending (see alpha channel) during blit operations.

```
HRESULT Blt(
  LPRECT lpDestRect,
  LPDIRECTDRAWSURFACE4 lpDDSrcSurface,
```

```
  LPRECT lpSrcRect,
  DWORD dwFlags,
  LPDDBLTFX lpDDBltFx
);
```

Parameters

lpDestRect

Address of a **RECT** structure that defines the upper-left and lower-right points of the rectangle to blit to on the destination surface. If this parameter is NULL, the entire destination surface will be used.

lpDDSrcSurface

Address of an **IDirectDrawSurface4** interface for the DirectDrawSurface object that is the source of the blit.

lpSrcRect

Address of a **RECT** structure that defines the upper-left and lower-right points of the rectangle to blit from on the source surface. If this parameter is NULL, the entire source surface will be used.

dwFlags

A combination of flags that determine the valid members of the associated **DDBLTFX** structure, specify color key information, or that request special behavior from the method. The following flags are defined.

Validation flags

DDBLT_COLORFILL

Uses the **dwFillColor** member of the **DDBLTFX** structure as the RGB color that fills the destination rectangle on the destination surface.

DDBLT_DDFX

Uses the **dwDDFX** member of the **DDBLTFX** structure to specify the effects to use for this blit.

DDBLT_DDROPS

Uses the **dwDDROP** member of the **DDBLTFX** structure to specify the raster operations (ROPS) that are not part of the Win32 API.

DDBLT_DEPTHFILL

Uses the **dwFillDepth** member of the **DDBLTFX** structure as the depth value with which to fill the destination rectangle on the destination z-buffer surface.

DDBLT_KEYDESTOVERRIDE

Uses the **ddckDestColorkey** member of the **DDBLTFX** structure as the color key for the destination surface.

DDBLT_KEYSRCOVERRIDE

Uses the **ddckSrcColorkey** member of the **DDBLTFX** structure as the color key for the source surface.

DDBLT_ROP

Uses the **dwROP** member of the **DDBLTFX** structure for the ROP for this blit. These ROPs are the same as those defined in the Win32 API.

DDBLT_ROTATIONANGLE
Uses the **dwRotationAngle** member of the **DDBLTFX** structure as the rotation angle (specified in 1/100th of a degree) for the surface.

Color key flags

DDBLT_KEYDEST
Uses the color key associated with the destination surface.

DDBLT_KEYSRC
Uses the color key associated with the source surface.

Behavior flags

DDBLT_ASYNC
Performs this blit asynchronously through the FIFO in the order received. If no room is available in the FIFO hardware, the call fails.

DDBLT_WAIT
Postpones the DDERR_WASSTILLDRAWING return value if the blitter is busy, and returns as soon as the blit can be set up or another error occurs.

Obsolete and unsupported flags

All "DDBLT_ALPHA" flag values.
Not currently implemented.

All "DDBLT_ZBUFFER" flag values
This method does not currently support z-aware blit operations. None of the flags beginning with "DDBLT_ZBUFFER" are supported in this release of DirectX 6.0.

lpDDBltFx
Address of the **DDBLTFX** structure.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDCLIPLIST
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDRECT
DDERR_NOALPHAHW
DDERR_NOBLTHW
DDERR_NOCLIPLIST
DDERR_NODDROPSHW
DDERR_NOMIRRORHW
DDERR_NORASTEROPHW
DDERR_NOROTATIONHW

DDERR_NOSTRETCHHW
DDERR_NOZBUFFERHW
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING

Remarks

This method is capable of synchronous or asynchronous blits (the default behavior), either display memory to display memory, display memory to system memory, system memory to display memory, or system memory to system memory. The blits can be performed by using source color keys, and destination color keys. Arbitrary stretching or shrinking will be performed if the source and destination rectangles are not the same size.

Typically, **IDirectDrawSurface4::Blt** returns immediately with an error if the blitter is busy and the blit could not be set up. Specify the DDBLT_WAIT flag to request a synchronous blit. When you include the DDBLT_WAIT flag, the method waits until the blit can be set up or another error occurs before it returns.

Note that **RECT** structures are defined so that the **right** and **bottom** members are exclusive—therefore, **right** minus **left** equals the width of the rectangle, not one less than the width.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawSurface4::BltBatch

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::BltBatch** method is not currently implemented.

```
HRESULT BltBatch(
  LPDDBLTBATCH lpDDBltBatch,
  DWORD dwCount,
  DWORD dwFlags
);
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawSurface4::BltFast

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::BltFast** method performs a source copy blit or transparent blit by using a source color key or destination color key.

```
HRESULT BltFast(
  DWORD dwX,
  DWORD dwY,
  LPDIRECTDRAWSURFACE4 lpDDSrcSurface,
  LPRECT lpSrcRect,
  DWORD dwTrans
);
```

Parameters

dwX and *dwY*
The x- and y-coordinates to blit to on the destination surface.

lpDDSrcSurface
Address of an **IDirectDrawSurface4** interface for the DirectDrawSurface object that is the source of the blit.

lpSrcRect
Address of a **RECT** structure that defines the upper-left and lower-right points of the rectangle to blit from on the source surface.

dwTrans
Type of transfer.

DDBLTFAST_DESTCOLORKEY
Specifies a transparent blit that uses the destination's color key.

DDBLTFAST_NOCOLORKEY
Specifies a normal copy blit with no transparency.

DDBLTFAST_SRCCOLORKEY
Specifies a transparent blit that uses the source's color key.

DDBLTFAST_WAIT
Postpones the DDERR_WASSTILLDRAWING message if the blitter is busy, and returns as soon as the blit can be set up or another error occurs.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_EXCEPTION
DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDRECT
DDERR_NOBLTHW
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING

Remarks

This method always attempts an asynchronous blit if it is supported by the hardware.

This method works only on display memory surfaces and cannot clip when blitting. If you use this method on a surface with an attached clipper, the call will fail and the method will return DDERR_UNSUPPORTED.

The software implementation of **IDirectDrawSurface4::BltFast** is 10 percent faster than the **IDirectDrawSurface4::Blt** method. However, there is no speed difference between the two if display hardware is being used.

Typically, **IDirectDrawSurface4::BltFast** returns immediately with an error if the blitter is busy and the blit cannot be set up. You can use the DDBLTFAST_WAIT flag, however, if you want this method to not return until either the blit can be set up or another error occurs.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawSurface4::ChangeUniquenessValue

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::ChangeUniquenessValue** method manually updates the uniqueness value for this surface.

```
HRESULT ChangeUniquenessValue();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_EXCEPTION
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

DirectDraw automatically updates uniqueness values whenever the contents of a surface change.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::GetUniquenessValue, Surface Uniqueness Values

IDirectDrawSurface4::DeleteAttachedSurface

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::DeleteAttachedSurface** method detaches two attached surfaces.

```
HRESULT DeleteAttachedSurface(
  DWORD dwFlags,
  LPDIRECTDRAWSURFACE4 lpDDSAttachedSurface
);
```

Parameters

dwFlags
This parameter is currently not used and must be set to 0.

lpDDSAttachedSurface
Address of the **IDirectDrawSurface4** interface for the DirectDrawSurface object to be detached. If this parameter is NULL, all attached surfaces are detached.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_CANNOTDETACHSURFACE
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST
DDERR_SURFACENOTATTACHED

Remarks

This method decrements the reference count of the surface being detached. If the reference count of the surface being detached reaches zero, it is lost and removed from memory.

Implicit attachments, those formed by DirectDraw rather than the **IDirectDrawSurface4::AddAttachedSurface** method, cannot be detached. Detaching surfaces from a flipping chain can alter other surfaces in the chain. If a front buffer is detached from a flipping chain, the next surface in the chain becomes the front buffer, and the following surface becomes the back buffer. If a back buffer is detached from a chain, the following surface becomes a back buffer. If a plain surface is detached from a chain, the chain simply becomes shorter. If a flipping chain has only two surfaces and they are detached, the chain is destroyed and both surfaces return to their previous designations.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::Flip

IDirectDrawSurface4::EnumAttachedSurfaces

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::EnumAttachedSurfaces** method enumerates all the surfaces attached to a given surface.

```
HRESULT EnumAttachedSurfaces(
  LPVOID lpContext,
  LPDDENUMSURFACESCALLBACK2 lpEnumSurfacesCallback
);
```

Parameters

lpContext
Address of the application-defined structure that is passed to the enumeration member every time it is called.

lpEnumSurfacesCallback
Address of the **EnumSurfacesCallback2** function that will be called for each surface that is attached to this surface.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_SURFACELOST

Remarks

This method enumerates only those surfaces that are directly attached to this surface. For example, in a flipping chain of three or more surfaces, only one surface will be enumerated, because each surface is attached only to the next surface in the flipping chain. In such a configuration, you can call **EnumAttachedSurfaces** on each successive surface to walk the entire flipping chain.

This method differs from its counterparts in previous interface versions in that it accepts a pointer to an **EnumSurfacesCallback2** function, rather than an **EnumSurfacesCallback** function.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawSurface4::EnumOverlayZOrders

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::EnumOverlayZOrders** method enumerates the overlay surfaces on the specified destination. The overlays can be enumerated in front-to-back or back-to-front order.

```
HRESULT EnumOverlayZOrders(
  DWORD dwFlags,
  LPVOID lpContext,
  LPDDENUMSURFACESCALLBACK2 lpfnCallback
);
```

Parameters

dwFlags
One of the following flags:

DDENUMOVERLAYZ_BACKTOFRONT
Enumerates overlays back to front.

DDENUMOVERLAYZ_FRONTTOBACK
Enumerates overlays front to back.

lpContext
Address of the user-defined context that will be passed to the callback function for each overlay surface.

lpfnCallback
Address of the **EnumSurfacesCallback2** callback function that will be called for each surface being overlaid on this surface.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method differs from its counterparts in previous interface versions in that it accepts a pointer to an **EnumSurfacesCallback2** function, rather than an **EnumSurfacesCallback** function.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawSurface4::Flip

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::Flip** method makes the surface memory associated with the DDSCAPS_BACKBUFFER surface become associated with the front-buffer surface.

```
HRESULT Flip(
  LPDIRECTDRAWSURFACE4 lpDDSurfaceTargetOverride,
  DWORD dwFlags
);
```

Parameters

lpDDSurfaceTargetOverride
Address of the **IDirectDrawSurface4** interface for an arbitrary surface in the flipping chain. The default for this parameter is NULL, in which case DirectDraw cycles through the buffers in the order they are attached to each other. If this parameter is not NULL, DirectDraw flips to the specified surface instead of the next surface in the flipping chain. The method fails if the specified surface is not a member of the flipping chain.

dwFlags
Flags specifying flip options.

DDFLIP_EVEN
For use only when displaying video in an overlay surface. The new surface contains data from the even field of a video signal. This flag cannot be used with the DDFLIP_ODD flag.

DDFLIP_INTERVAL2
DDFLIP_INTERVAL3
DDFLIP_INTERVAL4

These flags indicate how many vertical retraces to wait between each flip. The default is 1. DirectDraw will return DERR_WASSTILLDRAWING for each surface involved in the flip until the specified number of vertical retraces has occurred. If DDFLIP_INTERVAL2 is set, DirectDraw will flip on every second vertical sync; if DDFLIP_INTERVAL3, on every third sync; and if DDFLIP_INTERVAL4, on every fourth sync.

These flags are effective only if DDCAPS2_FLIPINTERVAL is set in the **DDCAPS** structure returned for the device.

DDFLIP_NOVSYNC
Causes DirectDraw to perform the physical flip as close as possible to the next scan line. Subsequent operations involving the two flipped surfaces will not check to see if the physical flip has finished—that is, they will not return DDERR_WASSTILLDRAWING for that reason (but may for other reasons). This allows an application to perform flips at a higher frequency than the monitor refresh rate, but may introduce visible artifacts.

If DDCAPS2_FLIPNOVSYNC is not set in the **DDCAPS** structure returned for the device, DDFLIP_NOVSYNC has no effect.

DDFLIP_ODD
For use only when displaying video in an overlay surface. The new surface contains data from the odd field of a video signal. This flag cannot be used with the DDFLIP_EVEN flag.

DDFLIP_WAIT
Typically, if the flip cannot be set up because the state of the display hardware is not appropriate, the DDERR_WASSTILLDRAWING error returns immediately and no flip occurs. Setting this flag causes the method to continue trying to flip if it receives the DDERR_WASSTILLDRAWING error from the HAL. The method does not return until the flipping operation has been successfully set up, or another error, such as DDERR_SURFACEBUSY, is returned.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOFLIPHW
DDERR_NOTFLIPPABLE
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING

Remarks

This method can be called only for a surface that has the DDSCAPS_FLIP and DDSCAPS_FRONTBUFFER capabilities. The display memory previously associated with the front buffer is associated with the back buffer.

The *lpDDSurfaceTargetOverride* parameter is used in rare cases when the back buffer is not the buffer that should become the front buffer. Typically this parameter is NULL.

The **IDirectDrawSurface4::Flip** method will always be synchronized with the vertical blank. If the surface has been assigned to a video port, this method updates the visible overlay surface and the video port's target surface.

For more information, see Flipping Surfaces.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::GetFlipStatus

IDirectDrawSurface4::FreePrivateData

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::FreePrivateData** method frees the specified private data associated with this surface.

```
HRESULT FreePrivateData(
  REFGUID guidTag,
);
```

Parameters

guidTag
Reference to (C++) or address of (C) the globally unique identifier that identifies the private data to be freed.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTFOUND

Remarks

DirectDraw calls this method automatically when a surface is released.

If the private data was set by using the DDSPD_IUNKNOWNPOINTER flag, this method calls the **IUnknown::Release** method on the associated interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::GetPrivateData, **IDirectDrawSurface4::SetPrivateData**

IDirectDrawSurface4::GetAttachedSurface

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::GetAttachedSurface** method obtains the attached surface that has the specified capabilities and increments the reference count of the retrieved interface.

```
HRESULT GetAttachedSurface(
  LPDDSCAPS2 lpDDSCaps,
  LPDIRECTDRAWSURFACE4 FAR *lplpDDAttachedSurface
);
```

Parameters

lpDDSCaps
Address of a **DDSCAPS2** structure that contains the hardware capabilities of the surface.

lplpDDAttachedSurface
Address of a variable that will contain a pointer to the retrieved surface's **IDirectDrawSurface4** interface. The retrieved surface is the one that matches the description according to the *lpDDSCaps* parameter.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTFOUND
DDERR_SURFACELOST

Remarks

Attachments are used to connect multiple DirectDrawSurface objects into complex structures, like the ones needed to support 3-D page flipping with z-buffers. This method fails if more than one surface is attached that matches the capabilities requested. In this case, the application must use the **IDirectDrawSurface4::EnumAttachedSurfaces** method to obtain the attached surfaces.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawSurface4::GetBltStatus

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::GetBltStatus** method obtains the blitter status.

```
HRESULT GetBltStatus(
  DWORD dwFlags
);
```

Parameters

dwFlags
One of the following flags:

DDGBS_CANBLT
Inquires whether a blit involving this surface can occur immediately, and returns DD_OK if the blit can be completed.

DDGBS_ISBLTDONE
Inquires whether the blit is done, and returns DD_OK if the last blit on this surface has completed.

Return Values

If the method succeeds, that means a blitter is present, the return value is DD_OK.

If the method fails, the return value is DDERR_WASSTILLDRAWING if the blitter is busy, DDERR_NOBLTHW if there is no blitter, or one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOBLTHW
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawSurface4::GetCaps

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::GetCaps** method retrieves the capabilities of the surface. These capabilities are not necessarily related to the capabilities of the display device.

```
HRESULT GetCaps(
  LPDDSCAPS2 lpDDSCaps
);
```

Parameters

lpDDSCaps

Address of a **DDSCAPS2** structure that will be filled with the hardware capabilities of the surface.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method differs from its counterpart in the **IDirectDrawSurface3** interface in that it accepts a pointer to a **DDSCAPS2** structure rather than the legacy **DDSCAPS** structure.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawSurface4::GetClipper

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::GetClipper** method retrieves the DirectDrawClipper object associated with this surface and increments the reference count of the returned clipper.

```
HRESULT GetClipper(
  LPDIRECTDRAWCLIPPER FAR *lplpDDClipper
);
```

Parameters

lplpDDClipper
Address of a pointer to the DirectDrawClipper object associated with the surface.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCLIPPERATTACHED

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::SetClipper

IDirectDrawSurface4::GetColorKey

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::GetColorKey** method retrieves the color key value for the DirectDrawSurface object.

```
HRESULT GetColorKey(
  DWORD dwFlags,
  LPDDCOLORKEY lpDDColorKey
);
```

Parameters

dwFlags
Determines which color key is requested.

DDCKEY_DESTBLT
Set if the structure specifies a color key or color space to be used as a destination color key for blit operations.

DDCKEY_DESTOVERLAY
Set if the structure specifies a color key or color space to be used as a destination color key for overlay operations.

DDCKEY_SRCBLT
Set if the structure specifies a color key or color space to be used as a source color key for blit operations.

DDCKEY_SRCOVERLAY
Set if the structure specifies a color key or color space to be used as a source color key for overlay operations.

lpDDColorKey
Address of the **DDCOLORKEY** structure that will be filled with the current values for the specified color key of the DirectDrawSurface object.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCOLORKEY
DDERR_NOCOLORKEYHW
DDERR_SURFACELOST
DDERR_UNSUPPORTED

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::SetColorKey

IDirectDrawSurface4::GetDC

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::GetDC** method creates a GDI-compatible handle of a device context for the surface.

```
HRESULT GetDC(
  HDC FAR *lphDC
);
```

Parameters

lphDC
Address for the returned handle to a device context.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_DCALREADYCREATED
DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING

Remarks

This method uses an internal version of the **IDirectDrawSurface4::Lock** method to lock the surface. The surface remains locked until the **IDirectDrawSurface4::ReleaseDC** method is called.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::Lock

IDirectDrawSurface4::GetDDInterface

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::GetDDInterface** method retrieves an interface to the DirectDraw object that was used to create the surface.

```
HRESULT GetDDInterface(
  LPVOID FAR *lplpDD
);
```

Parameters

lplpDD
Address of a variable that will be filled with a valid interface pointer if the call succeeds. Cast this pointer to an **IUnknown** interface pointer, then query for the desired DirectDraw interface.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method was not implemented in the **IDirectDraw** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawSurface4::GetFlipStatus

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::GetFlipStatus** method indicates whether the surface has finished its flipping process.

```
HRESULT GetFlipStatus(
  DWORD dwFlags
);
```

Parameters

dwFlags
One of the following flags:

DDGFS_CANFLIP
Inquires whether this surface can be flipped immediately and returns DD_OK if the flip can be completed.

DDGFS_ISFLIPDONE
Inquires whether the flip has finished and returns DD_OK if the last flip on this surface has completed.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is DDERR_WASSTILLDRAWING if the surface has not finished its flipping process, or one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_UNSUPPORTED

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::Flip

IDirectDrawSurface4::GetOverlayPosition

[This is preliminary documentation and subject to change.]

Given a visible, active overlay surface (DDSCAPS_OVERLAY flag set), the **IDirectDrawSurface4::GetOverlayPosition** method returns the display coordinates of the surface.

```
HRESULT GetOverlayPosition(
  LPLONG lplX,
  LPLONG lplY
);
```

Parameters

lplX and *lplY*
Addresses of the x- and y-display coordinates.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDPOSITION
DDERR_NOOVERLAYDEST
DDERR_NOTAOVERLAYSURFACE
DDERR_OVERLAYNOTVISIBLE
DDERR_SURFACELOST

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::SetOverlayPosition, **IDirectDrawSurface4::UpdateOverlay**

IDirectDrawSurface4::GetPalette

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::GetPalette** method retrieves the DirectDrawPalette object associated with this surface and increments the reference count of the returned palette.

```
HRESULT GetPalette(
  LPDIRECTDRAWPALETTE FAR *lplpDDPalette
);
```

Parameters

lplpDDPalette
Address of a variable that will be filled with a pointer to the palette object's **IDirectDrawPalette** interface.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOEXCLUSIVEMODE
DDERR_NOPALETTEATTACHED
DDERR_SURFACELOST
DDERR_UNSUPPORTED

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::SetPalette

IDirectDrawSurface4::GetPixelFormat

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::GetPixelFormat** method retrieves the color and pixel format of the surface.

```
HRESULT GetPixelFormat(
  LPDDPIXELFORMAT lpDDPixelFormat
);
```

Parameters

lpDDPixelFormat
Address of the **DDPIXELFORMAT** structure that will be filled with a detailed description of the current pixel and color space format of the surface.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawSurface4::GetPrivateData

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::GetPrivateData** method copies the private data associated with the surface to a provided buffer.

```
HRESULT GetPrivateData(
  REFGUID guidTag,
  LPVOID  lpBuffer,
  LPDWORD lpcbBufferSize
);
```

Parameters

guidTag
Reference to (C++) or address of (C) the globally unique identifier that identifies the private data to be retrieved.

lpBuffer
Address of a previously allocated buffer that will be filled with the requested private data if the call succeeds. The application calling this method is responsible for allocating and releasing this buffer.

lpcbBufferSize
Size of the buffer at *lpBuffer*, in bytes. If this value is less than the actual size of the private data (such as zero), the method sets this parameter to the required buffer size, and the method returns DDERR_MOREDATA.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_EXPIRED
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_MOREDATA
DDERR_NOTFOUND
DDERR_OUTOFMEMORY

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::SetPrivateData, **IDirectDrawSurface4::FreePrivateData**

IDirectDrawSurface4::GetSurfaceDesc

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::GetSurfaceDesc** method retrieves a description of the surface in its current condition.

```
HRESULT GetSurfaceDesc(
  LPDDSURFACEDESC2 lpDDSurfaceDesc
);
```

Parameters

lpDDSurfaceDesc

Address of a **DDSURFACEDESC2** structure that will be filled with the current description of this surface. You need only initialize this structure's **dwSize** member to the size, in bytes, of the structure prior to the call; no other preparation is required.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

DDSURFACEDESC, **DDSURFACEDESC2**

IDirectDrawSurface4::GetUniquenessValue

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::GetUniquenessValue** method retrieves the current uniqueness value for this surface.

```
HRESULT GetUniquenessValue(
  LPDWORD lpValue,
);
```

Parameters

lpValue
Address of a variable that will be filled with the surface's current uniqueness value, if the call succeeds.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

The only defined uniqueness value is 0, to indicate that the surface is likely to be changing beyond DirectDraw's control. Other uniqueness values are only significant if they differ from a previously cached uniqueness value. If the current value is different than a cached value, then the contents of the surface have changed.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::ChangeUniquenessValue, Surface Uniqueness Values

IDirectDrawSurface4::Initialize

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::Initialize** method initializes a DirectDrawSurface object.

```
HRESULT Initialize(
  LPDIRECTDRAW lpDD,
  LPDDSURFACEDESC2 lpDDSurfaceDesc
);
```

Parameters

lpDD
Address of the DirectDraw structure that represents the DirectDraw object.

lpDDSurfaceDesc
Address of a **DDSURFACEDESC2** structure that will be filled with the relevant details about the surface.

Return Values

The method returns DDERR_ALREADYINITIALIZED.

Remarks

This method is provided for compliance with the Component Object Model (COM) protocol. Because the DirectDrawSurface object is initialized when it is created, this method always returns DDERR_ALREADYINITIALIZED.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IUnknown::AddRef, **IUnknown::QueryInterface**, **IUnknown::Release**

IDirectDrawSurface4::IsLost

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::IsLost** method determines if the surface memory associated with a DirectDrawSurface object has been freed.

```
HRESULT IsLost();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DD_OK because the memory has not been freed.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST

You can use this method to determine when you need to reallocate surface memory. When a DirectDrawSurface object loses its surface memory, most methods return DDERR_SURFACELOST and perform no other action.

Remarks

Surfaces can lose their memory when the mode of the display card is changed, or when an application receives exclusive access to the display card and frees all of the surface memory currently allocated on the display card.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::Restore, Losing and Restoring Surfaces

IDirectDrawSurface4::Lock

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::Lock** method obtains a pointer to the surface memory.

```
HRESULT Lock(
  LPRECT lpDestRect,
  LPDDSURFACEDESC2 lpDDSurfaceDesc,
  DWORD dwFlags,
  HANDLE hEvent
);
```

Parameters

lpDestRect
Address of a **RECT** structure that identifies the region of surface that is being locked. If NULL, the entire surface will be locked.

lpDDSurfaceDesc
Address of a **DDSURFACEDESC2** structure that will be filled with the relevant details about the surface.

dwFlags
DDLOCK_EVENT
This flag is not currently implemented.
DDLOCK_NOSYSLOCK
Do not take the Win16Mutex (also known as Win16Lock). This flag is ignored when locking the primary surface.
DDLOCK_READONLY
Indicates that the surface being locked will only be read.

DDLOCK_SURFACEMEMORYPTR
Indicates that a valid memory pointer to the top of the specified rectangle should be returned. If no rectangle is specified, a pointer to the top of the surface is returned. This is the default.

DDLOCK_WAIT
If a lock cannot be obtained because a blit operation is in progress, the method retries until a lock is obtained or another error occurs, such as DDERR_SURFACEBUSY.

DDLOCK_WRITEONLY
Indicates that the surface being locked will be write enabled.

hEvent
This parameter is not used and must be set to NULL.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_WASSTILLDRAWING

Remarks

For more information on using this method, see Accessing Surface Memory Directly.

After retrieving a surface memory pointer, you can access the surface memory until a corresponding **IDirectDrawSurface4::Unlock** method is called. When the surface is unlocked, the pointer to the surface memory is invalid.

Do not call DirectDraw blit functions to blit from a locked region of a surface. If you do, the blit returns either DDERR_SURFACEBUSY or DDERR_LOCKEDSURFACES. Additionally, GDI blit functions will silently fail when used on a locked video memory surface.

Unless you include the DDLOCK_NOSYSLOCK flag, this method causes DirectDraw to hold the Win16Mutex (also known as Win16Lock) until you call the **IDirectDrawSurface4::Unlock** method. GUI debuggers cannot operate while the Win16Mutex is held.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::Unlock, **IDirectDrawSurface4::GetDC**, **IDirectDrawSurface4::ReleaseDC**

IDirectDrawSurface4::PageLock

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::PageLock** method prevents a system-memory surface from being paged out while a blit operation using direct memory access (DMA) transfers to or from system memory is in progress.

```
HRESULT PageLock(
  DWORD dwFlags
);
```

Parameters

dwFlags
This parameter is currently not used and must be set to 0.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_CANTPAGELOCK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST

Remarks

You must call this method to make use of DMA support. If you do not, the blit occurs using software emulation. For more information, see Using DMA.

The performance of the operating system could be negatively affected if too much memory is locked.

A lock count is maintained for each surface and is incremented each time **IDirectDrawSurface4::PageLock** is called for that surface. The count is decremented when **IDirectDrawSurface4::PageUnlock** is called. When the count reaches 0, the memory is unlocked and can then be paged by the operating system.

This method works only on system-memory surfaces; it will not page lock a display-memory surface or an emulated primary surface. If an application calls this method on a display memory surface, the method will do nothing except return DD_OK.

This method was not implemented in the **IDirectDraw** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::PageUnlock

IDirectDrawSurface4::PageUnlock

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::PageUnlock** method unlocks a system-memory surface, allowing it to be paged out.

```
HRESULT PageUnlock(
  DWORD dwFlags
);
```

Parameters

dwFlags
This parameter is currently not used and must be set to 0.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_CANTPAGEUNLOCK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTPAGELOCKED

DDERR_SURFACELOST

Remarks

A lock count is maintained for each surface and is incremented each time **IDirectDrawSurface4::PageLock** is called for that surface. The count is decremented when **IDirectDrawSurface4::PageUnlock** is called. When the count reaches 0, the memory is unlocked and can then be paged by the operating system.

This method works only on system-memory surfaces; it will not page unlock a display-memory surface or an emulated primary surface. If an application calls this method on a display-memory surface, this method will do nothing except return DD_OK.

This method was not implemented in the **IDirectDraw** interface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::PageLock

IDirectDrawSurface4::ReleaseDC

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::ReleaseDC** method releases the handle of a device context previously obtained by using the **IDirectDrawSurface4::GetDC** method.

```
HRESULT ReleaseDC(
  HDC hDC
);
```

Parameters

hDC
Handle to a device context previously obtained by **IDirectDrawSurface4::GetDC**.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST
DDERR_UNSUPPORTED

Remarks

This method also unlocks the surface previously locked when the **IDirectDrawSurface4::GetDC** method was called.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::GetDC

IDirectDrawSurface4::Restore

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::Restore** method restores a surface that has been lost. This occurs when the surface memory associated with the DirectDrawSurface object has been freed.

```
HRESULT Restore();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_IMPLICITLYCREATED
DDERR_INCOMPATIBLEPRIMARY
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOEXCLUSIVEMODE
DDERR_OUTOFMEMORY
DDERR_UNSUPPORTED
DDERR_WRONGMODE

Remarks

This method restores the memory allocated for a surface, but doesn't reload any bitmaps that may have existed in the surface before it was lost. For more information, see Losing and Restoring Surfaces.

Surfaces can be lost because the mode of the display card was changed or because an application received exclusive access to the display card and freed all of the surface memory currently allocated on the card. When a DirectDrawSurface object loses its surface memory, many methods will return DDERR_SURFACELOST and perform no other function. The **IDirectDrawSurface4::Restore** method will reallocate surface memory and reattach it to the DirectDrawSurface object.

A single call to this method will restore a DirectDrawSurface object's associated implicit surfaces (back buffers, and so on). An attempt to restore an implicitly created surface will result in an error. **IDirectDrawSurface4::Restore** will not work across explicit attachments created by using the **IDirectDrawSurface4::AddAttachedSurface** method—each of these surfaces must be restored individually.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::IsLost

IDirectDrawSurface4::SetClipper

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::SetClipper** method attaches a clipper object to or deletes one from a surface.

```
HRESULT SetClipper(
  LPDIRECTDRAWCLIPPER lpDDClipper
);
```

Parameters

lpDDClipper
Address of the **IDirectDrawClipper** interface for the DirectDrawClipper object that will be attached to the DirectDrawSurface object. If this parameter is NULL, the current DirectDrawClipper object will be detached.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE
DDERR_NOCLIPPERATTACHED

Remarks

When setting a clipper to a surface for the first time, this method increments the clipper's reference count; subsequent calls to do not affect the clipper's reference count. If you pass NULL as the *lpDDClipper* parameter, the clipper is removed from the surface, and the clipper's reference count is decremented. If you do not delete the clipper, the surface will automatically release its reference to the clipper when the surface itself is released. According to COM rules, your application is responsible for releasing any references it holds to the clipper when the object is no longer needed.

This method is primarily used by surfaces that are being overlaid on or blitted to the primary surface. However, it can be used on any surface. After a DirectDrawClipper object has been attached and a clip list is associated with it, the DirectDrawClipper object will be used for the **IDirectDrawSurface4::Blt**, **IDirectDrawSurface4::BltBatch**, and **IDirectDrawSurface4::UpdateOverlay** operations involving the parent DirectDrawSurface object. This method can also detach a DirectDrawSurface object's current DirectDrawClipper object.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::GetClipper

IDirectDrawSurface4::SetColorKey

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::SetColorKey** method sets the color key value for the DirectDrawSurface object if the hardware supports color keys on a per surface basis.

```
HRESULT SetColorKey(
  DWORD dwFlags,
  LPDDCOLORKEY lpDDColorKey
);
```

Parameters

dwFlags
Determines which color key is requested.

DDCKEY_COLORSPACE
Set if the structure contains a color space. Not set if the structure contains a single color key.

DDCKEY_DESTBLT
Set if the structure specifies a color key or color space to be used as a destination color key for blit operations.

DDCKEY_DESTOVERLAY
Set if the structure specifies a color key or color space to be used as a destination color key for overlay operations.

DDCKEY_SRCBLT
Set if the structure specifies a color key or color space to be used as a source color key for blit operations.

DDCKEY_SRCOVERLAY
Set if the structure specifies a color key or color space to be used as a source color key for overlay operations.

lpDDColorKey

Address of the **DDCOLORKEY** structure that contains the new color key values for the DirectDrawSurface object. This value can be NULL to remove a previously set color key.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE
DDERR_NOOVERLAYHW
DDERR_NOTAOVERLAYSURFACE
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING

Remarks

For transparent blits and overlays, you should set destination color on the destination surface and source color on the source surface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::GetColorKey

IDirectDrawSurface4::SetOverlayPosition

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::SetOverlayPosition** method changes the display coordinates of an overlay surface.

HRESULT SetOverlayPosition(

```
  LONG lX,
  LONG lY
);
```

Parameters

lX and *lY*
New x- and y-display coordinates.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDPOSITION
DDERR_NOOVERLAYDEST
DDERR_NOTAOVERLAYSURFACE
DDERR_OVERLAYNOTVISIBLE
DDERR_SURFACELOST
DDERR_UNSUPPORTED

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::GetOverlayPosition,
IDirectDrawSurface4::UpdateOverlay

IDirectDrawSurface4::SetPalette

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::SetPalette** method attaches a palette object to (or detaches one from) a surface. The surface uses this palette for all subsequent

operations. The palette change takes place immediately, without regard to refresh timing.

```
HRESULT SetPalette(
  LPDIRECTDRAWPALETTE lpDDPalette
);
```

Parameters

lpDDPalette

Address of the **IDirectDrawPalette** interface for the palette object to be used with this surface. If this parameter is NULL, the current palette will be detached.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDPIXELFORMAT
DDERR_INVALIDSURFACETYPE
DDERR_NOEXCLUSIVEMODE
DDERR_NOPALETTEATTACHED
DDERR_NOPALETTEHW
DDERR_NOT8BITCOLOR
DDERR_SURFACELOST
DDERR_UNSUPPORTED

Remarks

When setting a palette to a surface for the first time, this method increments the palette's reference count; subsequent calls to do not affect the palette's reference count. If you pass NULL as the *lpDDPalette* parameter, the palette is removed from the surface, and the palette's reference count is decremented. If you do not delete the palette, the surface will automatically release its reference to the palette when the surface itself is released. According to COM rules, your application is responsible for releasing any references it holds to the palette when the object is no longer needed.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::GetPalette, **IDirectDraw4::CreatePalette**

IDirectDrawSurface4::SetPrivateData

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::SetPrivateData** method associates data with the surface that is intended for use by the application, not by DirectDraw. Data is passed by value, and multiple sets of data can be associated with a single surface.

```
HRESULT SetPrivateData(
  REFGUID guidTag,
  LPVOID  lpData,
  DWORD   cbSize,
  DWORD   dwFlags
);
```

Parameters

guidTag
Reference to (C++) or address of (C) the globally unique identifier that identifies the private data to be set.

lpData
Address of a buffer that contains the data to be associated with the surface.

cbSize
Size of the buffer at *lpData*, in bytes.

dwFlags
Flags describing the type of data being passed, or requesting that the data be invalidated when the surface changes. The following flags are defined:

(none)
If no flags are specified, DirectDraw allocates memory to hold the data within the buffer, and copies the data into the new buffer. The buffer allocated by DirectDraw will automatically be freed as appropriate.

DDSPD_IUNKNOWNPOINTER

The data at *lpData* is a pointer to an **IUnknown** interface. DirectDraw automatically calls the **IUnknown::AddRef** method of this interface. When this data is no longer needed, DirectDraw automatically calls the **IUnknown::Release** method of this interface.

DDSPD_VOLATILE
The buffer at *lpData* is only valid while the surface remains unchanged from its current state. If the surface's contents change, subsequent calls to the **IDirectDrawSurface4::GetPrivateData** method will return DDERR_EXPIRED.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

Remarks

DirectDraw does not manage the memory at *lpData*. If this buffer was dynamically allocated, it is the caller's responsibility to free the memory.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::GetPrivateData, **IDirectDrawSurface4::FreePrivateData**

IDirectDrawSurface4::SetSurfaceDesc

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::SetSurfaceDesc** method sets the characteristics of an existing surface.

HRESULT SetSurfaceDesc(
LPDDSURFACEDESC2 *lpddsd2*,

DWORD *dwFlags*
);

Parameters

lpddsd2
Address of a **DDSURFACEDESC2** structure that contains the new surface characteristics.

dwFlags
This parameter is currently not used and must be set to 0.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDPARAMS
DDERR_INVALIDOBJECT
DDERR_SURFACELOST
DDERR_SURFACEBUSY
DDERR_INVALIDSURFACETYPE
DDERR_INVALIDPIXELFORMAT
DDERR_INVALIDCAPS
DDERR_UNSUPPORTED
DDERR_GENERIC

Remarks

Currently, this method can only be used to set the surface data and pixel format used by an explicit system memory surface. This is useful as it allows a surface to use data from a previously allocated buffer without copying. The new surface memory is allocated by the client application and, as such, the client application must also deallocate it. For more information about how this method is used, see Updating Surface Characteristics.

Using this method incorrectly will cause unpredictable behavior. The DirectDrawSurface object will not deallocate surface memory that it didn't allocate. Therefore, when the surface memory is no longer needed, it is your responsibility to deallocate it. However, when this method is called, DirectDraw frees the original surface memory that it implicitly allocated when creating the surface.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawSurface4::Unlock

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::Unlock** method notifies DirectDraw that the direct surface manipulations are complete.

```
HRESULT Unlock(
  LPRECT lpRect
);
```

Parameters

lpRect
Address of the **RECT** structure that was used to lock the surface in the corresponding call to the **IDirectDrawSurface4::Lock** method. This parameter can be NULL only if the entire surface was locked by passing NULL in the *lpDestRect* parameter of the corresponding call to the **IDirectDrawSurface4::Lock** method.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDRECT
DDERR_NOTLOCKED
DDERR_SURFACELOST

Remarks

Because it is possible to call **IDirectDrawSurface4::Lock** multiple times for the same surface with different destination rectangles, the pointer in *lpRect* links the calls to the **IDirectDrawSurface4::Lock** and **IDirectDrawSurface4::Unlock** methods.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::Lock

IDirectDrawSurface4::UpdateOverlay

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::UpdateOverlay** method repositions or modifies the visual attributes of an overlay surface. These surfaces must have the DDSCAPS_OVERLAY flag set.

```
HRESULT UpdateOverlay(
  LPRECT lpSrcRect,
  LPDIRECTDRAWSURFACE4 lpDDDestSurface,
  LPRECT lpDestRect,
  DWORD dwFlags,
  LPDDOVERLAYFX lpDDOverlayFx
);
```

Parameters

lpSrcRect
Address of a **RECT** structure that defines the x, y, width, and height of the region on the source surface being used as the overlay. This parameter can be NULL when hiding an overlay or to indicate that the entire overlay surface is to be used and that the overlay surface conforms to any boundary and size alignment restrictions imposed by the device driver.

lpDDDestSurface
Address of the **IDirectDrawSurface4** interface for the surface that is being overlaid.

lpDestRect
Address of a **RECT** structure that defines the x, y, width, and height of the region on the destination surface that the overlay should be moved to. This parameter can be NULL when hiding the overlay.

dwFlags
DDOVER_ADDDIRTYRECT

Adds a dirty rectangle to an emulated overlay surface.

DDOVER_ALPHADEST
Uses either the alpha information in pixel format or the alpha channel surface attached to the destination surface as the alpha channel for this overlay.

DDOVER_ALPHADESTCONSTOVERRIDE
Uses the **dwAlphaDestConst** member of the **DDOVERLAYFX** structure as the destination alpha channel for this overlay.

DDOVER_ALPHADESTNEG
Indicates that the destination surface becomes more transparent as the alpha value increases (0 is opaque).

DDOVER_ALPHADESTSURFACEOVERRIDE
Uses the **lpDDSAlphaDest** member of the **DDOVERLAYFX** structure as the alpha channel destination for this overlay.

DDOVER_ALPHAEDGEBLEND
Uses the **dwAlphaEdgeBlend** member of the **DDOVERLAYFX** structure as the alpha channel for the edges of the image that border the color key colors.

DDOVER_ALPHASRC
Uses either the alpha information in pixel format or the alpha channel surface attached to the source surface as the source alpha channel for this overlay.

DDOVER_ALPHASRCCONSTOVERRIDE
Uses the **dwAlphaSrcConst** member of the **DDOVERLAYFX** structure as the source alpha channel for this overlay.

DDOVER_ALPHASRCNEG
Indicates that the source surface becomes more transparent as the alpha value increases (0 is opaque).

DDOVER_ALPHASRCSURFACEOVERRIDE
Uses the **lpDDSAlphaSrc** member of the **DDOVERLAYFX** structure as the alpha channel source for this overlay.

DDOVER_AUTOFLIP
Automatically flip to the next surface in the flip chain each time a video port VSYNC occurs.

DDOVER_BOB
Display each field individually of the interlaced video stream without causing any artifacts.

DDOVER_BOBHARDWARE
Indicates that bob operations will be performed using hardware rather than software or emulated. This flag must be used with the DDOVER_BOB flag.

DDOVER_DDFX
Uses the overlay FX flags in the *lpDDOverlayFx* parameter to define special overlay effects.

DDOVER_HIDE
Turns off this overlay.

DDOVER_KEYDEST
Uses the color key associated with the destination surface.

DDOVER_KEYDESTOVERRIDE
Uses the **dckDestColorkey** member of the **DDOVERLAYFX** structure as the color key for the destination surface.

DDOVER_KEYSRC
Uses the color key associated with the source surface.

DDOVER_KEYSRCOVERRIDE
Uses the **dckSrcColorkey** member of the **DDOVERLAYFX** structure as the color key for the source surface.

DDOVER_OVERRIDEBOBWEAVE
Indicates that bob/weave decisions should not be overridden by other interfaces.

DDOVER_INTERLEAVED
Indicates that the surface memory is composed of interleaved fields.

DDOVER_SHOW
Turns on this overlay.

lpDDOverlayFx
Address of a **DDOVERLAYFX** structure that describes the effects to be used. This parameter can be NULL if the DDOVER_DDFX flag is not specified.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_DEVICEDOESNTOWNSURFACE
DDERR_GENERIC
DDERR_HEIGHTALIGN
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDRECT
DDERR_INVALIDSURFACETYPE
DDERR_NOSTRETCHHW
DDERR_NOTAOVERLAYSURFACE
DDERR_OUTOFCAPS
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_XALIGN

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

IDirectDrawSurface4::UpdateOverlayDisplay

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::UpdateOverlayDisplay** method is not currently implemented.

```
HRESULT UpdateOverlayDisplay(
  DWORD dwFlags
);
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::AddOverlayDirtyRect

IDirectDrawSurface4::UpdateOverlayZOrder

[This is preliminary documentation and subject to change.]

The **IDirectDrawSurface4::UpdateOverlayZOrder** method sets the z-order of an overlay.

```
HRESULT UpdateOverlayZOrder(
  DWORD dwFlags,
  LPDIRECTDRAWSURFACE4 lpDDSReference
);
```

Parameters

dwFlags
One of the following flags:
DDOVERZ_INSERTINBACKOF

Inserts this overlay in the overlay chain behind the reference overlay.

DDOVERZ_INSERTINFRONTOF
Inserts this overlay in the overlay chain in front of the reference overlay.

DDOVERZ_MOVEBACKWARD
Moves this overlay one position backward in the overlay chain.

DDOVERZ_MOVEFORWARD
Moves this overlay one position forward in the overlay chain.

DDOVERZ_SENDTOBACK
Moves this overlay to the back of the overlay chain.

DDOVERZ_SENDTOFRONT
Moves this overlay to the front of the overlay chain.

lpDDSReference
Address of the **IDirectDrawSurface4** interface for the DirectDraw surface to be used as a relative position in the overlay chain. This parameter is needed only for DDOVERZ_INSERTINBACKOF and DDOVERZ_INSERTINFRONTOF.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_NOTAOVERLAYSURFACE

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawSurface4::EnumOverlayZOrders

IDirectDrawVideoPort

[This is preliminary documentation and subject to change.]

Applications use the methods of the **IDirectDrawVideoPort** interface to channel live video data from a hardware video port to a DirectDraw surface. This section is a reference to the methods of this interface. For a conceptual overview, see Video Ports.

The methods of the **IDirectDrawVideoPort** interface can be organized into the following groups:

Color controls	**GetColorControls**
	SetColorControls
Fields and Signals	**GetFieldPolarity**
	GetVideoSignalStatus
Flipping	**Flip**
	SetTargetSurface
Formats	**GetInputFormats**
	GetOutputFormats
Timing and Synchronization	**GetVideoLine**
	WaitForSync
Video control	**StartVideo**
	StopVideo
	UpdateVideo
Zoom factors	**GetBandwidthInfo**

The **IDirectDrawVideoPort** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

IUnknown	**AddRef**
	QueryInterface
	Release

You can use the **LPDIRECTDRAWVIDEOPORT** data type to declare a variable that contains a pointer to an **IDirectDrawVideoPort** interface. The Dvp.h header file declares the **LPDIRECTDRAWVIDEOPORT** with the following code:

```
typedef struct IDirectDrawVideoPort    FAR *LPDIRECTDRAWVIDEOPORT;
```

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.

Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

IDirectDrawVideoPort::Flip

[This is preliminary documentation and subject to change.]

The **IDirectDrawVideoPort::Flip** method instructs the DirectDrawVideoPort object to write the next frame of video to a new surface.

```
HRESULT Flip(
  LPDIRECTDRAWSURFACE lpDDSurface,
  DWORD dwFlags
);
```

Parameters

lpDDSurface
Address of the **IDirectDrawSurface** interface for the surface that will receive the next frame of video. Setting this parameter to NULL causes DirectDraw to cycle through surfaces in the flipping chain in the order they were attached.

dwFlags
Flip options flags. This parameter can be one of the following values:

DDVPFLIP_VIDEO
The specified surface is to receive the normal video data.

DDVPFLIP_VBI
The specified surface is to receive only the data within the vertical blanking interval.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

Remarks

This method can be used to prevent tearing. Calls to **IDirectDrawVideoPort::Flip** are asynchronous—the actual flip operation will always be synchronized with the vertical blank of the video signal.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

IDirectDrawVideoPort::GetBandwidthInfo

[This is preliminary documentation and subject to change.]

The **IDirectDrawVideoPort::GetBandwidthInfo** method retrieves the minimum required overlay zoom factors and device limitations of a video port that uses the provided output pixel format.

```
HRESULT GetBandwidthInfo(
  LPDDPIXELFORMAT lpddpfFormat,
  DWORD dwWidth,
  DWORD dwHeight,
  DWORD dwFlags,
  LPDDVIDEOPORTBANDWIDTH lpBandwidth
);
```

Parameters

lpddpfFormat
Address of a **DDPIXELFORMAT** structure that describes the output pixel format for which bandwidth information will be retrieved.

dwWidth and *dwHeight*
Dimensions of an overlay or video data. These interpretation of these parameters depends on the value specified in the *dwFlags* parameter.

dwFlags
Flags indicating how the method is to interpret the *dwWidth* and *dwHeight* parameters. This parameter can be one of the following values:

DDVPB_OVERLAY
The *dwWidth* and *dwHeight* parameters indicate the size of the source overlay surface. Use this flag when the video port is dependent on the overlay source size.

DDVPB_TYPE
The *dwWidth* and *dwHeight* parameters are not set. The method will retrieve the device's dependency type in the **dwCaps** member of the associated **DDVIDEOPORTBANDWIDTH** structure. Use this flag when you call this method the first time.

DDVPB_VIDEOPORT

The *dwWidth* and *dwHeight* parameters indicate the prescale size of the video data that the video port writes to the frame buffer. Use this flag when the video port is dependent on the overlay zoom factor.

lpBandwidth

Address of a **DDVIDEOPORTBANDWIDTH** structure that will be filled with the retrieved bandwidth and device dependency information.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method will usually be called twice. When you make the first call, specify the DDVPB_TYPE flag in the *dwFlags* parameter to retrieve information about the device's overlay dependency type. Subsequent calls using the DDVPB_VIDEOPORT or DDVPB_OVERLAY flags must be interpreted considering the device's dependency type.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

IDirectDrawVideoPort::GetColorControls

[This is preliminary documentation and subject to change.]

The **IDirectDrawVideoPort::GetColorControls** method returns the current color control settings associated with the video port.

```
HRESULT GetColorControls(
  LPDDCOLORCONTROL lpColorControl
);
```

Parameters

lpColorControl

Address of a **DDCOLORCONTROL** structure that will be filled with the current settings of the video port's color control.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED

Remarks

The **dwFlags** member of the **DDCOLORCONTROL** structure indicate which of the color control options are supported.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

IDirectDrawVideoPort::GetFieldPolarity

[This is preliminary documentation and subject to change.]

The **IDirectDrawVideoPort::GetFieldPolarity** method retrieves the polarity of a video field.

```
HRESULT GetFieldPolarity(
  LPBOOL lpbFieldPolarity
);
```

Parameters

lpbFieldPolarity
Address of a variable that will be set to indicate the current field polarity. This value is set to TRUE if the current video field is the even field of an interlaced video signal and FALSE otherwise.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED
DDERR_VIDEONOTACTIVE

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

IDirectDrawVideoPort::GetInputFormats

[This is preliminary documentation and subject to change.]

The **IDirectDrawVideoPort::GetInputFormats** method retrieves the input formats supported by the DirectDrawVideoPort object.

```
HRESULT GetInputFormats(
  LPDWORD lpNumFormats,
  LPDDPIXELFORMAT lpFormats,
  DWORD dwFlags
);
```

Parameters

lpNumFormats
Address of a variable containing the number of entries that the array at *lpFormats* can hold. If this number is less than the total number of codes, the method fills the array with as many codes as will fit, sets the value at *lpNumFormats* to indicate the total number of codes, and returns DDERR_MOREDATA.

lpFormats
Address of an array of **DDPIXELFORMAT** structures that will be filled in with the input formats supported by this DirectDrawVideoPort object. If this parameter is NULL, the method sets *lpNumFormats* to the number of supported formats and then returns DD_OK.

dwFlags
Flags specifying the part of the video signal for which formats will be enumerated. This parameter can be one of the following values:

DDVPFORMAT_VIDEO
Returns formats for the video data.

DDVPFORMAT_VBI
Returns formats for the VBI data.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_MOREDATA

Remarks

This method can also be used to return the number of formats supported. To do this, set the *lpFormats* parameter to NULL. When the method returns, the variable at *lpNumFormats* contains the total number of supported input formats.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

IDirectDrawVideoPort::GetOutputFormats

[This is preliminary documentation and subject to change.]

The **IDirectDrawVideoPort::GetOutputFormats** method retrieves a list of output formats that the DirectDrawVideoPort object supports for a specified input format.

```
HRESULT GetOutputFormats(
  LPDDPIXELFORMAT lpInputFormat,
  LPDWORD lpNumFormats,
  LPDDPIXELFORMAT lpFormats,
  DWORD dwFlags
);
```

Parameters

lpInputFormat
Address of a **DDPIXELFORMAT** structure that describes the input format for which conversion information is requested.

lpNumFormats
Address of a variable containing the number of entries that the array at *lpFromats* can hold. If this number is less than the total number of codes, the method fills the array with as many codes as will fit, sets the value at *lpNumFormats* to indicate the total number of codes, and returns DDERR_MOREDATA.

lpFormats
Address of an array of **DDPIXELFORMAT** structures that will be filled in with the output formats supported by this DirectDrawVideoPort object. If this parameter is NULL, the method sets *lpNumFormats* to the number of supported formats and then returns DD_OK.

dwFlags
Flags specifying the part of the video signal for which formats will be enumerated. This parameter can be one of the following values:

DDVPFORMAT_VIDEO
Returns formats for the video data.

DDVPFORMAT_VBI
Returns formats for the VBI data.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_MOREDATA

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

IDirectDrawVideoPort::GetVideoLine

[This is preliminary documentation and subject to change.]

The **IDirectDrawVideoPort::GetVideoLine** method retrieves the current line of video being written to the frame buffer.

```
HRESULT GetVideoLine(
  LPDWORD lpdwLine
);
```

Parameters

lpdwLine
Address of a variable that will be filled with a value indicating the video line currently being written to the frame buffer.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED
DDERR_VERTICALBLANKINPROGRESS
DDERR_VIDEONOTACTIVE

Remarks

The value this method retrieves reflects the true video line being written, relative to the field height, before any prescaling occurs.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

IDirectDrawVideoPort::GetVideoSignalStatus

[This is preliminary documentation and subject to change.]

The **IDirectDrawVideoPort::GetVideoSignalStatus** method retrieves the status of the video signal currently being presented to the video port.

```
HRESULT GetVideoSignalStatus(
  LPDWORD lpdwStatus
);
```

Parameters

lpdwStatus

Address of a variable that will contain a return code indicating the quality of the video signal at the video port. The value will be set to one of the following codes:

DDVPSQ_NOSIGNAL

No video signal is present at the video port.

DDVPSQ_SIGNALOK

A valid video signal is present at the video port.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_UNSUPPORTED

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

IDirectDrawVideoPort::SetColorControls

[This is preliminary documentation and subject to change.]

The **IDirectDrawVideoPort::SetColorControls** method sets the color control settings associated with the video port.

```
HRESULT SetColorControls(
  LPDDCOLORCONTROL lpColorControl
);
```

Parameters

lpColorControl
Address of a **DDCOLORCONTROL** structure containing the new color control settings that will be applied to the video port.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_UNSUPPORTED

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

IDirectDrawVideoPort::SetTargetSurface

[This is preliminary documentation and subject to change.]

The **IDirectDrawVideoPort::SetTargetSurface** method sets the DirectDraw surface object that will receive the stream of live video data and/or the vertical blank interval data.

```
HRESULT SetTargetSurface(
  LPDIRECTDRAWSURFACE lpDDSurface,
  DWORD dwFlags
);
```

Parameters

lpDDSurface
Address of the DirectDrawSurface object that will receive the video data.

dwFlags
Value specifying the type of target surface.

DDVPTARGET_VIDEO
The specified surface should receive the normal video data and vertical interval data unless a separate surface was attached for this purpose.

DDVPTARGET_VBI
The specified surface should receive the data within the vertical blanking interval.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawVideoPort::StartVideo, **IDirectDrawVideoPort::StopVideo**, **IDirectDrawVideoPort::UpdateVideo**

IDirectDrawVideoPort::StartVideo

[This is preliminary documentation and subject to change.]

The **IDirectDrawVideoPort::StartVideo** method enables the hardware video port and starts the flow of video data into the currently specified surface.

```
HRESULT StartVideo(
  LPDDVIDEOPORTINFO lpVideoInfo
);
```

Parameters

lpVideoInfo
Address of a pointer to a **DDVIDEOPORTINFO** structure.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawVideoPort::SetTargetSurface, **IDirectDrawVideoPort::StopVideo**, **IDirectDrawVideoPort::UpdateVideo**

IDirectDrawVideoPort::StopVideo

[This is preliminary documentation and subject to change.]

The **IDirectDrawVideoPort::StopVideo** method stops the flow of video port data into the frame buffer.

```
HRESULT StopVideo();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is DDERR_INVALIDOBJECT.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawVideoPort::SetTargetSurface, I**IDirectDrawVideoPort::StartVideo**, **IDirectDrawVideoPort::UpdateVideo**

IDirectDrawVideoPort::UpdateVideo

[This is preliminary documentation and subject to change.]

The **IDirectDrawVideoPort::UpdateVideo** method updates parameters that govern the flow of video data from the video port to the DirectDrawSurface object.

```
HRESULT UpdateVideo(
  LPDDVIDEOPORTINFO lpVideoInfo
);
```

Parameters

lpVideoInfo
Address of a **DDVIDEOPORTINFO** structure that describes the video transfer parameters.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

See Also

IDirectDrawVideoPort::SetTargetSurface, **IDirectDrawVideoPort::StartVideo**, **IDirectDrawVideoPort::StopVideo**

IDirectDrawVideoPort::WaitForSync

[This is preliminary documentation and subject to change.]

The **IDirectDrawVideoPort::WaitForSync** method waits for VSYNC or until a given scan line is being drawn.

```
HRESULT WaitForSync(
  DWORD dwFlags,
  DWORD dwLine,
  DWORD dwTimeout
);
```

Parameters

dwFlags

Flag specifying how the method will wait for the video VSYNC or the specified line number.

DDVPWAIT_BEGIN

Return at the start of the vertical blanking interval.

DDVPWAIT_END

Return at the end of the vertical blanking interval.

DDVPWAIT_LINE

Return when the video counter either reaches or passes the line specified by the *dwLine* parameter.

dwLine

The video line determining when the method should return, relative to the field height, before prescaling. This parameter is ignored if the *dwFlags* parameter is set to DDVPWAIT_BEGIN or DDVPWAIT_END.

dwTimeout

Amount of time, in milliseconds, that the method will wait for the next video vertical blank before timing out. If this parameter is 0, the method waits 3 times the value specified in the **dwMicrosecondsPerField** member of the **DDVIDEOPORTDESC**.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED
DDERR_VIDEONOTACTIVE
DDERR_WASSTILLDRAWING

Remarks

This method helps the caller synchronize with the video vertical blank interval or with an arbitrary line of video data. The method blocks the calling thread until either the video VSYNC occurs or when the video line counter matches the specified line number.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.
Import Library: Use ddraw.lib.

Functions

[This is preliminary documentation and subject to change.]

This section contains information about the following DirectDraw global functions:

- **DirectDrawCreate**
- **DirectDrawCreateClipper**
- **DirectDrawEnumerate**
- **DirectDrawEnumerateEx**

DirectDrawCreate

[This is preliminary documentation and subject to change.]

The **DirectDrawCreate** function creates an instance of a DirectDraw object.

```
HRESULT WINAPI DirectDrawCreate(
  GUID FAR *lpGUID,
  LPDIRECTDRAW FAR *lplpDD,
  IUnknown FAR *pUnkOuter
);
```

Parameters

lpGUID

Address of the globally unique identifier (GUID) that represents the driver to be created. This can be NULL to indicate the active display driver, or you can pass one of the following flags to restrict the active display driver's behavior for debugging purposes:

DDCREATE_EMULATIONONLY

The DirectDraw object will use emulation for all features; it will not take advantage of any hardware supported features.

DDCREATE_HARDWAREONLY

The DirectDraw object will never emulate features not supported by the hardware. Attempts to call methods that require unsupported features will fail, returning DDERR_UNSUPPORTED.

lplpDD

Address of a variable that will be set to a valid **IDirectDraw** interface pointer if the call succeeds.

pUnkOuter

Allows for future compatibility with COM aggregation features. Presently, however, this method will return an error if this parameter is anything but NULL.

Return Values

If the function succeeds, the return value is DD_OK.

If the function fails, the return value may be one of the following error values:

DDERR_DIRECTDRAWALREADYCREATED

DDERR_GENERIC

DDERR_INVALIDDIRECTDRAWGUID

DDERR_INVALIDPARAMS

DDERR_NODIRECTDRAWHW

DDERR_OUTOFMEMORY

Remarks

This function attempts to initialize a DirectDraw object, and it then sets a pointer to the object if the call is successful.

On systems with multiple monitors, specifying NULL for *lpGUID* causes the DirectDraw object to run in emulation mode when the normal cooperative level is set. To make use of hardware acceleration on these systems, you must specify the device's GUID. For more information, see Devices and Acceleration in MultiMon Systems.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

DirectDrawCreateClipper

[This is preliminary documentation and subject to change.]

The **DirectDrawCreateClipper** function creates an instance of a DirectDrawClipper object not associated with a DirectDraw object.

```
HRESULT WINAPI DirectDrawCreateClipper(
  DWORD dwFlags,
  LPDIRECTDRAWCLIPPER FAR *lplpDDClipper,
  IUnknown FAR *pUnkOuter
);
```

Parameters

dwFlags
This parameter is currently not used and must be set to 0.

lplpDDClipper
Address of a pointer that will be filled with the address of the new DirectDrawClipper object.

pUnkOuter
Allows for future compatibility with COM aggregation features. Presently, however, this method will return an error if this parameter is anything but NULL.

Return Values

If the function succeeds, the return value is DD_OK.

If the function fails, the return value may be one of the following error values:

DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

Remarks

This function can be called before any DirectDraw objects are created. Because these DirectDrawClipper objects are not owned by any DirectDraw object, they are not automatically released when an application's objects are released. If the

application does not explicitly release the DirectDrawClipper objects, DirectDraw will release them when the application terminates.

To create a DirectDrawClipper object owned by a specific DirectDraw object, use the **IDirectDraw4::CreateClipper** method.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

IDirectDraw4::CreateClipper

DirectDrawEnumerate

[This is preliminary documentation and subject to change.]

This function is superseded by the **DirectDrawEnumerateEx** function.

The **DirectDrawEnumerate** function enumerates the primary DirectDraw display device and a non-display device (such as a 3-D accelerator that has no 2-D capabilities) if one is installed. The NULL entry always identifies the primary display device shared with GDI.

```
HRESULT WINAPI DirectDrawEnumerate(
  LPDDENUMCALLBACK lpCallback,
  LPVOID lpContext
);
```

Parameters

lpCallback
Address of a **DDEnumCallback** function that will be called with a description of each enumerated DirectDraw-enabled HAL.

lpContext
Address of an application-defined context that will be passed to the enumeration callback function each time it is called.

Return Values

If the function succeeds, the return value is DD_OK.

If the function fails, the return value is DDERR_INVALIDPARAMS.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.

See Also

DirectDrawEnumerateEx

DirectDrawEnumerateEx

[This is preliminary documentation and subject to change.]

The **DirectDrawEnumerateEx** function enumerates all DirectDraw devices installed on the system. The NULL entry always identifies the primary display device shared with GDI.

```
HRESULT WINAPI DirectDrawEnumerateEx(
  LPDDENUMCALLBACKEX lpCallback,
  LPVOID lpContext,
  DWORD  dwFlags
);
```

Parameters

lpCallback
Address of a **DDEnumCallbackEx** function that will be called with a description of each enumerated DirectDraw-enabled HAL.

lpContext
Address of an application-defined value that will be passed to the enumeration callback function each time it is called.

dwFlags
Flags specifying the enumeration scope. This parameter can be 0 or a combination of the following flags. If the value is 0, the function will enumerate only the primary display device.

DDENUM_ATTACHEDSECONDARYDEVICES
The function will enumerate the primary device, and any display devices that are attached to the desktop.

DDENUM_DETACHEDSECONDARYDEVICES
The function will enumerate the primary device, and any display devices that are not attached to the desktop.

DDENUM_NONDISPLAYDEVICES

The function will enumerate the primary device, and any non-display devices, such as 3-D accelerators that have no 2-D capabilities.

Return Values

If the function succeeds, the return value is DD_OK.

If the function fails, the return value is DDERR_INVALIDPARAMS.

Remarks

On systems with multiple monitors, this method enumerates multiple display devices. For more information, see Multiple Monitor Systems.

For Windows 98, this function is supported in DirectX 5.0 and later; for all other operating systems, DirectX 6.0 is required. Retrieve the **DirectDrawEnumerateEx** function's address from the Ddraw.dll dynamic-link library by calling the **GetProcAddress** Win32 function with the "DirectDrawEnumerateExA" (ANSI) or "DirectDrawEnumerateExW" (Unicode) process name strings. If **GetProcAddress** fails, then the installed version of the operating system does not support multiple monitors. For more information, see Enumerating Devices on MultiMon Systems.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in ddraw.h.
Import Library: Use ddraw.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT.

Callback Functions

[This is preliminary documentation and subject to change.]

This section contains information about the following callback functions used with DirectDraw:

- **DDEnumCallback**
- **DDEnumCallbackEx**
- **EnumModesCallback**
- **EnumModesCallback2**
- **EnumSurfacesCallback**
- **EnumSurfacesCallback2**
- **EnumVideoCallback**

DDEnumCallback

[This is preliminary documentation and subject to change.]

The **DDEnumCallback** function is an application-defined callback function for the **DirectDrawEnumerate** function.

```
BOOL WINAPI DDEnumCallback(
  GUID FAR *lpGUID,
  LPSTR  lpDriverDescription,
  LPSTR  lpDriverName,
  LPVOID lpContext
);
```

Parameters

lpGUID
Address of the unique identifier of the DirectDraw object.

lpDriverDescription
Address of a string containing the driver description.

lpDriverName
Address of a string containing the driver name.

lpContext
Address of an application-defined structure that will be passed to the callback function each time the function is called.

Return Values

The callback function returns non-zero to continue the enumeration.

The callback function returns zero to stop it.

Remarks

You can use the **LPDDENUMCALLBACK** data type to declare a variable that can contain a pointer to this callback function.

If UNICODE is defined, the string values will be returned as type **LPWSTR** rather than **LPSTR**.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: User-defined.

DDEnumCallbackEx

[This is preliminary documentation and subject to change.]

The **DDEnumCallbackEx** function is an application-defined callback function for the **DirectDrawEnumerateEx** function.

```
BOOL WINAPI DDEnumCallbackEx(
  GUID FAR *lpGUID,
  LPSTR    lpDriverDescription,
  LPSTR    lpDriverName,
  LPVOID   lpContext,
  HMONITOR hm
);
```

Parameters

lpGUID
Address of the unique identifier of the DirectDraw object.

lpDriverDescription
Address of a string containing the driver description.

lpDriverName
Address of a string containing the driver name.

lpContext
Address of an application-defined structure that will be passed to the callback function each time the function is called.

hm
Handle to the monitor associated with the enumerated DirectDraw object. This parameter will be NULL when the enumerated DirectDraw object is for the primary device, a non-display device (such as a 3-D accelerator with no 2-D capabilities), and devices not attached to the desktop.

Return Values

The callback function returns non-zero to continue the enumeration.

The callback function returns zero to stop it.

Remarks

You can use the **LPDDENUMCALLBACKEX** data type to declare a variable that can contain a pointer to this callback function.

If UNICODE is defined, the string values will be returned as type **LPWSTR** rather than **LPSTR**.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Header: Declared in ddraw.h.
Import Library: User-defined.

See Also

Enumerating Devices on MultiMon Systems, Multiple Monitor Systems

EnumModesCallback

[This is preliminary documentation and subject to change.]

The **EnumModesCallback** function is an application-defined callback function for the **IDirectDraw3::EnumDisplayModes** method, and its counterparts in earlier interfaces.

This callback function is superseded by the **EnumModesCallback2** function that is used with the **IDirectDraw4::EnumDisplayModes** method.

```
HRESULT WINAPI EnumModesCallback(
  LPDDSURFACEDESC lpDDSurfaceDesc,
  LPVOID lpContext
);
```

Parameters

lpDDSurfaceDesc
Address of a read-only **DDSURFACEDESC** structure that provides the monitor frequency and the mode that can be created.

lpContext
Address of an application-defined structure that will be passed to the callback function each time the function is called.

Return Values

The callback function returns DDENUMRET_OK to continue the enumeration.

The callback function returns DDENUMRET_CANCEL to stop it.

Remarks

You can use the **LPDDENUMMODESCALLBACK** data type to declare a variable that can contain a pointer to this callback function.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: User-defined.

EnumModesCallback2

[This is preliminary documentation and subject to change.]

The **EnumModesCallback2** function is an application-defined callback function for the **IDirectDraw4::EnumDisplayModes** method.

```
HRESULT WINAPI EnumModesCallback(
  LPDDSURFACEDESC2 lpDDSurfaceDesc,
  LPVOID lpContext
);
```

Parameters

lpDDSurfaceDesc
Address of a read-only **DDSURFACEDESC2** structure that provides the monitor frequency and the mode that can be created.

lpContext
Address of an application-defined structure that will be passed to the callback function each time the function is called.

Return Values

The callback function returns DDENUMRET_OK to continue the enumeration.

The callback function returns DDENUMRET_CANCEL to stop it.

Remarks

You can use the **LPDDENUMMODESCALLBACK2** data type to declare a variable that can contain a pointer to this callback function.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: User-defined.

EnumSurfacesCallback

[This is preliminary documentation and subject to change.]

The **EnumSurfacesCallback** function is an application-defined callback function for the **IDirectDraw2::EnumSurfaces**, **IDirectDrawSurface3::EnumAttachedSurfaces**, and **IDirectDrawSurface3::EnumOverlayZOrders** methods (and the versions from earlier interfaces).

```
HRESULT WINAPI EnumSurfacesCallback(
  LPDIRECTDRAWSURFACE lpDDSurface,
  LPDDSURFACEDESC lpDDSurfaceDesc,
  LPVOID lpContext
);
```

Parameters

lpDDSurface
Address of the **IDirectDrawSurface** interface for the attached surface.

lpDDSurfaceDesc
Address of a **DDSURFACEDESC** structure that describes the attached surface.

lpContext
Address of an application-defined structure that will be passed to the callback function each time the function is called.

Return Values

The callback function returns DDENUMRET_OK to continue the enumeration.

The callback function returns DDENUMRET_CANCEL to stop it.

Remarks

You can use the **LPDDENUMSURFACESCALLBACK** data type to declare a variable that can contain a pointer to this callback function.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: User-defined.

EnumSurfacesCallback2

[This is preliminary documentation and subject to change.]

The **EnumSurfacesCallback2** function is an application-defined callback function for the **IDirectDrawSurface4::EnumAttachedSurfaces** and **IDirectDrawSurface4::EnumOverlayZOrders** methods.

```
HRESULT WINAPI EnumSurfacesCallback2(
  LPDIRECTDRAWSURFACE4 lpDDSurface,
  LPDDSURFACEDESC2 lpDDSurfaceDesc,
  LPVOID lpContext
);
```

Parameters

lpDDSurface
Address of the **IDirectDrawSurface4** interface of the attached surface.

lpDDSurfaceDesc
Address of a **DDSURFACEDESC2** structure that describes the attached surface.

lpContext
Address of an application-defined structure that will be passed to the callback function each time the function is called.

Return Values

The callback function returns DDENUMRET_OK to continue the enumeration.

The callback function returns DDENUMRET_CANCEL to stop it.

Remarks

You can use the **LPDDENUMSURFACESCALLBACK2** data type to declare a variable that can contain a pointer to this callback function.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: User-defined.

EnumVideoCallback

[This is preliminary documentation and subject to change.]

The **EnumVideoCallback** function is an application-defined callback procedure for the **IDDVideoPortContainer::EnumVideoPorts** method.

```
HRESULT WINAPI EnumVideoCallback(
  LPDDVIDEOPORTCAPS lpDDVideoPortCaps,
  LPVOID lpContext
);
```

Parameters

lpDDVideoPortCaps
Pointer to the **DDVIDEOPORTCAPS** structure that contains the video port information, including the ID and capabilities. This data is read-only.

lpContext
Pointer to a caller-defined structure that is passed to the member every time it is called.

Return Values

The callback function returns DDENUMRET_OK to continue the enumeration.

The callback function returns DDENUMRET_CANCEL to stop it.

Remarks

Video-port related functions cannot be called from inside the **EnumVideoCallback** function. Attempts to do so will fail, returning DDERR_CURRENTLYNOTAVAIL.

You can use the **LPDDENUMVIDEOCALLBACK** data type to declare a variable that can contain a pointer to this callback function.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.
Import Library: User-defined.

Structures

[This is preliminary documentation and subject to change.]

This section contains information about the following structures used with DirectDraw:

- **DDBLTBATCH**
- **DDBLTFX**

- **DDCAPS**
- **DDCOLORCONTROL**
- **DDCOLORKEY**
- **DDDEVICEIDENTIFIER**
- **DDGAMMARAMP**
- **DDOVERLAYFX**
- **DDPIXELFORMAT**
- **DDSCAPS**
- **DDSCAPS2**
- **DDSURFACEDESC**
- **DDSURFACEDESC2**
- **DDVIDEOPORTBANDWIDTH**
- **DDVIDEOPORTCAPS**
- **DDVIDEOPORTCONNECT**
- **DDVIDEOPORTDESC**
- **DDVIDEOPORTINFO**
- **DDVIDEOPORTSTATUS**

Note

The memory for all DirectX structures should be initialized to zero before use. In addition, all structures that contain a **dwSize** member should set the member to the size of the structure, in bytes, before use. The following example performs these tasks on a common structure, **DDCAPS**:

```
DDCAPS ddcaps; // Can't use this yet.

ZeroMemory(&ddcaps, sizeof(DDCAPS));
ddcaps.dwSize = sizeof(DDCAPS);

// Now the structure can be used.
.
.
```

DDBLTBATCH

[This is preliminary documentation and subject to change.]

The **DDBLTBATCH** structure passes blit operations to the **IDirectDrawSurface4::BltBatch** method.

```
typedef struct _DDBLTBATCH{
   LPRECT           lprDest;
```

```
    LPDIRECTDRAWSURFACE lpDDSSrc;
    LPRECT              lprSrc;
    DWORD               dwFlags;
    LPDDBLTFX           lpDDBltFx;
} DDBLTBATCH,FAR *LPDDBLTBATCH;
```

Members

lprDest
Address of a **RECT** structure that defines the destination for the blit.

lpDDSSrc
Address of a DirectDrawSurface object that will be the source of the blit.

lprSrc
Address of a **RECT** structure that defines the source rectangle of the blit.

dwFlags
Optional control flags.

DDBLT_ALPHADEST
Uses either the alpha information in pixel format or the alpha channel surface attached to the destination surface as the alpha channel for this blit.

DDBLT_ALPHADESTCONSTOVERRIDE
Uses the **dwAlphaDestConst** member of the **DDBLTFX** structure as the alpha channel for the destination surface for this blit.

DDBLT_ALPHADESTNEG
Indicates that the destination surface becomes more transparent as the alpha value increases (0 is opaque).

DDBLT_ALPHADESTSURFACEOVERRIDE
Uses the **lpDDSAlphaDest** member of the **DDBLTFX** structure as the alpha channel for the destination surface for this blit.

DDBLT_ALPHAEDGEBLEND
Uses the **dwAlphaEdgeBlend** member of the **DDBLTFX** structure as the alpha channel for the edges of the image that border the color key colors.

DDBLT_ALPHASRC
Uses either the alpha information in pixel format or the alpha channel surface attached to the source surface as the alpha channel for this blit.

DDBLT_ALPHASRCCONSTOVERRIDE
Uses the **dwAlphaSrcConst** member of the **DDBLTFX** structure as the source alpha channel for this blit.

DDBLT_ALPHASRCNEG
Indicates that the source surface becomes more transparent as the alpha value increases (0 is opaque).

DDBLT_ALPHASRCSURFACEOVERRIDE
Uses the **lpDDSAlphaSrc** member of the **DDBLTFX** structure as the alpha channel source for this blit.

DDBLT_ASYNC
Processes this blit asynchronously through the FIFO hardware in the order received. If there is no room in the FIFO hardware, the call fails.

DDBLT_COLORFILL
Uses the **dwFillColor** member of the **DDBLTFX** structure as the RGB color that fills the destination rectangle on the destination surface.

DDBLT_DDFX
Uses the **dwDDFX** member of the **DDBLTFX** structure to specify the effects to be used for this blit.

DDBLT_DDROPS
Uses the **dwDDROP** member of the **DDBLTFX** structure to specify the raster operations (ROPs) that are not part of the Win32 API.

DDBLT_KEYDEST
Uses the color key associated with the destination surface.

DDBLT_KEYDESTOVERRIDE
Uses the **ddckDestColorkey** member of the **DDBLTFX** structure as the color key for the destination surface.

DDBLT_KEYSRC
Uses the color key associated with the source surface.

DDBLT_KEYSRCOVERRIDE
Uses the **ddckSrcColorkey** member of the **DDBLTFX** structure as the color key for the source surface.

DDBLT_ROP
Uses the **dwROP** member of the **DDBLTFX** structure for the ROP for this blit. The ROPs are the same as those defined in the Win32 API.

DDBLT_ROTATIONANGLE
Uses the **dwRotationAngle** member of the **DDBLTFX** structure as the rotation angle (specified in 1/100th of a degree) for the surface.

DDBLT_ZBUFFER
Performs a z-buffered blit using the z-buffers attached to the source and destination surfaces and the **dwZBufferOpCode** member of the **DDBLTFX** structure as the z-buffer opcode.

DDBLT_ZBUFFERDESTCONSTOVERRIDE
Performs a z-buffered blit using the **dwZDestConst** and **dwZBufferOpCode** members of the **DDBLTFX** structure as the z-buffer and z-buffer opcode, respectively, for the destination.

DDBLT_ZBUFFERDESTOVERRIDE
Performs a z-buffered blit using the **lpDDSZBufferDest** and **dwZBufferOpCode** members of the **DDBLTFX** structure as the z-buffer and z-buffer opcode, respectively, for the destination.

DDBLT_ZBUFFERSRCCONSTOVERRIDE
Performs a z-buffered blit using the **dwZSrcConst** and **dwZBufferOpCode** members of the **DDBLTFX** structure as the z-buffer and z-buffer opcode, respectively, for the source.

DDBLT_ZBUFFERSRCOVERRIDE
A z-buffered blit using the **lpDDSZBufferSrc** and **dwZBufferOpCode** members of the **DDBLTFX** structure as the z-buffer and z-buffer opcode, respectively, for the source.

lpDDBltFx
Address of a **DDBLTFX** structure specifying additional blit effects.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.

DDBLTFX

[This is preliminary documentation and subject to change.]

The **DDBLTFX** structure passes raster operations, effects, and override information to the **IDirectDrawSurface4::Blt** method. This structure is also part of the **DDBLTBATCH** structure used with the **IDirectDrawSurface4::BltBatch** method.

```
typedef struct _DDBLTFX{
   DWORD dwSize;
   DWORD dwDDFX;
   DWORD dwROP;
   DWORD dwDDROP;
   DWORD dwRotationAngle;
   DWORD dwZBufferOpCode;
   DWORD dwZBufferLow;
   DWORD dwZBufferHigh;
   DWORD dwZBufferBaseDest;
   DWORD dwZDestConstBitDepth;
   union
   {
      DWORD            dwZDestConst;
      LPDIRECTDRAWSURFACE lpDDSZBufferDest;
   } DUMMYUNIONNAMEN(1);
   DWORD dwZSrcConstBitDepth;
   union
   {
      DWORD            dwZSrcConst;
      LPDIRECTDRAWSURFACE lpDDSZBufferSrc;
   } DUMMYUNIONNAMEN(2);
   DWORD dwAlphaEdgeBlendBitDepth;
   DWORD dwAlphaEdgeBlend;
```

```
    DWORD dwReserved;
    DWORD dwAlphaDestConstBitDepth;
    union
    {
      DWORD            dwAlphaDestConst;
      LPDIRECTDRAWSURFACE lpDDSAlphaDest;
    } DUMMYUNIONNAMEN(3);
    DWORD dwAlphaSrcConstBitDepth;
    union
    {
      DWORD            dwAlphaSrcConst;
      LPDIRECTDRAWSURFACE lpDDSAlphaSrc;
    } DUMMYUNIONNAMEN(4);
    union
    {
      DWORD            dwFillColor;
      DWORD            dwFillDepth;
      DWORD            dwFillPixel;
      LPDIRECTDRAWSURFACE lpDDSPattern;
    } DUMMYUNIONNAMEN(5);
    DDCOLORKEY ddckDestColorkey;
    DDCOLORKEY ddckSrcColorkey;
  } DDBLTFX,FAR* LPDDBLTFX;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized before the structure is used.

dwDDFX

Type of FX operations.

DDBLTFX_ARITHSTRETCHY
: Uses arithmetic stretching along the y-axis for this blit.

DDBLTFX_MIRRORLEFTRIGHT
: Turns the surface on its y-axis. This blit mirrors the surface from left to right.

DDBLTFX_MIRRORUPDOWN
: Turns the surface on its x-axis. This blit mirrors the surface from top to bottom.

DDBLTFX_NOTEARING
: Schedules this blit to avoid tearing.

DDBLTFX_ROTATE180
: Rotates the surface 180 degrees clockwise during this blit.

DDBLTFX_ROTATE270
: Rotates the surface 270 degrees clockwise during this blit.

DDBLTFX_ROTATE90
Rotates the surface 90 degrees clockwise during this blit.

DDBLTFX_ZBUFFERBASEDEST
Adds the **dwZBufferBaseDest** member to each of the source z-values before comparing them with the destination z-values during this z-blit.

DDBLTFX_ZBUFFERRANGE
Uses the **dwZBufferLow** and **dwZBufferHigh** members as range values to specify limits to the bits copied from a source surface during this z-blit.

dwROP
Win32 raster operations. You can retrieve a list of supported raster operations by calling the **IDirectDraw4::GetCaps** method.

dwDDROP
DirectDraw raster operations.

dwRotationAngle
Rotation angle for the blit.

dwZBufferOpCode
Z-buffer compares.

dwZBufferLow
Low limit of a z-buffer.

dwZBufferHigh
High limit of a z-buffer.

dwZBufferBaseDest
Destination base value of a z-buffer.

dwZDestConstBitDepth
Bit depth of the destination z-constant.

dwZDestConst
Constant used as the z-buffer destination.

lpDDSZBufferDest
Surface used as the z-buffer destination.

dwZSrcConstBitDepth
Bit depth of the source z-constant.

dwZSrcConst
Constant used as the z-buffer source.

lpDDSZBufferSrc
Surface used as the z-buffer source.

dwAlphaEdgeBlendBitDepth
Bit depth of the constant for an alpha edge blend.

dwAlphaEdgeBlend
Alpha constant used for edge blending.

dwReserved
Reserved for future use.

dwAlphaDestConstBitDepth

Bit depth of the destination alpha constant.

dwAlphaDestConst
Constant used as the alpha channel destination.

lpDDSAlphaDest
Surface used as the alpha channel destination.

dwAlphaSrcConstBitDepth
Bit depth of the source alpha constant.

dwAlphaSrcConst
Constant used as the alpha channel source.

lpDDSAlphaSrc
Surface used as the alpha channel source.

dwFillColor
Color used to fill a surface when DDBLT_COLORFILL is specified. This value must be a pixel appropriate to the pixel format of the destination surface. For a palettized surface it would be a palette index, and for a 16-bit RGB surface it would be a 16-bit pixel value.

dwFillDepth
Depth value for the z-buffer.

dwFillPixel
Pixel value for RGBA or RGBZ fills. Applications that use RGBZ fills should use this member, not **dwFillColor** or **dwFillDepth**.

lpDDSPattern
Surface to use as a pattern. The pattern can be used in certain blit operations that combine a source and a destination.

ddckDestColorkey
Destination color key override.

ddckSrcColorkey
Source color key override.

Remarks

The unions in this structure have been updated to work with compilers that don't support nameless unions. If your compiler doesn't support nameless unions, define the NONAMELESSUNION token before including the Ddraw.h header file.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.

DDCAPS

[This is preliminary documentation and subject to change.]

The **DDCAPS** structure represents the capabilities of the hardware exposed through the DirectDraw object. This structure contains a **DDSCAPS** structure used in this context to describe what kinds of DirectDrawSurface objects can be created. It may not be possible to simultaneously create all of the surfaces described by these capabilities. This structure is used with the **IDirectDraw4::GetCaps** method.

The ddraw.h header file contains multiple versions of this structure. See Remarks for more information.

```
typedef struct _DDCAPS {
    DWORD   dwSize;
    DWORD   dwCaps;               // driver-specific caps
    DWORD   dwCaps2;              // more driver-specific caps
    DWORD   dwCKeyCaps;           // color key caps
    DWORD   dwFXCaps;             // stretching and effects caps
    DWORD   dwFXAlphaCaps;        // alpha caps
    DWORD   dwPalCaps;            // palette caps
    DWORD   dwSVCaps;             // stereo vision caps
    DWORD   dwAlphaBltConstBitDepths;      // alpha bit-depth members
    DWORD   dwAlphaBltPixelBitDepths;      //  .
    DWORD   dwAlphaBltSurfaceBitDepths;    //  .
    DWORD   dwAlphaOverlayConstBitDepths;  //  .
    DWORD   dwAlphaOverlayPixelBitDepths;  //  .
    DWORD   dwAlphaOverlaySurfaceBitDepths; //  .
    DWORD   dwZBufferBitDepths;     // Z-buffer bit depth
    DWORD   dwVidMemTotal;          // total video memory
    DWORD   dwVidMemFree;           // total free video memory
    DWORD   dwMaxVisibleOverlays;   // maximum visible overlays
    DWORD   dwCurrVisibleOverlays;  // overlays currently visible
    DWORD   dwNumFourCCCodes;       // number of supported FOURCC codes
    DWORD   dwAlignBoundarySrc;     // overlay alignment restrictions
    DWORD   dwAlignSizeSrc;         //  .
    DWORD   dwAlignBoundaryDest;    //  .
    DWORD   dwAlignSizeDest;        //  .
    DWORD   dwAlignStrideAlign;     // stride alignment
    DWORD   dwRops[DD_ROP_SPACE];   // supported raster ops
    DWORD   dwReservedCaps;         // reserved
    DWORD   dwMinOverlayStretch;    // overlay stretch factors
    DWORD   dwMaxOverlayStretch;    //  .
    DWORD   dwMinLiveVideoStretch;  // obsolete
    DWORD   dwMaxLiveVideoStretch;  //  .
    DWORD   dwMinHwCodecStretch;    //  .
    DWORD   dwMaxHwCodecStretch;    //  .
```

```
    DWORD   dwReserved1;          // reserved
    DWORD   dwReserved2;          //  .
    DWORD   dwReserved3;          //  .
    DWORD   dwSVBCaps;            // system-to-video blit related caps
    DWORD   dwSVBCKeyCaps;        //  .
    DWORD   dwSVBFXCaps;          //  .
    DWORD   dwSVBRops[DD_ROP_SPACE]; //  .
    DWORD   dwVSBCaps;            // video-to-system blit related caps
    DWORD   dwVSBCKeyCaps;        //  .
    DWORD   dwVSBFXCaps;          //  .
    DWORD   dwVSBRops[DD_ROP_SPACE]; //  .
    DWORD   dwSSBCaps;            // system-to-system blit related caps
    DWORD   dwSSBCKeyCaps;        //  .
    DWORD   dwSSBCFXCaps;         //  .
    DWORD   dwSSBRops[DD_ROP_SPACE]; //  .
    DWORD   dwMaxVideoPorts;      // maximum number of live video ports
    DWORD   dwCurrVideoPorts;     // current number of live video ports
    DWORD   dwSVBCaps2;           // additional system-to-video blit caps
    DWORD   dwNLVBCaps;           // nonlocal-to-local video memory blit caps
    DWORD   dwNLVBCaps2;          //  .
    DWORD   dwNLVBCKeyCaps;       //  .
    DWORD   dwNLVBFXCaps;         //  .
    DWORD   dwNLVBRops[DD_ROP_SPACE];//  .
    DDSCAPS2 ddsCaps;             // general surface caps
  } DDCAPS,FAR* LPDDCAPS;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized before the structure is used.

dwCaps

Driver-specific capabilities.

DDCAPS_3D

Indicates that the display hardware has 3-D acceleration.

DDCAPS_ALIGNBOUNDARYDEST

Indicates that DirectDraw will support only those overlay destination rectangles with the x-axis aligned to the **dwAlignBoundaryDest** boundaries of the surface.

DDCAPS_ALIGNBOUNDARYSRC

Indicates that DirectDraw will support only those overlay source rectangles with the x-axis aligned to the **dwAlignBoundarySrc** boundaries of the surface.

DDCAPS_ALIGNSIZEDEST

Indicates that DirectDraw will support only those overlay destination rectangles whose x-axis sizes, in pixels, are **dwAlignSizeDest** multiples.

DDCAPS_ALIGNSIZESRC
Indicates that DirectDraw will support only those overlay source rectangles whose x-axis sizes, in pixels, are **dwAlignSizeSrc** multiples.

DDCAPS_ALIGNSTRIDE
Indicates that DirectDraw will create display memory surfaces that have a stride alignment equal to the **dwAlignStrideAlign** value.

DDCAPS_ALPHA
Indicates that the display hardware supports alpha-only surfaces. (See alpha channel)

DDCAPS_BANKSWITCHED
Indicates that the display hardware is bank-switched and is potentially very slow at random access to display memory.

DDCAPS_BLT
Indicates that display hardware is capable of blit operations.

DDCAPS_BLTCOLORFILL
Indicates that display hardware is capable of color filling with a blitter.

DDCAPS_BLTDEPTHFILL
Indicates that display hardware is capable of depth filling z-buffers with a blitter.

DDCAPS_BLTFOURCC
Indicates that display hardware is capable of color-space conversions during blit operations.

DDCAPS_BLTQUEUE
Indicates that display hardware is capable of asynchronous blit operations.

DDCAPS_BLTSTRETCH
Indicates that display hardware is capable of stretching during blit operations.

DDCAPS_CANBLTSYSMEM
Indicates that display hardware is capable of blitting to or from system memory.

DDCAPS_CANCLIP
Indicates that display hardware is capable of clipping with blitting.

DDCAPS_CANCLIPSTRETCHED
Indicates that display hardware is capable of clipping while stretch blitting.

DDCAPS_COLORKEY
Supports some form of color key in either overlay or blit operations. More specific color key capability information can be found in the **dwCKeyCaps** member.

DDCAPS_COLORKEYHWASSIST
Indicates that the color key is partially hardware assisted. This means that other resources (CPU or video memory) might be used. If this bit is not set, full hardware support is in place.

DDCAPS_GDI
Indicates that display hardware is shared with GDI.

DDCAPS_NOHARDWARE
Indicates that there is no hardware support.

DDCAPS_OVERLAY
Indicates that display hardware supports overlays.

DDCAPS_OVERLAYCANTCLIP
Indicates that display hardware supports overlays but cannot clip them.

DDCAPS_OVERLAYFOURCC
Indicates that overlay hardware is capable of color-space conversions during overlay operations.

DDCAPS_OVERLAYSTRETCH
Indicates that overlay hardware is capable of stretching. The **dwMinOverlayStretch** and **dwMaxOverlayStretch** members contain valid data.

DDCAPS_PALETTE
Indicates that DirectDraw is capable of creating and supporting DirectDrawPalette objects for more surfaces than only the primary surface.

DDCAPS_PALETTEVSYNC
Indicates that DirectDraw is capable of updating a palette synchronized with the vertical refresh.

DDCAPS_READSCANLINE
Indicates that display hardware is capable of returning the current scan line.

DDCAPS_STEREOVIEW
Indicates that display hardware has stereo vision capabilities.

DDCAPS_VBI
Indicates that display hardware is capable of generating a vertical-blank interrupt.

DDCAPS_ZBLTS
Supports the use of z-buffers with blit operations.

DDCAPS_ZOVERLAYS
Supports the use of the **IDirectDrawSurface4::UpdateOverlayZOrder** method as a z-value for overlays to control their layering.

dwCaps2
More driver-specific capabilities.

DDCAPS2_AUTOFLIPOVERLAY
The overlay can be automatically flipped to the next surface in the flip chain each time a video port VSYNC occurs, allowing the video port and the overlay to double buffer the video without CPU overhead. This option is only valid when the surface is receiving data from a video port. If the video port data is non-interlaced or non-interleaved, it will flip on every VSYNC. If the data is being interleaved in memory, it will flip on every other VSYNC.

DDCAPS2_CANBOBHARDWARE

The overlay hardware can display each field of an interlaced video stream individually.

DDCAPS2_CANBOBINTERLEAVED
The overlay hardware can display each field individually of an interlaced video stream while it is interleaved in memory without causing any artifacts that might normally occur without special hardware support. This option is only valid when the surface is receiving data from a video port and is only valid when the video is zoomed at least two times in the vertical direction.

DDCAPS2_CANBOBNONINTERLEAVED
The overlay hardware can display each field individually of an interlaced video stream while it is not interleaved in memory without causing any artifacts that might normally occur without special hardware support. This option is only valid when the surface is receiving data from a video port and is only valid when the video is zoomed at least two times in the vertical direction.

DDCAPS2_CANCALIBRATEGAMMA
The system has a calibrator installed that can automatically adjust the gamma ramp so that the result will be identical on all systems that have a calibrator. To invoke the calibrator when setting new gamma levels, use the DDSGR_CALIBRATE flag when calling the **IDirectDrawGammaControl::SetGammaRamp** method. Calibrating gamma ramps incurs some processing overhead, and should not be used frequently.

DDCAPS2_CANDROPZ16BIT
16-bit RGBZ values can be converted into sixteen-bit RGB values. (The system does not support eight-bit conversions.)

DDCAPS2_CANFLIPODDEVEN
The driver is capable of performing odd and even flip operations, as specified by the DDFLIP_ODD and DDFLIP_EVEN flags used with the **IDirectDrawSurface4::Flip** method.

DDCAPS2_CANRENDERWINDOWED
The driver is capable of rendering in windowed mode.

DDCAPS2_CERTIFIED
Indicates that display hardware is certified.

DDCAPS2_COLORCONTROLPRIMARY
The primary surface contains color controls (for instance, gamma)

DDCAPS2_COLORCONTROLOVERLAY
The overlay surface contains color controls (such as brightness, sharpness)

DDCAPS2_COPYFOURCC
Indicates that the driver supports blitting any FOURCC surface to another surface of the same FOURCC.

DDCAPS2_FLIPINTERVAL
Indicates that the driver will respond to the DDFLIP_INTERVAL* flags. (see **IDirectDrawSurface4::Flip**).

DDCAPS2_FLIPNOVSYNC
Indicates that the driver will respond to the DDFLIP_NOVSYNC flag (see **IDirectDrawSurface4::Flip**).

DDCAPS2_NO2DDURING3DSCENE
Indicates that 2-D operations such as **IDirectDrawSurface4::Blt** and **IDirectDrawSurface4::Lock** cannot be performed on any surfaces that Direct3D® is using between calls to the **IDirect3DDevice3::BeginScene** and **IDirect3DDevice3::EndScene** methods.

DDCAPS2_NONLOCALVIDMEM
Indicates that the display driver supports surfaces in non-local video memory.

DDCAPS2_NONLOCALVIDMEMCAPS
Indicates that blit capabilities for non-local video memory surfaces differ from local video memory surfaces. If this flag is present, the DDCAPS2_NONLOCALVIDMEM flag will also be present.

DDCAPS2_NOPAGELOCKREQUIRED
DMA blit operations are supported on system memory surfaces that are not page locked.

DDCAPS2_PRIMARYGAMMA
Supports dynamic gamma ramps for the primary surface. For more information, see Gamma and Color Controls.

DDCAPS2_VIDEOPORT
Indicates that display hardware supports live video.

DDCAPS2_WIDESURFACES
Indicates that the display surfaces supports surfaces wider than the primary surface.

dwCKeyCaps
Color-key capabilities.

DDCKEYCAPS_DESTBLT
Supports transparent blitting with a color key that identifies the replaceable bits of the destination surface for RGB colors.

DDCKEYCAPS_DESTBLTCLRSPACE
Supports transparent blitting with a color space that identifies the replaceable bits of the destination surface for RGB colors.

DDCKEYCAPS_DESTBLTCLRSPACEYUV
Supports transparent blitting with a color space that identifies the replaceable bits of the destination surface for YUV colors.

DDCKEYCAPS_DESTBLTYUV
Supports transparent blitting with a color key that identifies the replaceable bits of the destination surface for YUV colors.

DDCKEYCAPS_DESTOVERLAY
Supports overlaying with color keying of the replaceable bits of the destination surface being overlaid for RGB colors.

DDCKEYCAPS_DESTOVERLAYCLRSPACE
Supports a color space as the color key for the destination of RGB colors.

DDCKEYCAPS_DESTOVERLAYCLRSPACEYUV
Supports a color space as the color key for the destination of YUV colors.

DDCKEYCAPS_DESTOVERLAYONEACTIVE
Supports only one active destination color key value for visible overlay surfaces .

DDCKEYCAPS_DESTOVERLAYYUV
Supports overlaying using color keying of the replaceable bits of the destination surface being overlaid for YUV colors.

DDCKEYCAPS_NOCOSTOVERLAY
Indicates there are no bandwidth trade-offs for using the color key with an overlay.

DDCKEYCAPS_SRCBLT
Supports transparent blitting using the color key for the source with this surface for RGB colors.

DDCKEYCAPS_SRCBLTCLRSPACE
Supports transparent blitting using a color space for the source with this surface for RGB colors.

DDCKEYCAPS_SRCBLTCLRSPACEYUV
Supports transparent blitting using a color space for the source with this surface for YUV colors.

DDCKEYCAPS_SRCBLTYUV
Supports transparent blitting using the color key for the source with this surface for YUV colors.

DDCKEYCAPS_SRCOVERLAY
Supports overlaying using the color key for the source with this overlay surface for RGB colors.

DDCKEYCAPS_SRCOVERLAYCLRSPACE
Supports overlaying using a color space as the source color key for the overlay surface for RGB colors.

DDCKEYCAPS_SRCOVERLAYCLRSPACEYUV
Supports overlaying using a color space as the source color key for the overlay surface for YUV colors.

DDCKEYCAPS_SRCOVERLAYONEACTIVE
Supports only one active source color key value for visible overlay surfaces.

DDCKEYCAPS_SRCOVERLAYYUV
Supports overlaying using the color key for the source with this overlay surface for YUV colors.

dwFXCaps
Driver-specific stretching and effects capabilities.

DDFXCAPS_BLTALPHA
Supports alpha-blended blit operations.

DDFXCAPS_BLTARITHSTRETCHY

Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink surfaces during a blit operation. Occurs along the y-axis (vertically).

DDFXCAPS_BLTARITHSTRETCHYN
Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink surfaces during a blit operation. Occurs along the y-axis (vertically), and works only for integer stretching (×1, ×2, and so on).

DDFXCAPS_BLTFILTER
Driver can do surface-reconstruction filtering for warped blits.

DDFXCAPS_BLTMIRRORLEFTRIGHT
Supports mirroring left to right in a blit operation.

DDFXCAPS_BLTMIRRORUPDOWN
Supports mirroring top to bottom in a blit operation.

DDFXCAPS_BLTROTATION
Supports arbitrary rotation in a blit operation.

DDFXCAPS_BLTROTATION90
Supports 90-degree rotations in a blit operation.

DDFXCAPS_BLTSHRINKX
Supports arbitrary shrinking of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSHRINKXN
Supports integer shrinking (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSHRINKY
Supports arbitrary shrinking of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_BLTSHRINKYN
Supports integer shrinking (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHX
Supports arbitrary stretching of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHXN
Supports integer stretching (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHY
Supports arbitrary stretching of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHYN
Supports integer stretching (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_BLTTRANSFORM
Supports geometric transformations (or warps) for blitted sprites. Transformations are not currently supported for explicit blit operations.

DDFXCAPS_OVERLAYALPHA
Supports alpha blending for overlay surfaces.

DDFXCAPS_OVERLAYARITHSTRETCHY
Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink overlay surfaces. Occurs along the y-axis (vertically).

DDFXCAPS_OVERLAYARITHSTRETCHYN
Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink overlay surfaces. Occurs along the y-axis (vertically), and works only for integer stretching (×1, ×2, and so on).

DDFXCAPS_OVERLAYFILTER
Supports surface-reconstruction filtering for warped overlay sprites. Filtering is not currently supported for explicitly displayed overlay surfaces (those displayed with calls to **IDirectDrawSurface4::UpdateOverlay**).

DDFXCAPS_OVERLAYMIRRORLEFTRIGHT
Supports mirroring of overlays across the vertical axis.

DDFXCAPS_OVERLAYMIRRORUPDOWN
Supports mirroring of overlays across the horizontal axis.

DDFXCAPS_OVERLAYSHRINKX
Supports arbitrary shrinking of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKXN
Supports integer shrinking (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKY
Supports arbitrary shrinking of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKYN
Supports integer shrinking (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSTRETCHX
Supports arbitrary stretching of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHXN
Supports integer stretching (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces.

This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHY
Supports arbitrary stretching of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHYN
Supports integer stretching (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYTRANSFORM
Supports geometric transformations (or warps) for overlay sprites. Transformations are not currently supported for explicitly displayed overlay surfaces (those displayed with calls to **IDirectDrawSurface4::UpdateOverlay**).

dwFXAlphaCaps
Driver-specific alpha capabilities.

DDFXALPHACAPS_BLTALPHAEDGEBLEND
Supports alpha blending around the edge of a source color-keyed surface. Used for blit operations.

DDFXALPHACAPS_BLTALPHAPIXELS
Supports alpha information in pixel format. The bit depth of alpha information in the pixel format can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for blit operations.

DDFXALPHACAPS_BLTALPHAPIXELSNEG
Supports alpha information in pixel format. The bit depth of alpha information in the pixel format can be 1, 2, 4, or 8. The alpha value becomes more transparent as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be used only if DDCAPS_ALPHA is set. Used for blit operations.

DDFXALPHACAPS_BLTALPHASURFACES
Supports alpha-only surfaces. The bit depth of an alpha-only surface can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for blit operations.

DDFXALPHACAPS_BLTALPHASURFACESNEG
Indicates that the alpha channel becomes more transparent as the alpha value increases. The depth of the alpha channel data can be 1, 2, 4, or 8. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be set only if DDCAPS_ALPHA has been set. Used for blit operations.

DDFXALPHACAPS_OVERLAYALPHAEDGEBLEND
Supports alpha blending around the edge of a source color-keyed surface. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHAPIXELS
Supports alpha information in pixel format. The bit depth of alpha information in pixel format can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHAPIXELSNEG
Supports alpha information in pixel format. The bit depth of alpha information in pixel format can be 1, 2, 4, or 8. The alpha value becomes more transparent as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be used only if DDCAPS_ALPHA has been set. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHASURFACES
Supports alpha-only surfaces. The bit depth of an alpha-only surface can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHASURFACESNEG
Indicates that the alpha channel becomes more transparent as the alpha value increases. The depth of the alpha channel data can be 1, 2, 4, or 8. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be used only if DDCAPS_ALPHA has been set. Used for overlays.

dwPalCaps
Palette capabilities.

DDPCAPS_1BIT
Supports palettes that contain 1 bit color entries (two colors).

DDPCAPS_2BIT
Supports palettes that contain 2 bit color entries (four colors).

DDPCAPS_4BIT
Supports palettes that contain 4 bit color entries (16 colors).

DDPCAPS_8BIT
Supports palettes that contain 8 bit color entries (256 colors).

DDPCAPS_8BITENTRIES
Specifies an index to an 8-bit color index. This field is valid only when used with the DDPCAPS_1BIT, DDPCAPS_2BIT, or DDPCAPS_4BIT capability and when the target surface is in 8 bits per pixel (bpp). Each color entry is 1 byte long and is an index to an 8-bpp palette on the destination surface.

DDPCAPS_ALPHA
Supports palettes that include an alpha component. For alpha-capable palettes, the **peFlags** member of for each **PALETTEENTRY** structure the palette contains is to be interpreted as a single 8-bit alpha value (in addition to the color data in the **peRed**, **peGreen**, and **peBlue** members). A palette created with this flag can only be attached to a texture surface.

DDPCAPS_ALLOW256
Supports palettes that can have all 256 entries defined.

DDPCAPS_PRIMARYSURFACE
Indicates that the palette is attached to the primary surface. Changing the palette has an immediate effect on the display unless the DDPCAPS_VSYNC capability is specified and supported.

DDPCAPS_PRIMARYSURFACELEFT
Indicates that the palette is attached to the primary surface on the left. Changing the palette has an immediate effect on the display unless the DDPCAPS_VSYNC capability is specified and supported.

DDPCAPS_VSYNC
Indicates that the palette can be modified synchronously with the monitor's refresh rate.

dwSVCaps
Stereo vision capabilities.

DDSVCAPS_ENIGMA
Indicates that the stereo view is accomplished using Enigma encoding.

DDSVCAPS_FLICKER
Indicates that the stereo view is accomplished using high-frequency flickering.

DDSVCAPS_REDBLUE
Indicates that the stereo view is accomplished when the viewer looks at the image through red and blue filters placed over the left and right eyes. All images must adapt their color spaces for this process.

DDSVCAPS_SPLIT
Indicates that the stereo view is accomplished with split-screen technology.

dwAlphaBltConstBitDepths
DDBD_2, DDBD_4, or DDBD_8. (Indicates 2-, 4-, or 8-bits per pixel.)

dwAlphaBltPixelBitDepths
DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

dwAlphaBltSurfaceBitDepths
DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

dwAlphaOverlayConstBitDepths
DDBD_2, DDBD_4, or DDBD_8. (Indicates 2-, 4-, or 8-bits per pixel.)

dwAlphaOverlayPixelBitDepths
DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

dwAlphaOverlaySurfaceBitDepths
DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

dwZBufferBitDepths

DDBD_8, DDBD_16, DDBD_24, or DDBD_32. (Indicates 8-, 16-, 24-, 32-bits per pixel.) This member is obsolete for DirectX 6.0 and later. Use the **IDirect3D3::EnumZBufferFormats** to retrieve information about supported depth buffer formats.

dwVidMemTotal
Total amount of display memory on the device, in bytes, less memory reserved for the primary surface and any private data structures reserved by the driver. (This value is the same as the total video memory reported by the **IDirectDraw4::GetAvailableVidMem** method.)

dwVidMemFree
Free display memory. This value equals the value in **dwVidMemTotal**, less any memory currently allocated by the application for surfaces. Unlike the **GetAvailableVidMem** method, which reports the memory available for a particular type of surface (like a texture), this value reflects the memory available for any type of surface.

dwMaxVisibleOverlays
Maximum number of visible overlays or overlay sprites.

dwCurrVisibleOverlays
Current number of visible overlays or overlay sprites.

dwNumFourCCCodes
Number of FourCC codes.

dwAlignBoundarySrc
Source rectangle alignment for an overlay surface, in pixels.

dwAlignSizeSrc
Source rectangle size alignment for an overlay surface, in pixels. Overlay source rectangles must have a pixel width that is a multiple of this value.

dwAlignBoundaryDest
Destination rectangle alignment for an overlay surface, in pixels.

dwAlignSizeDest
Destination rectangle size alignment for an overlay surface, in pixels. Overlay destination rectangles must have a pixel width that is a multiple of this value.

dwAlignStrideAlign
Stride alignment.

dwRops[DD_ROP_SPACE]
Raster operations supported.

dwReservedCaps
Reserved. Prior to DirectX 6.0, this member contained general surface capabilities, which are now contained in the **ddsCaps** member (a **DDSCAPS2** structure).

dwMinOverlayStretch and **dwMaxOverlayStretch**
Minimum and maximum overlay stretch factors multiplied by 1000. For example, 1.3 = 1300.

dwMinLiveVideoStretch and **dwMaxLiveVideoStretch**
These members are obsolete; do not use.

dwMinHwCodecStretch and **dwMaxHwCodecStretch**
These members are obsolete; do not use.

dwReserved1, **dwReserved2**, and **dwReserved3**
Reserved for future use.

dwSVBCaps
Driver-specific capabilities for system-memory-to-display-memory blits. Valid flags are identical to the blit-related flags used with the **dwCaps** member.

dwSVBCKeyCaps
Driver color-key capabilities for system-memory-to-display-memory blits. Valid flags are identical to the blit-related flags used with for the **dwCKeyCaps** member.

dwSVBFXCaps
Driver FX capabilities for system-memory-to-display-memory blits. Valid flags are identical to the blit-related flags used with the **dwFXCaps** member.

dwSVBRops[DD_ROP_SPACE]
Raster operations supported for system-memory-to-display-memory blits.

dwVSBCaps
Driver-specific capabilities for display-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with the **dwCaps** member.

dwVSBCKeyCaps
Driver color-key capabilities for display-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with for the **dwCKeyCaps** member.

dwVSBFXCaps
Driver FX capabilities for display-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with the **dwFXCaps** member.

dwVSBRops[DD_ROP_SPACE]
Raster operations supported for display-memory-to-system-memory blits.

dwSSBCaps
Driver-specific capabilities for system-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with the **dwCaps** member.

dwSSBCKeyCaps
Driver color-key capabilities for system-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with for the **dwCKeyCaps** member.

dwSSBCFXCaps
Driver FX capabilities for system-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with the **dwFXCaps** member.

dwSSBRops[DD_ROP_SPACE]
Raster operations supported for system-memory-to-system-memory blits.

dwMaxVideoPorts
Maximum number of live video ports.

dwCurrVideoPorts
Current number of live video ports.

dwSVBCaps2
More driver-specific capabilities for system-memory-to-video-memory blits. Valid flags are identical to the blit-related flags used with the **dwCaps2** member.

dwNLVBCaps
Driver-specific capabilities for nonlocal-to-local video memory blits. Valid flags are identical to the blit-related flags used with the **dwCaps** member.

dwNLVBCaps2
More driver-specific capabilities for nonlocal-to-local video memory blits. Valid flags are identical to the blit-related flags used with the **dwCaps2** member.

dwNLVBCKeyCaps
Driver color-key capabilities for nonlocal-to-local video memory blits. Valid flags are identical to the blit-related flags used with for the **dwCKeyCaps** member.

dwNLVBFXCaps
Driver FX capabilities for nonlocal-to-local video memory blits. Valid flags are identical to the blit-related flags used with the **dwFXCaps** member.

dwNLVBRops[DD_ROP_SPACE]
Raster operations supported for nonlocal-to-local video memory blits.

ddsCaps
DDSCAPS2 structure with general surface capabilities.

Remarks

For backward compatibility with previous versions of DirectX, the ddraw.h header file contains multiple definitions for the **DDCAPS** structure. The version that passes the preprocessor is determined by the value of the DIRECTDRAW_VERSION constant. For details, see Component Version Constants.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.

DDCOLORCONTROL

[This is preliminary documentation and subject to change.]

The **DDCOLORCONTROL** structure defines the color controls associated with a DirectDrawVideoPortObject, an overlay surface, or a primary surface.

```
typedef struct _DDCOLORCONTROL {
    DWORD   dwSize;
    DWORD   dwFlags;
```

```
    LONG    lBrightness;
    LONG    lContrast;
    LONG    lHue;
    LONG    lSaturation;
    LONG    lSharpness;
    LONG    lGamma;
    LONG    lColorEnable;
    DWORD   dwReserved1;
} DDCOLORCONTROL, FAR *LPDDCOLORCONTROL;
```

Members

dwSize

The the size of the structure, in bytes. This member must be initialized before use.

dwFlags

Flags specifying which structure members contain valid data . When the structure is returned by the **IDirectDrawColorControl::GetColorControls** method, it also indicates which options are supported by the device.

DDCOLOR_BRIGHTNESS
The **lBrightness** member contains valid data.

DDCOLOR_COLORENABLE
The **lColorEnable** member contains valid data.

DDCOLOR_CONTRAST
The **lContrast** member contains valid data.

DDCOLOR_GAMMA
The **lGamma** member contains valid data.

DDCOLOR_HUE
The **lHue** member contains valid data.

DDCOLOR_SATURATION
The **lSaturation** member contains valid data.

DDCOLOR_SHARPNESS
The **lSharpness** member contains valid data.

lBrightness

Luminance intensity, in IRE units times 100. The valid range is 0 to 10,000. The default is 750, which translates to 7.5 IRE.

lContrast

Relative difference between higher and lower intensity luminance values in IRE units times 100. The valid range is 0 to 20,000. The default value is 10,000 (100 IRE). Higher values of contrast cause darker luminance values to tend towards black, and cause lighter luminance values to tend towards white. Lower values of contrast cause all luminance values to move towards the middle luminance values.

lHue

Phase relationship of the chrominance components. Hue is specified in degrees and the valid range is -180 to 180. The default is 0.

lSaturation
Color intensity, in IRE units times 100. The valid range is 0 to 20,000. The default value is 10,000, which translates to 100 IRE.

lSharpness
Sharpness in arbitrary units. The valid range is 0 to 10. The default value is 5.

lGamma
Controls the amount of gamma correction applied to the luminance values. The valid range is 1 to 500 gamma units, with a default of 1.

lColorEnable
Flag indicating whether color is used. If this member is zero, color is not used; if it is 1, then color is used. The default value is 1.

dwReserved1
This member is reserved.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.

DDDEVICEIDENTIFIER

[This is preliminary documentation and subject to change.]

The **DDDEVICEIDENTIFIER** structure is passed to the **IDirectDraw4::GetDeviceIdentifier** method to obtain information about a device.

```
typedef struct tagDDDEVICEIDENTIFIER {
    char    szDriver[MAX_DDDEVICEID_STRING];
    char    szDescription[MAX_DDDEVICEID_STRING];
    LARGE_INTEGER  liDriverVersion;
    DWORD   dwVendorId;
    DWORD   dwDeviceId;
    DWORD   dwSubSysId;
    DWORD   dwRevision;
    GUID    guidDeviceIdentifier;
} DDDEVICEIDENTIFIER, * LPDDDEVICEIDENTIFIER;
```

Members

szDriver
Name of the driver.

szDescription
Description of the driver.

liDriverVersion
Version of the driver. It is legal to do less than and greater than comparisons on the whole 64 bits. Caution should be exercised if you use this element to identify problematic drivers. It is recommended that **guidDeviceIdentifier** be used for this purpose.

The data takes the following form:

```
wProduct = HIWORD(liDriverVersion.HighPart)
wVersion = LOWORD(liDriverVersion.HighPart)
wSubVersion = HIWORD(liDriverVersion.LowPart)
wBuild = LOWORD(liDriverVersion.LowPart)
```

dwVendorId
Identifier of the manufacturer. Can be 0 if unknown.

dwDeviceId
Identifier of the type of chipset. Can be 0 if unknown.

dwSubSysId
Identifier of the subsystem. Typically this means the particular board. Can be 0 if unknown.

dwRevision
Identifier of the revision level of the chipset. Can be 0 if unknown.

guidDeviceIdentifier
Unique identifier for the driver/chipset pair. Use this value if you wish to track changes to the driver/chipset in order to reprofile the graphics subsystem. It can also be used to identify particular problematic drivers.

Remarks

The values in **szDriver** and **szDescription** are for presentation to the user only. They should not be used to identify particular drivers, because different strings might be associated with the same device, or the same driver from different vendors might be described differently.

The **dwVendorId, dwDeviceId**, **dwSubSysId**, and **dwRevision** members can be used to identify particular chipsets, but use extreme caution.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.

DDGAMMARAMP

[This is preliminary documentation and subject to change.]

The **DDGAMMARAMP** structure contains red, green, and blue ramp data for the **IDirectDrawGammaControl::GetGammaRamp** and **IDirectDrawGammaControl::SetGammaRamp** methods.

```
typedef struct _DDGAMMARAMP {
    WORD red[256];
    WORD green[256];
    WORD blue[256];
} DDGAMMARAMP, FAR * LPDDGAMMARAMP;
```

Members

red, **green**, and **blue**
Array of 256 **WORD** elements that describe the red, green, and blue gamma ramps.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.

See Also

Gamma and Color Controls

DDCOLORKEY

[This is preliminary documentation and subject to change.]

The **DDCOLORKEY** structure describes a source color key, destination color key, or color space. A color key is specified if the low and high range values are the same. This structure is used with the **IDirectDrawSurface4::GetColorKey** and **IDirectDrawSurface4::SetColorKey** methods.

```
typedef struct _DDCOLORKEY{
    DWORD dwColorSpaceLowValue;
    DWORD dwColorSpaceHighValue;
} DDCOLORKEY,FAR* LPDDCOLORKEY;
```

Members

dwColorSpaceLowValue
Low value, inclusive, of the color range that is to be used as the color key.

dwColorSpaceHighValue
High value, inclusive, of the color range that is to be used as the color key.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.

DDOVERLAYFX

[This is preliminary documentation and subject to change.]

The **DDOVERLAYFX** structure passes override information to the **IDirectDrawSurface4::UpdateOverlay** method.

```
typedef struct _DDOVERLAYFX{
    DWORD dwSize;
    DWORD dwAlphaEdgeBlendBitDepth;
    DWORD dwAlphaEdgeBlend;
    DWORD dwReserved;
    DWORD dwAlphaDestConstBitDepth;
    union
    {
        DWORD            dwAlphaDestConst;
        LPDIRECTDRAWSURFACE lpDDSAlphaDest;
    } DUMMYUNIONNAMEN(1);
    DWORD dwAlphaSrcConstBitDepth;
    union
    {
        DWORD            dwAlphaSrcConst;
        LPDIRECTDRAWSURFACE lpDDSAlphaSrc;
    } DUMMYUNIONNAMEN(2);
    DDCOLORKEY dckDestColorkey;
    DDCOLORKEY dckSrcColorkey;

    DWORD dwDDFX;
    DWORD dwFlags;
} DDOVERLAYFX,FAR *LPDDOVERLAYFX;
```

Members

dwSize
Size of the structure, in bytes. This members must be initialized before the structure is used.

dwAlphaEdgeBlendBitDepth
Bit depth used to specify the constant for an alpha edge blend.

dwAlphaEdgeBlend
Constant to use as the alpha for an edge blend.

dwReserved
Reserved for future use.

dwAlphaDestConstBitDepth
Bit depth used to specify the alpha constant for a destination.

dwAlphaDestConst
Constant to use as the alpha channel for a destination.

lpDDSAlphaDest
Address of a surface to use as the alpha channel for a destination.

dwAlphaSrcConstBitDepth
Bit depth used to specify the alpha constant for a source.

dwAlphaSrcConst
Constant to use as the alpha channel for a source.

lpDDSAlphaSrc
Address of a surface to use as the alpha channel for a source.

dckDestColorkey
Destination color key override.

dckSrcColorkey
Source color key override.

dwDDFX
Overlay FX flags.

DDOVERFX_ARITHSTRETCHY
If stretching, use arithmetic stretching along the y-axis for this overlay.

DDOVERFX_MIRRORLEFTRIGHT
Mirror the overlay around the vertical axis.

DDOVERFX_MIRRORUPDOWN
Mirror the overlay around the horizontal axis.

dwFlags
This member is currently not used and must be set to 0.

Remarks

The unions in this structure have been updated to work with compilers that don't support nameless unions. If your compiler doesn't support nameless unions, define the NONAMELESSUNION token before including the Ddraw.h header file.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.

DDPIXELFORMAT

[This is preliminary documentation and subject to change.]

The **DDPIXELFORMAT** structure describes the pixel format of a DirectDrawSurface object for the **IDirectDrawSurface4::GetPixelFormat** method.

```
typedef struct _DDPIXELFORMAT{
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwFourCC;
    union
    {
       DWORD dwRGBBitCount;
       DWORD dwYUVBitCount;
       DWORD dwZBufferBitDepth;
       DWORD dwAlphaBitDepth;
       DWORD dwLuminanceBitCount; // new for DirectX 6.0
       DWORD dwBumpBitCount;      // new for DirectX 6.0
    } DUMMYUNIONNAMEN(1);
    union
    {
       DWORD dwRBitMask;
       DWORD dwYBitMask;
       DWORD dwStencilBitDepth;   // new for DirectX 6.0
       DWORD dwLuminanceBitMask;  // new for DirectX 6.0
       DWORD dwBumpDuBitMask;     // new for DirectX 6.0
    } DUMMYUNIONNAMEN(2);
    union
    {
       DWORD dwGBitMask;
       DWORD dwUBitMask;
       DWORD dwZBitMask;          // new for DirectX 6.0
       DWORD dwBumpDvBitMask;     // new for DirectX 6.0
    } DUMMYUNIONNAMEN(3);
    union
    {
       DWORD dwBBitMask;
       DWORD dwVBitMask;
       DWORD dwStencilBitMask;    // new for DirectX 6.0
```

```
    DWORD dwBumpLuminanceBitMask;  // new for DirectX 6.0
  } DUMMYUNIONNAMEN(4);
  union
  {
    DWORD dwRGBAlphaBitMask;
    DWORD dwYUVAlphaBitMask;
    DWORD dwLuminanceAlphaBitMask; // new for DirectX 6.0
    DWORD dwRGBZBitMask;
    DWORD dwYUVZBitMask;
  } DUMMYUNIONNAMEN(5);
} DDPIXELFORMAT, FAR* LPDDPIXELFORMAT;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags

Optional control flags.

DDPF_ALPHA
The pixel format describes an alpha-only surface.

DDPF_ALPHAPIXELS
The surface has alpha channel information in the pixel format.

DDPF_ALPHAPREMULT
The surface uses the premultiplied alpha format. That is, the color components in each pixel are premultiplied by the alpha component.

DDPF_BUMPLUMINANCE
The luminance data in the pixel format is valid, and the **dwLuminanceBitMask** member describes valid luminance bits for a luminance-only or luminance-alpha surface.

DDPF_BUMPDUDV
Bump-map data in the pixel format is valid. Bump-map information is in the **dwBumpBitCount**, **dwBumpDuBitMask**, **dwBumpDvBitMask**, and **dwBumpLuminanceBitMask** members.

DDPF_COMPRESSED
The surface will accept pixel data in the specified format and compress it during the write operation.

DDPF_FOURCC
The **dwFourCC** member is valid and contains a FOURCC code describing a non-RGB pixel format.

DDPF_LUMINANCE
The pixel format describes a luminance-only or luminance-alpha surface.

DDPF_PALETTEINDEXED1
DDPF_PALETTEINDEXED2

DDPF_PALETTEINDEXED4
DDPF_PALETTEINDEXED8
The surface is 1-, 2-, 4-, or 8-bit color indexed.

DDPF_PALETTEINDEXEDTO8
The surface is 1-, 2-, or 4-bit color indexed to an 8-bit palette.

DDPF_RGB
The RGB data in the pixel format structure is valid.

DDPF_RGBTOYUV
The surface will accept RGB data and translate it during the write operation to YUV data. The format of the data to be written will be contained in the pixel format structure. The DDPF_RGB flag will be set.

DDPF_STENCILBUFFER
The surface encodes stencil and depth information in each pixel of the z-buffer. This flag can only be used if the DDPF_ZBUFFER flag is also specified.

DDPF_YUV
The YUV data in the pixel format structure is valid.

DDPF_ZBUFFER
The pixel format describes a z-buffer surface.

DDPF_ZPIXELS
The surface contains z information in the pixels.

dwFourCC
FourCC code. For more information see, Four Character Codes (FOURCC).

dwRGBBitCount
RGB bits per pixel (4, 8, 16, 24, or 32).

dwYUVBitCount
YUV bits per pixel (4, 8, 16, 24, or 32).

dwZBufferBitDepth
Z-buffer bit depth (8, 16, 24, or 32).

dwAlphaBitDepth
Alpha channel bit depth (1, 2, 4, or 8) for an alpha-only surface (DDPF_ALPHA). For pixel formats that contain alpha information interleaved with color data (DDPF_ALPHAPIXELS), you must count the bits in the **dwRGBAlphaBitMask** member to obtain the bit-depth of the alpha component. For more information, see Remarks.

dwLuminanceBitCount
Total luminance bits per pixel. This member applies only to luminance-only and luminance-alpha surfaces.

dwBumpBitCount
Total bump-map bits per pixel in a bump-map surface.

dwRBitMask
Mask for red bits.

dwYBitMask

Mask for Y bits.

dwStencilBitDepth
Bit depth of the stencil buffer. This member specifies how many bits are reserved within each pixel of the z-buffer for stencil information (the total number of z-bits is equal to **dwZBufferBitDepth** minus **dwStencilBitDepth**).

dwLuminanceBitMask
Mask for luminance bits.

dwBumpDuBitMask
Mask for bump-map U_{Δ} bits.

dwGBitMask
Mask for green bits.

dwUBitMask
Mask for U bits.

dwZBitMask
Mask for z bits.

dwBumpDvBitMask
Mask for bump-map V_{Δ} bits.

dwBBitMask
Mask for blue bits.

dwVBitMask
Mask for V bits.

dwStencilBitMask
Mask for stencil bits within each z-buffer pixel.

dwBumpLuminanceBitMask
Mask for luminance in a bump-map pixel.

dwRGBAlphaBitMask and **dwYUVAlphaBitMask** and **dwLuminanceAlphaBitMask**
Masks for alpha channel.

dwRGBZBitMask and **dwYUVZBitMask**
Masks for z channel.

Remarks

The **dwAlphaBitDepth** member reflects the bit depth of an alpha-only pixel format (DDPF_ALPHA). For pixel formats that include the alpha component with color components (DDPF_ALPHAPIXELS), the alpha bit depth is obtained by counting the bits in the various mask members. The following example function returns the number of bits set in a given bitmask:

```
WORD GetNumberOfBits( DWORD dwMask )
{
   WORD wBits = 0;
   while( dwMask )
   {
```

```
        dwMask = dwMask & ( dwMask - 1 );
        wBits++;
    }
    return wBits;
}
```

The unions in this structure have been updated to work with compilers that don't support nameless unions. If your compiler doesn't support nameless unions, define the NONAMELESSUNION token before including the Ddraw.h header file.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.

See Also

Off-Screen Surface Formats

DDSCAPS

[This is preliminary documentation and subject to change.]

The **DDSCAPS** structure defines the capabilities of a DirectDrawSurface object. This structure is part of the **DDCAPS** and **DDSURFACEDESC** structures.

```
typedef struct _DDSCAPS{
    DWORD dwCaps;
} DDSCAPS, FAR* LPDDSCAPS;
```

Members

dwCaps
Capabilities of the surface. One or more of the following flags:

DDSCAPS_3D
Unsupported. Use the DDSCAPS_3DDEVICE instead.

DDSCAPS_3DDEVICE
Indicates that this surface can be used for 3-D rendering. Applications can use this flag to ensure that a device that can only render to a certain heap has off-screen surfaces allocated from the correct heap. If this flag is set for a heap, the surface is not allocated from that heap.

DDSCAPS_ALLOCONLOAD
Indicates that memory for the surface is not allocated until the surface is loaded by using the **IDirect3DTexture2::Load**.

DDSCAPS_ALPHA
Indicates that this surface contains alpha-only information.

DDSCAPS_BACKBUFFER
Indicates that this surface is the back buffer of a surface flipping structure. Typically, this capability is set by the **CreateSurface** method when the DDSCAPS_FLIP flag is used. Only the surface that immediately precedes the DDSCAPS_FRONTBUFFER surface has this capability set. The other surfaces are identified as back buffers by the presence of the DDSCAPS_FLIP flag, their attachment order, and the absence of the DDSCAPS_FRONTBUFFER and DDSCAPS_BACKBUFFER capabilities. If this capability is sent to the **CreateSurface** method, a stand-alone back buffer is being created. After this method is called, this surface could be attached to a front buffer, another back buffer, or both to form a flipping surface structure. For more information, see **IDirectDrawSurface4::AddAttachedSurface**. DirectDraw supports an arbitrary number of surfaces in a flipping structure.

DDSCAPS_COMPLEX
Indicates that a complex surface is being described. A complex surface results in the creation of more than one surface. The additional surfaces are attached to the root surface. The complex structure can be destroyed only by destroying the root.

DDSCAPS_FLIP
Indicates that this surface is a part of a surface flipping structure. When this capability is passed to the **CreateSurface** method, a front buffer and one or more back buffers are created. DirectDraw sets the DDSCAPS_FRONTBUFFER bit on the front-buffer surface and the DDSCAPS_BACKBUFFER bit on the surface adjacent to the front-buffer surface. The **dwBackBufferCount** member of the **DDSURFACEDESC** structure must be set to at least 1 in order for the method call to succeed. The DDSCAPS_COMPLEX capability must always be set when creating multiple surfaces by using the **CreateSurface** method.

DDSCAPS_FRONTBUFFER
Indicates that this surface is the front buffer of a surface flipping structure. This flag is typically set by the **CreateSurface** method when the DDSCAPS_FLIP capability is set. If this capability is sent to the **CreateSurface** method, a stand-alone front buffer is created. This surface will not have the DDSCAPS_FLIP capability. It can be attached to other back buffers to form a flipping structure by using **IDirectDrawSurface4::AddAttachedSurface**.

DDSCAPS_HWCODEC
Indicates that this surface should be able to have a stream decompressed to it by the hardware.

DDSCAPS_LIVEVIDEO
Indicates that this surface should be able to receive live video.

DDSCAPS_LOCALVIDMEM

Indicates that this surface exists in true, local video memory rather than non-local video memory. If this flag is specified then DDSCAPS_VIDEOMEMORY must be specified as well. This flag cannot be used with the DDSCAPS_NONLOCALVIDMEM flag.

DDSCAPS_MIPMAP
Indicates that this surface is one level of a mipmap. This surface will be attached to other DDSCAPS_MIPMAP surfaces to form the mipmap. This can be done explicitly by creating a number of surfaces and attaching them by using the **IDirectDrawSurface4::AddAttachedSurface** method, or implicitly by the **CreateSurface** method. If this capability is set, DDSCAPS_TEXTURE must also be set.

DDSCAPS_MODEX
Indicates that this surface is a 320×200 or 320×240 Mode X surface.

DDSCAPS_NONLOCALVIDMEM
Indicates that this surface exists in non-local video memory rather than true, local video memory. If this flag is specified, then DDSCAPS_VIDEOMEMORY flag must be specified as well. This cannot be used with the DDSCAPS_LOCALVIDMEM flag.

DDSCAPS_OFFSCREENPLAIN
Indicates that this surface is any off-screen surface that is not an overlay, texture, z-buffer, front-buffer, back-buffer, or alpha surface. It is used to identify plain surfaces.

DDSCAPS_OPTIMIZED
Not currently implemented.

DDSCAPS_OVERLAY
Indicates that this surface is an overlay. It may or may not be directly visible depending on whether it is currently being overlaid onto the primary surface. DDSCAPS_VISIBLE can be used to determine if it is being overlaid at the moment.

DDSCAPS_OWNDC
Indicates that this surface will have a device context (DC) association for a long period.

DDSCAPS_PALETTE
Indicates that this device driver allows unique DirectDrawPalette objects to be created and attached to this surface.

DDSCAPS_PRIMARYSURFACE
Indicates the surface is the primary surface. It represents what is visible to the user at the moment.

DDSCAPS_PRIMARYSURFACELEFT
Indicates that this surface is the primary surface for the left eye. It represents what is visible to the user's left eye at the moment. When this surface is created, the surface with the DDSCAPS_PRIMARYSURFACE capability represents what is seen by the user's right eye.

DDSCAPS_STANDARDVGAMODE

Indicates that this surface is a standard VGA mode surface, and not a Mode X surface. This flag cannot be used in combination with the DDSCAPS_MODEX flag.

DDSCAPS_SYSTEMMEMORY
Indicates that this surface memory was allocated in system memory.

DDSCAPS_TEXTURE
Indicates that this surface can be used as a 3-D texture. It does not indicate whether the surface is being used for that purpose.

DDSCAPS_VIDEOMEMORY
Indicates that this surface exists in display memory.

DDSCAPS_VIDEOPORT
Indicates that this surface can receive data from a video port.

DDSCAPS_VISIBLE
Indicates that changes made to this surface are immediately visible. It is always set for the primary surface, as well as for overlays while they are being overlaid and texture maps while they are being textured.

DDSCAPS_WRITEONLY
Indicates that only write access is permitted to the surface. Read access from the surface may generate a general protection (GP) fault, but the read results from this surface will not be meaningful.

DDSCAPS_ZBUFFER
Indicates that this surface is the z-buffer. The z-buffer contains information that cannot be displayed. Instead, it contains bit-depth information that is used to determine which pixels are visible and which are obscured.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.

DDSCAPS2

[This is preliminary documentation and subject to change.]

The **DDSCAPS2** structure defines the capabilities of a DirectDrawSurface object. This structure is part of the **DDSURFACEDESC2** structure.

```
typedef struct _DDSCAPS2 {
    DWORD    dwCaps;   // Surface capabilities
    DWORD    dwCaps2;  // More surface capabilities
    DWORD    dwCaps3;  // Not currently used
    DWORD    dwCaps4;  // .
} DDSCAPS2, FAR* LPDDSCAPS2;
```

Members

dwCaps

One or more flag values representing the capabilities of the surface. (The flags in this member are identical to those in the corresponding member of the **DDSCAPS** structure.)

DDSCAPS_3D
Unsupported. Use the DDSCAPS_3DDEVICE instead.

DDSCAPS_3DDEVICE
Indicates that this surface can be used for 3-D rendering. Applications can use this flag to ensure that a device that can only render to a certain heap has off-screen surfaces allocated from the correct heap. If this flag is set for a heap, the surface is not allocated from that heap.

DDSCAPS_ALLOCONLOAD
Indicates that memory for the surface is not allocated until the surface is loaded by using the **IDirect3DTexture2::Load**.

DDSCAPS_ALPHA
Indicates that this surface contains alpha-only information.

DDSCAPS_BACKBUFFER
Indicates that this surface is the back buffer of a surface flipping structure. Typically, this capability is set by the **CreateSurface** method when the DDSCAPS_FLIP flag is used. Only the surface that immediately precedes the DDSCAPS_FRONTBUFFER surface has this capability set. The other surfaces are identified as back buffers by the presence of the DDSCAPS_FLIP flag, their attachment order, and the absence of the DDSCAPS_FRONTBUFFER and DDSCAPS_BACKBUFFER capabilities. If this capability is sent to the **CreateSurface** method, a stand-alone back buffer is being created. After this method is called, this surface could be attached to a front buffer, another back buffer, or both to form a flipping surface structure. For more information, see **IDirectDrawSurface4::AddAttachedSurface**. DirectDraw supports an arbitrary number of surfaces in a flipping structure.

DDSCAPS_COMPLEX
Indicates that a complex surface is being described. A complex surface results in the creation of more than one surface. The additional surfaces are attached to the root surface. The complex structure can be destroyed only by destroying the root.

DDSCAPS_FLIP
Indicates that this surface is a part of a surface flipping structure. When this capability is passed to the **CreateSurface** method, a front buffer and one or more back buffers are created. DirectDraw sets the DDSCAPS_FRONTBUFFER bit on the front-buffer surface and the DDSCAPS_BACKBUFFER bit on the surface adjacent to the front-buffer surface. The **dwBackBufferCount** member of the **DDSURFACEDESC**

structure must be set to at least 1 in order for the method call to succeed. The DDSCAPS_COMPLEX capability must always be set when creating multiple surfaces by using the **CreateSurface** method.

DDSCAPS_FRONTBUFFER
Indicates that this surface is the front buffer of a surface flipping structure. This flag is typically set by the **CreateSurface** method when the DDSCAPS_FLIP capability is set. If this capability is sent to the **CreateSurface** method, a stand-alone front buffer is created. This surface will not have the DDSCAPS_FLIP capability. It can be attached to other back buffers to form a flipping structure by using **IDirectDrawSurface4::AddAttachedSurface**.

DDSCAPS_HWCODEC
Indicates that this surface should be able to have a stream decompressed to it by the hardware.

DDSCAPS_LIVEVIDEO
Indicates that this surface should be able to receive live video.

DDSCAPS_LOCALVIDMEM
Indicates that this surface exists in true, local video memory rather than non-local video memory. If this flag is specified then DDSCAPS_VIDEOMEMORY must be specified as well. This flag cannot be used with the DDSCAPS_NONLOCALVIDMEM flag.

DDSCAPS_MIPMAP
Indicates that this surface is one level of a mipmap. This surface will be attached to other DDSCAPS_MIPMAP surfaces to form the mipmap. This can be done explicitly by creating a number of surfaces and attaching them by using the **IDirectDrawSurface4::AddAttachedSurface** method, or implicitly by the **CreateSurface** method. If this capability is set, DDSCAPS_TEXTURE must also be set.

DDSCAPS_MODEX
Indicates that this surface is a 320×200 or 320×240 Mode X surface.

DDSCAPS_NONLOCALVIDMEM
Indicates that this surface exists in non-local video memory rather than true, local video memory. If this flag is specified, then DDSCAPS_VIDEOMEMORY flag must be specified as well. This cannot be used with the DDSCAPS_LOCALVIDMEM flag.

DDSCAPS_OFFSCREENPLAIN
Indicates that this surface is any off-screen surface that is not an overlay, texture, z-buffer, front-buffer, back-buffer, or alpha surface. It is used to identify plain surfaces.

DDSCAPS_OPTIMIZED
Not currently implemented.

DDSCAPS_OVERLAY
Indicates that this surface is an overlay. It may or may not be directly visible depending on whether it is currently being overlaid onto the primary surface.

DDSCAPS_VISIBLE can be used to determine if it is being overlaid at the moment.

DDSCAPS_OWNDC
Indicates that this surface will have a device context (DC) association for a long period.

DDSCAPS_PALETTE
Indicates that this device driver allows unique DirectDrawPalette objects to be created and attached to this surface.

DDSCAPS_PRIMARYSURFACE
Indicates the surface is the primary surface. It represents what is visible to the user at the moment.

DDSCAPS_PRIMARYSURFACELEFT
Indicates that this surface is the primary surface for the left eye. It represents what is visible to the user's left eye at the moment. When this surface is created, the surface with the DDSCAPS_PRIMARYSURFACE capability represents what is seen by the user's right eye.

DDSCAPS_STANDARDVGAMODE
Indicates that this surface is a standard VGA mode surface, and not a Mode X surface. This flag cannot be used in combination with the DDSCAPS_MODEX flag.

DDSCAPS_SYSTEMMEMORY
Indicates that this surface memory was allocated in system memory.

DDSCAPS_TEXTURE
Indicates that this surface can be used as a 3-D texture. It does not indicate whether the surface is being used for that purpose.

DDSCAPS_VIDEOMEMORY
Indicates that this surface exists in display memory.

DDSCAPS_VIDEOPORT
Indicates that this surface can receive data from a video port.

DDSCAPS_VISIBLE
Indicates that changes made to this surface are immediately visible. It is always set for the primary surface, as well as for overlays while they are being overlaid and texture maps while they are being textured.

DDSCAPS_WRITEONLY
Indicates that only write access is permitted to the surface. Read access from the surface may generate a general protection (GP) fault, but the read results from this surface will not be meaningful.

DDSCAPS_ZBUFFER
Indicates that this surface is the z-buffer. The z-buffer contains information that cannot be displayed. Instead, it contains bit-depth information that is used to determine which pixels are visible and which are obscured.

dwCaps2
Additional surface capabilities. This member can contain one or more of the following capability flags or, when using this structure with the

IDirectDrawSurface4::SetSurfaceDesc method, this member can contain an additional flag to indicate how the surface memory was allocated:

Capability flags

DDSCAPS2_HARDWAREDEINTERLACE
: Indicates that this surface will receive data from a video port using the de-interlacing hardware. This allows the driver to allocate memory for any extra buffers that may be required. The DDSCAPS_VIDEOPORT and DDSCAPS_OVERLAY flags must also be set.

DDSCAPS2_HINTANTIALIASING
: Indicates that the application intends to use antialiasing. Only valid if DDSCAPS_3DDEVICE is also set.

DDSCAPS2_HINTDYNAMIC
: Indicates to the driver that this surface will be locked very frequently (for procedural textures, dynamic light maps, and so on). This flag can only be used for texture surfaces (DDSCAPS_TEXTURE flag set in the **dwCaps** member). This flag cannot be used with the DDSCAPS2_HINTSTATIC or DDSCAPS2_OPAQUE flags.

DDSCAPS2_HINTSTATIC
: Indicates to the driver that this surface can be reordered or retiled on load. This operation will not change the size of the texture. It is relatively fast and symmetrical, since the application may lock these bits (although it will take a performance hit when doing so).

  This flag can only be used for texture surfaces (DDSCAPS_TEXTURE flag set in the **dwCaps** member). This flag cannot be used with the DDSCAPS2_HINTDYNAMIC or DDSCAPS2_OPAQUE flags.

DDSCAPS2_OPAQUE
: Indicates to the driver that this surface will never be locked again. The driver is free to optimize this surface by retiling and actual compression. Such a surface cannot be locked or used in blit operations, attempts to lock or blit a surface with this capability will fail. This flag can only be used for texture surfaces (DDSCAPS_TEXTURE flag set in the **dwCaps** member). This flag cannot be used with the DDSCAPS2_HINTDYNAMIC or DDSCAPS2_HINTSTATIC flags.

DDSCAPS2_TEXTUREMANAGE
: Indicates that the client would like this texture surface to be managed by DirectDraw and Direct3D Immediate Mode. This flag can only be used for texture surfaces (DDSCAPS_TEXTURE flag set in the **dwCaps** member). For more information, see Automatic Texture Management in the Direct3D Immediate Mode documentation. Do not use this flag if your application uses Direct3D Retained Mode. Instead, create textures in system memory and allow Retained Mode to manage them.

dwCaps3 and **dwCaps4**
: Not currently used.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.

DDSURFACEDESC

[This is preliminary documentation and subject to change.]

The **DDSURFACEDESC** structure contains a description of a surface. This structure is passed to the **IDirectDraw2::CreateSurface** method. The relevant members differ for each potential type of surface.

When using the **IDirectDraw4** interface, this structure is superseded by the **DDSURFACEDESC2** structure.

```
typedef struct _DDSURFACEDESC {
    DWORD           dwSize;
    DWORD           dwFlags;
    DWORD           dwHeight;
    DWORD           dwWidth;
    union
    {
        LONG        lPitch;
        DWORD       dwLinearSize;
    };
    DWORD           dwBackBufferCount;
    union
    {
        DWORD       dwMipMapCount;
        DWORD       dwZBufferBitDepth;
        DWORD       dwRefreshRate;
    };
    DWORD           dwAlphaBitDepth;
    DWORD           dwReserved;
    LPVOID          lpSurface;
    DDCOLORKEY      ddckCKDestOverlay;
    DDCOLORKEY      ddckCKDestBlt;
    DDCOLORKEY      ddckCKSrcOverlay;
    DDCOLORKEY      ddckCKSrcBlt;
    DDPIXELFORMAT   ddpfPixelFormat;
    DDSCAPS         ddsCaps;
} DDSURFACEDESC, FAR* LPDDSURFACEDESC;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags
Optional control flags. One or more of the following flags:

DDSD_ALL
Indicates that all input members are valid.

DDSD_ALPHABITDEPTH
Indicates that the **dwAlphaBitDepth** member is valid.

DDSD_BACKBUFFERCOUNT
Indicates that the **dwBackBufferCount** member is valid.

DDSD_CAPS
Indicates that the **ddsCaps** member is valid.

DDSD_CKDESTBLT
Indicates that the **ddckCKDestBlt** member is valid.

DDSD_CKDESTOVERLAY
Indicates that the **ddckCKDestOverlay** member is valid.

DDSD_CKSRCBLT
Indicates that the **ddckCKSrcBlt** member is valid.

DDSD_CKSRCOVERLAY
Indicates that the **ddckCKSrcOverlay** member is valid.

DDSD_HEIGHT
Indicates that the **dwHeight** member is valid.

DDSD_LINEARSIZE
Indicates that **dwLinearSize** member is valid.

DDSD_LPSURFACE
Indicates that the **lpSurface** member is valid.

DDSD_MIPMAPCOUNT
Indicates that the **dwMipMapCount** member is valid.

DDSD_PITCH
Indicates that the **lPitch** member is valid.

DDSD_PIXELFORMAT
Indicates that the **ddpfPixelFormat** member is valid.

DDSD_REFRESHRATE
Indicates that the **dwRefreshRate** member is valid.

DDSD_WIDTH
Indicates that the **dwWidth** member is valid.

DDSD_ZBUFFERBITDEPTH
Indicates that the **dwZBufferBitDepth** member is valid.

dwHeight and **dwWidth**
Dimensions of the surface to be created, in pixels.

lPitch
Distance, in bytes, to the start of next line. When used with the **IDirectDrawSurface4::GetSurfaceDesc** method, this is a return value. When creating a surface from existing memory or when calling the **IDirectDrawSurface4::SetSurfaceDesc** method, this is an input value that must be a **DWORD** multiple.

dwLinearSize
The size of the buffer. Currently returned only for compressed texture surfaces.

dwBackBufferCount
Number of back buffers.

dwMipMapCount
Number of mipmap levels.

dwZBufferBitDepth
Depth of z-buffer. This member is obsolete for DirectX 6.0 and later. Use the **IDirect3D3::EnumZBufferFormats** to retrieve information about supported depth buffer formats.

dwRefreshRate
Refresh rate (used when the display mode is described). The value of 0 indicates an adapter fault.

dwAlphaBitDepth
Depth of alpha buffer.

dwReserved
Reserved.

lpSurface
Address of the associated surface memory. When calling **IDirectDrawSurface4::Lock**, this member contains a valid pointer to surface memory after the call returns. When creating a surface from existing memory or when using the **IDirectDrawSurface4::SetSurfaceDesc** method, this member is an input value that is the address of system memory allocated by the calling application. Do not set this member if your application needs DirectDraw to allocate and manage surface memory.

ddckCKDestOverlay
DDCOLORKEY structure that describes the destination color key to be used for an overlay surface.

ddckCKDestBlt
DDCOLORKEY structure that describes the destination color key for blit operations.

ddckCKSrcOverlay
DDCOLORKEY structure that describes the source color key to be used for an overlay surface.

ddckCKSrcBlt
DDCOLORKEY structure that describes the source color key for blit operations.

ddpfPixelFormat

DDPIXELFORMAT structure that describes the surface's pixel format.

ddsCaps
DDSCAPS structure containing the surface's capabilities.

Remarks

This structure is similar to the **DDSURFACEDESC2** structure, but contains a **DDSCAPS** structure as the **ddsCaps** member, rather than a **DDSCAPS2** structure. Unlike **DDSURFACEDESC2**, this structure contains the **dwZBufferBitDepth** member.

QuickInfo

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.

DDSURFACEDESC2

[This is preliminary documentation and subject to change.]

The **DDSURFACEDESC2** structure contains a description of a surface. This structure is used to pass surface parameters to the **IDirectDraw4::CreateSurface** and **IDirectDrawSurface4::SetSurfaceDesc** methods. It is also used to retrieve information about a surface in calls to **IDirectDrawSurface4::Lock** and **IDirectDrawSurface4::GetSurfaceDesc**. The relevant members differ for each potential type of surface.

```
typedef struct _DDSURFACEDESC2 {
    DWORD       dwSize;
    DWORD       dwFlags;
    DWORD       dwHeight;
    DWORD       dwWidth;
    union
    {
        LONG    lPitch;
        DWORD   dwLinearSize;
    } DUMMYUNIONNAMEN(1);
    DWORD       dwBackBufferCount;
    union
    {
        DWORD   dwMipMapCount;
        DWORD   dwRefreshRate;
    } DUMMYUNIONNAMEN(2);
    DWORD       dwAlphaBitDepth;
    DWORD       dwReserved;
```

```
    LPVOID       lpSurface;
    DDCOLORKEY   ddckCKDestOverlay;
    DDCOLORKEY   ddckCKDestBlt;
    DDCOLORKEY   ddckCKSrcOverlay;
    DDCOLORKEY   ddckCKSrcBlt;
    DDPIXELFORMAT ddpfPixelFormat;
    DDSCAPS2     ddsCaps;
    DWORD        dwTextureStage;
} DDSURFACEDESC2, FAR* LPDDSURFACEDESC2;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags

Optional control flags. One or more of the following flags:

DDSD_ALL
Indicates that all input members are valid.

DDSD_ALPHABITDEPTH
Indicates that the **dwAlphaBitDepth** member is valid.

DDSD_BACKBUFFERCOUNT
Indicates that the **dwBackBufferCount** member is valid.

DDSD_CAPS
Indicates that the **ddsCaps** member is valid.

DDSD_CKDESTBLT
Indicates that the **ddckCKDestBlt** member is valid.

DDSD_CKDESTOVERLAY
Indicates that the **ddckCKDestOverlay** member is valid.

DDSD_CKSRCBLT
Indicates that the **ddckCKSrcBlt** member is valid.

DDSD_CKSRCOVERLAY
Indicates that the **ddckCKSrcOverlay** member is valid.

DDSD_HEIGHT
Indicates that the **dwHeight** member is valid.

DDSD_LINEARSIZE
Indicates that the **dwLinearSize** member is valid.

DDSD_LPSURFACE
Indicates that the **lpSurface** member is valid.

DDSD_MIPMAPCOUNT
Indicates that the **dwMipMapCount** member is valid.

DDSD_PITCH
Indicates that the **lPitch** member is valid.

DDSD_PIXELFORMAT
Indicates that the **ddpfPixelFormat** member is valid.

DDSD_REFRESHRATE
Indicates that the **dwRefreshRate** member is valid.

DDSD_TEXTURESTAGE
Indicates that the **dwTextureStage** member is valid.

DDSD_WIDTH
Indicates that the **dwWidth** member is valid.

DDSD_ZBUFFERBITDEPTH
Obsolete; see remarks.

dwHeight and **dwWidth**
Dimensions of the surface to be created, in pixels.

lPitch
Distance, in bytes, to the start of next line. When used with the **IDirectDrawSurface4::GetSurfaceDesc** method, this is a return value. When used with the **IDirectDrawSurface4::SetSurfaceDesc** method, this is an input value that must be a **DWORD** multiple. See remarks for more information.

dwLinearSize
The size of the buffer. Currently returned only for compressed texture surfaces.

dwBackBufferCount
Number of back buffers.

dwMipMapCount
Number of mipmap levels.

dwRefreshRate
Refresh rate (used when the display mode is described). The value of 0 indicates an adapter fault.

dwAlphaBitDepth
Depth of alpha buffer.

dwReserved
Reserved.

lpSurface
Address of the associated surface memory. When calling **IDirectDrawSurface4::Lock**, this member is a valid pointer to surface memory. When calling **IDirectDrawSurface4::SetSurfaceDesc**, this member is a pointer to system memory that the caller explicitly allocates for the DirectDrawSurface object. See remarks for more information.

ddckCKDestOverlay
DDCOLORKEY structure that describes the destination color key to be used for an overlay surface.

ddckCKDestBlt
DDCOLORKEY structure that describes the destination color key for blit operations.

ddckCKSrcOverlay

DDCOLORKEY structure that describes the source color key to be used for an overlay surface.

ddckCKSrcBlt
DDCOLORKEY structure that describes the source color key for blit operations.

ddpfPixelFormat
DDPIXELFORMAT structure that describes the surface's pixel format.

ddsCaps
DDSCAPS2 structure containing the surface's capabilities.

dwTextureStage
Stage identifier used to bind a texture to a specific stage in 3-D device's multitexture cascade. Although not required for all hardware, setting this member is recommended for best performance on the largest variety of 3-D accelerators. Hardware that requires explicitly assigned textures will expose the D3DDEVCAPS_SEPARATETEXTUREMEMORIES 3-D device capability in the **D3DDEVICEDESC** structure that is filled by the **IDirect3DDevice3::GetCaps** method.

Remarks

The **lPitch** and **lpSurface** members are output values when calling the **IDirectDrawSurface4::GetSurfaceDesc** method. When creating surfaces from existing memory, or updating surface characteristics, these members are input values that describe the pitch and location of memory allocated by the calling application for use by DirectDraw. DirectDraw does not attempt to manage or free memory allocated by the application. For more information, see Creating Client Memory Surfaces and Updating Surface Characteristics.

This structure is nearly identical to the **DDSURFACEDESC** structure, but contains a **DDSCAPS2** structure as the **ddsCaps** member. Unlike **DDSURFACEDESC**, this structure doesn't contain the **dwZBufferBitDepth** member. Z-buffer depth is provided in the **ddpfPixelFormat** member.

The unions in this structure were written to work with compilers that don't support nameless unions. If your compiler doesn't support nameless unions, define the NONAMELESSUNION token before including the Ddraw.h header file.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in ddraw.h.

DDVIDEOPORTBANDWIDTH

[This is preliminary documentation and subject to change.]

The **DDVIDEOPORTBANDWIDTH** structure describes the bandwidth characteristics of an overlay surface when used with a particular video port and pixel format configuration. This structure is used with the **IDirectDrawVideoPort::GetBandwidthInfo** method.

```
typedef struct _DDVIDEOPORTBANDWIDTH {
  DWORD dwSize;
  DWORD dwCaps;
  DWORD dwOverlay;
  DWORD dwColorkey;
  DWORD dwYInterpolate;
  DWORD dwYInterpAndColorkey;
  DWORD dwReserved1;
  DWORD dwReserved2;
} DDVIDEOPORTBANDWIDTH,*LPDDVIDEOPORTBANDWIDTH;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before use.

dwCaps
Flag values specifying device dependency. This member can be one of the following values:

DDVPBCAPS_DESTINATION
This device's capabilities are described in terms of the overlay's minimum stretch factor. Bandwidth information provided for this device refers to the destination overlay size.

DDVPBCAPS_SOURCE
This device's capabilities are described in terms of the required source overlay size. Bandwidth information provided for this device refers to the source overlay size.

dwOverlay
Stretch factor or overlay source size at which an overlay is supported multiplied by 1000. For example 1.3 = 1300, or .75 = 750.

dwColorkey
Stretch factor or overlay source size at which an overlay with color keying is supported multiplied by 1000. For example 1.3 = 1300, or .75 = 750.

dwYInterpolate
Stretch factor or overlay source size at which an overlay with y-axis interpolation is supported multiplied by 1000. For example 1.3 = 1300, or .75 = 750.

dwYInterpAndColorkey
Stretch factor or overlay source size at which an overlay with y-axis interpolation and color keying is supported multiplied by 1000. For example 1.3 = 1300, or .75 = 750.

dwReserved1 and **dwReserved2**
Reserved; set to zero.

Remarks

When DDVPBCAPS_DESTINATION is specified, the stretch factors described in the other members describe the minimum stretch factor required to display an overlay with the dimensions given when calling the **GetBandwidthInfo** method. Stretch factor values under 1000 mean that the video port is capable of shrinking an overlay when displayed, and values over 1000 mean that the overlay must be stretched larger than their source to be displayed.

When DDVPBCAPS_SOURCE is specified, the stretch factors described in the other members describe how much you must shrink the overlay source in order for it to be displayed. In this case, the best possible value is 1000, meaning that no shrinking is required. Smaller values tell you that the source rectangle you specified when calling **GetBandwidthInfo** were too large and must be smaller. For example, if the stretch factor is 750 and you specified 320 pixels for the *dwWidth* parameter, then you will not be able to display the overlay at that size. To successfully display the overlay, you must use a source rectangle 240 pixels wide to successfully display the overlay.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.

DDVIDEOPORTCAPS

[This is preliminary documentation and subject to change.]

The **DDVIDEOPORTCAPS** structure describes the capabilities and alignment restrictions of a video port. This structure is used with the **IDDVideoPortContainer::EnumVideoPorts** method.

```
typedef struct _DDVIDEOPORTCAPS {
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwMaxWidth;
    DWORD dwMaxVBIWidth;
    DWORD dwMaxHeight;
    DWORD dwVideoPortID;
    DWORD dwCaps;
    DWORD dwFX;
    DWORD dwNumAutoFlipSurfaces;
    DWORD dwAlignVideoPortBoundary;
    DWORD dwAlignVideoPortPrescaleWidth;
```

```
    DWORD dwAlignVideoPortCropBoundary;
    DWORD dwAlignVideoPortCropWidth;
    DWORD dwPreshrinkXStep;
    DWORD dwPreshrinkYStep;
    DWORD dwNumVBIAutoFlipSurfaces;
    DWORD dwNumPreferredAutoflip;
    WORD  wNumFilterTapsX;
    WORD  wNumFilterTapsY;
} DDVIDEOPORTCAPS, *LPDDVIDEOPORTCAPS;
```

Members

dwSize
Size of the structure, in bytes. This must be initialized before use.

dwFlags
Flag values indicating the fields that contain valid data. The following flags are defined:

DDVPD_AUTOFLIP
The **dwNumAutoFlipSurfaces** member is valid.

DDVPD_ALIGN
The **dwAlignVideoPortBoundary**, **dwAlignVideoPortPrescaleWidth**, **dwAlignVideoPortCropBoundary**, and **dwAlignVideoPortCropWidth** are valid.

DDVPD_CAPS
The **dwCaps** member is valid.

DDVPD_FILTERQUALITY
The **wNumFilterTapsX** and **wNumFilterTapsY** members are valid.

DDVPD_FX
The **dwFX** member is valid.

DDVPD_HEIGHT
The **dwMaxHeight** member is valid.

DDVPD_ID
The **dwVideoPortID** member is valid.

DDVPD_PREFERREDAUTOFLIP
The **dwNumPreferredAutoflip** member is valid.

DDVPD_WIDTH
The **dwMaxWidth** member is valid.

dwMaxWidth
Maximum width of the video port field.

dwMaxVBIWidth
Maximum width of the VBI data.

dwMaxHeight
Maximum height of the video port field.

dwVideoPortID
Zero-based index identifying the video port.

dwCaps
Video port capabilities.

DDVPCAPS_AUTOFLIP
Flip can be performed automatically to avoid tearing when a VREF occurs. If the data is being interleaved in memory, it will flip on every other VREF.

DDVPCAPS_COLORCONTROL
Can perform color control operations on incoming data before writing to the frame buffer.

DDVPCAPS_INTERLACED
Supports interlaced video.

DDVPCAPS_NONINTERLACED
Supports non-interlaced video.

DDVPCAPS_OVERSAMPLEDVBI
Can accept VBI data in a different format or width than the regular video data.

DDVPCAPS_READBACKFIELD
Supports the **IDirectDrawVideoPort::GetFieldPolarity** method.

DDVPCAPS_READBACKLINE
Supports the **IDirectDrawVideoPort::GetVideoLine** method.

DDVPCAPS_SHAREABLE
Supports two genlocked video streams that share the video port, where one stream uses the even fields and the other uses the odd fields. Separate parameters (including address, scaling, cropping, and so on) are maintained for both fields.

DDVPCAPS_SKIPEVENFIELDS
Even fields of video can be automatically discarded.

DDVPCAPS_SKIPODDFIELDS
Odd fields of video can be automatically discarded.

DDVPCAPS_SYNCMASTER
Can drive the graphics sync (refresh rate) based on the video port sync.

DDVPCAPS_SYSTEMMEMORY
Capable of writing to surfaces created in system memory.

DDVPCAPS_VBIANDVIDEOINDEPENDENT
Indicates that the VBI and video portions of the video stream can be controlled by independent processes.

DDVPCAPS_VBISURFACE
Data within the VBI can be written to a different surface.

dwFX
Additional video port capabilities.

DDVPFX_CROPTOPDATA
Limited cropping is available to crop VBI data.

DDVPFX_CROPX
Incoming data can be cropped in the x-direction before it is written to the surface.

DDVPFX_CROPY
Incoming data can be cropped in the y-direction before it is written to the surface.

DDVPFX_IGNOREVBIXCROP
The video port can ignore the left and right cropping coordinates when cropping oversampled VBI data.

DDVPFX_INTERLEAVE
Supports interleaving interlaced fields in memory.

DDVPFX_MIRRORLEFTRIGHT
Supports mirroring left to right as the video data is written into the frame buffer.

DDVPFX_MIRRORUPDOWN
Supports mirroring top to bottom as the video data is written into the frame buffer.

DDVPFX_PRESHRINKX
Data can be arbitrarily shrunk in the x-direction before it is written to the surface.

DDVPFX_PRESHRINKY
Data can be arbitrarily shrunk in the y-direction before it is written to the surface.

DDVPFX_PRESHRINKXB
Data can be binary shrunk (1/2, 1/4, 1/8, and so on) in the x-direction before it is written to the surface.

DDVPFX_PRESHRINKYB
Data can be binary shrunk (1/2, 1/4, 1/8, and so on) in the y-direction before it is written to the surface.

DDVPCAPS_PRESHRINKXS
Data can be shrunk in the x-direction by increments of 1/*x*, where *x* is specified in the **dwShrinkXStep** member.

DDVPCAPS_PRESHRINKYS
Data can be shrunk in the y-direction by increments of 1/*y*, where *y* is specified in the **dwShrinkYStep**

DDVPFX_PRESTRETCHX
Data can be arbitrarily stretched in the x-direction before it is written to the surface.

DDVPFX_PRESTRETCHY
Data can be arbitrarily stretched in the y-direction before it is written to the surface.

DDVPFX_PRESTRETCHXN
Data can be integer stretched in the x-direction before it is written to the surface. (1x, 2x, 3x, and so forth)

DDVPFX_PRESTRETCHYN
Data can be integer stretched in the y-direction before it is written to the surface. (1x, 2x, 3x, and so forth)

DDVPFX_VBICONVERT
Data within the VBI can be converted independently of the remaining video data.

DDVPFX_VBINOINTERLEAVE
Interleaving can be disabled for data within the VBI.

DDVPFX_VBINOSCALE
Scaling can be disabled for data within the VBI.

dwNumAutoFlipSurfaces
Maximum number of auto-flippable surfaces supported by the video port.

dwAlignVideoPortBoundary
Byte restriction of placement within the surface.

dwAlignVideoPortPrescaleWidth
Byte restriction of width after prescaling.

dwAlignVideoPortCropBoundary
Byte restriction of left cropping.

dwAlignVideoPortCropWidth
Byte restriction of cropping width.

dwPreshrinkXStep
Width can be shrunk in the x-direction in steps of 1/**dwPreshrinkXStep**.

dwPreshrinkYStep
Height can be shrunk in the y-direction in steps of 1/**dwPreshrinkYStep**.

dwNumVBIAutoFlipSurfaces
Maximum number of auto-flipping surfaces capable of receiving data transmitted during the vertical blanking interval (VBI) independent from the remainder of the video stream. When constructing the auto-flip chain, the number of VBI surfaces must equal the number of surfaces receiving the remainder of the video data.

dwNumPreferredAutoflip
Optimal number of auto-flippable surfaces supported by the hardware.

wNumFilterTapsX and **wNumFilterTapsY**
Number of taps the prescaler filter uses in the x- and y-directions. The value of 0 indicates that no prescaling is performed in that direction, 1 indicates that the prescaler performs replication, 2 indicates that the prescaler uses two taps, and so on.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.

DDVIDEOPORTCONNECT

[This is preliminary documentation and subject to change.]

The **DDVIDEOPORTCONNECT** structure describes a video port connection. This structure is used with the **IDDVideoPortContainer::GetVideoPortConnectInfo** method.

```
typedef struct _DDVIDEOPORTCONNECT{
    DWORD dwSize;
    DWORD dwPortWidth;
    GUID  guidTypeID;
    DWORD dwFlags;
    ULONG_PTR dwReserved1;
} DDVIDEOPORTCONNECT,*LPDDVIDEOPORTCONNECT;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before use.

dwPortWidth
Width of the video port. This value represents the number of physical pins on the video port, not the width of a surface in memory. This member must always be set, even when the value in the **guidTypeID** member assumes a certain size.

guidTypeID
A GUID that describes the sync characteristics of the video port. The following port types are predefined:

DDVPTYPE_E_HREFH_VREFH
External syncs where HREF is active high and VREF is active high.

DDVPTYPE_E_HREFH_VREFL
External syncs where HREF is active high and VREF is active low.

DDVPTYPE_E_HREFL_VREFH
External syncs where HREF is active low and VREF is active high.

DDVPTYPE_E_HREFL_VREFL
External syncs where HREF is active low and VREF is active low.

DDVPTYPE_CCIR656
Sync information is embedded in the data stream according to the CCIR656 specification.

DDVPTYPE_BROOKTREE
Sync information is embedded in the data stream using the Brooktree definition.

DDVPTYPE_PHILIPS
Sync information is embedded in the data stream using the Philips definition.

dwFlags

Flags describing the capabilities of the video-port connection. This member can be set by DirectDraw when connection information is being retrieved or by the client when connection information is being set. This member can be a combination of the following flags:

DDVPCONNECT_DOUBLECLOCK
Indicates that the video port either supports double-clocking data or should double-clock data. This flag is only valid with an external sync.

DDVPCONNECT_VACT
Indicates that the video port either supports using an external VACT signal or should use the external VACT signal. This flag is only valid with an external sync.

DDVPCONNECT_INVERTPOLARITY
Indicates that the video port is capable of inverting the field polarities or is to invert field polarities.

When a video port inverts field polarities, it treats even fields as odd fields and vice versa.

DDVPCONNECT_DISCARDSVREFDATA
The video port discards any data written during the VREF period, causing it to not be written to the frame buffer. This flag is read-only.

DDVPCONNECT_HALFLINE
The video port will write half lines into the frame buffer, sometimes causing the data to be displayed incorrectly. This flag is read-only.

DDVPCONNECT_INTERLACED
Indicates that the signal is interlaced. This flag is only used by the client when creating a video port object.

DDVPCONNECT_SHAREEVEN
The physical video port is shareable, and that this video port object will use the even fields. This flag is only used by the client when creating the video port object.

DDVPCONNECT_SHAREODD
The physical video port is shareable, and that this video port object will use the odd fields. This flag is only used by the client when creating the video port object.

dwReserved1
Reserved; set to zero.

Remarks

This structure is used independently and as a member of the **DDVIDEOPORTDESC** structure.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for

Windows 95.
Header: Declared in dvp.h.

DDVIDEOPORTDESC

[This is preliminary documentation and subject to change.]

The **DDVIDEOPORTDESC** structure describes a video-port object to be created. This structure is used with the **IDDVideoPortContainer::CreateVideoPort** method.

```
typedef struct _DDVIDEOPORTDESC {
    DWORD dwSize;
    DWORD dwFieldWidth;
    DWORD dwVBIWidth;
    DWORD dwFieldHeight;
    DWORD dwMicrosecondsPerField;
    DWORD dwMaxPixelsPerSecond;
    DWORD dwVideoPortID;
    DWORD dwReserved1;
    DDVIDEOPORTCONNECT VideoPortType;
    ULONG_PTR dwReserved2;
    ULONG_PTR dwReserved3;
} DDVIDEOPORTDESC, *LPDDVIDEOPORTDESC;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before use.

dwFieldWidth
Width of incoming video stream, in pixels.

dwVBIWidth
Width of the VBI data in the incoming video stream, in pixels.

dwFieldHeight
Field height for fields in the incoming video stream, in scan lines.

dwMicrosecondsPerField
Time interval, in microseconds, between live video VREF periods. This number should be rounded up to the nearest microsecond.

dwMaxPixelsPerSecond
Maximum pixel rate per second.

dwVideoPortID
The zero-based ID of the physical video port to be used.

dwReserved1
Reserved; set to zero.

VideoPortType

A **DDVIDEOPORTCONNECT** structure describing the connection characteristics of the video port.

dwReserved2 and **dwReserved3**
Reserved; set to zero.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.

DDVIDEOPORTINFO

[This is preliminary documentation and subject to change.]

The **DDVIDEOPORTINFO** structure describes the transfer of video data to a surface. This structure is used with the **IDirectDrawVideoPort::StartVideo** method.

```
typedef struct _DDVIDEOPORTINFO{
    DWORD dwSize;
    DWORD dwOriginX;
    DWORD dwOriginY;
    DWORD dwVPFlags;
    RECT rCrop;
    DWORD dwPrescaleWidth;
    DWORD dwPrescaleHeight;
    LPDDPIXELFORMAT lpddpfInputFormat;
    LPDDPIXELFORMAT lpddpfVBIInputFormat;
    LPDDPIXELFORMAT lpddpfVBIOutputFormat;
    DWORD dwVBIHeight;
    ULONG_PTR dwReserved1;
    ULONG_PTR dwReserved2;
} DDVIDEOPORTINFO,*LPDDVIDEOPORTINFO;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before use.

dwOriginX and **dwOriginY**
X- and y-coordinates for the origin of the video data in the surface.

dwVPFlags
Video port options.

DDVP_AUTOFLIP
Perform automatic flipping. For more information, see Auto-flipping.

DDVP_CONVERT
Perform conversion using the information in the **lpddpfVBIOutputFormat** member.

DDVP_CROP
Perform cropping using the rectangle specified by the **rCrop** member.

DDVP_IGNOREVBIXCROP
The video port should ignore left and right cropping coordinates when cropping oversampled VBI data.

DDVP_INTERLEAVE
Interlaced fields should be interleaved in memory.

DDVP_MIRRORLEFTRIGHT
Mirror image data from left to right as it is written into the frame buffer.

DDVP_MIRRORUPDOWN
Mirror image data from top to bottom as it is written into the frame buffer.

DDVP_OVERRIDEBOBWEAVE
Override automatic display method chosen by the driver, using only the display method set by the caller when creating the overlay surface.

DDVP_PRESCALE
Perform pre-scaling or pre-zooming based on the values in the **dwPrescaleHeight** and **dwPrescaleWidth** members.

DDVP_SKIPEVENFIELDS
Ignore input of even fields.

DDVP_SKIPODDFIELDS
Ignore input of odd fields.

DDVP_SYNCMASTER
Indicates that the video port VREF should drive the graphics VREF, locking the refresh rate to the video port.

DDVP_VBICONVERT
The **lpddpfVBIOutputFormat** member contains data that should be used to convert VBI data.

DDVP_VBINOSCALE
VBI data should not be scaled.

DDVP_VBINOINTERLEAVE
Interleaving can be disabled for data within the VBI.

rCrop
Cropping rectangle. This member is optional.

dwPrescaleWidth
Pre-scaling or zooming in the x-direction. This member is optional.

dwPrescaleHeight
Pre-scaling or zooming in the y-direction. This member is optional.

lpddpfInputFormat

A **DDPIXELFORMAT** structure describing the pixel format to be written to the video port. This will often be identical to the surface's pixel format, but can differ if the video port is to perform conversion.

lpddpfVBIInputFormatand **lpddpfVBIOutputFormat**
DDPIXELFORMAT structures describing the input and output pixel formats of the data within the vertical blanking interval.

dwVBIHeight
The amount of data within the vertical blanking interval, in scan lines.

dwReserved1 and **dwReserved2**
Reserved; set to zero.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.

DDVIDEOPORTSTATUS

[This is preliminary documentation and subject to change.]

The **DDVIDEOPORTSTATUS** structure describes the status of a video-port object. This structure is used with the **IDDVideoPortContainer::QueryVideoPortStatus** method.

```
typedef struct _DDVIDEOPORTSTATUS {
    DWORD dwSize;
    BOOL  bInUse;
    DWORD dwFlags;
    DWORD dwReserved1;
    DDVIDEOPORTCONNECT VideoPortType;
    ULONG_PTR dwReserved2;
    ULONG_PTR dwReserved3;
} DDVIDEOPORTSTATUS, *LPDDVIDEOPORTSTATUS;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before use.

bInUse
Value indicating the current status of the video port. This member is TRUE if the video port is currently being used, and FALSE otherwise.

dwFlags
Flags

DDVPSTATUS_VBIONLY
The video port interface is only controlling the VBI portion of the video stream.

DDVPSTATUS_VIDEOONLY
The video port interface is only controlling the video portion of the video stream.

dwReserved1
Reserved; set to zero.

VideoPortType
A **DDVIDEOPORTCONNECT** structure that receives information about the video-port connection.

dwReserved2 and **dwReserved3**
Reserved; set to zero.

QuickInfo

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 95 or later. Available as a redistributable for Windows 95.
Header: Declared in dvp.h.

Return Values

[This is preliminary documentation and subject to change.]

Errors are represented by negative values and cannot be combined. This table lists the values that can be returned by all methods of the **IDirectDraw4**, **IDirectDrawSurface4**, **IDirectDrawPalette**, **IDirectDrawClipper**, **IDDVideoPortContainer**, and **IDirectDrawVideoPort** interfaces. For a list of the error codes that each method can return, see the method description.

DD_OK
The request completed successfully.

DDERR_ALREADYINITIALIZED
The object has already been initialized.

DDERR_BLTFASTCANTCLIP
A DirectDrawClipper object is attached to a source surface that has passed into a call to the **IDirectDrawSurface4::BltFast** method.

DDERR_CANNOTATTACHSURFACE
A surface cannot be attached to another requested surface.

DDERR_CANNOTDETACHSURFACE
A surface cannot be detached from another requested surface.

DDERR_CANTCREATEDC
Windows can not create any more device contexts (DCs), or a DC was requested for a palette-indexed surface when the surface had no palette and

the display mode was not palette-indexed (in this case DirectDraw cannot select a proper palette into the DC).

DDERR_CANTDUPLICATE
Primary and 3-D surfaces, or surfaces that are implicitly created, cannot be duplicated.

DDERR_CANTLOCKSURFACE
Access to this surface is refused because an attempt was made to lock the primary surface without DCI support.

DDERR_CANTPAGELOCK
An attempt to page lock a surface failed. Page lock will not work on a display-memory surface or an emulated primary surface.

DDERR_CANTPAGEUNLOCK
An attempt to page unlock a surface failed. Page unlock will not work on a display-memory surface or an emulated primary surface.

DDERR_CLIPPERISUSINGHWND
An attempt was made to set a clip list for a DirectDrawClipper object that is already monitoring a window handle.

DDERR_COLORKEYNOTSET
No source color key is specified for this operation.

DDERR_CURRENTLYNOTAVAIL
No support is currently available.

DDERR_DCALREADYCREATED
A device context (DC) has already been returned for this surface. Only one DC can be retrieved for each surface.

DDERR_DEVICEDOESNTOWNSURFACE
Surfaces created by one DirectDraw device cannot be used directly by another DirectDraw device.

DDERR_DIRECTDRAWALREADYCREATED
A DirectDraw object representing this driver has already been created for this process.

DDERR_EXCEPTION
An exception was encountered while performing the requested operation.

DDERR_EXCLUSIVEMODEALREADYSET
An attempt was made to set the cooperative level when it was already set to exclusive.

DDERR_EXPIRED
The data has expired and is therefore no longer valid.

DDERR_GENERIC
There is an undefined error condition.

DDERR_HEIGHTALIGN
The height of the provided rectangle is not a multiple of the required alignment.

DDERR_HWNDALREADYSET

The DirectDraw cooperative level window handle has already been set. It cannot be reset while the process has surfaces or palettes created.

DDERR_HWNDSUBCLASSED
DirectDraw is prevented from restoring state because the DirectDraw cooperative level window handle has been subclassed.

DDERR_IMPLICITLYCREATED
The surface cannot be restored because it is an implicitly created surface.

DDERR_INCOMPATIBLEPRIMARY
The primary surface creation request does not match with the existing primary surface.

DDERR_INVALIDCAPS
One or more of the capability bits passed to the callback function are incorrect.

DDERR_INVALIDCLIPLIST
DirectDraw does not support the provided clip list.

DDERR_INVALIDDIRECTDRAWGUID
The globally unique identifier (GUID) passed to the **DirectDrawCreate** function is not a valid DirectDraw driver identifier.

DDERR_INVALIDMODE
DirectDraw does not support the requested mode.

DDERR_INVALIDOBJECT
DirectDraw received a pointer that was an invalid DirectDraw object.

DDERR_INVALIDPARAMS
One or more of the parameters passed to the method are incorrect.

DDERR_INVALIDPIXELFORMAT
The pixel format was invalid as specified.

DDERR_INVALIDPOSITION
The position of the overlay on the destination is no longer legal.

DDERR_INVALIDRECT
The provided rectangle was invalid.

DDERR_INVALIDSTREAM
The specified stream contains invalid data.

DDERR_INVALIDSURFACETYPE
The requested operation could not be performed because the surface was of the wrong type.

DDERR_LOCKEDSURFACES
One or more surfaces are locked, causing the failure of the requested operation.

DDERR_MOREDATA
There is more data available than the specified buffer size can hold.

DDERR_NO3D
No 3-D hardware or emulation is present.

DDERR_NOALPHAHW

No alpha acceleration hardware is present or available, causing the failure of the requested operation.

DDERR_NOBLTHW
No blitter hardware is present.

DDERR_NOCLIPLIST
No clip list is available.

DDERR_NOCLIPPERATTACHED
No DirectDrawClipper object is attached to the surface object.

DDERR_NOCOLORCONVHW
The operation cannot be carried out because no color-conversion hardware is present or available.

DDERR_NOCOLORKEY
The surface does not currently have a color key.

DDERR_NOCOLORKEYHW
The operation cannot be carried out because there is no hardware support for the destination color key.

DDERR_NOCOOPERATIVELEVELSET
A create function is called without the **IDirectDraw4::SetCooperativeLevel** method being called.

DDERR_NODC
No DC has ever been created for this surface.

DDERR_NODDROPSHW
No DirectDraw raster operation (ROP) hardware is available.

DDERR_NODIRECTDRAWHW
Hardware-only DirectDraw object creation is not possible; the driver does not support any hardware.

DDERR_NODIRECTDRAWSUPPORT
DirectDraw support is not possible with the current display driver.

DDERR_NOEMULATION
Software emulation is not available.

DDERR_NOEXCLUSIVEMODE
The operation requires the application to have exclusive mode, but the application does not have exclusive mode.

DDERR_NOFLIPHW
Flipping visible surfaces is not supported.

DDERR_NOFOCUSWINDOW
An attempt was made to create or set a device window without first setting the focus window.

DDERR_NOGDI
No GDI is present.

DDERR_NOHWND
Clipper notification requires a window handle, or no window handle has been previously set as the cooperative level window handle.

DDERR_NOMIPMAPHW
The operation cannot be carried out because no mipmap capable texture mapping hardware is present or available.

DDERR_NOMIRRORHW
The operation cannot be carried out because no mirroring hardware is present or available.

DDERR_NONONLOCALVIDMEM
An attempt was made to allocate non-local video memory from a device that does not support non-local video memory.

DDERR_NOOPTIMIZEHW
The device does not support optimized surfaces.

DDERR_NOOVERLAYDEST
The **IDirectDrawSurface4::GetOverlayPosition** method is called on an overlay that the **IDirectDrawSurface4::UpdateOverlay** method has not been called on to establish a destination.

DDERR_NOOVERLAYHW
The operation cannot be carried out because no overlay hardware is present or available.

DDERR_NOPALETTEATTACHED
No palette object is attached to this surface.

DDERR_NOPALETTEHW
There is no hardware support for 16- or 256-color palettes.

DDERR_NORASTEROPHW
The operation cannot be carried out because no appropriate raster operation hardware is present or available.

DDERR_NOROTATIONHW
The operation cannot be carried out because no rotation hardware is present or available.

DDERR_NOSTRETCHHW
The operation cannot be carried out because there is no hardware support for stretching.

DDERR_NOT4BITCOLOR
The DirectDrawSurface object is not using a 4-bit color palette and the requested operation requires a 4-bit color palette.

DDERR_NOT4BITCOLORINDEX
The DirectDrawSurface object is not using a 4-bit color index palette and the requested operation requires a 4-bit color index palette.

DDERR_NOT8BITCOLOR
The DirectDrawSurface object is not using an 8-bit color palette and the requested operation requires an 8-bit color palette.

DDERR_NOTAOVERLAYSURFACE
An overlay component is called for a non-overlay surface.

DDERR_NOTEXTUREHW

The operation cannot be carried out because no texture-mapping hardware is present or available.

DDERR_NOTFLIPPABLE
An attempt has been made to flip a surface that cannot be flipped.

DDERR_NOTFOUND
The requested item was not found.

DDERR_NOTINITIALIZED
An attempt was made to call an interface method of a DirectDraw object created by **CoCreateInstance** before the object was initialized.

DDERR_NOTLOADED
The surface is an optimized surface, but it has not yet been allocated any memory.

DDERR_NOTLOCKED
An attempt is made to unlock a surface that was not locked.

DDERR_NOTPAGELOCKED
An attempt is made to page unlock a surface with no outstanding page locks.

DDERR_NOTPALETTIZED
The surface being used is not a palette-based surface.

DDERR_NOVSYNCHW
The operation cannot be carried out because there is no hardware support for vertical blank synchronized operations.

DDERR_NOZBUFFERHW
The operation to create a z-buffer in display memory or to perform a blit using a z-buffer cannot be carried out because there is no hardware support for z-buffers.

DDERR_NOZOVERLAYHW
The overlay surfaces cannot be z-layered based on the z-order because the hardware does not support z-ordering of overlays.

DDERR_OUTOFCAPS
The hardware needed for the requested operation has already been allocated.

DDERR_OUTOFMEMORY
DirectDraw does not have enough memory to perform the operation.

DDERR_OUTOFVIDEOMEMORY
DirectDraw does not have enough display memory to perform the operation.

DDERR_OVERLAPPINGRECTS
Operation could not be carried out because the source and destination rectangles are on the same surface and overlap each other.

DDERR_OVERLAYCANTCLIP
The hardware does not support clipped overlays.

DDERR_OVERLAYCOLORKEYONLYONEACTIVE
An attempt was made to have more than one color key active on an overlay.

DDERR_OVERLAYNOTVISIBLE

The **IDirectDrawSurface4::GetOverlayPosition** method is called on a hidden overlay.

DDERR_PALETTEBUSY
Access to this palette is refused because the palette is locked by another thread.

DDERR_PRIMARYSURFACEALREADYEXISTS
This process has already created a primary surface.

DDERR_REGIONTOOSMALL
The region passed to the **IDirectDrawClipper::GetClipList** method is too small.

DDERR_SURFACEALREADYATTACHED
An attempt was made to attach a surface to another surface to which it is already attached.

DDERR_SURFACEALREADYDEPENDENT
An attempt was made to make a surface a dependency of another surface to which it is already dependent.

DDERR_SURFACEBUSY
Access to the surface is refused because the surface is locked by another thread.

DDERR_SURFACEISOBSCURED
Access to the surface is refused because the surface is obscured.

DDERR_SURFACELOST
Access to the surface is refused because the surface memory is gone. Call the **IDirectDrawSurface4::Restore** method on this surface to restore the memory associated with it.

DDERR_SURFACENOTATTACHED
The requested surface is not attached.

DDERR_TOOBIGHEIGHT
The height requested by DirectDraw is too large.

DDERR_TOOBIGSIZE
The size requested by DirectDraw is too large. However, the individual height and width are valid sizes.

DDERR_TOOBIGWIDTH
The width requested by DirectDraw is too large.

DDERR_UNSUPPORTED
The operation is not supported.

DDERR_UNSUPPORTEDFORMAT
The pixel format requested is not supported by DirectDraw.

DDERR_UNSUPPORTEDMASK
The bitmask in the pixel format requested is not supported by DirectDraw.

DDERR_UNSUPPORTEDMODE
The display is currently in an unsupported mode.

DDERR_VERTICALBLANKINPROGRESS

A vertical blank is in progress.

DDERR_VIDEONOTACTIVE
The video port is not active.

DDERR_WASSTILLDRAWING
The previous blit operation that is transferring information to or from this surface is incomplete.

DDERR_WRONGMODE
This surface cannot be restored because it was created in a different mode.

DDERR_XALIGN
The provided rectangle was not horizontally aligned on a required boundary.

Pixel Format Masks

[This is preliminary documentation and subject to change.]

This section contains information about the pixel formats supported by the hardware-emulation layer (HEL). The following topics are discussed:

- Texture Map Formats
- Off-Screen Surface Formats

Texture Map Formats

[This is preliminary documentation and subject to change.]

A wide range of texture pixel formats are supported by the HEL. The following table shows these formats. The Masks column contains the red, green, blue, and alpha masks for each set of pixel format flags and bit depths.

Pixel format flags	Bit depth	Masks
DDPF_RGB | DDPF_PALETTEINDEXED1	1	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED1 | DDPF_PALETTEINDEXEDTO8	1	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB |	2	R: 0x00000000

DDPF_PALETTEINDEXED2		G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED2 | DDPF_PALETTEINDEXEDTO8	2	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED4	4	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED4 | DDPF_PALETTEINDEXEDTO8	4	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED8	8	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB	8	R: 0x000000E0 G: 0x0000001C B: 0x00000003 A: 0x00000000
DDPF_RGB | DDPF_ALPHAPIXELS	16	R: 0x00000F00 G: 0x000000F0

		B: 0x0000000F A: 0x0000F000
DDPF_RGB	16	R: 0x0000F800 G: 0x000007E0 B: 0x0000001F A: 0x00000000
DDPF_RGB	16	R: 0x0000001F G: 0x000007E0 B: 0x0000F800 A: 0x00000000
DDPF_RGB	16	R: 0x00007C00 G: 0x000003E0 B: 0x0000001F A: 0x00000000
DDPF_RGB | DDPF_ALPHAPIXELS	16	R: 0x00007C00 G: 0x000003E0 B: 0x0000001F A: 0x00008000
DDPF_RGB	24	R: 0x00FF0000 G: 0x0000FF00 B: 0x000000FF A: 0x00000000
DDPF_RGB	24	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000

		A: 0x00000000
DDPF_RGB	32	R: 0x00FF0000 G: 0x0000FF00 B: 0x000000FF A: 0x00000000
DDPF_RGB	32	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000 A: 0x00000000
DDPF_RGB | DDPF_ALPHAPIXELS	32	R: 0x00FF0000 G: 0x0000FF00 B: 0x000000FF A: 0xFF000000
DDPF_RGB | DDPF_ALPHAPIXELS	32	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000 A: 0xFF000000

The HEL can create these formats in system memory. The DirectDraw device driver for a 3-D–accelerated display card may create textures of other formats in display memory. Such a driver exports the DDSCAPS_TEXTURE flag to indicate that it can create textures.

Off-Screen Surface Formats

[This is preliminary documentation and subject to change.]

The following table shows the pixel formats for off-screen plain surfaces supported by the DirectX® 5 HEL. The Masks column contains the red, green, blue, and alpha masks for each set of pixel format flags and bit depths.

Pixel format flags	Bit depth	Masks
DDPF_RGB |	1	R: 0x00000000

DDPF_PALETTEINDEXED1		G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED2	2	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED4	4	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED8	8	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB	16	R: 0x0000F800 G: 0x000007E0 B: 0x0000001F A: 0x00000000
DDPF_RGB	16	R: 0x00007C00 G: 0x000003E0 B: 0x0000001F A: 0x00000000
DDPF_RGB	24	R: 0x00FF0000

		G: 0x0000FF00 B: 0x000000FF A: 0x00000000
DDPF_RGB	24	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000 A: 0x00000000
DDPF_RGB	32	R: 0x00FF0000 G: 0x0000FF00 B: 0x000000FF A: 0x00000000
DDPF_RGB	32	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000 A: 0x00000000
DDPF_RGB | DDPF_ZPIXELS	32	R: 0x0000F800 G: 0x000007E0 B: 0x0000001F Z: 0xFFFF0000
DDPF_RGB | DDPF_ZPIXELS	32	R: 0x00007C00 G: 0x000003E0 B: 0x0000001F Z: 0xFFFF0000

In addition to supporting a wide range of off-screen surface formats, the HEL also supports surfaces intended for use by Direct3D, or other 3-D renderers.

Four Character Codes (FOURCC)

[This is preliminary documentation and subject to change.]

DirectDraw utilizes a special set of codes that are four characters in length. These codes, called four character codes or FOURCCs, are stored in file headers of files containing multimedia data such as bitmap images, sound, or video. FOURCCs describe the software technology that was used to produce multimedia data. By implication, they also describe the format of the data itself.

DirectDraw applications use FOURCCs for image color and format conversion. If an application calls the **IDirectDrawSurface4::GetPixelFormat** method to request the pixel format of a surface whose format is not RGB, the **dwFourCC** member of the **DDPIXELFORMAT** structure identifies the FOURCC when the method returns. For more information, see Converting Color and Format.

In addition, the **biCompression** member of the **BITMAPINFOHEADER** structure can be set to a FOURCC to indicate the codec used to compress or decompress an image.

FOURCCs are registered with Microsoft by the vendors of the respective multimedia software technologies. Some common FOURCCs appear in the following list.

FOURCC	Company	Technology Name
AUR2	AuraVision Corporation	AuraVision Aura 2: YUV 422
AURA	AuraVision Corporation	AuraVision Aura 1: YUV 411
CHAM	Winnov, Inc.	MM_WINNOV_CAVIARA_CHAMPAGNE
CVID	Supermac	Cinepak by Supermac
CYUV	Creative Labs, Inc	Creative Labs YUV
DXT1	Microsoft Corporation	DirectX Texture Compression format 1
DXT2	Microsoft Corporation	DirectX Texture Compression format 2
DXT3	Microsoft Corporation	DirectX Texture Compression format 3
DXT4	Microsoft Corporation	DirectX Texture Compression format 4
DXT5	Microsoft Corporation	DirectX Texture Compression format 5
FVF1	Iterated Systems, Inc.	Fractal Video Frame
IF09	Intel Corporation	Intel Intermediate YUV9
IV31	Intel Corporation	Indeo 3.1
JPEG	Microsoft Corporation	Still Image JPEG DIB
MJPG	Microsoft Corporation	Motion JPEG Dib Format
MRLE	Microsoft Corporation	Run Length Encoding
MSVC	Microsoft Corporation	Video 1
PHMO	IBM Corporation	Photomotion
RT21	Intel Corporation	Indeo 2.1
ULTI	IBM Corporation	Ultimotion

V422	Vitec Multimedia	24 bit YUV 4:2:2
V655	Vitec Multimedia	16 bit YUV 4:2:2
VDCT	Vitec Multimedia	Video Maker Pro DIB
VIDS	Vitec Multimedia	YUV 4:2:2 CCIR 601 for V422
YU92	Intel Corporation	YUV
YUV8	Winnov, Inc.	MM_WINNOV_CAVIAR_YUV8
YUV9	Intel Corporation	YUV9
YUYV	Canopus, Co., Ltd.	BI_YUYV, Canopus
ZPEG	Metheus	Video Zipper

DirectDraw Visual Basic Reference

[This is preliminary documentation and subject to change.]

This section contains reference information for the API elements that DirectDraw provides. Reference material is divided into the following categories:

- Classes
- Types
- Enumerations
- Error Codes
- Pixel Format Masks
- Four Character Codes (FOURCC)

Classes

[This is preliminary documentation and subject to change.]

This section contains reference information about the classes used with the DirectDraw component. The following classes are covered:

- **DirectDraw4**
- **DirectDrawClipper**
- **DirectDrawColorControl**
- **DirectDrawEnum**
- **DirectDrawEnumModes**
- **DirectDrawEnumSurfaces**
- **DirectDrawGammaControl**
- **DirectDrawPalette**

- **DirectDrawSurface4**
- **DirectX7**

DirectDraw4

[#] [This is preliminary documentation and subject to change.]

Applications use the methods of the **DirectDraw4** class to create DirectDraw objects and work with system-level variables. This section is a reference to the methods of this class. For a conceptual overview, see The DirectDraw Object.

The methods of the **DirectDraw4** class can be organized into the following groups:

Cooperative levels	**SetCooperativeLevel** **TestCooperativeLevel**
Creating objects	**CreateClipper** **CreatePalette** **CreateSurface** **CreateSurfaceFromFile** **CreateSurfaceFromResource** **GetDirect3D** **LoadPaletteFromBitmap**
Device capabilities	**GetCaps**
Display modes	**GetDisplayMode** **GetDisplayModesEnum** **GetMonitorFrequency** **RestoreDisplayMode** **SetDisplayMode** **WaitForVerticalBlank**
Display status	**GetScanLine** **GetVerticalBlankStatus**
Miscellaneous	**GetAvailableTotalMem** **GetFourCCCodes** **GetFreeMem**

[#] IDH__dx_DirectDraw4_ddraw_vb

	GetNumFourCCCodes
Surface management	**DuplicateSurface** **FlipToGDISurface** **GetGDISurface** **GetSurfaceFromDC** **GetSurfacesEnum** **RestoreAllSurfaces**

The **DirectDraw4** class extends the features of previous versions of the class by offering methods enabling more flexible surface management than previous versions.

DirectDraw4.CreateClipper

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.CreateClipper** method creates a **DirectDrawClipper** object.

object.**CreateClipper(***flags* **As Long) As DirectDrawClipper**

object
Object expression that resolves to a **DirectDraw4** object.

flags
Argument that is currently not used and must be set to 0.

Return Values

If the method succeeds, a **DirectDrawClipper** object is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCOOPERATIVELEVELSET
DDERR_OUTOFMEMORY

For information on trapping errors, see the Visual Basic Error Trapping topic.

[#] IDH__dx_DirectDraw4.CreateClipper_ddraw_vb

Remarks

The DirectDrawClipper object can be attached to a DirectDrawSurface and used during **DirectDrawSurface4.Blt** and **DirectDrawSurface4.UpdateOverlay** operations.

See Also

DirectDrawSurface4.GetClipper, **DirectDrawSurface4.SetClipper**

DirectDraw4.CreatePalette

[This is preliminary documentation and subject to change.]

The **DirectDraw4.CreatePalette** method creates a **DirectDrawPalette** object for this DirectDraw object.

```
object.CreatePalette( _
    flags As CONST_DDPCAPSFLAGS, _
    pe() As PALETTEENTRY) As DirectDrawPalette
```

object
Object expression that resolves to a **DirectDraw4** object.

flags
One or more of the constants of the **CONST_DDPCAPSFLAGS** enumeration:

DDPCAPS_1BIT
Indicates that the index is 1 bit. There are two entries in the color table.

DDPCAPS_2BIT
Indicates that the index is 2 bits. There are four entries in the color table.

DDPCAPS_4BIT
Indicates that the index is 4 bits. There are 16 entries in the color table.

DDPCAPS_8BIT
Indicates that the index is 8 bits. There are 256 entries in the color table.

DDPCAPS_8BITENTRIES
Indicates that the index refers to an 8-bit color index. This flag is valid only when used with the DDPCAPS_1BIT, DDPCAPS_2BIT, or DDPCAPS_4BIT flag, and when the target surface is in 8 bpp. Each color entry is 1 byte long and is an index to a destination surface's 8-bpp palette.

DDPCAPS_ALPHA
Indicates that the **flags** member of the associated **PALETTEENTRY** type is to be interpreted as a single 8-bit alpha value. A palette created with this flag can only be attached to a texture (a surface created with the DDSCAPS_TEXTURE capability flag).

DDPCAPS_ALLOW256
Indicates that this palette can have all 256 entries defined.

IDH__dx_DirectDraw4.CreatePalette_ddraw_vb

DDPCAPS_INITIALIZE
This flag is obsolete and ignored by DirectDraw.

DDPCAPS_PRIMARYSURFACE
This palette is attached to the primary surface. Changing this palette's color table immediately affects the display unless DDPSETPAL_VSYNC is specified and supported.

DDPCAPS_PRIMARYSURFACELEFT
This palette is the one attached to the left eye primary surface. Changing this palette's color table immediately affects the left eye display unless DDPSETPAL_VSYNC is specified and supported.

DDPCAPS_VSYNC
This palette can have modifications to it synced with the monitors refresh rate.

pe()
An array of 2, 4, 16, or 256 **PALETTEENTRY** types that will initialize this **DirectDrawPalette** object.

Return Values

If the method succeeds, a **DirectDrawPalette** object is returned.

Error Codes

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCOOPERATIVELEVELSET
DDERR_OUTOFMEMORY
DDERR_UNSUPPORTED

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDraw4.CreateSurface

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.CreateSurface** method creates a **DirectDrawSurface4** object for this DirectDraw object.

object.**CreateSurface(_**
dd **As DDSURFACEDESC2) As DirectDrawSurface4**

object
Object expression that resolves to a **DirectDraw4** object.

[#] IDH__dx_DirectDraw4.CreateSurface_ddraw_vb

dd

A **DDSURFACEDESC2** type that describes the requested surface. A **DDSCAPS2** type is a member of **DDSURFACEDESC2**.

Return Values

If the method succeeds, a **DirectDrawSurface4** object is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INCOMPATIBLEPRIMARY
DDERR_INVALIDCAPS
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDPIXELFORMAT
DDERR_NOALPHAHW
DDERR_NOCOOPERATIVELEVELSET
DDERR_NODIRECTDRAWHW
DDERR_NOEMULATION
DDERR_NOEXCLUSIVEMODE
DDERR_NOFLIPHW
DDERR_NOMIPMAPHW
DDERR_NOOVERLAYHW
DDERR_NOZBUFFERHW
DDERR_OUTOFMEMORY
DDERR_OUTOFVIDEOMEMORY
DDERR_PRIMARYSURFACEALREADYEXISTS
DDERR_UNSUPPORTEDMODE

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDraw4.CreateSurfaceFromFile

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.CreateSurfaceFromFile** method creates a **DirectDrawSurface4** object for this DirectDraw object and attaches the specified bitmap image to the **DirectDrawSurface** object.

[#] IDH__dx_DirectDraw4.CreateSurfaceFromFile_ddraw_vb

object.**CreateSurfaceFromFile(_**
file **As String**, **_**
dd **As DDSURFACEDESC2) As DirectDrawSurface4**

object
Object expression that resolves to a **DirectDraw4** object.

file
Name of the bitmap image to load onto the surface that is created.

dd
A **DDSURFACEDESC2** type that describes the requested surface. A **DDSCAPS2** type is a member of **DDSURFACEDESC2**.

Return Values

If the method succeeds, a **DirectDrawSurface4** object is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INCOMPATIBLEPRIMARY
DDERR_INVALIDCAPS
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDPIXELFORMAT
DDERR_NOALPHAHW
DDERR_NOCOOPERATIVELEVELSET
DDERR_NODIRECTDRAWHW
DDERR_NOEMULATION
DDERR_NOEXCLUSIVEMODE
DDERR_NOFLIPHW
DDERR_NOMIPMAPHW
DDERR_NOOVERLAYHW
DDERR_NOZBUFFERHW
DDERR_OUTOFMEMORY
DDERR_OUTOFVIDEOMEMORY
DDERR_PRIMARYSURFACEALREADYEXISTS
DDERR_UNSUPPORTEDMODE

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDraw4.CreateSurfaceFromResource

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.CreateSurfaceFromResource** method creates a **DirectDrawSurface4** object for this DirectDraw object and attaches the specified resource to the **DirectDrawSurface** object.

```
object.CreateSurfaceFromResource( _
    file As String, _
    resName As String, _
    ddsd As DDSURFACEDESC2) As DirectDrawSurface4
```

object
Object expression that resolves to a **DirectDraw4** object.

file
Filename of the resource that is loaded onto the surface that is created. If the resource is part of the executable, specifying an empty string for this argument will locate the resource. This argument can also be the name of an OCX where the resource is located.

resName
Name of the resource that is loaded onto the surface that is created.

ddsd
A **DDSURFACEDESC2** type that describes the requested surface. A **DDSCAPS2** type is a member of **DDSURFACEDESC2**.

Return Values

If the method succeeds, a **DirectDrawSurface4** object is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INCOMPATIBLEPRIMARY
DDERR_INVALIDCAPS
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDPIXELFORMAT
DDERR_NOALPHAHW
DDERR_NOCOOPERATIVELEVELSET
DDERR_NODIRECTDRAWHW

[#] IDH__dx_DirectDraw4.CreateSurfaceFromResource_ddraw_vb

DDERR_NOEMULATION
DDERR_NOEXCLUSIVEMODE
DDERR_NOFLIPHW
DDERR_NOMIPMAPHW
DDERR_NOOVERLAYHW
DDERR_NOZBUFFERHW
DDERR_OUTOFMEMORY
DDERR_OUTOFVIDEOMEMORY
DDERR_PRIMARYSURFACEALREADYEXISTS
DDERR_UNSUPPORTEDMODE

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

If you supply an empty string as the module name and you run the application in the Visual Basic environment, then the resource won't be found. Passing an empty string will succeed only if you're using a stand-alone executable.

DirectDraw4.DuplicateSurface

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.DuplicateSurface** method duplicates a **DirectDrawSurface4** object.

object.**DuplicateSurface(_**
ddIn **As DirectDrawSurface4) As DirectDrawSurface4**

object
Object expression that resolves to a **DirectDraw4** object.

ddIn
A **DirectDrawSurface4** object that is the surface to be duplicated.

Return Values

If the method succeeds, a newly duplicated **DirectDrawSurface4** object is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_CANTDUPLICATE

[#] IDH__dx_DirectDraw4.DuplicateSurface_ddraw_vb

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY
DDERR_SURFACELOST

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method creates a new **DirectDrawSurface4** object that points to the same surface memory as an existing DirectDrawSurface object. This duplicate can be used just like the original object. The surface memory is released after the last object referencing it is released. A primary surface, 3-D surface, or implicitly created surface cannot be duplicated.

DirectDraw4.FlipToGDISurface

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.FlipToGDISurface** method makes the surface that GDI writes to the primary surface.

object.**FlipToGDISurface()**

object
Object expression that resolves to a **DirectDraw4** object.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTFOUND

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method can be called at the end of a page-flipping application to ensure that the display memory that GDI is writing to is visible to the user.

See Also

DirectDraw4.GetGDISurface

[#] IDH__dx_DirectDraw4.FlipToGDISurface_ddraw_vb

DirectDraw4.GetAvailableTotalMem

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.GetAvailableTotalMem** method retrieves the total amount of display memory available for a given type of surface.

```
object.GetAvailableTotalMem( _
    ddsCaps As DDSCAPS2) As Long
```

object
Object expression that resolves to a **DirectDraw4** object.

ddsCaps
A **DDSCAPS2** type that indicates the hardware capabilities of the proposed surface.

Return Value

If the method succeeds, the return value is the amount of total memory.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDCAPS
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NODIRECTDRAWHW

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method provides only a snapshot of the current display-memory state. The amount of free display memory is subject to change as surfaces are created and released. Therefore, you should use the free memory value only as an approximation. In addition, a particular display adapter card may make no distinction between two different memory types. For example, the adapter might use the same portion of display memory to store z-buffers and textures. So, allocating one type of surface (for example, a z-buffer) can affect the amount of display memory available for another type of surface (for example, textures). Therefore, it is best to first allocate an application's fixed resources (such as front and back buffers , and z-buffers)

[#] IDH__dx_DirectDraw4.GetAvailableTotalMem_ddraw_vb

before determining how much memory is available for dynamic use (such as texture mapping).

DirectDraw4.GetCaps

[This is preliminary documentation and subject to change.]

The **DirectDraw4.GetCaps** method fills in the capabilities of the device driver for the hardware and the hardware-emulation layer (HEL).

object.**GetCaps(***hwCaps* **As DDCAPS**, *helCaps* **As DDCAPS)**

object
Object expression that resolves to a **DirectDraw4** object.

hwCaps
A **DDCAPS** type that will be filled with the capabilities of the hardware, as reported by the device driver. Set this argument to NOTHING if device driver capabilities are not to be retrieved.

helCaps
A **DDCAPS** type that will be filled with the capabilities of the HEL. Set this argument to NOTHING if HEL capabilities are not to be retrieved.

Error Codes

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

You can only set one of the two arguments to NOTHING in Visual Basic to exclude it. If you set both to NOTHING the method will fail, returning DDERR_INVALIDPARAMS.

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDraw4.GetDirect3D

[This is preliminary documentation and subject to change.]

The **DirectDraw4.GetDirect3D** method creates a **Direct3D3** object.

*object.***GetDirect3D() As Direct3D3**

object
Object expression that resolves to a **DirectDraw4** object.

IDH__dx_DirectDraw4.GetCaps_ddraw_vb
IDH__dx_DirectDraw4.GetDirect3D_ddraw_vb

Return Value

If the method succeeds, a **Direct3D3** object is returned.

Error Codes

E_INVALIDINTERFACE
E_OUTOFMEMORY

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

The object returned by a successful function call must be assigned to a **Direct3D3** object variable. For example, in Visual Basic:

```
Dim Direct3D as Direct3D3
Set Direct3D = object.GetDirect3D()
```

DirectDraw4.GetDisplayMode

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.GetDisplayMode** method retrieves the current display mode.

***object*.GetDisplayMode(*surface* As DDSURFACEDESC2)**

object
Object expression that resolves to a **DirectDraw4** object.

surface
A **DDSURFACEDESC2** type that will be filled with a description of the surface display mode.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTEDMODE

For information on trapping errors, see the Visual Basic Error Trapping topic.

[#] IDH__dx_DirectDraw4.GetDisplayMode_ddraw_vb

Remarks

An application should not save the information returned by this method to restore the display mode on clean-up. The application should use the **DirectDraw4.RestoreDisplayMode** method to restore the mode on clean-up, thereby avoiding mode-setting conflicts that could arise in a multiprocess environment.

See Also

DirectDraw4.SetDisplayMode, **DirectDraw4.RestoreDisplayMode**, **DirectDraw4.GetDisplayModesEnum**

DirectDraw4.GetDisplayModesEnum

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.GetDisplayModesEnum** method returns a **DirectDrawEnumModes** object filled with display mode information.

object.**GetDisplayModesEnum(_**
flags **As CONST_DDEDMFLAGS**, **_**
ddsd **As DDSURFACEDESC2) As DirectDrawEnumModes**

object
Object expression that resolves to a **DirectDraw4** object.

flags
One of the following constants of the **CONST_DDEDMFLAGS** enumeration:

DDEDM_REFRESHRATES
Enumerates modes with different refresh rates. This guarantees that a particular mode will be enumerated only once. This flag specifies whether the refresh rate is taken into account when determining if a mode is unique.

DDEDM_STANDARDVGAMODES
Enumerates Mode 13 in addition to the 320x200x8 Mode X mode.

ddsd
A **DDSURFACEDESC2** type that will be filled with a description of the surface display mode.

Return Value

If the method succeeds, a **DirectDrawEnumModes** object is returned which you can then query for a description of the display modes.

[#] IDH__dx_DirectDraw4.GetDisplayModesEnum_ddraw_vb

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
E_OUTOFMEMORY

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

An application should not save the information returned by this method to restore the display mode on clean-up. The application should use the **DirectDraw4.RestoreDisplayMode** method to restore the mode on clean-up, thereby avoiding mode-setting conflicts that could arise in a multiprocess environment.

See Also

DirectDraw4.SetDisplayMode, **DirectDraw4.RestoreDisplayMode**, **DirectDraw4.GetDisplayModesEnum**

DirectDraw4.GetFourCCCodes

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.GetFourCCCodes** method retrieves the FOURCC codes supported by the DirectDraw object.

object.**GetFourCCCodes(***ccCodes()* **As Long)**

object
Object expression that resolves to a **DirectDraw4** object.

ccCodes
An array of variables that will be filled with the Four Character Codes(FOURCC) supported by this DirectDraw object.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

[#] IDH__dx_DirectDraw4.GetFourCCCodes_ddraw_vb

Remarks

To retrieve the number of codes supported by a **DirectDraw4** object, use **DirectDraw4.GetNumFourCCCodes**.

DirectDraw4.GetFreeMem

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.GetFreeMem** method retrieves the total amount of display memory currenty free.

object.**GetFreeMem(***ddsCaps* **As DDSCAPS2) As Long**

object
Object expression that resolves to a **DirectDraw4** object.

ddsCaps
A **DDSCAPS2** type that indicates the hardware capabilities of the proposed surface.

Return Value

The amount of total memory.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDCAPS
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NODIRECTDRAWHW

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method provides only a snapshot of the current display-memory state. The amount of free display memory is subject to change as surfaces are created and released. Therefore, you should use the free memory value only as an approximation. In addition, a particular display adapter card may make no distinction between two different memory types. For example, the adapter might use the same portion of display memory to store z-buffers and textures. So, allocating one type of surface (for example, a z-buffer) can affect the amount of display memory available for another type of surface (for example, textures). Therefore, it is best to first allocate an application's fixed resources (such as front and back buffers , and z-buffers)

[#] IDH__dx_DirectDraw4.GetFreeMem_ddraw_vb

before determining how much memory is available for dynamic use (such as texture mapping).

DirectDraw4.GetGDISurface

[This is preliminary documentation and subject to change.]

The **DirectDraw4.GetGDISurface** method retrieves the DirectDrawSurface object that currently represents the surface memory that GDI is treating as the primary surface.

object.**GetGDISurface() As DirectDrawSurface4**

object
Object expression that resolves to a **DirectDraw4** object.

Return Value

If the method succeeds, a **DirectDrawSurface4** is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTFOUND

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

DirectDraw4.FlipToGDISurface

DirectDraw4.GetMonitorFrequency

[This is preliminary documentation and subject to change.]

The **DirectDraw4.GetMonitorFrequency** method retrieves the frequency of the monitor being driven by the DirectDraw object.

object.**GetMonitorFrequency() As Long**

object
Object expression that resolves to a **DirectDraw4** object.

IDH__dx_DirectDraw4.GetGDISurface_ddraw_vb
IDH__dx_DirectDraw4.GetMonitorFrequency_ddraw_vb

Return Values

The monitor frequency, reported in Hz.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDraw4.GetNumFourCCCodes

[This is preliminary documentation and subject to change.]

The **DirectDraw4.GetNumFourCCCodes** method retrieves the number of FOURCC codes supported by the **DirectDraw** object.

object.**GetFourCCCodes() As Long**

object
Object expression that resolves to a **DirectDraw4** object.

Return Value

The number of supported Four Character Codes(FOURCC).

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method is typically called before calling **DirectDraw4.GetFourCCCodes**.

DirectDraw4.GetScanLine

[This is preliminary documentation and subject to change.]

IDH__dx_DirectDraw4.GetNumFourCCCodes_ddraw_vb
IDH__dx_DirectDraw4.GetScanLine_ddraw_vb

The **DirectDraw4.GetScanLine** method retrieves the scan line that is currently being drawn on the monitor.

object.**GetScanLine(***lines* **As Long) As Long**

object
Object expression that resolves to a **DirectDraw4** object.

lines
The current scan line.

Return Value

If the method succeeds, the return value is DD_OK, indicating that the calling application can continue executing.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED
DDERR_VERTICALBLANKINPROGRESS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Scan lines are reported as zero-based integers. The returned scan line value is between 0 and *n*, where scan line 0 is the first visible scan line on the screen and *n* is the last visible scan line, plus any scan lines that occur during the vertical blank period. So, in a case where an application is running at 640×480, and there are 12 scan lines during vblank, the values returned by this method will range from 0 to 491.

See Also

DirectDraw4.GetVerticalBlankStatus, **DirectDraw4.WaitForVerticalBlank**

DirectDraw4.GetSurfaceFromDC

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.GetSurfaceFromDC** method retrieves the **DirectDrawSurface4** object for a surface based on its GDI device context handle.

[#] IDH__dx_DirectDraw4.GetSurfaceFromDC_ddraw_vb

object.**GetSurfaceFromDC(***hdc* **As Long) As DirectDrawSurface4**

object
Object expression that resolves to a **DirectDraw4** object.

hdc
The handle to a display device context.

Return Value

If the method succeeds, a **DirectDrawSurface4** is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_GENERIC
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY
DDERR_NOTFOUND

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method will succeed only for device context handles that identify surfaces already associated with the DirectDraw object.

See Also

Surfaces and Device Contexts

DirectDraw4.GetSurfacesEnum

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.GetSurfacesEnum** method returns a **DirectDrawEnumSurfaces** object which is used to enumerate the attached surfaces of the DirectDraw4 object.

object.**GetSurfacesEnum(_**
flags **As CONST_DDENUMSURFACESFLAGS, _**
desc **As DDSURFACEDESC2) As DirectDrawEnumSurfaces**

object
Object expression that resolves to a **DirectDraw4** object.

[#] IDH__dx_DirectDraw4.GetSurfacesEnum_ddraw_vb

flags

A **CONST_DDENUMSURFACESFLAGS** enumeration containing a combination of one search type flag and one matching flag. The search type flag determines how the method searches for matching surfaces; you can search for surfaces that can be created using the description in the *desc* parameter or you can search for existing surfaces that already match that description. The matching flag determines whether the method enumerates all surfaces, only those that match, or only those that don't match the description in the *desc* parameter.

Search type flags

DDENUMSURFACES_CANBECREATED
Enumerates the first surface that can be created and meets the search criterion. This flag can only be used with the DDENUMSURFACES_MATCH flag.

DDENUMSURFACES_DOESEXIST
Enumerates the already existing surfaces that meet the search criterion.

Matching flags

DDENUMSURFACES_ALL
Enumerates all of the surfaces that meet the search criterion. This flag can only be used with the DDENUMSURFACES_DOESEXIST search type flag.

DDENUMSURFACES_MATCH
Searches for any surface that matches the surface description.

DDENUMSURFACES_NOMATCH
Searches for any surface that does not match the surface desc

desc

A **DDSURFACEDESC2** type that defines the surface of interest. This parameter can be NOTHING if the flags parameter includes the DDENUMSURFACES_ALL flag.

Return Value

If the method succeeds, a **DirectDrawEnumSurfaces** object is returned which you can then query for a description of the attached surfaces.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

E_OUTOFMEMORY

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

If the DDENUMSURFACES_CANBECREATED flag is set, this method attempts to temporarily create a surface that meets the search criterion.

DirectDraw4.GetVerticalBlankStatus

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.GetVerticalBlankStatus** method retrieves the status of the vertical blank.

object.**GetVerticalBlankStatus() As Long**

object
Object expression that resolves to a **DirectDraw4** object.

Return Value

The status of the vertical blank. This argument is zero if a vertical blank is occurring, and non-zero otherwise.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

To synchronize with the vertical blank, use the **DirectDraw4.WaitForVerticalBlank** method.

See Also

DirectDraw4.GetScanLine, **DirectDraw4.WaitForVerticalBlank**

[#] IDH__dx_DirectDraw4.GetVerticalBlankStatus_ddraw_vb

DirectDraw4.LoadPaletteFromBitmap

[This is preliminary documentation and subject to change.]

The **DirectDraw4.LoadPaletteFromBitmap** method creates a **DirectDrawPalette** object based on the palette of the specified bitmap for this DirectDraw object.

object.**LoadPaletteFromBitmap(_**
bName **As String) As DirectDrawPalette**

object
Object expression that resolves to a **DirectDraw4** object.

bName
The name of the bitmap from which you want to load the palette.

Return Value

If the method succeeds, a **DirectDrawPalette** object is returned.

Error Codes

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCOOPERATIVELEVELSET
DDERR_OUTOFMEMORY
DDERR_UNSUPPORTED

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDraw4.RestoreAllSurfaces

[This is preliminary documentation and subject to change.]

The **DirectDraw4.RestoreAllSurfaces** method restores all the surfaces created for the DirectDraw object, in the order they were created.

object.**RestoreAllSurfaces()**

object
Object expression that resolves to a **DirectDraw4** object.

IDH__dx_DirectDraw4.LoadPaletteFromBitmap_ddraw_vb
IDH__dx_DirectDraw4.RestoreAllSurfaces_ddraw_vb

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method is provided for convenience. Effectively, this method calls the **DirectDrawSurface4.Restore** method for each surface created by this DirectDraw object.

See Also

DirectDrawSurface4.Restore, Losing and Restoring Surfaces

DirectDraw4.RestoreDisplayMode

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.RestoreDisplayMode** method resets the mode of the display device hardware for the primary surface to what it was before the **DirectDraw4.SetDisplayMode** method was called. Exclusive-level access is required to use this method.

object.**RestoreDisplayMode()**

object
Object expression that resolves to a **DirectDraw4** object.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_LOCKEDSURFACES
DDERR_NOEXCLUSIVEMODE

For information on trapping errors, see the Visual Basic Error Trapping topic.

[#] IDH__dx_DirectDraw4.RestoreDisplayMode_ddraw_vb

See Also

DirectDraw4.SetDisplayMode, **DirectDrawEnumModes**, **DirectDraw4.SetCooperativeLevel**

DirectDraw4.SetCooperativeLevel

[This is preliminary documentation and subject to change.]

The **DirectDraw4.SetCooperativeLevel** method determines the top-level behavior of the application.

```
object.SetCooperativeLevel( _
    hdl As Long, _
    flags As CONST_DDSCLFLAGS)
```

object
Object expression that resolves to a **DirectDraw4** object.

hdl
Argument specifying the window handle used for the application. Set to the calling application's top-level window handle (not a handle for any child windows created by the top-level window). This argument can be NOTHING when the DDSCL_NORMAL flag is specified in the *flags* argument.

flags
One or more of the following constants from the **CONST_DDSCLFLAGS** enumeration:

DDSCL_ALLOWMODEX
Allows the use of Mode X display modes. This flag can only be used if the DDSCL_EXCLUSIVE and DDSCL_FULLSCREEN flags are present.

DDSCL_ALLOWREBOOT
Allows CTRL+ALT+DELl to function while in exclusive (full-screen) mode.

DDSCL_CREATEDEVICEWINDOW
This flag is supported in Windows 98 and Windows 2000 only. Indicates that DirectDraw is to create and manage a default device window for this DirectDraw object. For more information, see Focus and Device Windows.

DDSCL_EXCLUSIVE
Requests the exclusive level. This flag must be used with the DDSCL_FULLSCREEN flag.

DDSCL_FULLSCREEN
Indicates that the exclusive-mode owner will be responsible for the entire primary surface. GDI can be ignored. This flag must be used with the DDSCL_EXCLUSIVE flag.

DDSCL_MULTITHREADED
Requests multithread-safe DirectDraw behavior. This causes Direct3D to take the global critical section more frequently.

IDH__dx_DirectDraw4.SetCooperativeLevel_ddraw_vb

DDSCL_NORMAL
Indicates that the application will function as a regular Windows application. This flag cannot be used with the DDSCL_ALLOWMODEX, DDSCL_EXCLUSIVE, or DDSCL_FULLSCREEN flags.

DDSCL_NOWINDOWCHANGES
Indicates that DirectDraw is not allowed to minimize or restore the application window on activation.

DDSCL_SETDEVICEWINDOW
This flag is supported in Windows 98 and Windows 2000 only. Indicates that the *hdl* argument is the window handle of the device window for this DirectDraw object. This flag cannot be used with the DDSCL_SETFOCUSWINDOW flag.

DDSCL_SETFOCUSWINDOW
This flag is supported in Windows 98 and Windows 2000 only. Indicates that the *hdl* argument is the window handle of the focus window for this DirectDraw object. This flag cannot be used with the DDSCL_SETDEVICEWINDOW flag.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_EXCLUSIVEMODEALREADYSET
DDERR_HWNDALREADYSET
DDERR_HWNDSUBCLASSED
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method must be called by the same thread that created the application window.

An application must set either the DDSCL_EXCLUSIVE or DDSCL_NORMAL flag.

The DDSCL_EXCLUSIVE flag must be set to call functions that can have drastic performance consequences for other applications. For more information, see Cooperative Levels.

See Also

DirectDraw4.SetDisplayMode, **DirectDraw4.GetDisplayModesEnum**, Mode X and Mode 13 Display Modes, Focus and Device Windows.

DirectDraw4.SetDisplayMode

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.SetDisplayMode** method sets the mode of the display-device hardware.

```
object.SetDisplayMode( _
    w As Long, _
    h As Long, _
    bpp As Long, _
    ref As Long, _
    mode As CONST_DDSDMFLAGS)
```

object
: **Object** expression that resolves to a **DirectDraw4** object.

w and *h*
: The width and height of the new mode.

bpp
: The bits per pixel (bpp) of the new mode.

ref
: The refresh rate of the new mode. Set this value to 0 to request the default refresh rate for the driver.

flags
: One of the constants from the **CONST_DDSDMFLAGS** enumeration describing additional options. Currently, the only valid flag is DDSDM_STANDARDVGAMODE, which causes the method to set Mode 13 instead of Mode X 320x200x8 mode. If you are setting another resolution, bit depth, or a Mode X mode, do not use this flag and set the argument to 0.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_GENERIC
DDERR_INVALIDMODE
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_LOCKEDSURFACES

[#] IDH__dx_DirectDraw4.SetDisplayMode_ddraw_vb

DDERR_NOEXCLUSIVEMODE
DDERR_SURFACEBUSY
DDERR_UNSUPPORTED
DDERR_UNSUPPORTEDMODE
DDERR_WASSTILLDRAWING

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method must be called by the same thread that created the application window.

If another application changes the display mode, the primary surface will be lost and will return DDERR_SURFACELOST until it is recreated to match the new display mode.

See Also

DirectDraw4.RestoreDisplayMode, **DirectDraw4.GetDisplayModesEnum**, **DirectDraw4.SetCooperativeLevel**, Setting Display Modes, Restoring Display Modes

DirectDraw4.TestCooperativeLevel

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.TestCooperativeLevel** method reports the current cooperative-level status of the DirectDraw device for a windowed or full-screen application.

object.**TestCooperativeLevel() As Long**

object
Object expression that resolves to a **DirectDraw4** object.

Return Values

If the method succeeds, the return value is DD_OK, indicating that the calling application can continue executing.

Error Codes

If the method fails or if the DD_OK was not returned, the error code may be one of the following values (see remarks):

DDERR_INVALIDOBJECT
DDERR_EXCLUSIVEMODEALREADYSET

[#] IDH__dx_DirectDraw4.TestCooperativeLevel_ddraw_vb

DDERR_NOEXCLUSIVEMODE
DD_OK
DDERR_WRONGMODE

Remarks

This method is particularly useful to applications that use the WM_ACTIVATEAPP and WM_DISPLAYCHANGE system messages as a notification to restore surfaces or re-create DirectDraw objects. A zero for a return value always indicates that the application can continue execution without restoring or re-creating surfaces, but the failure codes are interpreted differently depending on the cooperative-level that the application uses. For more information, see Testing Cooperative Levels.

DirectDraw4.WaitForVerticalBlank

[#] [This is preliminary documentation and subject to change.]

The **DirectDraw4.WaitForVerticalBlank** method helps the application synchronize itself with the vertical-blank interval.

object.**WaitForVerticalBlank(_**
flags **As CONST_DDWAITVBFLAGS, _**
handle **As Long) As Long**

object
Object expression that resolves to a **DirectDraw4** object.

flags
One of the following constants of the **CONST_DDWAITVBFLAGS** enumeration specifying how long to wait for the vertical blank.

DDWAITVB_BLOCKBEGIN
Returns when the vertical-blank interval begins.

DDWAITVB_BLOCKBEGINEVENT
Triggers an event when the vertical blank begins. This value is not currently supported.

DDWAITVB_BLOCKEND
Returns when the vertical-blank interval ends and the display begins.

handle
The handle of the event to be triggered when the vertical blank begins. This argument is not currently used.

Return Values

If the method succeeds, the return value is DD_OK, indicating that the calling application can continue executing.

[#] IDH__dx_DirectDraw4.WaitForVerticalBlank_ddraw_vb

Error Codes

If the method fails or if the return value is not DD_OK, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED
DD_OK
DDERR_WASSTILLDRAWING

See Also

DirectDraw4.GetVerticalBlankStatus, **DirectDraw4.GetScanLine**

DirectDrawClipper

[This is preliminary documentation and subject to change.]

Applications use the methods of the **DirectDrawClipper** class to manage clip lists. This section is a reference to the methods of this class. For a conceptual overview, see Clippers.

The methods of the **DirectDrawClipper** class can be organized into the following groups:

Clip list	**GetClipList**
	GetClipListSize
	IsClipListChanged
	SetClipList
Handles	**GetHWnd**
	SetHWnd

DirectDrawClipper.GetClipList

[This is preliminary documentation and subject to change.]

The **DirectDrawClipper.GetClipList** method retrieves a copy of the clip list associated with a DirectDrawClipper object.

object.**GetClipList(***rects()* **As RECT)**

object

IDH__dx_DirectDrawClipper_ddraw_vb
IDH__dx_DirectDrawClipper.GetClipList_ddraw_vb

Object expression that resolves to a **DirectDrawClipper** object.

rects()
An array of **RECT** types is filled with the clip list.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_GENERIC
DDERR_INVALIDCLIPLIST
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCLIPLIST
DDERR_REGIONTOOSMALL

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

DirectDrawClipper.SetClipList

DirectDrawClipper.GetClipListSize

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawClipper.GetClipListSize** method retrieves the size of the clip list associated with a DirectDrawClipper object.

object.**GetClipListSize() As Long**

object
Object expression that resolves to a **DirectDrawClipper** object.

Return Value

If the method succeeds, the size of the clip list, in bytes, is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_GENERIC
DDERR_INVALIDPARAMS
DDERR_NOCLIPLIST

[#] IDH__dx_DirectDrawClipper.GetClipListSize_ddraw_vb

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDrawClipper.GetHWnd

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawClipper.GetHWnd** method retrieves the window handle previously associated with this DirectDrawClipper object by the **DirectDrawClipper.SetHWnd** method.

object.**GetHWnd() As Long**

object
Object expression that resolves to a **DirectDrawClipper** object.

Return Value

If the method succeeds, the window handle is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

DirectDrawClipper.SetHWnd

DirectDrawClipper.IsClipListChanged

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawClipper.IsClipListChanged** method monitors the status of the clip list if a window handle is associated with a DirectDrawClipper object.

object.**IsClipListChanged() As Long**

object
Object expression that resolves to a **DirectDrawClipper** object.

IDH__dx_DirectDrawClipper.GetHWnd_ddraw_vb
IDH__dx_DirectDrawClipper.IsClipListChanged_ddraw_vb

Return Value

If the method succeeds, the status of the clip list is returned. The result is non-zero if the clip list has changed and zero if it hasn't.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDrawClipper.SetClipList

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawClipper.SetClipList** method sets or deletes the clip list used by the **DirectDrawSurface4.Blt** and **DirectDrawSurface4.UpdateOverlay** methods on surfaces to which the parent DirectDrawClipper object is attached.

```
object.SetClipList( _
    count As Long, _ ,
    rects() as RECT)
```

object
Object expression that resolves to a **DirectDrawClipper** object.

count
The number of **RECT** types in the rects() array.

rects()
An array of **RECT** types that describe the clip list.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_CLIPPERISUSINGHWND
DDERR_INVALIDCLIPLIST
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

For information on trapping errors, see the Visual Basic Error Trapping topic.

[#] IDH__dx_DirectDrawClipper.SetClipList_ddraw_vb

Remarks

The clip list cannot be set if a window handle is already associated with the DirectDrawClipper object. Note that the **DirectDrawSurface4.BltFast** method cannot clip.

See Also

DirectDrawClipper.GetClipList, **DirectDrawSurface4.Blt**, **DirectDrawSurface4.BltFast**, **DirectDrawSurface4.UpdateOverlay**

DirectDrawClipper.SetHWnd

[This is preliminary documentation and subject to change.]

The **DirectDrawClipper.SetHWnd** method sets the window handle that will obtain the clipping information.

object.**SetHWnd(***hdl* **As Long)**

object
Object expression that resolves to a **DirectDrawClipper** object.

hdl
The window handle that will obtain the clipping information.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDCLIPLIST
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

DirectDrawClipper.GetHWnd

DirectDrawColorControl

[This is preliminary documentation and subject to change.]

The **DirectDrawColorControl** class allows you to get and set color controls:

IDH__dx_DirectDrawClipper.SetHWnd_ddraw_vb
IDH__dx_DirectDrawColorControl_ddraw_vb

Color controls	**GetColorControls** **SetColorControls**

DirectDrawColorControl.GetColorControls

[This is preliminary documentation and subject to change.]

The **DirectDrawColorControl.GetColorControls** method returns the current color control settings associated with the specified overlay or primary surface. The **IFlags** member of the **DDCOLORCONTROL** type indicates which of the color control options are supported.

object.**GetColorControls(***colorControl* **As DDCOLORCONTROL)**

object
Object expression that resolves to a **DirectDrawColorControl** object.

colorControl
A **DDCOLORCONTROL** type that will receive the current control settings of the specified surface.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

DirectDrawColorControl.SetColorControls, Using Color Controls, Gamma and Color Controls

DirectDrawColorControl.SetColorControls

[This is preliminary documentation and subject to change.]

The **DirectDrawColorControl.SetColorControls** method sets the color control settings associated with the specified overlay or primary surface.

IDH__dx_DirectDrawColorControl.GetColorControls_ddraw_vb
IDH__dx_DirectDrawColorControl.SetColorControls_ddraw_vb

object.**SetColorControls(***colorControl* **As DDCOLORCONTROL)**

object
Object expression that resolves to a **DirectDrawColorControl** object.

colorControl
A **DDCOLORCONTROL** type containing the new values to be applied to the specified surface.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

DirectDrawColorControl.GetColorControls, Using Color Controls, Gamma and Color Controls

DirectDrawEnum

[This is preliminary documentation and subject to change.]

Applications use the methods of the **DirectDrawEnum** class to obtain information on video driver display adapters that are installed on the computer. This object is created and filled with data by the **DirectX7.GetDDEnum** method.

DirectDraw Enumeration	**GetCount**
	GetDescription
	GetGuid
	GetName

DirectDrawEnum.GetCount

[This is preliminary documentation and subject to change.]

The **DirectDrawEnum.GetCount** method returns the number of DirectDraw drivers installed on the system.

object.**GetCount() As Long**

IDH__dx_DirectDrawEnum_ddraw_vb
IDH__dx_DirectDrawEnum.GetCount_ddraw_vb

object
Object expression that resolves to a **DirectDrawEnum** object.

Return Value

The number of entries in the enumeration object.

Remarks

Each entry represents a DirectDraw driver description. To get individual driver descriptions use **DirectDrawEnum.GetDescription**, **DirectDrawEnum.GetGuid** and **DirectDrawEnum.GetName**.

DirectDrawEnum.GetDescription

[This is preliminary documentation and subject to change.]

The **DirectDrawEnum.GetDescription** method returns the driver description of the specified DirectDraw device.

object.**GetDescription(***index* **As Long) As String**

object
Object expression that resolves to a **DirectDrawEnum** object.
index
The particular DirectDraw device in the DirectDrawEnum object.

Return Value

The driver description of the device.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

DirectDrawEnum.GetGuid

[This is preliminary documentation and subject to change.]

The **DirectDrawEnum.GetGuid** returns the unique identifier of the specified DirectDraw device.

object.**GetGuid(***index* **As Long) As String**

object

IDH__dx_DirectDrawEnum.GetDescription_ddraw_vb
IDH__dx_DirectDrawEnum.GetGuid_ddraw_vb

Object expression that resolves to a **DirectDrawEnum** object.

index
The particular DirectDraw device in the DirectDrawEnum object.

Return Value

The unique identifier of the device.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

DirectDrawEnum.GetName

[This is preliminary documentation and subject to change.]

The **DirectDrawEnum.GetName** returns the driver name of the specified DirectDraw device.

object.**GetName(***index* **As Long) As String**

object
Object expression that resolves to a **DirectDrawEnum** object.

index
The particular DirectDraw device in the DirectDrawEnum object.

Return Value

The driver name of the device.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

DirectDrawEnumModes

[This is preliminary documentation and subject to change.]

Applications use the methods of the **DirectDrawEnumModes** class to enumerate the computer's available video modes. This object is created and filled with data by the **DirectDraw4.GetDisplayModesEnum** method.

DirectDraw Mode Enumeration	**GetCount**
	GetItem

IDH__dx_DirectDrawEnum.GetName_ddraw_vb
IDH__dx_DirectDrawEnumModes_ddraw_vb

DirectDrawEnumModes.GetCount

[This is preliminary documentation and subject to change.]

The **DirectDrawEnumModes.GetCount** method returns the number of available video modes.

object.**GetCount() As Long**

object
Object expression that resolves to a **DirectDrawEnumModes** object.

Return Value

The number of available video modes.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

Each entry represents a video mode. To get a description of an individual video mode use **DirectDrawEnumModes.GetItem**.

DirectDrawEnumModes.GetItem

[This is preliminary documentation and subject to change.]

The **DirectDrawEnumModes.GetItem** method returns a video mode description for the specified element in the enumeration object.

object.**GetItem(***index* **As Long,** *info* **As DDSURFACEDESC2)**

object
Object expression that resolves to a **DirectDrawEnumModes** object.

index
Number specifying which element of the array to be accessed. Each element is an available video mode.

info
A **DDSURFACEDESC2** type that will be filled with video mode information.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

IDH__dx_DirectDrawEnumModes.GetCount_ddraw_vb
IDH__dx_DirectDrawEnumModes.GetItem_ddraw_vb

Remarks

The number of video modes in the **DirectDrawEnumModes** object can be obtained from the **DirectDrawEnumModes.GetCount**.

DirectDrawEnumSurfaces

[This is preliminary documentation and subject to change.]

Applications use the methods of the **DirectDrawEnumSurfaces** class to enumerate all of the created surfaces. This object is created and filled with data by the **DirectDrawSurface4.GetAttachedSurfaceEnum** and **DirectDrawSurface4.GetOverlayZOrdersEnum** methods.

DirectDraw Surface Enumeration	**GetCount**
	GetItem

DirectDrawEnumSurfaces.GetCount

[This is preliminary documentation and subject to change.]

The **DirectDrawEnumSurfaces.GetCount** method returns the number of created surfaces.

object.**GetCount() As Long**

object
Object expression that resolves to a **DirectDrawEnumSurfaces** object.

Return Value

The number of created surfaces.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

Each entry represents created DirectDraw surfaces. To get a description of a surface use **DirectDrawEnumSurfaces.GetItem**.

IDH__dx_DirectDrawEnumSurfaces_ddraw_vb
IDH__dx_DirectDrawEnumSurfaces.GetCount_ddraw_vb

DirectDrawEnumSurfaces.GetItem

[This is preliminary documentation and subject to change.]

The **DirectDrawEnumSurfaces.GetItem** method returns a specific surface from the list of created surfaces of the **DirectDrawEnumSurfaces** object.

object.**GetItem(***index* **As Long) As DirectDrawSurface4**

object
Object expression that resolves to a **DirectDrawEnumSurfaces** object.

index
Number specifying which element of the array to be accessed. Each element represents a created surface.

Return Value

An **DirectDrawSurface4** object describing the surface is returned.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

To obtain the number of created surfaces call the **DirectDrawEnumSurfaces.GetCount** method.

DirectDrawGammaControl

[This is preliminary documentation and subject to change.]

Applications use the methods of the **DirectDrawGammaControl** class to adjust the red, green, and blue gamma ramp levels of the primary surface. This section is a reference to the methods of this class. This object is created with a call to the **DirectDrawSurface4.GetDirectDrawGammaControl**.

For a conceptual overview, see Gamma and Color Controls.

Gamma ramps	**GetGammaRamp**
	SetGammaRamp

IDH__dx_DirectDrawEnumSurfaces.GetItem_ddraw_vb
IDH__dx_DirectDrawGammaControl_ddraw_vb

DirectDrawGammaControl.GetGammaRamp

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawGammaControl.GetGammaRamp** method retrieves the red, green, and blue gamma ramps for the primary surface.

object.**GetGammaRamp(_**
flags **As CONST_DDSGRFLAGS, _**
gammaRamp **As DDGAMMARAMP)**

object
Object expression that resolves to a **DirectDrawGammaControl** object.

flags
One of the constants of the **CONST_DDSGRFLAGS** enumeration indicating if gamma calibration is desired. Set this argument DDSGR_CALIBRATE to request that the calibrator adjust the gamma ramp according to the physical properties of the display, making the result identical on all systems. If calibration is not needed, set this argument to 0.

gammaRamp
A **DDGAMMARAMP** type that will be filled with the current red, green, and blue gamma ramps. This type maps color values in the frame buffer to the color values that will be passed to the DAC (Digital-to-Analog Converter).

Error Codes

If the method fails, the error code may be one of the following:

DDERR_EXCEPTION
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

DirectDrawGammaControl.SetGammaRamp

DirectDrawGammaControl.SetGammaRamp

[#] [This is preliminary documentation and subject to change.]

IDH__dx_DirectDrawGammaControl.GetGammaRamp_ddraw_vb
IDH__dx_DirectDrawGammaControl.SetGammaRamp_ddraw_vb

The **DirectDrawGammaControl.SetGammaRamp** method sets the red, green, and blue gamma ramps for the primary surface.

object.**SetGammaRamp(_**
flags **As CONST_DDSGRFLAGS**, _
gammaRamp **As DDGAMMARAMP)**

object
Object expression that resolves to a **DirectDrawGammaControl** object.

flags
One of the constants of the **CONST_DDSGRFLAGS** enumeration indicating if gamma calibration is desired. Set this argument DDSGR_CALIBRATE to request that the calibrator adjust the gamma ramp according to the physical properties of the display, making the result identical on all systems. If calibration is not needed, set this argument to 0.

gammaRamp
A **DDGAMMARAMP** type that contains the new red, green, and blue gamma ramp entries. Each array maps color values in the frame buffer to the color values that will be passed to the DAC (Digital-to-Analog Converter).

Error Codes

If the method fails, the error code may be one of the following:

DDERR_EXCEPTION
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Not all systems support gamma calibration. To determine if gamma calibration is supported, call **DirectDraw4.GetCaps**, and examine the **lCaps2** member of the associated **DDCAPS** type after the method returns. If the DDCAPS_CANCALIBRATEGAMMA capability flag is present, then gamma calibration is supported.

Calibrating gamma ramps incurs some processing overhead, and should not be used frequently.

Including the DDSGR_CALIBRATE flag in the *flags* argument when running on systems that do not support gamma calibration will not cause this method to fail. The method succeeds, setting new gamma ramp values without calibration.

See Also

DirectDrawGammaControl.GetGammaRamp

DirectDrawPalette

[This is preliminary documentation and subject to change.]

Applications use the methods of the **DirectDrawPalette** class to create DirectDrawPalette objects and work with system-level variables. This section is a reference to the methods of this class. For a conceptual overview, see Palettes.

The methods of the **DirectDrawPalette** class can be organized into the following groups:

Palette capabilities	**GetCaps**
Palette entries	**GetEntries**
	SetEntries

DirectDrawPalette.GetCaps

[This is preliminary documentation and subject to change.]

The **DirectDrawPalette.GetCaps** method retrieves the capabilities of this palette object.

object.**GetCaps() As CONST_DDPCAPSFLAGS**

object
Object expression that resolves to a **DirectDrawPalette** object.

Return Value

One of the following constants of the **CONST_DDPCAPSFLAGS** enumeration indicating the capabilities of the palette object:

DDPCAPS_1BIT
DDPCAPS_2BIT
DDPCAPS_4BIT
DDPCAPS_8BIT
DDPCAPS_8BITENTRIES
DDPCAPS_ALPHA

IDH__dx_DirectDrawPalette_ddraw_vb
IDH__dx_DirectDrawPalette.GetCaps_ddraw_vb

DDPCAPS_ALLOW256
DDPCAPS_PRIMARYSURFACE
DDPCAPS_PRIMARYSURFACELEFT
DDPCAPS_VSYNC

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDrawPalette.GetEntries

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawPalette.GetEntries** method queries palette values from a DirectDrawPalette object.

```
object.GetEntries( _
    start As Long, _
    count As Long, _
    val() As PALETTEENTRY)
```

object
Object expression that resolves to a **DirectDrawPalette** object.

start
The start of the entries that should be retrieved sequentially.

count
The number of palette entries that can fit in the address specified in *val()*. The colors of each palette entry are returned in sequence, from the value of the *start* argument through the value of the *count* argument minus 1. (These arguments are set by **DirectDrawPalette.SetEntries**.)

val()
An array of variables of type **PALETTEENTRY**. The palette entries are 1 byte each if the DDPCAPS_8BITENTRIES flag was set in the flags argument of the **DirectDraw4.CreatePalette** and 4 bytes otherwise. Each field is a color description.

Error Codes

If the method fails, the error code may be one of the following:

[#] IDH__dx_DirectDrawPalette.GetEntries_ddraw_vb

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTPALETTIZED

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

DirectDrawPalette.SetEntries

DirectDrawPalette.SetEntries

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawPalette.SetEntries** method changes entries in a DirectDrawPalette object immediately.

```
object.SetEntries( _
    start As Long, _
    count As Long, _
    val() As PALETTEENTRY)
```

object
Object expression that resolves to a **DirectDrawPalette** object.

start
The first entry to be set.

count
The number of palette entries to be changed.

val()
An arrary of variables of type **PALETTEENTRY**. The palette entries are 1 byte each if the DDPCAPS_8BITENTRIES flag was set in the flags argument of the **DirectDraw4.CreatePalette** and 4 bytes otherwise. Each field is a color description.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOPALETTEATTACHED
DDERR_NOTPALETTIZED
DDERR_UNSUPPORTED

[#] IDH__dx_DirectDrawPalette.SetEntries_ddraw_vb

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

DirectDrawPalette.GetEntries, **DirectDrawSurface4.SetPalette**

DirectDrawSurface4

[#] [This is preliminary documentation and subject to change.]

Applications use the methods of the **DirectDrawSurface4** class to create DirectDrawSurface objects and work with system-level variables. This section is a reference to the methods of this class. For a conceptual overview, see Surfaces.

The methods of the **DirectDrawSurface4** class can be organized into the following groups:

Allocating memory	**IsLost**
	Restore
Attaching surfaces	**AddAttachedSurface**
	DeleteAttachedSurface
	GetAttachedSurface
	GetAttachedSurfaceEnum
Blitting	**Blt**
	BltColorFill
	BltFast
	BltFx
	BltToDC
	GetBltStatus
Color keying	**GetColorKey**
	SetColorKey
Device contexts	**GetDC**
	ReleaseDC
Drawing and Text	**DrawBox**
	DrawCircle

[#] IDH__dx_DirectDrawSurface4_ddraw_vb

	DrawEllipse
	DrawLine
	DrawRoundedBox
	DrawText
	GetDrawStyle
	GetDrawWidth
	GetFillColor
	GetFillStyle
	GetFontTransparency
	GetForeColor
	SetDrawStyle
	SetDrawWidth
	SetFillColor
	SetFillStyle
	SetFont
	SetFontTransparency
	SetForeColor
Flipping	**Flip**
	GetFlipStatus
Locking	**GetLockedPixel**
	GetLockedSurfaceBits
	Lock
	SetLockedPixel
	SetLockedSurfaceBits
	Unlock
Miscellaneous	**GetDirectDraw**
	GetDirectDrawColorControl
	GetDirectDrawGammaControl
	GetTexture
Overlays	**GetOverlayZOrdersEnum**
	UpdateOverlay
	UpdateOverlayZOrder
Surface capabilities	**GetCaps**

Surface clipper	**GetClipper** **SetClipper**
Surface characteristics	**ChangeUniquenessValue** **GetPixelFormat** **GetSurfaceDesc** **GetUniquenessValue**
Surface palettes	**GetPalette** **SetPalette**

DirectDrawSurface4.AddAttachedSurface

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.AddAttachedSurface** method attaches the specified surface to this surface.

object.**AddAttachedSurface(***ddS* **As DirectDrawSurface4)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

ddS
A **DirectDrawSurface4** object for the surface to be attached.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_CANNOTATTACHSURFACE
DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACEALREADYATTACHED
DDERR_SURFACELOST
DDERR_WASSTILLDRAWING

For information on trapping errors, see the Visual Basic Error Trapping topic.

[#] IDH__dx_DirectDrawSurface4.AddAttachedSurface_ddraw_vb

Remarks

You can explicitly unattach the surface by using the **DirectDrawSurface4.DeleteAttachedSurface** method. Unlike complex surfaces that you create with a single call to **DirectDraw4.CreateSurface**, surfaces attached with this method are not automatically released. It is the application's responsibility to release such surfaces.

Possible attachments include z-buffers, alpha channels, and back buffers. Some attachments automatically break other attachments. For example, the 3-D z-buffer can be attached only to one back buffer at a time. Attachment is not bidirectional, and a surface cannot be attached to itself. Emulated surfaces (in system memory) cannot be attached to nonemulated surfaces. Unless one surface is a texture map, the two attached surfaces must be the same size. A flipping surface cannot be attached to another flipping surface of the same type; however, attaching two surfaces of different types is allowed. For example, a flipping z-buffer can be attached to a regular flipping surface. If a nonflipping surface is attached to another nonflipping surface of the same type, the two surfaces will become a flipping chain. If a nonflipping surface is attached to a flipping surface, it becomes part of the existing flipping chain. Additional surfaces can be added to this chain, and each call of the **DirectDrawSurface4.Flip** method will advance one step through the surfaces.

See Also

DirectDrawSurface4.DeleteAttachedSurface,
DirectDrawSurface4.GetAttachedSurfaceEnum

DirectDrawSurface4.Blt

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.Blt** method performs a bit block transfer. This method does not support z-buffering or alpha blending (see alpha channel) during blit operations.

```
object.Blt( _
    destRect As RECT, _
    ddS As DirectDrawSurface4, _
    srcRect As RECT, _
    flags As CONST_DDBLTFLAGS) As Long
```

object
Object expression that resolves to a **DirectDrawSurface4** object.

destRect
A **RECT** type that defines the upper-left and lower-right points of the rectangle to blit to on the destination surface. If this argument is NOTHING, the entire destination surface will be used.

[#] IDH__dx_DirectDrawSurface4.Blt_ddraw_vb

ddS

A **DirectDrawSurface4** object for the DirectDrawSurface object that is the source of the blit.

srcRect

A **RECT** type that defines the upper-left and lower-right points of the rectangle to blit from on the source surface. If this argument is NOTHING, the entire source surface will be used.

flags

Combination of constants from the **CONST_DDBLTFLAGS** enumeration that determines the valid members of the associated **DDBLTFX** type, which specify color key information, or request special behavior from the method. The following flags are defined.

Validation flags

DDBLT_COLORFILL

Uses the **lFill** member of the **DDBLTFX** structure as the RGB color that fills the destination rectangle on the destination surface.

DDBLT_DDFX

Uses the **lDDFX** member of the **DDBLTFX** structure to specify the effects to use for this blit.

DDBLT_DDROPS

Uses the **lROP** member of the **DDBLTFX** structure to specify the raster operations (ROPS) that are not part of the Win32 API.

DDBLT_KEYDESTOVERRIDE

Uses the **ddckDestColorKey_high** and **ddckDestColorKey_low** members of the **DDBLTFX** structure as the color key for the destination surface.

DDBLT_KEYSRCOVERRIDE

Uses the **ddckSrcColorKey_high** and **ddckSrcColorKey_low** members of the **DDBLTFX** structure as the color key for the source surface.

DDBLT_ROP

Uses the **lROP** member of the **DDBLTFX** structure for the ROP for this blit. These ROPs are the same as those defined in the Win32 API.

DDBLT_ROTATIONANGLE

Uses the **lRotationAngle** member of the **DDBLTFX** structure as the rotation angle (specified in 1/100th of a degree) for the surface.

Color key flags

DDBLT_KEYDEST

Uses the color key associated with the destination surface.

DDBLT_KEYSRC

Uses the color key associated with the source surface.

Behavior flags

DDBLT_ASYNC

Performs this blit asynchronously through the FIFO in the order received. If no room is available in the FIFO hardware, the call fails.

DDBLT_WAIT

Postpones the DDERR_WASSTILLDRAWING return value if the blitter is busy, and returns as soon as the blit can be set up or another error occurs.

Obsolete and unsupported flags

All "DDBLT_ALPHA" flag values.

Obsolete.

All "DDBLT_ZBUFFER" flag values

This method does not currently support z-aware blit operations. None of the flags beginning with "DDBLT_ZBUFFER" are supported in this release of DirectX 6.0.

Return Values

The return value may be one of the following:

DDERR_GENERIC
DDERR_INVALIDCLIPLIST
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDRECT
DDERR_NOALPHAHW
DDERR_NOBLTHW
DDERR_NOCLIPLIST
DDERR_NODDROPSHW
DDERR_NOMIRRORHW
DDERR_NORASTEROPHW
DDERR_NOROTATIONHW
DDERR_NOSTRETCHHW
DDERR_NOZBUFFERHW
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method is capable of synchronous or asynchronous blits (the default behavior), either display memory to display memory, display memory to system memory, system memory to display memory, or system memory to system memory. The blits can be performed by using source color keys, and destination color keys. Arbitrary

stretching or shrinking will be performed if the source and destination rectangles are not the same size.

Typically, **DirectDrawSurface4.Blt** returns immediately with an error if the blitter is busy and the blit could not be set up. Specify the DDBLT_WAIT flag to request a synchronous blit. When you include the DDBLT_WAIT flag, the method waits until the blit can be set up or another error occurs before it returns.

Note that *dstRect* and *srcRect* arguments are defined so that the **right** and **bottom** members are exclusive—therefore, **right** minus **left** equals the width of the rectangle, not one less than the width.

DirectDrawSurface4.BltColorFill

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.BltColorFill** method performs a bit block transfer of a single color to the specified destination rectangle.

```
object.BltColorFill( _
    destRect As RECT, _
    fillvalue As Long) As Long
```

object
Object expression that resolves to a **DirectDrawSurface4** object.

destRect
A **RECT** type that defines the upper-left and lower-right points of the rectangle to blit to on the destination surface. If this argument is NOTHING, the entire destination surface will be used.

fillvalue
The color to blit.

Return Values

The return value may be one of the following:

DDERR_GENERIC
DDERR_INVALIDCLIPLIST
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDRECT
DDERR_NOALPHAHW
DDERR_NOBLTHW
DDERR_NOCLIPLIST

[#] IDH__dx_DirectDrawSurface4.BltColorFill_ddraw_vb

DDERR_NODDROPSHW
DDERR_NOMIRRORHW
DDERR_NORASTEROPHW
DDERR_NOROTATIONHW
DDERR_NOSTRETCHHW
DDERR_NOZBUFFERHW
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_UNSUPPORTED

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDrawSurface4.BltFast

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.BltFast** method performs a source copy blit or transparent blit by using a source color key or destination color key.

```
object.BltFast( _
    dx As Long, _
    dy As Long, _
    ddS As DirectDrawSurface4, _
    srcRect As RECT, _
    trans As CONST_DDBLTFASTFLAGS) As Long
```

object
Object expression that resolves to a **DirectDrawSurface4** object.

dx and *dy*
The x- and y-coordinates to blit to on the destination surface.

ddS
A **DirectDrawSurface4** object for the DirectDrawSurface object that is the source of the blit.

srcRect
A **RECT** type that defines the upper-left and lower-right points of the rectangle to blit from on the source surface.

trans
One of the constants from the **CONST_DDBLTFASTFLAGS** enumeration which identifies the type of transfer.

DDBLTFAST_DESTCOLORKEY
Specifies a transparent blit that uses the destination's color key.

DDBLTFAST_NOCOLORKEY
Specifies a normal copy blit with no transparency.

[#] IDH__dx_DirectDrawSurface4.BltFast_ddraw_vb

DDBLTFAST_SRCCOLORKEY
Specifies a transparent blit that uses the source's color key.

DDBLTFAST_WAIT
Postpones the DDERR_WASSTILLDRAWING message if the blitter is busy, and returns as soon as the blit can be set up or another error occurs.

Return Value

The value may be one of the following:

DDERR_EXCEPTION
DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDRECT
DDERR_NOBLTHW
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING

Remarks

This method always attempts an asynchronous blit if it is supported by the hardware.

This method works only on display memory surfaces and cannot clip when blitting. If you use this method on a surface with an attached clipper, the call will fail and the method will return DDERR_UNSUPPORTED.

The software implementation of **DirectDrawSurface4.BltFast** is 10 percent faster than the **DirectDrawSurface4.Blt** method. However, there is no speed difference between the two if display hardware is being used.

Typically, **DirectDrawSurface4.BltFast** returns immediately with an error if the blitter is busy and the blit cannot be set up. You can use the DDBLTFAST_WAIT flag, however, if you want this method to not return until either the blit can be set up or another error occurs.

DirectDrawSurface4.BltFx

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.BltFx** method performs a bit block transfer with additional blit effect behavior specified in the *BltFx* argument.

[#] IDH__dx_DirectDrawSurface4.BltFx_ddraw_vb

object.**BltFx(_**
destRect **As RECT**, _
ddS **As DirectDrawSurface4**, _
srcRect **As RECT**, _
flags **As CONST_DDBLTFLAGS**, _
BltFx **As DDBLTFX) As Long**

object
Object expression that resolves to a **DirectDrawSurface4** object.

destRect
A **RECT** type that defines the upper-left and lower-right points of the rectangle to blit to on the destination surface. If this argument is NOTHING, the entire destination surface will be used.

ddS
A **DirectDrawSurface4** object that is the source for the blit.

srcRect
A **RECT** type that defines the upper-left and lower-right points of the rectangle to blit from on the source surface. If this argument is NOTHING, the entire source surface will be used.

flags
Combination of constants of the **CONST_DDBLTFLAGS** enumeration that determines the valid members of the associated **DDBLTFX** type, specify color key information, or that request special behavior from the method. The following flags are defined.

Validation flags

DDBLT_COLORFILL
Uses the **lFillColor** member of the **DDBLTFX** type as the RGB color that fills the destination rectangle on the destination surface.

DDBLT_DDFX
Uses the **lDDFX** member of the **DDBLTFX** type to specify the effects to use for this blit.

DDBLT_DDROPS
Uses the **lROP** member of the **DDBLTFX** type to specify the raster operations (ROPS) that are not part of the Win32 API.

DDBLT_KEYDESTOVERRIDE
Uses the **ddckDestColorKey_high** and **ddckDestColorKey_low** members of the **DDBLTFX** type as the color key for the destination surface.

DDBLT_KEYSRCOVERRIDE
Uses the **ddckSrcColorKey_high** and **ddckSrcColorKey_low** member of the **DDBLTFX** type as the color key for the source surface.

DDBLT_ROP
Uses the **lROP** member of the **DDBLTFX** type for the ROP for this blit. These ROPs are the same as those defined in the Win32 API.

DDBLT_ROTATIONANGLE

Uses the **lRotationAngle** member of the **DDBLTFX** type as the rotation angle (specified in 1/100th of a degree) for the surface.

Color key flags

DDBLT_KEYDEST
Uses the color key associated with the destination surface.

DDBLT_KEYSRC
Uses the color key associated with the source surface.

Behavior flags

DDBLT_ASYNC
Performs this blit asynchronously through the FIFO in the order received. If no room is available in the FIFO hardware, the call fails.

DDBLT_WAIT
Postpones the DDERR_WASSTILLDRAWING return value if the blitter is busy, and returns as soon as the blit can be set up or another error occurs.

Obsolete and unsupported flags

All "DDBLT_ALPHA" flag values.
Obsolete.

All "DDBLT_ZBUFFER" flag values
This method does not currently support z-aware blit operations. None of the flags beginning with "DDBLT_ZBUFFER" are supported in this release of DirectX 6.0.

BltFx
A **DDBLTFX** type specifying additional blit effect operations to be performed.

Return Value

The value may be one of the following:

DDERR_GENERIC
DDERR_INVALIDCLIPLIST
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDRECT
DDERR_NOALPHAHW
DDERR_NOBLTHW
DDERR_NOCLIPLIST
DDERR_NODDROPSHW
DDERR_NOMIRRORHW
DDERR_NORASTEROPHW
DDERR_NOROTATIONHW
DDERR_NOSTRETCHHW

DDERR_NOZBUFFERHW
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_UNSUPPORTED

DirectDrawSurface4.BltToDC

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.BltToDC** method performs a bit block transfer to the specified device context.

```
object.BltToDC( _
    hdc As LONG, _
    srcRect As RECT, _
    destRect As RECT)
```

object
Object expression that resolves to a **DirectDrawSurface4** object.

hdc
Handle to a device context.

srcRect
A **RECT** type that defines the upper-left and lower-right points of the rectangle to blit from on the source surface. If this argument is NOTHING, the entire source surface will be used.

destRect
A **RECT** type that defines the upper-left and lower-right points of the rectangle to blit to on the destination surface. If this argument is NOTHING, the entire source surface will be used.

Return Value

The value may be one of the following:

DDERR_GENERIC
DDERR_INVALIDCLIPLIST
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDRECT
DDERR_NOALPHAHW
DDERR_NOBLTHW
DDERR_NOCLIPLIST

[#] IDH__dx_DirectDrawSurface4.BltToDC_ddraw_vb

DDERR_NODDROPSHW
DDERR_NOMIRRORHW
DDERR_NORASTEROPHW
DDERR_NOROTATIONHW
DDERR_NOSTRETCHHW
DDERR_NOZBUFFERHW
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_UNSUPPORTED

DirectDrawSurface4.ChangeUniquenessValue

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.ChangeUniquenessValue** method manually updates the uniqueness value for this surface.

object.**ChangeUniquenessValue()**

object
Object expression that resolves to a **DirectDrawSurface4** object.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_EXCEPTION
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

DirectDraw automatically updates uniqueness values whenever the contents of a surface change.

See Also

DirectDrawSurface4.GetUniquenessValue

[#] IDH__dx_DirectDrawSurface4.ChangeUniquenessValue_ddraw_vb

DirectDrawSurface4.DeleteAttachedSurface

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.DeleteAttachedSurface** method detaches two attached surfaces.

object.**DeleteAttachedSurface(***ddS* **As DirectDrawSurface4)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

ddS
A **DirectDrawSurface4** object for the DirectDrawSurface object to be detached. If this argument is NOTHING, all attached surfaces are detached.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_CANNOTDETACHSURFACE
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST
DDERR_SURFACENOTATTACHED

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Implicit attachments, those formed by DirectDraw rather than the **DirectDrawSurface4.AddAttachedSurface** method, cannot be detached. Detaching surfaces from a flipping chain can alter other surfaces in the chain. If a front buffer is detached from a flipping chain, the next surface in the chain becomes the front buffer, and the following surface becomes the back buffer. If a back buffer is detached from a chain, the following surface becomes a back buffer. If a plain surface is detached from a chain, the chain simply becomes shorter. If a flipping chain has only two surfaces and they are detached, the chain is destroyed and both surfaces return to their previous designations.

See Also

DirectDrawSurface4.Flip

[#] IDH__dx_DirectDrawSurface4.DeleteAttachedSurface_ddraw_vb

DirectDrawSurface4.DrawBox

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.DrawBox** method draws a box on the surface.

```
object.DrawBox( _
    x1 As Long, _
    y1 As Long, _
    x2 As Long, _
    y2 As Long)
```

object
Object expression that resolves to a **DirectDrawSurface4** object.

x1, y1, x2, y2
The upper-left and bottom-right points of the box to be drawn.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

The default draw style is a solid line and the default fill style is set to transparent. Both styles can be changed with a call to **DirectDrawSurface4.SetDrawStyle** and **DirectDrawSurface4.SetFillStyle**, respectively.

DirectDrawSurface4.DrawCircle

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.DrawCircle** method draws a circle on the surface.

```
object.DrawCircle( _
    x1 As Long, _
    y1 As Long, _
    r As Long)
```

object
Object expression that resolves to a **DirectDrawSurface4** object.

x1, y1, r
The center point and the radius of the circle to be drawn.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

IDH__dx_DirectDrawSurface4.DrawBox_ddraw_vb
IDH__dx_DirectDrawSurface4.DrawCircle_ddraw_vb

Remarks

The default draw style is a solid line and the default fill style is set to transparent. Both styles can be changed with a call to **DirectDrawSurface4.SetDrawStyle** and **DirectDrawSurface4.SetFillStyle**, respectively.

DirectDrawSurface4.DrawEllipse

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.DrawEllipse** method draws a ellipse on the surface.

object.

object
Object expression that resolves to a **DirectDrawSurface4** object.

x1, y1, x2, y2
The upper-left and the lower-right of the bounding rectangle of the ellipse to be drawn.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

The default draw style is a solid line and the default fill style is set to transparent. Both styles can be changed with a call to **DirectDrawSurface4.SetDrawStyle** and **DirectDrawSurface4.SetFillStyle**, respectively.

DirectDrawSurface4.DrawLine

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.DrawLine** method draws a line on the surface.

```
object.DrawLine( _
    x1 As Long, _
    y1 As Long, _
    x2 As Long, _
    y2 As Long)
```

object
Object expression that resolves to a **DirectDrawSurface4** object.

x1, y1, x2, y2
The end points of the line to be drawn.

IDH__dx_DirectDrawSurface4.DrawEllipse_ddraw_vb
IDH__dx_DirectDrawSurface4.DrawLine_ddraw_vb

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

The default draw style is a solid line can be changed with a call to **DirectDrawSurface4.SetDrawStyle**.

DirectDrawSurface4.DrawRoundedBox

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.DrawRoundedBox** method draws a rounded box on the surface.

```
object.DrawRoundedBox( _
    x1 As Long, _
    y1 As Long, _
    x2 As Long, _
    y2 As Long, _
    rw As Long, _
    rh As Long)
```

object
: **Object** expression that resolves to a **DirectDrawSurface4** object.

x1, y1, x2, y2
: The upper-left and lower-right points of the rectangle.

rw
: The width of the ellipse used to draw the rounded corners.

rh
: The height of the ellipse used to draw the rounded corners.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

The default draw style is a solid line and the default fill style is set to transparent. Both styles can be changed with a call to **DirectDrawSurface4.SetDrawStyle** and **DirectDrawSurface4.SetFillStyle**, respectively.

[#] IDH__dx_DirectDrawSurface4.DrawRoundedBox_ddraw_vb

DirectDrawSurface4.DrawText

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.DrawText** method draws text on the surface.

```
object.DrawText( _
    x As Long, _
    y As Long, _
    text As String, _
    b As Boolean)
```

object
Object expression that resolves to a **DirectDrawSurface4** object.

x, y
The location on the surface to draw text.

text
The text to display.

b
Boolean value indicating whether to draw to the current cursor position, see **Remarks**.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

To append text to the end of the last call to **DirectDrawSurface4.DrawText**, you must pass 0,0 as the *x* and *y* coordinate and declare *b* as **TRUE**. If you declare *b* as **FALSE**, then calling this method will result in text being displayed at the specified *x* and *y* coordinates.

DirectDrawSurface4.Flip

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.Flip** method makes the surface memory associated with the DDSCAPS_BACKBUFFER surface become associated with the front-buffer surface.

```
object.Flip( _
    ddS As DirectDrawSurface4, _
    flags As CONST_DDFLIPFLAGS)
```

object

IDH__dx_DirectDrawSurface4.DrawText_ddraw_vb
IDH__dx_DirectDrawSurface4.Flip_ddraw_vb

Object expression that resolves to a **DirectDrawSurface4** object.

ddS

A **DirectDrawSurface4** object for an arbitrary surface in the flipping chain. The default for this argument is NOTHING, in which case DirectDraw cycles through the buffers in the order they are attached to each other. If this argument is not NOTHING, DirectDraw flips to the specified surface instead of the next surface in the flipping chain. The method fails if the specified surface is not a member of the flipping chain.

flags

One or more constansts of the **CONST_DDFLIPFLAGS** enumeration specifying flip options.

DDFLIP_EVEN

For use only when displaying video in an overlay surface. The new surface contains data from the even field of a video signal. This flag cannot be used with the DDFLIP_ODD flag.

DDFLIP_INTERFVAL2
DDFLIP_INTERFVAL3
DDFLIP_INTERFVAL4

These flags indicate how many vertical retraces to wait between each flip. The default is 1. DirectDraw will return DERR_WASSTILLDRAWING for each surface involved in the flip until the specified number of vertical retraces has occurred. If DDFLIP_INTERVAL2 is set, DirectDraw will flip on every second vertical sync; if DDFLIP_INTERVAL3, on every third sync; and if DDFLIP_INTERVAL4, on every fourth sync.

These flags are effective only if DDCAPS2_FLIPINTERVAL is set in the DDCAPS structure returned for the device.

DDFLIP_ODD

For use only when displaying video in an overlay surface. The new surface contains data from the odd field of a video signal. This flag cannot be used with the DDFLIP_EVEN flag.

DDFLIP_WAIT

Typically, if the flip cannot be set up because the state of the display hardware is not appropriate, the DDERR_WASSTILLDRAWING error returns immediately and no flip occurs. Setting this flag causes the method to continue trying to flip if it receives the DDERR_WASSTILLDRAWING error from the HAL. The method does not return until the flipping operation has been successfully set up, or another error, such as DDERR_SURFACEBUSY, is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_GENERIC

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS
DDERR_NOFLIPHW
DDERR_NOTFLIPPABLE
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method can be called only for a surface that has the DDSCAPS_FLIP and DDSCAPS_FRONTBUFFER capabilities. The display memory previously associated with the front buffer is associated with the back buffer.

The *ddS* argument is used in rare cases when the back buffer is not the buffer that should become the front buffer. Typically this argument is NOTHING.

The **DirectDrawSurface4.Flip** method will always be synchronized with the vertical blank. If the surface has been assigned to a video port, this method updates the visible overlay surface and the video port's target surface.

For more information, see Flipping Surfaces.

See Also

DirectDrawSurface4.GetFlipStatus

DirectDrawSurface4.GetAttachedSurface

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetAttachedSurface** method obtains the attached surface that has the specified capabilities.

object.**GetAttachedSurface(_**
caps **As DDSCAPS2) As DirectDrawSurface4**

object
Object expression that resolves to a **DirectDrawSurface4** object.

caps
A **DDSCAPS2** type that contains the hardware capabilities of the surface.

[#] IDH__dx_DirectDrawSurface4.GetAttachedSurface_ddraw_vb

Return Values

If the method succeeds, a **DirectDrawSurface4** object is returned. The retrieved surface is the one that matches the description according to the *caps* argument.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTFOUND
DDERR_SURFACELOST

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Attachments are used to connect multiple DirectDrawSurface objects into complex types, like the ones needed to support 3-D page flipping with z-buffers. This method fails if more than one surface is attached that matches the capabilities requested. In this case, the application must use the **DirectDrawSurface4.GetAttachedSurfaceEnum** method to obtain the attached surfaces.

The object returned by a successful function call must be assigned to a **DirectDrawSurface4** object variable. For example, in Visual Basic:

```
Dim DDrawSurface as DirectDrawSurface4
Set DDrawSurface = object.GetAttachedSurface()
```

DirectDrawSurface4.GetAttachedSurfaceEnum

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetAttachedSurfaceEnum** method returns a **DirectDrawEnumSurfaces** object that is filled with attached surfaces information .

***object*.GetAttachedSurfaceEnum() As DirectDrawEnumSurfaces**

object
Object expression that resolves to a **DirectDrawSurface4** object.

[#] IDH__dx_DirectDrawSurface4.GetAttachedSurfaceEnum_ddraw_vb

Return Value

If the method succeeds, a **DirectDrawEnumSurfaces** enumeration interface is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

The number of attached surfaces is obtained with a call to **DirectDrawEnumSurfaces.GetCount** and a description of the attached surface is obtained with a call to **DirectDrawEnumSurfaces.GetItem**

The object returned by a successful function call must be assigned to a **DirectDrawEnumSurfaces** object variable. For example, in Visual Basic:

```
Dim SurfaceEnum as DirectDrawEnumSurfaces
Set SurfaceEnum = object.GetAttachedSurfacesEnum()
```

DirectDrawSurface4.GetBltStatus

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetBltStatus** method obtains the blitter status.

***object*.GetBltStatus(*flags* As CONST_DDGBSFLAGS) As Long**

object
Object expression that resolves to a **DirectDrawSurface4** object.

flags
One of the constants of the **CONST_DDGBSFLAGS** enumeration:

DDGBS_CANBLT
Inquires whether a blit involving this surface can occur immediately, and returns DD_OK if the blit can be completed.

DDGBS_ISBLTDONE
Inquires whether the blit is done, and returns DD_OK if the last blit on this surface has completed.

[#] IDH__dx_DirectDrawSurface4.GetBltStatus_ddraw_vb

Return Values

If the method succeeds, depending on which constant is specified in the *flags* argument, zero is returned for FALSE and non-zero for TRUE.

Error Codes

If the method fails, the error code is DDERR_WASSTILLDRAWING if the surface has not finished its flipping process, or one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DD_OK
DDERR_UNSUPPORTED

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDrawSurface4.GetCaps

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetCaps** method retrieves the capabilities of the surface. These capabilities are not necessarily related to the capabilities of the display device.

object.**GetCaps(***caps* **As DDSCAPS2)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

caps
A **DDSCAPS2** type that will be filled with the hardware capabilities of the surface.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

[#] IDH__dx_DirectDrawSurface4.GetCaps_ddraw_vb

DirectDrawSurface4.GetClipper

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetClipper** method retrieves the DirectDrawClipper object associated with this surface.

object.**GetClipper() As DirectDrawClipper**

object
Object expression that resolves to a **DirectDrawSurface4** object.

Return Value

If the method succeeds, a **DirectDrawClipper** object associated with the surface is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCLIPPERATTACHED

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

DirectDrawSurface4.SetClipper

Remarks

The object returned by a successful function call must be assigned to a **DirectDrawClipper** object variable. For example, in Visual Basic:

```
Dim Clipper as DirectDrawClipper
Set Clipper = object.GetClipper()
```

DirectDrawSurface4.GetColorKey

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetColorKey** method retrieves the color key value for the DirectDrawSurface object.

IDH__dx_DirectDrawSurface4.GetClipper_ddraw_vb
IDH__dx_DirectDrawSurface4.GetColorKey_ddraw_vb

object.**GetColorKey(***flags* **As Long,** *val* **As DDCOLORKEY)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

flags
The color key requested.

DDCKEY_DESTBLT
Set if the type specifies a color key or color space to be used as a destination color key for blit operations.

DDCKEY_DESTOVERLAY
Set if the type specifies a color key or color space to be used as a destination color key for overlay operations.

DDCKEY_SRCBLT
Set if the type specifies a color key or color space to be used as a source color key for blit operations.

DDCKEY_SRCOVERLAY
Set if the type specifies a color key or color space to be used as a source color key for overlay operations.

val
A **DDCOLORKEY** type that will be filled with the current values for the specified color key of the DirectDrawSurface object.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCOLORKEY
DDERR_NOCOLORKEYHW
DDERR_SURFACELOST
DDERR_UNSUPPORTED

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

DirectDrawSurface4.SetColorKey

DirectDrawSurface4.GetDC

[#] [This is preliminary documentation and subject to change.]

[#] IDH__dx_DirectDrawSurface4.GetDC_ddraw_vb

The **DirectDrawSurface4.GetDC** method creates a GDI-compatible handle of a device context for the surface.

object.**GetDC() As Long**

object
Object expression that resolves to a **DirectDrawSurface4** object.

Return Value

If the method succeeds, the handle to a device context is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_DCALREADYCREATED
DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method uses an internal version of the **DirectDrawSurface4.Lock** method to lock the surface. The surface remains locked until the **DirectDrawSurface4.ReleaseDC** method is called.

See Also

DirectDrawSurface4.Lock

DirectDrawSurface4.GetDirectDraw

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetDirectDraw** method retrieves the DirectDraw object that was used to create the surface.

[#] IDH__dx_DirectDrawSurface4.GetDirectDraw_ddraw_vb

object.**GetDirectDraw() As DirectDraw4**

object
Object expression that resolves to a **DirectDrawSurface4** object.

Return Value

If the method succeeds, a **DirectDraw4** object is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

The object returned by a successful function call must be assigned to a **DirectDraw4** object variable. For example:

```
Dim DDrawObject as DirectDraw4
Set DDrawObject = object.GetDDInterface()
```

DirectDrawSurface4.GetDirectDrawColorControl

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetDirectDrawColorControl** method returns the DirectDrawColorControl object used with the surface.

object.**GetDirectDrawColorControl() As DirectDrawColorControl**

object
Object expression that resolves to a **DirectDrawSurface4** object.

Return Value

If the method succeeds, a **DirectDrawColorControl** object is returned.

[#] IDH__dx_DirectDrawSurface4.GetDirectDrawColorControl_ddraw_vb

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDrawSurface4.GetDirectDrawGammaControl

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetDirectDrawGammaControl** method returns the **DirectDrawGammaControl** object used with the surface.

object.**GetDirectDrawGammaControl() As DirectDrawGammaControl**

object
Object expression that resolves to a **DirectDrawSurface4** object.

Return Value

If the method succeeds, a **DirectDrawGammaControl** object is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDrawSurface4.GetDrawStyle

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetDrawStyle** returns the style set by **DirectDrawSurface4.SetDrawStyle**.

object.**getDrawStyle() As Long**

object
Object expression that resolves to a **DirectDrawSurface4** object.

IDH__dx_DirectDrawSurface4.GetDirectDrawGammaControl_ddraw_vb
IDH__dx_DirectDrawSurface4.GetDrawStyle_ddraw_vb

Return Values

The style set by **DirectDrawSurface4.SetDrawStyle**.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

The default draw style is a solid line.

DirectDrawSurface4.GetDrawWidth

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetDrawWidth** returns the width set by **DirectDrawSurface4.SetDrawWidth**.

object.**getDrawWidth() As Long**

object
Object expression that resolves to a **DirectDrawSurface4** object.

Return Values

The width set by **DirectDrawSurface4.SetDrawWidth**.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

The default draw width is 1.

DirectDrawSurface4.GetFillColor

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetFillColor** returns the fill color set by **DirectDrawSurface4.SetFillColor**.

object.**GetFillColor() As Long**

IDH__dx_DirectDrawSurface4.GetDrawWidth_ddraw_vb
IDH__dx_DirectDrawSurface4.GetFillColor_ddraw_vb

object
Object expression that resolves to a **DirectDrawSurface4** object.

Return Values

The fill color value set by **DirectDrawSurface4.SetFillColor**.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

The default fill color is black.

DirectDrawSurface4.GetFillStyle

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetFillStyle** returns the fill style set by **DirectDrawSurface4.SetFillStyle**.

object.**GetFillStyle() As Long**

object
Object expression that resolves to a **DirectDrawSurface4** object.

Return Values

The the fill style set by **DirectDrawSurface4.SetFillStyle**.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

The default fill style is transparent.

DirectDrawSurface4.GetFlipStatus

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetFlipStatus** method indicates whether the surface has finished its flipping process.

object.**GetFlipStatus(***flags* **As CONST_DDGFSFLAGS) As Long**

IDH__dx_DirectDrawSurface4.GetFillStyle_ddraw_vb
IDH__dx_DirectDrawSurface4.GetFlipStatus_ddraw_vb

object
Object expression that resolves to a **DirectDrawSurface4** object.

flags
One of the following constants of the **CONST_DDGFSFLAGS** enumeration:

DDGFS_CANFLIP
Inquires whether this surface can be flipped immediately and returns DD_OK if the flip can be completed.

DDGFS_ISFLIPDONE
Inquires whether the flip has finished and returns DD_OK if the last flip on this surface has completed.

Return Values

If the method succeeds, depending on which constant is specified in the *flags* argument, zero is returned for FALSE and non-zero for TRUE.

Error Codes

The error code is DDERR_WASSTILLDRAWING if the surface has not finished its flipping process, or one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DD_OK
DDERR_UNSUPPORTED

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

DirectDrawSurface4.Flip

DirectDrawSurface4.GetFontTransparency

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetFontTransparency** returns the font transparency set by **DirectDrawSurface4.SetFontTransparency**.

[#] IDH__dx_DirectDrawSurface4.GetFontTransparency_ddraw_vb

object.**GetFontTransparency() As Boolean**

object
Object expression that resolves to a **DirectDrawSurface4** object.

Return Values

If the font is set to be transparent, non-zero is returned and zero if the font is not set to be transparent.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

DirectDrawSurface4.GetForeColor

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetForeColor** returns the fore color set by **DirectDrawSurface4.SetForeColor**.

object.**GetForeColor() As Long**

object
Object expression that resolves to a **DirectDrawSurface4** object.

Return Values

The fore color set by **DirectDrawSurface4.SetForeColor**.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

DirectDrawSurface4.GetLockedPixel

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetLockedPixel** method returns the specified pixel set by **DirectDrawSurface4.SetLockedPixel**.

object.**GetLockedPixel(_**
x **As Long, _**
y **As Long) As Long**

IDH__dx_DirectDrawSurface4.GetForeColor_ddraw_vb
IDH__dx_DirectDrawSurface4.GetLockedPixel_ddraw_vb

object
Object expression that resolves to a **DirectDrawSurface4** object.

x and *y*
The coordinates of the locked pixel.

Return Values

Returns the color of the locked pixel. This value is in the same format as the pixel format.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

DirectDrawSurface4.GetLockedSurfaceBits

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetLockedSurfaceBits** method fills an array with the locked surface. After manipulating the locked bits, a call to **DirectDrawSurface4.SetLockedSurfaceBits** will update the surface.

object.**GetLockedSurfaceBits(***memory* **As Any)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

memory
An array filled with the locked surface bits. This array is comprised of the same data type as the pixel format of the locked surface.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

The pixel format can be obtained with a call to **DirectDrawSurface4.GetPixelFormat**.

DirectDrawSurface4.GetOverlayZOrdersEnum

[This is preliminary documentation and subject to change.]

IDH__dx_DirectDrawSurface4.GetLockedSurfaceBits_ddraw_vb
IDH__dx_DirectDrawSurface4.GetOverlayZOrdersEnum_ddraw_vb

The **DirectDrawSurface4.GetOverlayZOrdersEnum** method returns a **DirectDrawEnumSurfaces** object that is filled with overlay data. The created **DirectDrawEnumSurfaces** object can then be called to enumerate the overlay surfaces on the specified destination. The overlays can be enumerated in front-to-back or back-to-front order..

```
object.GetOverlayZOrdersEnum( _
    flags As CONST_DDENUMOVERLAYZFLAGS) _
    As DirectDrawEnumSurfaces
```

object
Object expression that resolves to a **DirectDrawSurface4** object.

flags
One of the following constants of the **CONST_DDENUMOVERLAYZFLAGS** enumeration:

DDENUMOVERLAYZ_BACKTOFRONT
Enumerates overlays back to front.

DDENUMOVERLAYZ_FRONTTOBACK
Enumerates overlays front to back.

Return Value

If the method succeeds, a **DirectDrawEnumSurfaces** is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

The object returned by a successful function call must be assigned to a **DirectDrawEnumSurfaces** object variable. For example:

```
Dim SurfaceEnum as DirectDrawEnumSurfaces
Set SurfaceEnum = object.GetOverlayZOrdersEnum(flags)
```

DirectDrawSurface4.GetPalette

[#] [This is preliminary documentation and subject to change.]

[#] IDH__dx_DirectDrawSurface4.GetPalette_ddraw_vb

The **DirectDrawSurface4.GetPalette** method retrieves the DirectDrawPalette object associated with this surface.

object.**GetPalette() As DirectDrawPalette**

object
Object expression that resolves to a **DirectDrawSurface4** object.

Return Value

If the method succeeds, a **DirectDrawPalette** object is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOEXCLUSIVEMODE
DDERR_NOPALETTEATTACHED
DDERR_SURFACELOST
DDERR_UNSUPPORTED

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

DirectDrawSurface4.SetPalette**Remarks**

The object returned by a successful function call must be assigned to a **DirectDrawEnumSurfaces** object variable. For example:

```
Dim DDPalette as DirectDrawPalette
Set DDPalette = object.GetPalette()
```

DirectDrawSurface4.GetPixelFormat

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetPixelFormat** method retrieves the color and pixel format of the surface.

object.**GetPixelFormat(***pf* **As DDPIXELFORMAT)**

[#] IDH__dx_DirectDrawSurface4.GetPixelFormat_ddraw_vb

object
Object expression that resolves to a **DirectDrawSurface4** object.

pf
A **DDPIXELFORMAT** type that will be filled with a detailed description of the current pixel and color space format of the surface.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDrawSurface4.GetSurfaceDesc

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetSurfaceDesc** method retrieves a description of the surface in its current condition.

object.**GetSurfaceDesc(***surface* **As DDSURFACEDESC2)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

surface
A **DDSURFACEDESC2** type that will be filled with the current description of this surface.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

[#] IDH__dx_DirectDrawSurface4.GetSurfaceDesc_ddraw_vb

DirectDrawSurface4.GetTexture

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetTexture** method returns a **Direct3DTexture2** object.

object.**GetTexture() As Direct3DTexture2**

object
Object expression that resolves to a **DirectDrawSurface4** object.

Return Values

If the method succeeds, a **Direct3DTexture2** object is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDrawSurface4.GetUniquenessValue

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.GetUniquenessValue** method retrieves the current uniqueness value for this surface.

object.**GetUniquenessValue() As Long**

object
Object expression that resolves to a **DirectDrawSurface4** object.

Return Values

If the method succeeds, the surface's current uniqueness value is returned.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT

IDH__dx_DirectDrawSurface4.GetTexture_ddraw_vb
IDH__dx_DirectDrawSurface4.GetUniquenessValue_ddraw_vb

DDERR_INVALIDPARAMS

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

The only defined uniqueness value is 0, to indicate that the surface is likely to be changing beyond DirectDraw's control. Other uniqueness values are only significant if they differ from a previously cached uniqueness value. If the current value is different than a cached value, then the contents of the surface have changed.

See Also

DirectDrawSurface4.ChangeUniquenessValue

DirectDrawSurface4.IsLost

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.IsLost** method determines if the surface memory associated with a DirectDrawSurface object has been freed.

object.**IsLost() As Long**

object
Object expression that resolves to a **DirectDrawSurface4** object.

Return Value

If the method succeeds, zero (FALSE) is returned indicating the surface has not been freed and nonzero if it has been freed.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

You can use this method to find out if you need reallocate surface memory by calling the **DirectDrawSurface4.Restore** method. When a DirectDrawSurface

[#] IDH__dx_DirectDrawSurface4.IsLost_ddraw_vb

object loses its surface memory, most methods return DDERR_SURFACELOST and perform no other action.

Surfaces can lose their memory when the mode of the display card is changed, or when an application receives exclusive access to the display card and frees all of the surface memory currently allocated on the display card.

See Also

DirectDrawSurface4.Restore, Losing and Restoring Surfaces

DirectDrawSurface4.Lock

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.Lock** method obtains a pointer to the surface memory.

```
object.Lock( _
    r As RECT, _
    desc As DDSURFACEDESC2, _
    flags As CONST_DDLOCKFLAGS, _
    hnd As Long)
```

object
Object expression that resolves to a **DirectDrawSurface4** object.

r
A **RECT** type that defines the upper-left and lower-right points of the rectangle that identifies the region of surface that is being locked. If NOTHING, the entire surface will be locked.

desc
A **DDSURFACEDESC2** type that will be filled with the relevant details about the surface.

flags
One or more of the following constants of the **CONST_DDLOCKFLAGS** enumeration that describes the type of lock to be performed:

DDLOCK_EVENT
This flag is not currently implemented.

DDLOCK_NOSYSLOCK
If possible, do not take the Win16Mutex (also known as Win16Lock). This flag is ignored when locking the primary surface.

DDLOCK_READONLY
Indicates that the surface being locked will only be read.

DDLOCK_SURFACEMEMORYPTR

[#] IDH__dx_DirectDrawSurface4.Lock_ddraw_vb

Indicates that a valid memory pointer to the top of the specified rectangle should be returned. If no rectangle is specified, a pointer to the top of the surface is returned. This is the default.

DDLOCK_WAIT
If a lock cannot be obtained because a blit operation is in progress, the method retries until a lock is obtained or another error occurs, such as DDERR_SURFACEBUSY.

DDLOCK_WRITEONLY
Indicates that the surface being locked will be write enabled.

hnd
This parameter is not used and must be set to NOTHING.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_WASSTILLDRAWING

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

For more information on using this method, see Accessing Surface Memory Directly.

After retrieving a surface memory pointer, you can access the surface memory until a corresponding **DirectDrawSurface4.Unlock** method is called. When the surface is unlocked, the pointer to the surface memory is invalid.

Do not call DirectDraw blit functions to blit from a locked region of a surface. If you do, the blit returns either DDERR_SURFACEBUSY or DDERR_LOCKEDSURFACES. Additionally, GDI blit functions will silently fail when used on a locked video memory surface.

This method often causes DirectDraw to hold the Win16Mutex (also known as Win16Lock) until you call the **DirectDrawSurface4.Unlock** method. GUI debuggers cannot operate while the Win16Mutex is held.

See Also

DirectDrawSurface4.Unlock, **DirectDrawSurface4.GetDC**, **DirectDrawSurface4.ReleaseDC**

DirectDrawSurface4.ReleaseDC

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.ReleaseDC** method releases the handle of a device context previously obtained by using the **DirectDrawSurface4.GetDC** method.

object.**ReleaseDC(***hdc* **As Long)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

hdc
The handle to a device context previously obtained by **DirectDrawSurface4.GetDC**.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST
DDERR_UNSUPPORTED

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method also unlocks the surface previously locked when the **DirectDrawSurface4.GetDC** method was called.

See Also

DirectDrawSurface4.GetDC

DirectDrawSurface4.Restore

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.Restore** method restores a surface that has been lost. This occurs when the surface memory associated with the DirectDrawSurface object has been freed.

object.**Restore()**

IDH__dx_DirectDrawSurface4.ReleaseDC_ddraw_vb
IDH__dx_DirectDrawSurface4.Restore_ddraw_vb

object
Object expression that resolves to a **DirectDrawSurface4** object.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_GENERIC
DDERR_IMPLICITLYCREATED
DDERR_INCOMPATIBLEPRIMARY
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOEXCLUSIVEMODE
DDERR_OUTOFMEMORY
DDERR_UNSUPPORTED
DDERR_WRONGMODE

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

This method restores the memory allocated for a surface, but doesn't reload any bitmaps that may have existed in the surface before it was lost.

Surfaces can be lost because the mode of the display card was changed or because an application received exclusive access to the display card and freed all of the surface memory currently allocated on the card. When a DirectDrawSurface object loses its surface memory, many methods will return DDERR_SURFACELOST and perform no other function. The **DirectDrawSurface4.Restore** method will reallocate surface memory and reattach it to the DirectDrawSurface object.

A single call to this method will restore a DirectDrawSurface object's associated implicit surfaces (back buffers, and so on). An attempt to restore an implicitly created surface will result in an error. **DirectDrawSurface4.Restore** will not work across explicit attachments created by using the **DirectDrawSurface4.AddAttachedSurface** method—each of these surfaces must be restored individually.

See Also

DirectDrawSurface4.IsLost

DirectDrawSurface4.SetClipper

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.SetClipper** method attaches a clipper object to or deletes one from a surface.

object.**SetClipper(***val* **As DirectDrawClipper)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

val
A **DirectDrawClipper** object for the DirectDrawClipper object that will be attached to the DirectDrawSurface object. If this argument is NOTHING, the current DirectDrawClipper object will be detached.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE
DDERR_NOCLIPPERATTACHED

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

If you pass NOTHING as the *val* argument, the clipper is removed from the surface

This method is primarily used by surfaces that are being overlaid on or blitted to the primary surface. However, it can be used on any surface. After a DirectDrawClipper object has been attached and a clip list is associated with it, the DirectDrawClipper object will be used for the **DirectDrawSurface4.Blt**, and **DirectDrawSurface4.UpdateOverlay** operations involving the parent DirectDrawSurface object. This method can also detach a DirectDrawSurface object's current DirectDrawClipper object.

See Also

DirectDrawSurface4.GetClipper

[#] IDH__dx_DirectDrawSurface4.SetClipper_ddraw_vb

DirectDrawSurface4.SetColorKey

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.SetColorKey** method sets the color key value for the DirectDrawSurface object if the hardware supports color keys on a per surface basis.

```
object.SetColorKey( _
    flags As CONST_DDCKEYFLAGS, _
    val As DDCOLORKEY)
```

object
Object expression that resolves to a **DirectDrawSurface4** object.

flags
One of the following constants of the **CONST_DDCKEYFLAGS** enumeration specifying the type of color key requested.

DDCKEY_COLORSPACE
Set if the type contains a color space. Not set if the type contains a single color key.

DDCKEY_DESTBLT
Set if the type specifies a color key or color space to be used as a destination color key for blit operations.

DDCKEY_DESTOVERLAY
Set if the type specifies a color key or color space to be used as a destination color key for overlay operations.

DDCKEY_SRCBLT
Set if the type specifies a color key or color space to be used as a source color key for blit operations.

DDCKEY_SRCOVERLAY
Set if the type specifies a color key or color space to be used as a source color key for overlay operations.

val
A **DDCOLORKEY** type that contains the new color key values for the DirectDrawSurface object. This value can be NOTHING to remove a previously set color key.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE

[#] IDH__dx_DirectDrawSurface4.SetColorKey_ddraw_vb

DDERR_NOOVERLAYHW
DDERR_NOTAOVERLAYSURFACE
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

For transparent blits and overlays, you should set destination color on the destination surface and source color on the source surface.

See Also

DirectDrawSurface4.GetColorKey

DirectDrawSurface4.SetDrawStyle

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.SetDrawStyle** sets the draw style.

object.**SetDrawStyle(***drawStyle* **As Long)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

drawStyle
One of the following draw styles to be set.

Setting	Description
0	(Default) Solid
1	Dash
2	Dot
3	Dash-Dot
4	Dash-Dot-Dot
5	Transparent
6	Inside Solid

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

[#] IDH__dx_DirectDrawSurface4.SetDrawStyle_ddraw_vb

See Also

DirectDrawSurface4.GetDrawStyle

DirectDrawSurface4.SetDrawWidth

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.SetDrawWidth** sets the width of the line used in drawing methods.

***object*.setDrawWidth(*drawWidth* As Long)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

drawWidth
The width of the line used in drawing methods. The default value is 1. Any value between 1 and 32,767 is valid.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

See Also

DirectDrawSurface4.GetDrawWidth

DirectDrawSurface4.SetFillColor

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.SetFillColor** specifies the fill color used in drawing methods.

***object*.SetFillColor(*color* As Long)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

color
An RGB value to be used as the fill color in drawing methods. The default value is black (&H00000000).

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

IDH__dx_DirectDrawSurface4.SetDrawWidth_ddraw_vb
IDH__dx_DirectDrawSurface4.SetFillColor_ddraw_vb

See Also

DirectDrawSurface4.GetFillColor

DirectDrawSurface4.SetFillStyle

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.SetFillStyle** specifies the fill style used in the drawing methods.

object.**SetFillStyle(***fillStyle* **As Long)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

fillStyle
One of the following fill styles to be used in drawing methods.

Setting	Description
0	Solid
1	(Default) Transparent
2	Horizontal Line
3	Vertical Line
4	Upward Diagonal
5	Downward Diagonal
6	Cross
7	Diagonal Cross

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

See Also

DirectDrawSurface4.GetFillStyle

DirectDrawSurface4.SetFont

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.SetFont** specifies the font to be used in **DirectDrawSurface4.DrawText**.

object.**SetFont(***font* **As IFont)**

IDH__dx_DirectDrawSurface4.SetFillStyle_ddraw_vb
IDH__dx_DirectDrawSurface4.SetFont_ddraw_vb

object
Object expression that resolves to a **DirectDrawSurface4** object.

font
The font as specified in the **IFont** class.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

DirectDrawSurface4.SetFontTransparency

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.SetFontTransparency** specified whether the font to be used when the **DirectDrawSurface4.DrawText** is called is transparent.

object.**SetFontTransparency(***b* **As Boolean)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

b
Set to D_TRUE if the font is transparent or D_FALSE if the font is not transparent.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

DirectDrawSurface4.SetForeColor

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.SetForeColor** method sets the foreground color used in drawing methods.

object.**SetForeColor(***color* **As Long)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

color
An RGB value to set as the foreground color.

IDH__dx_DirectDrawSurface4.SetFontTransparency_ddraw_vb
IDH__dx_DirectDrawSurface4.SetForeColor_ddraw_vb

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

See Also

DirectDrawSurface4.GetForeColor

DirectDrawSurface4.SetLockedPixel

[This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.SetLockedPixel** method sets a single pixel to the specified color and updates the locked surface.

```
object.SetLockedPixel( _
    x As Long, _
    y As Long, _
    col As Long)
```

object
Object expression that resolves to a **DirectDrawSurface4** object.

x and *y*
The coordinates of the pixel to set.

col
An RGB value to for the set pixel.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

You must first lock the surface with a call to **DirectDrawSurface4.Lock**.

See Also

DirectDrawSurface4.GetLockedPixel

DirectDrawSurface4.SetLockedSurfaceBits

[This is preliminary documentation and subject to change.]

IDH__dx_DirectDrawSurface4.SetLockedPixel_ddraw_vb
IDH__dx_DirectDrawSurface4.SetLockedSurfaceBits_ddraw_vb

The **DirectDrawSurface4.SetLockedSurfaceBits** method updates the locked surface.

object.**SetLockedSurfaceBits(***memory* **As Any)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

memory
An array filled with the manipulated surface bits. This must be the same array used in the method **DirectDrawSurface4.GetLockedSurfaceBits**.

Error Codes

If the method fails, an error is raised and **Err.Number** will be set.

Remarks

To manipulate the locked surface, call **DirectDrawSurface4.GetLockedSurfaceBits**. You must first lock the surface with a call to **DirectDrawSurface4.Lock**.

DirectDrawSurface4.SetPalette

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.SetPalette** method attaches a palette object to (or detaches one from) a surface. The surface uses this palette for all subsequent operations. The palette change takes place immediately, without regard to refresh timing.

object.**SetPalette(***ddp* **As DirectDrawPalette)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

ddp
A **DirectDrawPalette** object for the palette object to be used with this surface. If this argument is NOTHING, the current palette will be detached.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDPIXELFORMAT

[#] IDH__dx_DirectDrawSurface4.SetPalette_ddraw_vb

DDERR_INVALIDSURFACETYPE
DDERR_NOEXCLUSIVEMODE
DDERR_NOPALETTEATTACHED
DDERR_NOPALETTEHW
DDERR_NOT8BITCOLOR
DDERR_SURFACELOST
DDERR_UNSUPPORTED

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

If you pass NOTHING as the *ddp* argument, the palette is removed from the surface.

See Also

DirectDrawSurface4.GetPalette, **DirectDraw4.CreatePalette**

DirectDrawSurface4.Unlock

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.Unlock** method notifies DirectDraw that the direct surface manipulations are complete.

object.**Unlock(***r* **As RECT)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

r
A **RECT** type that is used to lock the surface in the corresponding call to the **DirectDrawSurface4.Lock** method. This argument can be NOTHING only if the entire surface was locked by passing NOTHING in the *r* argument of the corresponding call to the **DirectDrawSurface4.Lock** method.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDRECT
DDERR_NOTLOCKED

[#] IDH__dx_DirectDrawSurface4.Unlock_ddraw_vb

DDERR_SURFACELOST

For information on trapping errors, see the Visual Basic Error Trapping topic.

Remarks

Because it is possible to call **DirectDrawSurface4.Lock** multiple times for the same surface with different destination rectangles, the *r* parameter links the calls to the **DirectDrawSurface4.Lock** and **DirectDrawSurface4.Unlock** methods.

See Also

DirectDrawSurface4.Lock

DirectDrawSurface4.UpdateOverlay

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.UpdateOverlay** method repositions or modifies the visual attributes of an overlay surface. These surfaces must have the DDSCAPS_OVERLAY flag set in the DDSCAPS2 type when the surface is created.

```
object.UpdateOverlay( _
    RECT As RECT, _
    ddS As DirectDrawSurface4, _
    rectD As RECT, _
    flags As CONST_DDOVERFLAGS)
```

object
Object expression that resolves to a **DirectDrawSurface4** object.

RECT
A variable of type **RECT** that defines the x, y, width, and height of the region on the source surface being used as the overlay. This argument can be NOTHING when hiding an overlay or to indicate that the entire overlay surface is to be used and that the overlay surface conforms to any boundary and size alignment restrictions imposed by the device driver.

ddS
A **DirectDrawSurface4** object for the surface that is being overlaid.

rectD
A variable of type **RECT** that defines the x, y, width, and height of the region on the destination surface that the overlay should be moved to. This argument can be NOTHING when hiding the overlay.

flags

[#] IDH__dx_DirectDrawSurface4.UpdateOverlay_ddraw_vb

One or more of the following constants of the **CONST_DDOVERFLAGS** enumeration:

DDOVER_ADDDIRTYRECT
This flag is not used.

DDOVER_ALPHADEST
Uses either the alpha information in pixel format or the alpha channel surface attached to the destination surface as the alpha channel for this overlay.

DDOVER_ALPHADESTNEG
Indicates that the destination surface becomes more transparent as the alpha value increases (0 is opaque).

DDOVER_ALPHASRC
Uses either the alpha information in pixel format or the alpha channel surface attached to the source surface as the source alpha channel for this overlay.

DDOVER_ALPHASRCNEG
Indicates that the source surface becomes more transparent as the alpha value increases (0 is opaque).

DDOVER_AUTOFLIP
Automatically flip to the next surface in the flip chain each time a video port VSYNC occurs.

DDOVER_BOB
Display each field individually of the interlaced video stream without causing any artifacts.

DDOVER_BOBHARDWARE
Indicates that bob operations will be performed using hardware rather than software or emulated. This flag must be used with the DDOVER_BOB flag.

DDOVER_HIDE
Turns off this overlay.

DDOVER_KEYDEST
Uses the color key associated with the destination surface.

DDOVER_KEYSRC
Uses the color key associated with the source surface.

DDOVER_OVERRIDEBOBWEAVE
Indicates that bob/weave decisions should not be overridden by other classes.

DDOVER_INTERLEAVED
Indicates that the surface memory is composed of interleaved fields.

DDOVER_SHOW
Turns on this overlay.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_DEVICEDOESNTOWNSURFACE

DDERR_GENERIC

DDERR_HEIGHTALIGN
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDRECT
DDERR_INVALIDSURFACETYPE
DDERR_NOSTRETCHHW
DDERR_NOTAOVERLAYSURFACE
DDERR_OUTOFCAPS
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_XALIGN

For information on trapping errors, see the Visual Basic Error Trapping topic.

DirectDrawSurface4.UpdateOverlayZOrder

[#] [This is preliminary documentation and subject to change.]

The **DirectDrawSurface4.UpdateOverlayZOrder** method sets the z-order of an overlay.

object.**UpdateOverlayZOrder(_**
flags **As CONST_DDOVERZFLAGS, _**
ddS **As DirectDrawSurface4)**

object
Object expression that resolves to a **DirectDrawSurface4** object.

flags
One of the following constants of the **CONST_DDOVERZFLAGS** enumeration:

DDOVERZ_INSERTINBACKOF
Inserts this overlay in the overlay chain behind the reference overlay.

DDOVERZ_INSERTINFRONTOF
Inserts this overlay in the overlay chain in front of the reference overlay.

DDOVERZ_MOVEBACKWARD
Moves this overlay one position backward in the overlay chain.

DDOVERZ_MOVEFORWARD
Moves this overlay one position forward in the overlay chain.

DDOVERZ_SENDTOBACK
Moves this overlay to the back of the overlay chain.

[#] IDH__dx_DirectDrawSurface4.UpdateOverlayZOrder_ddraw_vb

DDOVERZ_SENDTOFRONT
Moves this overlay to the front of the overlay chain.

ddS
A **DirectDrawSurface4** object for the DirectDraw surface to be used as a relative position in the overlay chain. This argument is needed only for DDOVERZ_INSERTINBACKOF and DDOVERZ_INSERTINFRONTOF.

Error Codes

If the method fails, the error code may be one of the following:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTAOVERLAYSURFACE

For information on trapping errors, see the Visual Basic Error Trapping topic.

See Also

DirectDrawSurface4.GetOverlayZOrdersEnum

DirectDraw Global Methods

[This is preliminary documentation and subject to change.]

The **DirectX7** class is the main class of any DirectX for Visual Basic application. This class has methods that are used with all of the DirectX components. The methods pertaining to DirectDraw are:

- **DirectX7.DirectDrawCreate**
- **DirectX7.GetDDEnum**

Types

[This is preliminary documentation and subject to change.]

This section contains information about the following types used with DirectDraw:

- **DDBLTFX**
- **DDCAPS**
- **DDCOLORCONTROL**
- **DDCOLORKEY**
- **DDGAMMARAMP**
- **DDPIXELFORMAT**
- **DDSCAPS2**
- **DDSURFACEDESC2**

- **DXDRIVERINFO**
- **PALETTEENTRY**
- **RECT**

DDBLTFX

[#] [This is preliminary documentation and subject to change.]

The **DDBLTFX** type passes raster operations, effects, and override information to the **DirectDrawSurface4.Blt** method.

```
Type DDBLTFX
    ddckDestColorKey_high As Long
    ddckDestColorKey_low As Long
    ddckSrcColorKey_high As Long
    ddckSrcColorKey_low As Long
    lAlphaDestConst As Long
    lAlphaDestConstBitDepth As Long
    lAlphaEdgeBlend As Long
    lAlphaEdgeBlendBitDepth As Long
    lAlphaSrcConst As Long
    lAlphaSrcConstBitDepth As Long
    lDDFX As CONST_DDBLTFXFLAGS
    lDDROP As Long
    lFill As Long
    lReserved As Long
    lROP As Long
    lRotationAngle As Long
    lZBufferBaseDest As Long
    lZBufferHigh As Long
    lZBufferLow As Long
    lZBufferOpCode As Long
    lZDestConst As Long
    lZDestConstBitDepth As Long
    lZSrcConst As Long
    lZSrcConstBitDepth As Long
End Type
```

ddckDestColorKey_high
High value, inclusive, of the color range that is to be used as the destination color key.

ddckDestColorKey_low
Low value, inclusive, of the color range that is to be used as the destination color key.

[#] IDH__dx_DDBLTFX_ddraw_vb

ddckSrcColorKey_high
High value, inclusive, of the color range that is to be used as the source color key.

ddckSrcColorKey_low
Low value, inclusive, of the color range that is to be used as the source color key.

lAlphaDestConst
Constant used as the alpha channel destination.

lAlphaDestConstBitDepth
Bit depth of the destination alpha constant.

lAlphaEdgeBlend
Alpha constant used for edge blending.

lAlphaEdgeBlendBitDepth
Bit depth of the constant for an alpha edge blend.

lAlphaSrcConst
Constant used as the alpha channel source.

lAlphaSrcConstBitDepth
Bit depth of the source alpha constant.

lDDFX
Type of FX operations. One of the following constants of the **CONST_DDBLTFXFLAGS** enumeration:

DDBLTFX_ARITHSTRETCHY
Uses arithmetic stretching along the y-axis for this blit.

DDBLTFX_MIRRORLEFTRIGHT
Turns the surface on its y-axis. This blit mirrors the surface from left to right.

DDBLTFX_MIRRORUPDOWN
Turns the surface on its x-axis. This blit mirrors the surface from top to bottom.

DDBLTFX_NOTEARING
Schedules this blit to avoid tearing.

DDBLTFX_ROTATE180
Rotates the surface 180 degrees clockwise during this blit.

DDBLTFX_ROTATE270
Rotates the surface 270 degrees clockwise during this blit.

DDBLTFX_ROTATE90
Rotates the surface 90 degrees clockwise during this blit.

DDBLTFX_ZBUFFERBASEDEST
Adds the **lZBufferBaseDest** member to each of the source z-values before comparing them with the destination z-values during this z-blit.

DDBLTFX_ZBUFFERRANGE
Uses the **lZBufferLow** and **lZBufferHigh** members as range values to specify limits to the bits copied from a source surface during this z-blit.

lDDROP

DirectDraw raster operations.

lFill
Color used to fill a surface when DDBLT_COLORFILL is specified. This value must be a pixel appropriate to the pixel format of the destination surface. For a palettized surface it would be a palette index, and for a 16-bit RGB surface it would be a 16-bit pixel value.

lReserved
Reserved for future use.

lROP
Win32 raster operations. You can retrieve a list of supported raster operations by calling the **DirectDraw4.GetCaps** method.

lRotationAngle
Rotation angle for the blit.

lZBufferBaseDest
Destination base value of a z-buffer.

lZBufferHigh
High limit of a z-buffer.

lZBufferLow
Low limit of a z-buffer.

lZBufferOpCode
Z-buffer compares.

lZDestConst
Constant used as the z-buffer destination.

lZDestConstBitDepth
Bit depth of the destination z-constant.

lZSrcConst
Constant used as the z-buffer source.

lZSrcConstBitDepth
Bit depth of the source z-constant.

DDCAPS

[#] [This is preliminary documentation and subject to change.]

The **DDCAPS** type represents the capabilities of the hardware exposed through the DirectDraw object. This type contains a **DDSCAPS2** type used in this context to describe what kinds of DirectDrawSurface objects can be created. It may not be possible to simultaneously create all of the surfaces described by these capabilities. This type is used with the **DirectDraw4.GetCaps** method.

```
Type DDCAPS
    ddsCaps As DDSCAPS2
    lAlignBoundaryDest As Long
```

[#] IDH__dx_DDCAPS_ddraw_vb

lAlignBoundarySrc As Long
lAlignSizeDest As Long
lAlignSizeSrc As Long
lAlignStrideAlign As Long
lAlphaBltConstBitDepths As Long
lAlphaBltPixelBitDepths As Long
lAlphaBltSurfaceBitDepths As Long
lAlphaOverlayConstBitDepths As Long
lAlphaOverlayPixelBitDepths As Long
lAlphaOverlaySurfaceBitDepths As Long
lCaps As CONST_DDCAPS1FLAGS
lCaps2 As CONST_DDCAPS2FLAGS
lCKeyCaps As CONST_DDCKEYCAPSFLAGS
lCurrVideoPorts As Long
lCurrVisibleOverlays As Long
lFXAlphaCaps As CONST_DDFXALPHACAPSFLAGS
lFXCaps As CONST_DDFXCAPSFLAGS
lMaxHwCodecStretch As Long
lMaxLiveVideoStretch As Long
lMaxOverlayStretch As Long
lMaxVideoPorts As Long
lMaxVisibleOverlays As Long
lMinHwCodecStretch As Long
lMinLiveVideoStretch As Long
lMinOverlayStretch As Long
lNLVBCaps As CONST_DDCAPS1FLAGS
lNLVBCaps2 As CONST_DDCAPS2FLAGS
lNLVBCKeyCaps As CONST_DDCKEYCAPSFLAGS
lNLVBFXCaps As CONST_DDFXCAPSFLAGS
lNLVBRops (0 To 7) As Long
lNumFourCCCodes As Long
lPalCaps As CONST_DDPCAPSFLAGS
lReserved1 As Long
lReserved2 As Long
lReserved3 As Long
lReservedCaps As Long
lRops (0 To 7) As Long
lSSBCaps As CONST_DDCAPS1FLAGS
lSSBCKeyCaps As CONST_DDCKEYCAPSFLAGS
lSSBFXCaps As CONST_DDFXCAPSFLAGS
lSSBRops (0 To 7) As Long
lSVBCaps As CONST_DDCAPS1FLAGS
lSVBCaps2 As CONST_DDCAPS2FLAGS
lSVBCKeyCaps As CONST_DDCKEYCAPSFLAGS
lSVBFXCaps As CONST_DDFXCAPSFLAGS
lSVBRops (0 To 7) As Long

```
    lSVCaps As CONST_DDSTEREOCAPSFLAGS
    lVidMemFree As Long
    lVidMemTotal As Long
    lVSBCaps As CONST_DDCAPS1FLAGS
    lVSBCKeyCaps As CONST_DDCKEYCAPSFLAGS
    lVSBFXCaps As CONST_DDFXCAPSFLAGS
    lVSRops (0 To 7) As Long
    lZBufferBitDepths As Long
End Type
```

ddsCaps
A **DDSCAPS2** type used for further capability descriptions.

lAlignBoundaryDest
Destination rectangle alignment for an overlay surface, in pixels.

lAlignBoundarySrc
Source rectangle alignment for an overlay surface, in pixels.

lAlignSizeDest
Destination rectangle size alignment for an overlay surface, in pixels. Overlay destination rectangles must have a pixel width that is a multiple of this value.

lAlignSizeSrc
Source rectangle size alignment for an overlay surface, in pixels. Overlay source rectangles must have a pixel width that is a multiple of this value.

lAlignStrideAlign
Stride alignment.

lAlphaBltConstBitDepths
DDBD_2, DDBD_4, or DDBD_8. (Indicates 2-, 4-, or 8-bits per pixel.)

lAlphaBltPixelBitDepths
DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

lAlphaBltSurfaceBitDepths
DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

lAlphaOverlayConstBitDepths
DDBD_2, DDBD_4, or DDBD_8. (Indicates 2-, 4-, or 8-bits per pixel.)

lAlphaOverlayPixelBitDepths
DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

lAlphaOverlaySurfaceBitDepths
DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

lCaps
Constants of the **CONST_DDCAPS1FLAGS** describing driver-specific capabilities.

DDCAPS_3D

Indicates that the display hardware has 3-D acceleration.

DDCAPS_ALIGNBOUNDARYDEST
Indicates that DirectDraw will support only those overlay destination rectangles with the x-axis aligned to the **lAlignBoundaryDest** boundaries of the surface.

DDCAPS_ALIGNBOUNDARYSRC
Indicates that DirectDraw will support only those overlay source rectangles with the x-axis aligned to the **lAlignBoundarySrc** boundaries of the surface.

DDCAPS_ALIGNSIZEDEST
Indicates that DirectDraw will support only those overlay destination rectangles whose x-axis sizes, in pixels, are **lAlignSizeDest** multiples.

DDCAPS_ALIGNSIZESRC
Indicates that DirectDraw will support only those overlay source rectangles whose x-axis sizes, in pixels, are **lAlignSizeSrc** multiples.

DDCAPS_ALIGNSTRIDE
Indicates that DirectDraw will create display memory surfaces that have a stride alignment equal to the **lAlignStrideAlign** value.

DDCAPS_ALPHA
Indicates that the display hardware supports alpha-only surfaces. (See alpha channel)

DDCAPS_BANKSWITCHED
Indicates that the display hardware is bank-switched and is potentially very slow at random access to display memory.

DDCAPS_BLT
Indicates that display hardware is capable of blit operations.

DDCAPS_BLTCOLORFILL
Indicates that display hardware is capable of color filling with a blitter.

DDCAPS_BLTDEPTHFILL
Indicates that display hardware is capable of depth filling z-buffers with a blitter.

DDCAPS_BLTFOURCC
Indicates that display hardware is capable of color-space conversions during blit operations.

DDCAPS_BLTQUEUE
Indicates that display hardware is capable of asynchronous blit operations.

DDCAPS_BLTSTRETCH
Indicates that display hardware is capable of stretching during blit operations.

DDCAPS_CANBLTSYSMEM
Indicates that display hardware is capable of blitting to or from system memory.

DDCAPS_CANCLIP
Indicates that display hardware is capable of clipping with blitting.

DDCAPS_CANCLIPSTRETCHED

Indicates that display hardware is capable of clipping while stretch blitting.

DDCAPS_COLORKEY
Supports some form of color key in either overlay or blit operations. More specific color key capability information can be found in the **lCKeyCaps** member.

DDCAPS_COLORKEYHWASSIST
Indicates that the color key is partially hardware assisted. This means that other resources (CPU or video memory) might be used. If this bit is not set, full hardware support is in place.

DDCAPS_GDI
Indicates that display hardware is shared with GDI.

DDCAPS_NOHARDWARE
Indicates that there is no hardware support.

DDCAPS_OVERLAY
Indicates that display hardware supports overlays.

DDCAPS_OVERLAYCANTCLIP
Indicates that display hardware supports overlays but cannot clip them.

DDCAPS_OVERLAYFOURCC
Indicates that overlay hardware is capable of color-space conversions during overlay operations.

DDCAPS_OVERLAYSTRETCH
Indicates that overlay hardware is capable of stretching. The **lMinOverlayStretch** and **lMaxOverlayStretch** members contain valid data.

DDCAPS_PALETTE
Indicates that DirectDraw is capable of creating and supporting DirectDrawPalette objects for more surfaces than only the primary surface.

DDCAPS_PALETTEVSYNC
Indicates that DirectDraw is capable of updating a palette synchronized with the vertical refresh.

DDCAPS_READSCANLINE
Indicates that display hardware is capable of returning the current scan line.

DDCAPS_STEREOVIEW
Indicates that display hardware has stereo vision capabilities.

DDCAPS_VBI
Indicates that display hardware is capable of generating a vertical-blank interrupt.

DDCAPS_ZBLTS
Supports the use of z-buffers with blit operations.

DDCAPS_ZOVERLAYS
Supports the use of the **DirectDrawSurface4.UpdateOverlayZOrder** method as a z-value for overlays to control their layering.

lCaps2

Constants of the **CONST_DDCAPS2FLAGS** enumeration describing more driver-specific capabilities.

DDCAPS2_AUTOFLIPOVERLAY
The overlay can be automatically flipped to the next surface in the flip chain each time a video port VSYNC occurs, allowing the video port and the overlay to double buffer the video without CPU overhead. This option is only valid when the surface is receiving data from a video port. If the video port data is non-interlaced or non-interleaved, it will flip on every VSYNC. If the data is being interleaved in memory, it will flip on every other VSYNC.

DDCAPS2_CANBOBHARDWARE
The overlay hardware can display each field of an interlaced video stream individually.

DDCAPS2_CANBOBINTERLEAVED
The overlay hardware can display each field individually of an interlaced video stream while it is interleaved in memory without causing any artifacts that might normally occur without special hardware support. This option is only valid when the surface is receiving data from a video port and is only valid when the video is zoomed at least two times in the vertical direction.

DDCAPS2_CANBOBNONINTERLEAVED
The overlay hardware can display each field individually of an interlaced video stream while it is not interleaved in memory without causing any artifacts that might normally occur without special hardware support. This option is only valid when the surface is receiving data from a video port and is only valid when the video is zoomed at least two times in the vertical direction.

DDCAPS2_CANCALIBRATEGAMMA
The system has a calibrator installed that can automatically adjust the gamma ramp so that the result will be identical on all systems that have a calibrator. To invoke the calibrator when setting new gamma levels, use the DDSGR_CALIBRATE flag when calling the **DirectDrawGammaControl.SetGammaRamp** method. Calibrating gamma ramps incurs some processing overhead, and should not be used frequently.

DDCAPS2_CANDROPZ16BIT
16-bit RGBZ values can be converted into sixteen-bit RGB values. (The system does not support eight-bit conversions.)

DDCAPS2_CANFLIPODDEVEN
The driver is capable of performing odd and even flip operations, as specified by the DDFLIP_ODD and DDFLIP_EVEN flags used with the **DirectDrawSurface4.Flip** method.

DDCAPS2_CANRENDERWINDOWED
The driver is capable of rendering in windowed mode.

DDCAPS2_CERTIFIED
Indicates that display hardware is certified.

DDCAPS2_COLORCONTROLPRIMARY
The primary surface contains color controls (for instance, gamma)

DDCAPS2_COLORCONTROLOVERLAY
The overlay surface contains color controls (such as brightness, sharpness)

DDCAPS2_COPYFOURCC
Indicates that the driver supports blitting any FOURCC surface to another surface of the same FOURCC.

DDCAPS2_NO2DDURING3DSCENE
Indicates that 2-D operations such as **DirectDrawSurface4.Blt** and **DirectDrawSurface4.Lock** cannot be performed on any surfaces that Direct3D® is using between calls to the **Direct3DDevice3.BeginScene** and **Direct3DDevice3.EndScene** methods.

DDCAPS2_NONLOCALVIDMEM
Indicates that the display driver supports surfaces in non-local video memory.

DDCAPS2_NONLOCALVIDMEMCAPS
Indicates that blit capabilities for non-local video memory surfaces differ from local video memory surfaces. If this flag is present, the DDCAPS2_NONLOCALVIDMEM flag will also be present.

DDCAPS2_NOPAGELOCKREQUIRED
DMA blit operations are supported on system memory surfaces that are not page locked.

DDCAPS2_PRIMARYGAMMA
Supports dynamic gamma ramps for the primary surface. For more information, see Gamma and Color Controls.

DDCAPS2_VIDEOPORT
Indicates that display hardware supports live video.

DDCAPS2_WIDESURFACES
Indicates that the display surfaces supports surfaces wider than the primary surface.

lCKeyCaps
Constants of the **CONST_DDCKEYCAPSFLAGS** enumeration describing color-key capabilities.

DDCKEYCAPS_DESTBLT
Supports transparent blitting with a color key that identifies the replaceable bits of the destination surface for RGB colors.

DDCKEYCAPS_DESTBLTCLRSPACE
Supports transparent blitting with a color space that identifies the replaceable bits of the destination surface for RGB colors.

DDCKEYCAPS_DESTBLTCLRSPACEYUV
Supports transparent blitting with a color space that identifies the replaceable bits of the destination surface for YUV colors.

DDCKEYCAPS_DESTBLTYUV
Supports transparent blitting with a color key that identifies the replaceable bits of the destination surface for YUV colors.

DDCKEYCAPS_DESTOVERLAY

Supports overlaying with color keying of the replaceable bits of the destination surface being overlaid for RGB colors.

DDCKEYCAPS_DESTOVERLAYCLRSPACE
Supports a color space as the color key for the destination of RGB colors.

DDCKEYCAPS_DESTOVERLAYCLRSPACEYUV
Supports a color space as the color key for the destination of YUV colors.

DDCKEYCAPS_DESTOVERLAYONEACTIVE
Supports only one active destination color key value for visible overlay surfaces .

DDCKEYCAPS_DESTOVERLAYYUV
Supports overlaying using color keying of the replaceable bits of the destination surface being overlaid for YUV colors.

DDCKEYCAPS_NOCOSTOVERLAY
Indicates there are no BANDWIDTH trade-offs for using the color key with an overlay.

DDCKEYCAPS_SRCBLT
Supports transparent blitting using the color key for the source with this surface for RGB colors.

DDCKEYCAPS_SRCBLTCLRSPACE
Supports transparent blitting using a color space for the source with this surface for RGB colors.

DDCKEYCAPS_SRCBLTCLRSPACEYUV
Supports transparent blitting using a color space for the source with this surface for YUV colors.

DDCKEYCAPS_SRCBLTYUV
Supports transparent blitting using the color key for the source with this surface for YUV colors.

DDCKEYCAPS_SRCOVERLAY
Supports overlaying using the color key for the source with this overlay surface for RGB colors.

DDCKEYCAPS_SRCOVERLAYCLRSPACE
Supports overlaying using a color space as the source color key for the overlay surface for RGB colors.

DDCKEYCAPS_SRCOVERLAYCLRSPACEYUV
Supports overlaying using a color space as the source color key for the overlay surface for YUV colors.

DDCKEYCAPS_SRCOVERLAYONEACTIVE
Supports only one active source color key value for visible overlay surfaces.

DDCKEYCAPS_SRCOVERLAYYUV
Supports overlaying using the color key for the source with this overlay surface for YUV colors.

lCurrVideoPorts
Current number of live video ports.

lCurrVisibleOverlays
Current number of visible overlays or overlay sprites.

lFXAlphaCaps
Constants of the **CONST_DDFXALPHACAPSFLAGS** enumeration decribing driver-specific alpha capabilities.

DDFXALPHACAPS_BLTALPHAEDGEBLEND
Supports alpha blending around the edge of a source color-keyed surface. Used for blit operations.

DDFXALPHACAPS_BLTALPHAPIXELS
Supports alpha information in pixel format. The bit depth of alpha information in the pixel format can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for blit operations.

DDFXALPHACAPS_BLTALPHAPIXELSNEG
Supports alpha information in pixel format. The bit depth of alpha information in the pixel format can be 1, 2, 4, or 8. The alpha value becomes more transparent as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be used only if DDCAPS_ALPHA is set. Used for blit operations.

DDFXALPHACAPS_BLTALPHASURFACES
Supports alpha-only surfaces. The bit depth of an alpha-only surface can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for blit operations.

DDFXALPHACAPS_BLTALPHASURFACESNEG
Indicates that the alpha channel becomes more transparent as the alpha value increases. The depth of the alpha channel data can be 1, 2, 4, or 8. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be set only if DDCAPS_ALPHA has been set. Used for blit operations.

DDFXALPHACAPS_OVERLAYALPHAEDGEBLEND
Supports alpha blending around the edge of a source color-keyed surface. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHAPIXELS
Supports alpha information in pixel format. The bit depth of alpha information in pixel format can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHAPIXELSNEG
Supports alpha information in pixel format. The bit depth of alpha information in pixel format can be 1, 2, 4, or 8. The alpha value becomes more transparent as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be used only if DDCAPS_ALPHA has been set. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHASURFACES
Supports alpha-only surfaces. The bit depth of an alpha-only surface can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHASURFACESNEG
Indicates that the alpha channel becomes more transparent as the alpha value increases. The depth of the alpha channel data can be 1, 2, 4, or 8. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be used only if DDCAPS_ALPHA has been set. Used for overlays.

lFXCaps
Constants of the **CONST_DDFXCAPSFLAGS** enumeration describing driver-specific stretching and effects capabilities.

DDFXCAPS_BLTALPHA
Supports alpha-blended blit operations.

DDFXCAPS_BLTARITHSTRETCHY
Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink surfaces during a blit operation. Occurs along the y-axis (vertically).

DDFXCAPS_BLTARITHSTRETCHYN
Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink surfaces during a blit operation. Occurs along the y-axis (vertically), and works only for integer stretching (×1, ×2, and so on).

DDFXCAPS_BLTFILTER
Driver can do surface-reconstruction filtering for warped blits.

DDFXCAPS_BLTMIRRORLEFTRIGHT
Supports mirroring left to right in a blit operation.

DDFXCAPS_BLTMIRRORUPDOWN
Supports mirroring top to bottom in a blit operation.

DDFXCAPS_BLTROTATION
Supports arbitrary rotation in a blit operation.

DDFXCAPS_BLTROTATION90
Supports 90-degree rotations in a blit operation.

DDFXCAPS_BLTSHRINKX
Supports arbitrary shrinking of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSHRINKXN
Supports integer shrinking (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSHRINKY
Supports arbitrary shrinking of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_BLTSHRINKYN

Supports integer shrinking (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHX
Supports arbitrary stretching of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHXN
Supports integer stretching (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHY
Supports arbitrary stretching of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHYN
Supports integer stretching (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_BLTTRANSFORM
Supports geometric transformations (or warps) for blitted sprites. Transformations are not currently supported for explicit blit operations.

DDFXCAPS_OVERLAYALPHA
Supports alpha blending for overlay surfaces.

DDFXCAPS_OVERLAYFILTER
Supports surface-reconstruction filtering for warped overlay sprites. Filtering is not currently supported for explicitly displayed overlay surfaces (those displayed with calls to **DirectDrawSurface4.UpdateOverlay**).

DDFXCAPS_OVERLAYMIRRORUPDOWN
Supports mirroring of overlays across the horizontal axis.

DDFXCAPS_OVERLAYSHRINKX
Supports arbitrary shrinking of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKXN
Supports integer shrinking (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKY
Supports arbitrary shrinking of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKYN
Supports integer shrinking (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This

flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSTRETCHX
Supports arbitrary stretching of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHXN
Supports integer stretching (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHY
Supports arbitrary stretching of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHYN
Supports integer stretching (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYTRANSFORM
Supports geometric transformations (or warps) for overlay sprites. Transformations are not currently supported for explicitly displayed overlay surfaces (those displayed with calls to **DirectDrawSurface4.UpdateOverlay**).

lMinHwCodecStretch and **lMaxHwCodecStretch**
These members are obsolete; do not use.

lMinLiveVideoStretch and **lMaxLiveVideoStretch**
These members are obsolete; do not use.

lMinOverlayStretch and **lMaxOverlayStretch**
Minimum and maximum overlay stretch factors multiplied by 1000. For example, 1.3 = 1300.

lMaxVideoPorts
Maximum number of live video ports.

lMaxVisibleOverlays
Maximum number of visible overlays or overlay sprites.

lNLVBCaps
Constants of the **CONST_DDCAPS1FLAGS** enumeration describing driver-specific capabilities for nonlocal-to-local video memory blits. Valid flags are identical to the blit-related flags used with the **lCaps** member.

lNLVBCaps2

Constants of the **CONST_DDCAPS2FLAGS** enumeration describing more driver-specific capabilities for nonlocal-to-local video memory blits. Valid flags are identical to the blit-related flags used with the **lCaps2** member.

lNLVBCKeyCaps
Constants of the **CONST_DDCKEYCAPSFLAGS** enumeration describing driver color-key capabilities for nonlocal-to-local video memory blits. Valid flags are identical to the blit-related flags used with for the **lCKeyCaps** member.

lNLVBFXCaps
Constants of the **CONST_DDFXCAPSFLAGS** enumeration describing driver FX capabilities for nonlocal-to-local video memory blits. Valid flags are identical to the blit-related flags used with the **lFXCaps** member.

lNLVBRops[0 to 7**]**
Raster operations supported for nonlocal-to-local video memory blits.

lNumFourCCCodes
Number of FourCC codes.

lPalCaps
Constants of the **CONST_DDPCAPSFLAGS** enumeration describing palette capabilities.

DDPCAPS_1BIT
Indicates that the index is 1 bit. There are two entries in the color table.

DDPCAPS_2BIT
Indicates that the index is 2 bits. There are four entries in the color table.

DDPCAPS_4BIT
Indicates that the index is 4 bits. There are 16 entries in the color table.

DDPCAPS_8BIT
Indicates that the index is 8 bits. There are 256 entries in the color table.

DDPCAPS_8BITENTRIES
Specifies an index to an 8-bit color index. This field is valid only when used with the DDPCAPS_1BIT, DDPCAPS_2BIT, or DDPCAPS_4BIT capability and when the target surface is in 8 bits per pixel (bpp). Each color entry is 1 byte long and is an index to an 8-bpp palette on the destination surface.

DDPCAPS_ALLOW256
Indicates that this palette can have all 256 entries defined.

DDPCAPS_PRIMARYSURFACE
Indicates that the palette is attached to the primary surface. Changing the palette has an immediate effect on the display unless the DDPCAPS_VSYNC capability is specified and supported.

DDPCAPS_PRIMARYSURFACELEFT
Indicates that the palette is attached to the primary surface on the left. Changing the palette has an immediate effect on the display unless the DDPCAPS_VSYNC capability is specified and supported.

DDPCAPS_VSYNC
Indicates that the palette can be modified synchronously with the monitor's refresh rate.

lReserved1, **lReserved2**, **lReserved3, and IReservedCaps**
Reserved for future use.

lRops[0 To 7**]**
Raster operations supported.

lSSBCaps
Constants of the **CONST_DDCAPS1FLAGS** enumeration describing driver-specific capabilities for system-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with the **lCaps** member.

lSSBCKeyCaps
Constants of the **CONST_DDCKEYCAPSFLAGS** enumeration describing driver color-key capabilities for system-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with for the **lCKeyCaps** member.

lSSBCFXCaps
Constants of the **CONST_DDFXCAPSFLAGS** enumeration describing driver FX capabilities for system-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with the **lFXCaps** member.

lSSBRops[0 To 7**]**
Raster operations supported for system-memory-to-system-memory blits.

lSVBCaps
Constants of the **CONST_DDCAPS1FLAGS** enumeration describing driver-specific capabilities for system-memory-to-display-memory blits. Valid flags are identical to the blit-related flags used with the **lCaps** member.

lSVBCaps2
Constants of the **CONST_DDCAPS2FLAGS** enumeration describing more driver-specific capabilities for system-memory-to-video-memory blits. Valid flags are identical to the blit-related flags used with the **lCaps2** member.

lSVBCKeyCaps
Constants of the **CONST_DDCKEYCAPSFLAGS** enumeration describing driver color-key capabilities for system-memory-to-display-memory blits. Valid flags are identical to the blit-related flags used with for the **lCKeyCaps** member.

lSVBFXCaps
Constants of the **CONST_DDFXCAPSFLAGS** enumeration describing driver FX capabilities for system-memory-to-display-memory blits. Valid flags are identical to the blit-related flags used with the **lFXCaps** member.

lSVBRops[0 To 7**]**
Raster operations supported for system-memory-to-display-memory blits.

lSVCaps
Constants of the **CONST_DDSTEREOCAPSFLAGS** enumeration describing stereo vision capabilities.

DDSVCAPS_ENIGMA
Indicates that the stereo view is accomplished using Enigma encoding.

DDSVCAPS_FLICKER
Indicates that the stereo view is accomplished using high-frequency flickering.

DDSVCAPS_REDBLUE
Indicates that the stereo view is accomplished when the viewer looks at the image through red and blue filters placed over the left and right eyes. All images must adapt their color spaces for this process.

DDSVCAPS_SPLIT
Indicates that the stereo view is accomplished with split-screen technology.

lVidMemFree
Amount of free display memory.

lVidMemTotal
Total amount of display memory.

lVSBCaps
Constants of the **CONST_DDCAPS1FLAGS** enumeration describing driver-specific capabilities for display-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with the **lCaps** member.

lVSBCKeyCaps
Constants of the **CONST_DDCKEYCAPSFLAGS** enumeration describing driver color-key capabilities for display-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with for the **lCKeyCaps** member.

lVSBFXCaps
Constants of the **CONST_DDFXCAPSFLAGS** enumeration describing driver FX capabilities for display-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with the **lFXCaps** member.

lVSBRops[0 To 7**]**
Raster operations supported for display-memory-to-system-memory blits.

lZBufferBitDepths
DDBD_8, DDBD_16, or DDBD_24. (Indicates 8-, 16-, 24-bits per pixel.) 32-bit z-buffers are not supported.

DDCOLORCONTROL

[#] [This is preliminary documentation and subject to change.]

The **DDCOLORCONTROL** type defines the color controls associated with a DirectDrawVideoPortObject, an overlay surface, or a primary surface.

```
Type DDCOLORCONTROL
    lBrightness As Long
    lColorEnable As Long
    lContrast As Long
    lFlags As CONST_DDCOLORFLAGS
    lGamma As Long
    lHue As Long
    lReserved1 As Long
    lSaturation As Long
```

[#] IDH__dx_DDCOLORCONTROL_ddraw_vb

```
    lSharpness As Long
End Type
```

lBrightness
Luminance intensity, in IRE units times 100. The valid range is 0 to 10,000. The default is 750, which translates to 7.5 IRE.

lColorEnable
Flag indicating whether color is used. If this member is zero, color is not used; if it is 1, then color is used. The default value is 1.

lContrast
Relative difference between higher and lower intensity luminance values in IRE units times 100. The valid range is 0 to 20,000. The default value is 10,000 (100 IRE). Higher values of contrast cause darker luminance values to tend towards black, and cause lighter luminance values to tend towards white. Lower values of contrast cause all luminance values to move towards the middle luminance values.

lFlags
Constants of the **CONST_DDCOLORFLAGS** enumeration specifying which type members contain valid data . When the type is returned by the **DirectDrawColorControl.GetColorControls** method, it also indicates which options are supported by the device.

DDCOLOR_BRIGHTNESS
The **lBrightness** member contains valid data.

DDCOLOR_COLORENABLE
The **lColorEnable** member contains valid data.

DDCOLOR_CONTRAST
The **lContrast** member contains valid data.

DDCOLOR_GAMMA
The **lGamma** member contains valid data.

DDCOLOR_HUE
The **lHue** member contains valid data.

DDCOLOR_SATURATION
The **lSaturation** member contains valid data.

DDCOLOR_SHARPNESS
The **lSharpness** member contains valid data.

lGamma
Controls the amount of gamma correction applied to the luminance values. The valid range is 1 to 500 gamma units, with a default of 1.

lHue
Phase relationship of the chrominance components. Hue is specified in degrees and the valid range is -180 to 180. The default is 0.

lReserved1
This member is reserved.

lSaturation

Color intensity, in IRE units times 100. The valid range is 0 to 20,000. The default value is 10,000, which translates to 100 IRE.

lSharpness
Sharpness in arbitrary units. The valid range is 0 to 10. The default value is 5.

DDCOLORKEY

[This is preliminary documentation and subject to change.]

The **DDCOLORKEY** type describes a source color key, destination color key, or color space. A color key is specified if the low and high range values are the same. This type is used with the **DirectDrawSurface4.GetColorKey** and **DirectDrawSurface4.SetColorKey** methods.

```
Type COLORKEY
   high As Long
   low As Long
End Type
```

high
High value, inclusive, of the color range that is to be used as the color key.

low
Low value, inclusive, of the color range that is to be used as the color key.

DDGAMMARAMP

[This is preliminary documentation and subject to change.]

The **DDGAMMARAMP** type contains red, green, and blue ramp data for the **DirectDrawGammaControl.GetGammaRamp** and **DirectDrawGammaControl.SetGammaRamp** methods.

```
Type DDGAMMARAMP
   blue(0 To 255) As Integer
   green(0 To 255) As Integer
   red(0 To 255) As Integer
End Type
```

red, **green**, and **blue**
Describes the red, green, and blue gamma ramps.

See Also

Gamma and Color Controls

IDH__dx_DDCOLORKEY_ddraw_vb
IDH__dx_DDGAMMARAMP_ddraw_vb

DDPIXELFORMAT

[#] [This is preliminary documentation and subject to change.]

The **DDPIXELFORMAT** type describes the pixel format of a DirectDrawSurface object for the **DirectDrawSurface4.GetPixelFormat** method.

```
Type DDPIXELFORMAT
    lAlphaBitDepth As Long
    lBBitMask As Long
    lBumpDuBitMask As Long
    lBumpDvBitMask As Long
    lBumpLuminanceBitMask As Long
    lFlags As CONST_DDPIXELFORMATFLAGS
    lFourCC As Long
    lGBitMask As Long
    lLuminanceAlphaBitMask As Long
    lLuminanceBitCount As Long
    lLuminanceBitMask As Long
    lRBitMask As Long
    lRGBAlphaBitMask As Long
    lRGBBitCount As Long
    lRGBZBitMask As Long
    lStencilBitDepth As Long
    lStencilBitMask As Long
    lUBitMask As Long
    lVBitMask As Long
    lYBitMask As Long
    lYUVAlphaBitMask As Long
    lYUVBitCount As Long
    lYUVZBitMask As Long
    lZBufferBitDepth As Long
End Type
```

lAlphaBitDepth
Alpha channel bit depth (1, 2, 4, or 8) for an alpha-only surface (DDPF_ALPHA). For pixel formats that contain alpha information interleaved with color data (DDPF_ALPHAPIXELS), you must count the bits in the **lRGBAlphaBitMask** member to obtain the bit-depth of the alpha component.

lBBitMask
Mask for blue bits.

lBumpBitCount
Total bump-map bits per pixel in a bump-map surface.

lBumpDuBitMask
Mask for bump-map U_{Δ} bits.

[#] IDH__dx_DDPIXELFORMAT_ddraw_vb

lBumpDvBitMask
Mask for bump-map V_Δ bits.

lBumpLuminanceBitMask
Mask for luminance in a bump-map pixel.

lFlags
Constants of the **CONST_DDPIXELFORMATFLAGS** enumeration describing optional control flags.

DDPF_ALPHA
The pixel format describes an alpha-only surface.

DDPF_ALPHAPIXELS
The surface has alpha channel information in the pixel format.

DDPF_ALPHAPREMULT
The surface uses the premultiplied alpha format. That is, the color components in each pixel are premultiplied by the alpha component.

DDPF_BUMPLUMINANCE
The luminance data in the pixel format is valid, and the **lLuminanceBitMask** member descibes valid luminance bits for a luminance-only or luminance-alpha surface.

DDPF_BUMPDUDV
Bump-map data in the pixel format is valid. Bump-map information is in the **lBumpBitCount**, **lBumpDuBitMask**, **lBumpDvBitMask**, and **lBumpLuminanceBitMask** members.

DDPF_COMPRESSED
The surface will accept pixel data in the specified format and compress it during the write operation.

DDPF_FOURCC
The **lFourCC** member is valid and contains a FOURCC code describing a non-RGB pixel format.

DDPF_LUMINANCE
The pixel format describes a luminance-only or luminance-alpha surface.

DDPF_PALETTEINDEXED1
DDPF_PALETTEINDEXED2
DDPF_PALETTEINDEXED4
DDPF_PALETTEINDEXED8
The surface is 1-, 2-, 4-, or 8-bit color indexed.

DDPF_PALETTEINDEXEDTO8
The surface is 1-, 2-, or 4-bit color indexed to an 8-bit palette.

DDPF_RGB
The RGB data in the pixel format type is valid.

DDPF_RGBTOYUV
The surface will accept RGB data and translate it during the write operation to YUV data. The format of the data to be written will be contained in the pixel format type. The DDPF_RGB flag will be set.

DDPF_STENCILBUFFER

The surface encodes stencil and depth information in each pixel of the z-buffer.

DDPF_YUV
The YUV data in the pixel format type is valid.

DDPF_ZBUFFER
The pixel format describes a z-buffer-only surface.

DDPF_ZPIXELS
The surface contains z information in the pixels.

lFourCC
FourCC code. For more information see, Four Character Codes (FOURCC).

lGBitMask
Mask for green bits.

lRGBAlphaBitMask and **lYUVAlphaBitMask** and **lLuminanceAlphaBitMask**
Masks for alpha channel.

lLuminanceBitCount
Total luminance bits per pixel. This member applies only to luminance-only and luminance-alpha surfaces.

lLuminanceBitMask
Mask for luminance bits.

lRBitMask
Mask for red bits.

lRGBAlphaBitMask and **lYUVAlphaBitMask**
Masks for alpha channel.

lRGBBitCount
RGB bits per pixel (4, 8, 16, 24, or 32).

lRGBZBitMask and **lYUVZBitMask**
Masks for z channel.

lStencilBitDepth
Bit depth of the stencil buffer. This member specifies how many bits are reserved within each pixel of the z-buffer for stencil information (the total number of z-bits is equal to **lZBufferBitDepth** minus **lStencilBitDepth**).

lStencilBitMask
Mask for stencil bits within each z-buffer pixel.

lUBitMask
Mask for U bits.

lVBitMask
Mask for V bits.

lYBitMask
Mask for Y bits.

lYUVBitCount
YUV bits per pixel (4, 8, 16, 24, or 32).

lZBitMask
Mask for z bits.

lZBufferBitDepth
Z-buffer bit depth (8, 16, or 24). 32-bit z-buffers are not supported.

DDSCAPS2

[This is preliminary documentation and subject to change.]

The **DDSCAPS2** type defines the capabilities of a DirectDrawSurface object. This type is part of the **DDSURFACEDESC2** type.

```
Type DDSCAPS2
  lCaps As CONST_DDSURFACECAPSFLAGS   // Surface capabilities
  lCaps2 As CONST_DDSURFACECAPS2FLAGS  // More surface
Capabilities
  lCaps3 As Long  // Not currently used
  lCaps4 As Long  // Not currently used
End Type
```

lCaps
One or more constants of the **CONST_DDSURFACECAPSFLAGS** enumeration representing the capabilities of the surface.

DDSCAPS_3DDEVICE
Indicates that this surface can be used for 3-D rendering. Applications can use this flag to ensure that a device that can only render to a certain heap has off-screen surfaces allocated from the correct heap. If this flag is set for a heap, the surface is not allocated from that heap.

DDSCAPS_ALLOCONLOAD
Not used, ignored by DirectDraw and Direct3D.

DDSCAPS_ALPHA
Indicates that this surface contains alpha-only information.

DDSCAPS_BACKBUFFER
Indicates that this surface is the back buffer of a surface flipping type. Typically, this capability is set by the **CreateSurface** method when the DDSCAPS_FLIP flag is used. Only the surface that immediately precedes the DDSCAPS_FRONTBUFFER surface has this capability set. The other surfaces are identified as back buffers by the presence of the DDSCAPS_FLIP flag, their attachment order, and the absence of the DDSCAPS_FRONTBUFFER and DDSCAPS_BACKBUFFER capabilities. If this capability is sent to the **CreateSurface** method, a stand-alone back buffer is being created. After this method is called, this surface could be attached to a front buffer, another back buffer, or both to form a flipping surface type. For more information, see **DirectDrawSurface4.AddAttachedSurface**. DirectDraw supports an arbitrary number of surfaces in a flipping type.

DDSCAPS_COMPLEX

IDH__dx_DDSCAPS2_ddraw_vb

Indicates that a complex surface is being described. A complex surface results in the creation of more than one surface. The additional surfaces are attached to the root surface. The complex type can be destroyed only by destroying the root.

DDSCAPS_FLIP
Indicates that this surface is a part of a surface flipping type. When this capability is passed to the **CreateSurface** method, a front buffer and one or more back buffers are created. DirectDraw sets the DDSCAPS_FRONTBUFFER bit on the front-buffer surface and the DDSCAPS_BACKBUFFER bit on the surface adjacent to the front-buffer surface. The **lBackBufferCount** member of the **DDSURFACEDESC2** type must be set to at least 1 in order for the method call to succeed. The DDSCAPS_COMPLEX capability must always be set when creating multiple surfaces by using the **CreateSurface** method.

DDSCAPS_FRONTBUFFER
Indicates that this surface is the front buffer of a surface flipping type. This flag is typically set by the **CreateSurface** method when the DDSCAPS_FLIP capability is set. If this capability is sent to the **CreateSurface** method, a stand-alone front buffer is created. This surface will not have the DDSCAPS_FLIP capability. It can be attached to other back buffers to form a flipping type by using **DirectDrawSurface4.AddAttachedSurface**.

DDSCAPS_HWCODEC
Indicates that this surface should be able to have a stream decompressed to it by the hardware.

DDSCAPS_LIVEVIDEO
Indicates that this surface should be able to receive live video.

DDSCAPS_LOCALVIDMEM
Indicates that this surface exists in true, local video memory rather than non-local video memory. If this flag is specified then DDSCAPS_VIDEOMEMORY must be specified as well. This flag cannot be used with the DDSCAPS_NONLOCALVIDMEM flag.

DDSCAPS_MIPMAP
Indicates that this surface is one level of a mipmap. This surface will be attached to other DDSCAPS_MIPMAP surfaces to form the mipmap. This can be done explicitly by creating a number of surfaces and attaching them by using the **DirectDrawSurface4.AddAttachedSurface** method, or implicitly by the **CreateSurface** method. If this capability is set, DDSCAPS_TEXTURE must also be set.

DDSCAPS_MODEX
Indicates that this surface is a 320×200 or 320×240 Mode X surface.

DDSCAPS_NONLOCALVIDMEM
Indicates that this surface exists in non-local video memory rather than true, local video memory. If this flag is specified, then DDSCAPS_VIDEOMEMORY flag must be specified as well. This cannot be used with the DDSCAPS_LOCALVIDMEM flag.

DDSCAPS_OFFSCREENPLAIN
Indicates that this surface is any off-screen surface that is not an overlay, texture, z-buffer, front-buffer, back-buffer, or alpha surface. It is used to identify plain surfaces.

DDSCAPS_OPTIMIZED
Not currently implemented.

DDSCAPS_OVERLAY
Indicates that this surface is an overlay. It may or may not be directly visible depending on whether it is currently being overlaid onto the primary surface. DDSCAPS_VISIBLE can be used to determine if it is being overlaid at the moment.

DDSCAPS_OWNDC
Indicates that this surface will have a device context (DC) association for a long period.

DDSCAPS_PALETTE
Indicates that this device driver allows unique DirectDrawPalette objects to be created and attached to this surface.

DDSCAPS_PRIMARYSURFACE
Indicates the surface is the primary surface. It represents what is visible to the user at the moment.

DDSCAPS_PRIMARYSURFACELEFT
Indicates that this surface is the primary surface for the left eye. It represents what is visible to the user's left eye at the moment. When this surface is created, the surface with the DDSCAPS_PRIMARYSURFACE capability represents what is seen by the user's right eye.

DDSCAPS_RESERVED2
Reserved for future use.

DDSCAPS_STANDARDVGAMODE
Indicates that this surface is a standard VGA mode surface, and not a Mode X surface. This flag cannot be used in combination with the DDSCAPS_MODEX flag.

DDSCAPS_SYSTEMMEMORY
Indicates that this surface memory was allocated in system memory.

DDSCAPS_TEXTURE
Indicates that this surface can be used as a 3-D texture. It does not indicate whether the surface is being used for that purpose.

DDSCAPS_VIDEOMEMORY
Indicates that this surface exists in display memory.

DDSCAPS_VIDEOPORT
Indicates that this surface can receive data from a video port.

DDSCAPS_VISIBLE
Indicates that changes made to this surface are immediately visible. It is always set for the primary surface, as well as for overlays while they are being overlaid and texture maps while they are being textured.

DDSCAPS_WRITEONLY
Indicates that only write access is permitted to the surface. Read access from the surface may generate a general protection (GP) fault, but the read results from this surface will not be meaningful.

DDSCAPS_ZBUFFER
Indicates that this surface is the z-buffer. The z-buffer contains information that cannot be displayed. Instead, it contains bit-depth information that is used to determine which pixels are visible and which are obscured.

lCaps2
Additional surface capabilities. This member can contain one or more of the following capability constants of the **CONST_DDSURFACECAPS2FLAGS** enumeration and can contain an additional flag to indicate how the surface memory was allocated:

Capability flags

DDSCAPS2_HARDWAREDEINTERLACE
Indicates that this surface will receive data from a video port using the de-interlacing hardware. This allows the driver to allocate memory for any extra buffers that may be required. The DDSCAPS_VIDEOPORT and DDSCAPS_OVERLAY flags must also be set.

DDSCAPS2_HINTDYNAMIC
Indicates to the driver that this surface will be locked very frequently (for procedural textures, dynamic lightmaps, etc). This flag can only be used for texture surfaces (DDSCAPS_SYSTEMMEMORY flag set in the **lCaps** member). This flag cannot be used with the DDSCAPS2_HINTSTATIC or DDSCAPS2_OPAQUE flags.

DDSCAPS2_HINTSTATIC
Indicates to the driver that this surface can be reordered or retiled on load. This operation will not change the size of the texture. It is relatively fast and symmetrical, since the application may lock these bits (although it will take a performance hit when doing so). This flag can only be used for texture surfaces (DDSCAPS_SYSTEMMEMORY flag set in the **lCaps** member). This flag cannot be used with the DDSCAPS2_HINTDYNAMIC or DDSCAPS2_OPAQUE flags.

DDSCAPS2_OPAQUE
Indicates to the driver that this surface will never be locked again. The driver is free to optimize this surface by retiling and actual compression. Such a surface cannot be locked or used in blit operations, attempts to lock or blit a surface with this capability will fail. This flag can only be used for texture surfaces (DDSCAPS_SYSTEMMEMORY flag set in the **lCaps** member). This flag cannot be used with the DDSCAPS2_HINTDYNAMIC or DDSCAPS2_HINTSTATIC flags.

DDSCAPS2_TEXTUREMANAGE
Indicates that the client would like this texture surface to be managed by DirectDraw and Direct3D. This flag can only be used for texture surfaces (DDSCAPS_TEXTURE flag set in the **lCaps** member).

lCaps3 and **lCaps4**
Not currently used.

DDSURFACEDESC2

[#] [This is preliminary documentation and subject to change.]

The **DDSURFACEDESC2** type contains a description of a surface. This type is passed to the **DirectDraw4.CreateSurface** method. The relevant members differ for each potential type of surface.

```
Type DDSURFACEDESC2
    ddckCKDestBlt As DDCOLORKEY
    ddckCKDestOverlay As DDCOLORKEY
    ddckCKSrcBlt As DDCOLORKEY
    ddckCKSrcOverlay As DDCOLORKEY
    ddpfPixelFormat As DDPIXELFORMAT
    ddsCaps As DDSCAPS2
    lAlphaBitDepth As Long
    lBackBufferCount As Long
    lFlags As CONST_DDSURFACEDESCFLAGS
    lHeight As Long
    lLinearSize As Long
    lMipMapCount As Long
    lPitch As Long
    lRefreshRate As Long
    lTextureStage As Long
    lWidth As Long
    lZBufferBitDepth As Long
End Type
```

ddckCKDestBlt
DDCOLORKEY type that describes the destination color key for blit operations.

ddckCKDestOverlay
DDCOLORKEY type that describes the destination color key to be used for an overlay surface.

ddckCKSrcBlt
DDCOLORKEY type that describes the source color key for blit operations.

ddckCKSrcOverlay
DDCOLORKEY type that describes the source color key to be used for an overlay surface.

ddpfPixelFormat
DDPIXELFORMAT type that describes the surface's pixel format.

[#] IDH__dx_DDSURFACEDESC2_ddraw_vb

ddsCaps
DDSCAPS2 type containing the surface's capabilities.

lAlphaBitDepth
Depth of alpha buffer.

lBackBufferCount
Number of back buffers.

lFlags
Optional control flags. One or more of the following constants of the **CONST_DDSURFACEDESCFLAGS** enumeration:

DDSD_ALL
Indicates that all input members are valid.

DDSD_ALPHABITDEPTH
Indicates that the **lAlphaBitDepth** member is valid.

DDSD_BACKBUFFERCOUNT
Indicates that the **lBackBufferCount** member is valid.

DDSD_CAPS
Indicates that the **ddsCaps** member is valid.

DDSD_CKDESTBLT
Indicates that the **ddckCKDestBlt** member is valid.

DDSD_CKDESTOVERLAY
Indicates that the **ddckCKDestOverlay** member is valid.

DDSD_CKSRCBLT
Indicates that the **ddckCKSrcBlt** member is valid.

DDSD_CKSRCOVERLAY
Indicates that the **ddckCKSrcOverlay** member is valid.

DDSD_HEIGHT
Indicates that the **lHeight** member is valid.

DDSD_LINEARSIZE
Not used.

DDSD_LPSURFACE
Indicates that the **lpSurface** member is valid.

DDSD_MIPMAPCOUNT
Indicates that the **lMipMapCount** member is valid.

DDSD_PITCH
Indicates that the **lPitch** member is valid.

DDSD_PIXELFORMAT
Indicates that the **ddpfPixelFormat** member is valid.

DDSD_REFRESHRATE
Indicates that the **lRefreshRate** member is valid.

DDSD_TEXTURESTAGE
Indicates that the **lTextureStage** member is valid.

DDSD_WIDTH

Indicates that the **lWidth** member is valid.

DDSD_ZBUFFERBITDEPTH
Indicates that the **lZBufferBitDepth** member is valid.

lHeight and **lWidth**
Dimensions of the surface to be created, in pixels.

lLinearSize
Not currently used.

lMipMapCount
Number of mipmap levels.

lPitch
Distance, in bytes, to the start of next line. When used with the **DirectDrawSurface4.GetSurfaceDesc** method, this is a return value. When creating a surface from existing memory this is an input value that must be a multiple.

lRefreshRate
Refresh rate (used when the display mode is described). The value of 0 indicates an adapter fault.

lTextureStage
Stage identifier used to bind a texture to a specific stage in 3-D device's multitexture cascade. Although not required for all hardware, setting this member is recommended for best performance on the largest variety of 3-D accelerators. Hardware that requires explicitly assigned textures will expose the D3DDEVCAPS_SEPARATETEXTUREMEMORIES 3-D device capability in the **D3DDEVICEDESC** structure that is filled by the **Direct3DDevice3.GetCaps** method.

lZBufferBitDepth
Depth of z-buffer. 32-bit z-buffers are not supported.

Remarks

The **lPitch** member is an output values when calling the **DirectDrawSurface4.GetSurfaceDesc** method. When creating surfaces from existing memory, or updating surface characteristics, these members are input values that describe the pitch and location of memory allocated by the calling application for use by DirectDraw. DirectDraw does not attempt to manage or free memory allocated by the application. For more information, see Creating Client Memory Surfaces and Updating Surface Characteristics.

DXDRIVERINFO

[#] [This is preliminary documentation and subject to change.]

[#] IDH__dx_DXDRIVERINFO_ddraw_vb

The **DXDRIVERINFO** type is used in the enumeration methods for DirectDraw, DirectSound and Direct3D to hold driver information.

```
Type DXDRIVERINFO
   strDescription As String
   strGuid As String
   strName As String
End Type
```

strDescription
The textual description of the DirectDraw device.

strGuid
The GUID that identifies the DirectDraw driver being enumerated.

strName
The name of the DirectDraw driver corresponding to this device.

Remarks

This type is also used in DirectSound and Direct3D.

PALETTEENTRY

[#] [This is preliminary documentation and subject to change.]

The **PALETTEENTRY** type specifies the color and usage of an entry in a logical color palette.

```
Type PALETTEENTRY
   blue As Byte
   flags As Byte
   green As Byte
   red As Byte
End Type
```

blue
Specifies a blue intensity value for the palette entry.

flags
Specifies how the palette entry is to be used. The **peFlags** member may be set to NOTHING or one of the following values:

Value	Meaning
PC_EXPLICIT	Specifies that the low-order word of the logical palette entry designates a hardware palette index. This flag allows the application to show the contents of the

[#] IDH__dx_PALETTEENTRY_ddraw_vb

	display device palette.
PC_NOCOLLAPSE	Specifies that the color be placed in an unused entry in the system palette instead of being matched to an existing color in the system palette. If there are no unused entries in the system palette, the color is matched normally. Once this color is in the system palette, colors in other logical palettes can be matched to this color.
PC_RESERVED	Specifies that the logical palette entry be used for palette animation. This flag prevents other windows from matching colors to the palette entry since the color frequently changes. If an unused system-palette entry is available, the color is placed in that entry. Otherwise, the color is not available for animation.

green
Specifies a green intensity value for the palette entry.

red
Specifies a red intensity value for the palette entry.

RECT

[#] [This is preliminary documentation and subject to change.]

The **RECT** type defines the coordinates of the upper-left and lower-right corners of a rectangle.

```
Type RECT
    Bottom As Long
    Left As Long
    Right As Long
    Top As Long
End Type
```

Bottom
Specifies the y-coordinate of the lower-right corner of the rectangle.

Left
Specifies the x-coordinate of the upper-left corner of the rectangle.

Right
Specifies the x-coordinate of the lower-right corner of the rectangle.

Top

[#] IDH__dx_RECT_ddraw_vb

Specifies the y-coordinate of the upper-left corner of the rectangle.

Enumerations

[This is preliminary documentation and subject to change.]

DirectDraw uses enumerations to group constants and to take advantage of the statement completion feature of Visual Basic. The enumerations used in DirectSound are:

- **CONST_DDBITDEPTHFLAGS**
- **CONST_DDBLTFASTFLAGS**
- **CONST_DDBLTFLAGS**
- **CONST_DDBLTFXFLAGS**
- **CONST_DDCAPS1FLAGS**
- **CONST_DDCAPS2FLAGS**
- **CONST_DDCKEYCAPSFLAGS**
- **CONST_DDCKEYFLAGS**
- **CONST_DDCOLORFLAGS**
- **CONST_DDEDMFLAGS**
- **CONST_DDENUMOVERLAYZFLAGS**
- **CONST_DDENUMSURFACESFLAGS**
- **CONST_DDFLIPFLAGS**
- **CONST_DDFXALPHACAPSFLAGS**
- **CONST_DDFXCAPSFLAGS**
- **CONST_DDGBSFLAGS**
- **CONST_DDGFSFLAGS**
- **CONST_DDLOCKFLAGS**
- **CONST_DDOVERFLAGS**
- **CONST_DDOVERLAYFXFLAGS**
- **CONST_DDOVERZFLAGS**
- **CONST_DDPCAPSFLAGS**
- **CONST_DDPIXELFORMATFLAGS**
- **CONST_DDRAW**
- **CONST_DDSCLFLAGS**
- **CONST_DDSDMFLAGS**
- **CONST_DDSGRFLAGS**
- **CONST_DDSTEREOCAPSFLAGS**
- **CONST_DDSURFACECAPS2FLAGS**

- **CONST_DDSURFACECAPSFLAGS**
- **CONST_DDSURFACEDESCFLAGS**
- **CONST_DDWAITVBFLAGS**

CONST_DDBITDEPTHFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_DDBITDEPTHFLAGS** enumeration is used to specify the bit depth.

```
Enum CONST_DDBITDEPTHFLAGS
   DDBD_1 = 16384
   DDBD_16 = 1024
   DDBD_2 = 8192
   DDBD_24 = 512
   DDBD_32 = 256
   DDBD_4 = 4096
   DDBD_8 = 2048
End Enum
```

DDBD_1 to 32
: The bits per pixel.

CONST_DDBLTFASTFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_DDBLTFASTFLAGS** enumeration is used in the trans parameter of the **DirectDrawSurface4.BltFast** method to determine the type of transfer.

```
Enum CONST_DDBLTFASTFLAGS
   DDBLTFAST_DESTCOLORKEY = 2
   DDBLTFAST_NOCOLORKEY = 0
   DDBLTFAST_SRCCOLORKEY = 1
   DDBLTFAST_WAIT = 16
End Enum
```

DDBLTFAST_DESTCOLORKEY
: Specifies a transparent blit that uses the destination's color key.

DDBLTFAST_NOCOLORKEY
: Specifies a normal copy blit with no transparency.

DDBLTFAST_SRCCOLORKEY
: Specifies a transparent blit that uses the source's color key.

DDBLTFAST_WAIT

IDH__dx_CONST_DDBITDEPTHFLAGS_ddraw_vb
IDH__dx_CONST_DDBLTFASTFLAGS_ddraw_vb

Postpones the DDERR_WASSTILLDRAWING message if the blitter is busy, and returns as soon as the blit can be set up or another error occurs.

CONST_DDBLTFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDBLTFLAGS** enumeration is used in the flags parameter of the **DirectDrawSurface4.Blt** and **DirectDrawSurface4.BltFx** methods to determine the valid members of the associated **DDBLTFX** type. The **DDBLTFX** type specifies color key information or request special behavior from the methods.

```
Enum CONST_DDBLTFLAGS
    DDBLT_ASYNC = 512
    DDBLT_COLORFILL = 1024
    DDBLT_DDFX = 2048
    DDBLT_DDROPS = 4096
    DDBLT_KEYDEST = 8192
    DDBLT_KEYDESTOVERRIDE = 16384
    DDBLT_KEYSRC = 32768
    DDBLT_KEYSRCOVERRIDE = 65536
    DDBLT_ROP = 131072
    DDBLT_ROTATIONANGLE = 262144
    DDBLT_WAIT = 16777216
End Enum
```

Validation flags

DDBLT_COLORFILL

Uses the **lFill** member of the **DDBLTFX** structure as the RGB color that fills the destination rectangle on the destination surface.

DDBLT_DDFX

Uses the **lDDFX** member of the **DDBLTFX** structure to specify the effects to use for this blit.

DDBLT_DDROPS

Uses the **lROP** member of the **DDBLTFX** structure to specify the raster operations (ROPS) that are not part of the Win32 API.

DDBLT_KEYDESTOVERRIDE

Uses the **ddckDestColorKey_high** and **ddckDestColorKey_low** members of the **DDBLTFX** structure as the color key for the destination surface.

DDBLT_KEYSRCOVERRIDE

Uses the **ddckSrcColorKey_high** and **ddckSrcColorKey_low** members of the **DDBLTFX** structure as the color key for the source surface.

DDBLT_ROP

[#] IDH__dx_CONST_DDBLTFLAGS_ddraw_vb

Uses the **lROP** member of the **DDBLTFX** structure for the ROP for this blit. These ROPs are the same as those defined in the Win32 API.

DDBLT_ROTATIONANGLE
Uses the **lRotationAngle** member of the **DDBLTFX** structure as the rotation angle (specified in 1/100th of a degree) for the surface.

Color key flags

DDBLT_KEYDEST
Uses the color key associated with the destination surface.

DDBLT_KEYSRC
Uses the color key associated with the source surface.

Behavior flags

DDBLT_ASYNC
Performs this blit asynchronously through the FIFO in the order received. If no room is available in the FIFO hardware, the call fails.

DDBLT_WAIT
Postpones the DDERR_WASSTILLDRAWING return value if the blitter is busy, and returns as soon as the blit can be set up or another error occurs.

CONST_DDBLTFXFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDBLTFXFLAGS** enumeration is used in the **lDDFX** member of the **DDBLTFX** type to specify the type of FX operation.

```
Enum CONST_DDBLTFXFLAGS
    DDBLTFX_ARITHSTRETCHY = 1
    DDBLTFX_MIRRORLEFTRIGHT = 2
    DDBLTFX_MIRRORUPDOWN = 4
    DDBLTFX_NOTEARING = 8
    DDBLTFX_ROTATE180 = 16
    DDBLTFX_ROTATE270 = 32
    DDBLTFX_ROTATE90 = 64
    DDBLTFX_ZBUFFERBASEDEST = 256
    DDBLTFX_ZBUFFERRANGE = 128
End Enum
```

DDBLTFX_ARITHSTRETCHY
Uses arithmetic stretching along the y-axis for this blit.

DDBLTFX_MIRRORLEFTRIGHT
Turns the surface on its y-axis. This blit mirrors the surface from left to right.

DDBLTFX_MIRRORUPDOWN
Turns the surface on its x-axis. This blit mirrors the surface from top to bottom.

[#] IDH__dx_CONST_DDBLTFXFLAGS_ddraw_vb

DDBLTFX_NOTEARING
Schedules this blit to avoid tearing.

DDBLTFX_ROTATE180
Rotates the surface 180 degrees clockwise during this blit.

DDBLTFX_ROTATE270
Rotates the surface 270 degrees clockwise during this blit.

DDBLTFX_ROTATE90
Rotates the surface 90 degrees clockwise during this blit.

DDBLTFX_ZBUFFERBASEDEST
Adds the **lZBufferBaseDest** member of the **DDBLTFX** type to each of the source z-values before comparing them with the destination z-values during this z-blit.

DDBLTFX_ZBUFFERRANGE
Uses the **lZBufferLow** and **lZBufferHigh** members of the **DDBLTFX** type as range values to specify limits to the bits copied from a source surface during this z-blit.

CONST_DDCAPS1FLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDCAPS1FLAGS** enumeration is used by the **lCaps**, **lNLVBCaps**, **lSSBCaps**, **lSVBCaps**, and the **lVSBCaps** members of the **DDCAPS** type to describe hardware capabilities.

```
Enum CONST_DDCAPS1FLAGS
    DDCAPS_3D = 1
    DDCAPS_ALIGNBOUNDARYDEST = 2
    DDCAPS_ALIGNBOUNDARYSRC = 8
    DDCAPS_ALIGNSIZEDEST = 4
    DDCAPS_ALIGNSIZESRC = 16
    DDCAPS_ALIGNSTRIDE = 32
    DDCAPS_ALPHA = 8388608
    DDCAPS_BANKSWITCHED = 134217728
    DDCAPS_BLT = 64
    DDCAPS_BLTCOLORFILL = 67108864
    DDCAPS_BLTDEPTHFILL = 268435456
    DDCAPS_BLTFOURCC = 256
    DDCAPS_BLTQUEUE = 128
    DDCAPS_BLTSTRETCH = 512
    DDCAPS_CANBLTSYSMEM = -2147483648
    DDCAPS_CANCLIP = 536870912
    DDCAPS_CANCLIPSTRETCHED = 1073741824
    DDCAPS_COLORKEY = 4194304
```

[#] IDH__dx_CONST_DDCAPS1FLAGS_ddraw_vb

```
    DDCAPS_COLORKEYHWASSIST = 16777216
    DDCAPS_GDI = 1024
    DDCAPS_NOHARDWARE = 33554432
    DDCAPS_OVERLAY = 2048
    DDCAPS_OVERLAYCANTCLIP = 4096
    DDCAPS_OVERLAYFOURCC = 8192
    DDCAPS_OVERLAYSTRETCH = 16384
    DDCAPS_PALETTE = 32768
    DDCAPS_PALETTEVSYNC = 65536
    DDCAPS_READSCANLINE = 131072
    DDCAPS_STEREOVIEW = 262144
    DDCAPS_VBI = 524288
    DDCAPS_ZBLTS = 1048576
    DDCAPS_ZOVERLAYS = 2097152
End Enum
```

DDCAPS_3D
: Indicates that the display hardware has 3-D acceleration.

DDCAPS_ALIGNBOUNDARYDEST
: Indicates that DirectDraw will support only those overlay destination rectangles with the x-axis aligned to the **lAlignBoundaryDest** boundaries of the surface.

DDCAPS_ALIGNBOUNDARYSRC
: Indicates that DirectDraw will support only those overlay source rectangles with the x-axis aligned to the **lAlignBoundarySrc** boundaries of the surface.

DDCAPS_ALIGNSIZEDEST
: Indicates that DirectDraw will support only those overlay destination rectangles whose x-axis sizes, in pixels, are **lAlignSizeDest** multiples.

DDCAPS_ALIGNSIZESRC
: Indicates that DirectDraw will support only those overlay source rectangles whose x-axis sizes, in pixels, are **lAlignSizeSrc** multiples.

DDCAPS_ALIGNSTRIDE
: Indicates that DirectDraw will create display memory surfaces that have a stride alignment equal to the **lAlignStrideAlign** value.

DDCAPS_ALPHA
: Indicates that the display hardware supports alpha-only surfaces. (See alpha channel)

DDCAPS_BANKSWITCHED
: Indicates that the display hardware is bank-switched and is potentially very slow at random access to display memory.

DDCAPS_BLT
: Indicates that display hardware is capable of blit operations.

DDCAPS_BLTCOLORFILL
: Indicates that display hardware is capable of color filling with a blitter.

DDCAPS_BLTDEPTHFILL

Indicates that display hardware is capable of depth filling z-buffers with a blitter.

DDCAPS_BLTFOURCC
Indicates that display hardware is capable of color-space conversions during blit operations.

DDCAPS_BLTQUEUE
Indicates that display hardware is capable of asynchronous blit operations.

DDCAPS_BLTSTRETCH
Indicates that display hardware is capable of stretching during blit operations.

DDCAPS_CANBLTSYSMEM
Indicates that display hardware is capable of blitting to or from system memory.

DDCAPS_CANCLIP
Indicates that display hardware is capable of clipping with blitting.

DDCAPS_CANCLIPSTRETCHED
Indicates that display hardware is capable of clipping while stretch blitting.

DDCAPS_COLORKEY
Supports some form of color key in either overlay or blit operations. More specific color key capability information can be found in the **lCKeyCaps** member.

DDCAPS_COLORKEYHWASSIST
Indicates that the color key is partially hardware assisted. This means that other resources (CPU or video memory) might be used. If this bit is not set, full hardware support is in place.

DDCAPS_GDI
Indicates that display hardware is shared with GDI.

DDCAPS_NOHARDWARE
Indicates that there is no hardware support.

DDCAPS_OVERLAY
Indicates that display hardware supports overlays.

DDCAPS_OVERLAYCANTCLIP
Indicates that display hardware supports overlays but cannot clip them.

DDCAPS_OVERLAYFOURCC
Indicates that overlay hardware is capable of color-space conversions during overlay operations.

DDCAPS_OVERLAYSTRETCH
Indicates that overlay hardware is capable of stretching. The **lMinOverlayStretch** and **lMaxOverlayStretch** members contain valid data.

DDCAPS_PALETTE
Indicates that DirectDraw is capable of creating and supporting DirectDrawPalette objects for more surfaces than only the primary surface.

DDCAPS_PALETTEVSYNC
Indicates that DirectDraw is capable of updating a palette synchronized with the vertical refresh.

DDCAPS_READSCANLINE
Indicates that display hardware is capable of returning the current scan line.

DDCAPS_STEREOVIEW
Indicates that display hardware has stereo vision capabilities.

DDCAPS_VBI
Indicates that display hardware is capable of generating a vertical-blank interrupt.

DDCAPS_ZBLTS
Supports the use of z-buffers with blit operations.

DDCAPS_ZOVERLAYS
Supports the use of the **DirectDrawSurface4.UpdateOverlayZOrder** method as a z-value for overlays to control their layering.

CONST_DDCAPS2FLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDCAPS2FLAGS** enumeration is used in the **lCaps2**, **lNLVBCaps2**, **lSVBCaps2** members of the **DDCAPS** type to describe additional driver-specific capabilities.

```
Enum CONST_DDCAPS2FLAGS
    DDCAPS2_AUTOFLIPOVERLAY = 8
    DDCAPS2_CANBOBINTERLEAVED = 16
    DDCAPS2_CANBOBNONINTERLEAVED = 32
    DDCAPS2_CANDROPZ16BIT = 256
    DDCAPS2_CANFLIPODDEVEN = 8192
    DDCAPS2_CERTIFIED = 1
    DDCAPS2_COLORCONTROLOVERLAY = 64
    DDCAPS2_COLORCONTROLPRIMARY = 128
    DDCAPS2_NO2DDURING3DSCENE = 2
    DDCAPS2_NONLOCALVIDMEM = 512
    DDCAPS2_NONLOCALVIDMEMCAPS = 1024
    DDCAPS2_NOPAGELOCKREQUIRED = 2048
    DDCAPS2_VIDEOPORT = 4
    DDCAPS2_WIDESURFACES = 4096
End Enum
```

DDCAPS2_AUTOFLIPOVERLAY
The overlay can be automatically flipped to the next surface in the flip chain each time a video port VSYNC occurs, allowing the video port and the overlay to double buffer the video without CPU overhead. This option is only valid when the surface is receiving data from a video port. If the video port data is

---

[#] IDH__dx_CONST_DDCAPS2FLAGS_ddraw_vb

non-interlaced or non-interleaved, it will flip on every VSYNC. If the data is being interleaved in memory, it will flip on every other VSYNC.

DDCAPS2_CANBOBHARDWARE
The overlay hardware can display each field of an interlaced video stream individually.

DDCAPS2_CANBOBINTERLEAVED
The overlay hardware can display each field individually of an interlaced video stream while it is interleaved in memory without causing any artifacts that might normally occur without special hardware support. This option is only valid when the surface is receiving data from a video port and is only valid when the video is zoomed at least two times in the vertical direction.

DDCAPS2_CANBOBNONINTERLEAVED
The overlay hardware can display each field individually of an interlaced video stream while it is not interleaved in memory without causing any artifacts that might normally occur without special hardware support. This option is only valid when the surface is receiving data from a video port and is only valid when the video is zoomed at least two times in the vertical direction.

DDCAPS2_CANCALIBRATEGAMMA
The system has a calibrator installed that can automatically adjust the gamma ramp so that the result will be identical on all systems that have a calibrator. To invoke the calibrator when setting new gamma levels, use the DDSGR_CALIBRATE flag when calling the **DirectDrawGammaControl.SetGammaRamp** method. Calibrating gamma ramps incurs some processing overhead, and should not be used frequently.

DDCAPS2_CANDROPZ16BIT
16-bit RGBZ values can be converted into sixteen-bit RGB values. (The system does not support eight-bit conversions.)

DDCAPS2_CANFLIPODDEVEN
The driver is capable of performing odd and even flip operations, as specified by the DDFLIP_ODD and DDFLIP_EVEN flags used with the **DirectDrawSurface4.Flip** method.

DDCAPS2_CANRENDERWINDOWED
The driver is capable of rendering in windowed mode.

DDCAPS2_CERTIFIED
Indicates that display hardware is certified.

DDCAPS2_COLORCONTROLOVERLAY
The overlay surface contains color controls (such as brightness, sharpness)

DDCAPS2_COLORCONTROLPRIMARY
The primary surface contains color controls (for instance, gamma)

DDCAPS2_COPYFOURCC
Indicates that the driver supports blitting any FOURCC surface to another surface of the same FOURCC.

DDCAPS2_NO2DDURING3DSCENE

Indicates that 2-D operations such as **DirectDrawSurface4.Blt** and **DirectDrawSurface4.Lock** cannot be performed on any surfaces that Direct3D® is using between calls to the **Direct3DDevice3.BeginScene** and **Direct3DDevice3.EndScene** methods.

DDCAPS2_NONLOCALVIDMEM
Indicates that the display driver supports surfaces in non-local video memory.

DDCAPS2_NONLOCALVIDMEMCAPS
Indicates that blit capabilities for non-local video memory surfaces differ from local video memory surfaces. If this flag is present, the DDCAPS2_NONLOCALVIDMEM flag will also be present.

DDCAPS2_NOPAGELOCKREQUIRED
DMA blit operations are supported on system memory surfaces that are not page locked.

DDCAPS2_PRIMARYGAMMA
Supports dynamic gamma ramps for the primary surface.

DDCAPS2_VIDEOPORT
Indicates that display hardware supports live video.

DDCAPS2_WIDESURFACES
Indicates that the display surfaces supports surfaces wider than the primary surface.

CONST_DDCKEYCAPSFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDCKEYCAPSFLAGS** enumeration is used by the lCKeyCaps, lNLVBCKeyCaps, lSSBCKeyCaps, lSVBCKeyCaps, and lVSBCKeyCaps members of the DDCAPS type to describe the color-key capabilities of the hardware.

```
Enum CONST_DDCKEYCAPSFLAGS
    DDCKEYCAPS_DESTBLT = 1
    DDCKEYCAPS_DESTBLTCLRSPACE = 2
    DDCKEYCAPS_DESTBLTCLRSPACEYUV = 4
    DDCKEYCAPS_DESTBLTYUV = 8
    DDCKEYCAPS_DESTOVERLAY = 16
    DDCKEYCAPS_DESTOVERLAYCLRSPACE = 32
    DDCKEYCAPS_DESTOVERLAYCLRSPACEYUV = 64
    DDCKEYCAPS_DESTOVERLAYONEACTIVE = 128
    DDCKEYCAPS_DESTOVERLAYYUV = 256
    DDCKEYCAPS_NOCOSTOVERLAY = 262144
    DDCKEYCAPS_SRCBLT = 512
    DDCKEYCAPS_SRCBLTCLRSPACE = 1024
    DDCKEYCAPS_SRCBLTCLRSPACEYUV = 2048
    DDCKEYCAPS_SRCBLTYUV = 4096
```

[#] IDH__dx_CONST_DDCKEYCAPSFLAGS_ddraw_vb

```
    DDCKEYCAPS_SRCOVERLAY = 8192
    DDCKEYCAPS_SRCOVERLAYCLRSPACE = 16384
    DDCKEYCAPS_SRCOVERLAYCLRSPACEYUV = 32768
    DDCKEYCAPS_SRCOVERLAYONEACTIVE = 65536
    DDCKEYCAPS_SRCOVERLAYYUV = 131072
End Enum
```

DDCKEYCAPS_DESTBLT
: Supports transparent blitting with a color key that identifies the replaceable bits of the destination surface for RGB colors.

DDCKEYCAPS_DESTBLTCLRSPACE
: Supports transparent blitting with a color space that identifies the replaceable bits of the destination surface for RGB colors.

DDCKEYCAPS_DESTBLTCLRSPACEYUV
: Supports transparent blitting with a color space that identifies the replaceable bits of the destination surface for YUV colors.

DDCKEYCAPS_DESTBLTYUV
: Supports transparent blitting with a color key that identifies the replaceable bits of the destination surface for YUV colors.

DDCKEYCAPS_DESTOVERLAY
: Supports overlaying with color keying of the replaceable bits of the destination surface being overlaid for RGB colors.

DDCKEYCAPS_DESTOVERLAYCLRSPACE
: Supports a color space as the color key for the destination of RGB colors.

DDCKEYCAPS_DESTOVERLAYCLRSPACEYUV
: Supports a color space as the color key for the destination of YUV colors.

DDCKEYCAPS_DESTOVERLAYONEACTIVE
: Supports only one active destination color key value for visible overlay surfaces .

DDCKEYCAPS_DESTOVERLAYYUV
: Supports overlaying using color keying of the replaceable bits of the destination surface being overlaid for YUV colors.

DDCKEYCAPS_NOCOSTOVERLAY
: Indicates there are no BANDWIDTH trade-offs for using the color key with an overlay.

DDCKEYCAPS_SRCBLT
: Supports transparent blitting using the color key for the source with this surface for RGB colors.

DDCKEYCAPS_SRCBLTCLRSPACE
: Supports transparent blitting using a color space for the source with this surface for RGB colors.

DDCKEYCAPS_SRCBLTCLRSPACEYUV
: Supports transparent blitting using a color space for the source with this surface for YUV colors.

DDCKEYCAPS_SRCBLTYUV
Supports transparent blitting using the color key for the source with this surface for YUV colors.

DDCKEYCAPS_SRCOVERLAY
Supports overlaying using the color key for the source with this overlay surface for RGB colors.

DDCKEYCAPS_SRCOVERLAYCLRSPACE
Supports overlaying using a color space as the source color key for the overlay surface for RGB colors.

DDCKEYCAPS_SRCOVERLAYCLRSPACEYUV
Supports overlaying using a color space as the source color key for the overlay surface for YUV colors.

DDCKEYCAPS_SRCOVERLAYONEACTIVE
Supports only one active source color key value for visible overlay surfaces.

DDCKEYCAPS_SRCOVERLAYYUV
Supports overlaying using the color key for the source with this overlay surface for YUV colors.

CONST_DDCKEYFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_DDCKEYFLAGS** enumeration is used in the *flags* parameter of the **DirectDrawSurface4.SetColorKey** method to specify the type of color key requested.

```
Enum CONST_DDCKEYFLAGS
    DDCKEY_COLORSPACE = 1
    DDCKEY_DESTBLT = 2
    DDCKEY_DESTOVERLAY = 4
    DDCKEY_SRCBLT = 8
    DDCKEY_SRCOVERLAY = 16
End Enum
```

DDCKEY_COLORSPACE
Set if the type contains a color space. Not set if the type contains a single color key.

DDCKEY_DESTBLT
Set if the type specifies a color key or color space to be used as a destination color key for blit operations.

DDCKEY_DESTOVERLAY
Set if the type specifies a color key or color space to be used as a destination color key for overlay operations.

DDCKEY_SRCBLT

IDH__dx_CONST_DDCKEYFLAGS_ddraw_vb

Set if the type specifies a color key or color space to be used as a source color key for blit operations.

DDCKEY_SRCOVERLAY
Set if the type specifies a color key or color space to be used as a source color key for overlay operations.

CONST_DDCOLORFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_DDCOLORFLAGS** enumeration is used in the **lFlags** member of the **DDCOLORCONTROL** type to specify which members of the **DDCOLORCONTROL** type contain valid data.

```
Enum CONST_DDCOLORFLAGS
    DDCOLOR_BRIGHTNESS = 1
    DDCOLOR_COLORENABLE = 64
    DDCOLOR_CONTRAST = 2
    DDCOLOR_GAMMA = 32
    DDCOLOR_HUE = 4
    DDCOLOR_SATURATION = 8
    DDCOLOR_SHARPNESS = 16
End Enum
```

DDCOLOR_BRIGHTNESS
The **lBrightness** member contains valid data.

DDCOLOR_COLORENABLE
The **lColorEnable** member contains valid data.

DDCOLOR_CONTRAST
The **lContrast** member contains valid data.

DDCOLOR_GAMMA
The **lGamma** member contains valid data.

DDCOLOR_HUE
The **lHue** member contains valid data.

DDCOLOR_SATURATION
The **lSaturation** member contains valid data.

DDCOLOR_SHARPNESS
The **lSharpness** member contains valid data.

CONST_DDEDMFLAGS

[This is preliminary documentation and subject to change.]

IDH__dx_CONST_DDCOLORFLAGS_ddraw_vb
IDH__dx_CONST_DDEDMFLAGS_ddraw_vb

The **CONST_DDEDMFLAGS** enumeration is used in the *flags* parameter of the **DirectDraw4.GetDisplayModesEnum** method to specify the type of enumeration.

```
Enum CONST_DDEDMFLAGS
   DDEDM_DEFAULT = 0
   DDEDM_REFRESHRATES = 1
   DDEDM_STANDARDVGAMODES = 2
End Enum
```

DDEDM_REFRESHRATES
: Enumerates modes with different refresh rates. This guarantees that a particular mode will be enumerated only once. This flag specifies whether the refresh rate is taken into account when determining if a mode is unique.

DDEDM_STANDARDVGAMODES
: Enumerates Mode 13 in addition to the 320x200x8 Mode X mode.

CONST_DDENUMOVERLAYZFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDENUMOVERLAYZFLAGS** enumeration is used by the *flags* parameter of the **DirectDrawSurface4.GetOverlayZOrdersEnum** method to describe how overlays should be enumerated.

```
Enum CONST_DDENUMOVERLAYZFLAGS
   DDENUMOVERLAYZ_BACKTOFRONT = 0
   DDENUMOVERLAYZ_FRONTTOBACK = 1
End Enum
```

DDENUMOVERLAYZ_BACKTOFRONT
: Enumerates overlays back to front.

DDENUMOVERLAYZ_FRONTTOBACK
: Enumerates overlays front to back.

CONST_DDENUMSURFACESFLAGS

[#] [This is preliminary documentation and subject to change.]

IDH__dx_CONST_DDENUMOVERLAYZFLAGS_ddraw_vb
IDH__dx_CONST_DDENUMSURFACESFLAGS_ddraw_vb

The **CONST_DDENUMSURFACESFLAGS** enumeration is used by the **DirectDraw4.GetSurfacesEnum** method to control how the method enumerates attached surfaces.

```
Enum CONST_DDENUMSURFACESFLAGS
    DDENUMSURFACES_ALL = 1
    DDENUMSURFACES_CANBECREATED = 8
    DDENUMSURFACES_DOESEXIST = 16
    DDENUMSURFACES_MATCH = 2
    DDENUMSURFACES_NOMATCH = 4
End Enum
```

Search type flags

DDENUMSURFACES_CANBECREATED
: Enumerates the first surface that can be created and meets the search criterion. This flag can only be used with the DDENUMSURFACES_MATCH flag.

DDENUMSURFACES_DOESEXIST
: Enumerates the already existing surfaces that meet the search criterion.

Matching flags

DDENUMSURFACES_ALL
: Enumerates all of the surfaces that meet the search criterion. This flag can only be used with the DDENUMSURFACES_DOESEXIST search type flag.

DDENUMSURFACES_MATCH
: Searches for any surface that matches the surface description.

DDENUMSURFACES_NOMATCH
: Searches for any surface that does not match the surface description.

CONST_DDFLIPFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDFLIPFLAGS** enumeration is used in the *flags* parameter of the **DirectDrawSurface4.Flip** method to specify flip options.

```
Enum CONST_DDFLIPFLAGS
    DDFLIP_EVEN = 2
    DDFLIP_INTERFVAL2 = 536870912
    DDFLIP_INTERFVAL3 = 805306368
    DDFLIP_INTERFVAL4 = 1073741824
    DDFLIP_NOVSYNC = 8
    DDFLIP_ODD = 4
    DDFLIP_WAIT = 1
End Enum
```

[#] IDH__dx_CONST_DDFLIPFLAGS_ddraw_vb

DDFLIP_EVEN
For use only when displaying video in an overlay surface. The new surface contains data from the even field of a video signal. This flag cannot be used with the DDFLIP_ODD flag.

DDFLIP_ODD
For use only when displaying video in an overlay surface. The new surface contains data from the odd field of a video signal. This flag cannot be used with the DDFLIP_EVEN flag.

DDFLIP_WAIT
Typically, if the flip cannot be set up because the state of the display hardware is not appropriate, the DDERR_WASSTILLDRAWING error returns immediately and no flip occurs. Setting this flag causes the method to continue trying to flip if it receives the DDERR_WASSTILLDRAWING error from the HAL. The method does not return until the flipping operation has been successfully set up, or another error, such as DDERR_SURFACEBUSY, is returned.

CONST_DDFXALPHACAPSFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDFXALPHACAPSFLAGS** enumeration is used in the **lFXAlphaCaps** member of the **DDCAPS** type to describe driver-specific alpha capabilities.

```
Enum CONST_DDFXALPHACAPSFLAGS
    DDFXALPHACAPS_BLTALPHAEDGEBLEND = 1
    DDFXALPHACAPS_BLTALPHAPIXELS = 2
    DDFXALPHACAPS_BLTALPHAPIXELSNEG = 4
    DDFXALPHACAPS_BLTALPHASURFACES = 8
    DDFXALPHACAPS_BLTALPHASURFACESNEG = 16
    DDFXALPHACAPS_OVERLAYALPHAEDGEBLEND = 32
    DDFXALPHACAPS_OVERLAYALPHAPIXELS = 64
    DDFXALPHACAPS_OVERLAYALPHAPIXELSNEG = 128
    DDFXALPHACAPS_OVERLAYALPHASURFACES = 256
    DDFXALPHACAPS_OVERLAYALPHASURFACESNEG = 512
End Enum
```

DDFXALPHACAPS_BLTALPHAEDGEBLEND
Supports alpha blending around the edge of a source color-keyed surface. Used for blit operations.

DDFXALPHACAPS_BLTALPHAPIXELS

[#] IDH__dx_CONST_DDFXALPHACAPSFLAGS_ddraw_vb

Supports alpha information in pixel format. The bit depth of alpha information in the pixel format can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for blit operations.

DDFXALPHACAPS_BLTALPHAPIXELSNEG
Supports alpha information in pixel format. The bit depth of alpha information in the pixel format can be 1, 2, 4, or 8. The alpha value becomes more transparent as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be used only if DDCAPS_ALPHA is set. Used for blit operations.

DDFXALPHACAPS_BLTALPHASURFACES
Supports alpha-only surfaces. The bit depth of an alpha-only surface can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for blit operations.

DDFXALPHACAPS_BLTALPHASURFACESNEG
Indicates that the alpha channel becomes more transparent as the alpha value increases. The depth of the alpha channel data can be 1, 2, 4, or 8. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be set only if DDCAPS_ALPHA has been set. Used for blit operations.

DDFXALPHACAPS_OVERLAYALPHAEDGEBLEND
Supports alpha blending around the edge of a source color-keyed surface. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHAPIXELS
Supports alpha information in pixel format. The bit depth of alpha information in pixel format can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHAPIXELSNEG
Supports alpha information in pixel format. The bit depth of alpha information in pixel format can be 1, 2, 4, or 8. The alpha value becomes more transparent as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be used only if DDCAPS_ALPHA has been set. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHASURFACES
Supports alpha-only surfaces. The bit depth of an alpha-only surface can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHASURFACESNEG
Indicates that the alpha channel becomes more transparent as the alpha value increases. The depth of the alpha channel data can be 1, 2, 4, or 8. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be used only if DDCAPS_ALPHA has been set. Used for overlays.

CONST_DDFXCAPSFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDFXCAPSFLAGS** enumeration is used in the **lFXCaps**, **lNLVBFXCaps**, **lSSBFXCaps**, **lSVBFXCaps** and **lVSBFXCaps** members of the **DDCAPS** type to describe driver-specific stretching and effects capabilities, nonlocal-to-local video memory blit capabilities, system-memory-to-system-memory blit capabilities, system-memory-to-display-memory blit capabilities and display-memory-to-system-memory blit capabilities.

```
Enum CONST_DDFXCAPSFLAGS
   DDFXCAPS_BLTALPHA = 1
   DDFXCAPS_BLTARITHSTRETCHY = 32
   DDFXCAPS_BLTARITHSTRETCHYN = 16
   DDFXCAPS_BLTFILTER = 32
   DDFXCAPS_BLTMIRRORLEFTRIGHT = 64
   DDFXCAPS_BLTMIRRORUPDOWN = 128
   DDFXCAPS_BLTROTATION = 256
   DDFXCAPS_BLTROTATION90 = 512
   DDFXCAPS_BLTSHRINKX = 1024
   DDFXCAPS_BLTSHRINKXN = 2048
   DDFXCAPS_BLTSHRINKY = 4096
   DDFXCAPS_BLTSHRINKYN = 8192
   DDFXCAPS_BLTSTRETCHX = 16384
   DDFXCAPS_BLTSTRETCHXN = 32768
   DDFXCAPS_BLTSTRETCHY = 65536
   DDFXCAPS_BLTSTRETCHYN = 131072
   DDFXCAPS_BLTTRANSFORM = 2
   DDFXCAPS_OVERLAYALPHA = 4
   DDFXCAPS_OVERLAYARITHSTRETCHY = 262144
   DDFXCAPS_OVERLAYARITHSTRETCHYN = 8
   DDFXCAPS_OVERLAYFILTER = 262144
   DDFXCAPS_OVERLAYMIRRORLEFTRIGHT = 134217728
   DDFXCAPS_OVERLAYMIRRORUPDOWN = 268435456
   DDFXCAPS_OVERLAYSHRINKX = 524288
   DDFXCAPS_OVERLAYSHRINKXN = 1048576
   DDFXCAPS_OVERLAYSHRINKY = 2097152
   DDFXCAPS_OVERLAYSHRINKYN = 4194304
   DDFXCAPS_OVERLAYSTRETCHX = 8388608
   DDFXCAPS_OVERLAYSTRETCHXN = 16777216
   DDFXCAPS_OVERLAYSTRETCHY = 33554432
   DDFXCAPS_OVERLAYSTRETCHYN = 67108864
   DDFXCAPS_OVERLAYTRANSFORM = 536870912
End Enum
```

[#] IDH__dx_CONST_DDFXCAPSFLAGS_ddraw_vb

DDFXCAPS_BLTALPHA
Supports alpha-blended blit operations.

DDFXCAPS_BLTARITHSTRETCHY
Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink surfaces during a blit operation. Occurs along the y-axis (vertically).

DDFXCAPS_BLTARITHSTRETCHYN
Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink surfaces during a blit operation. Occurs along the y-axis (vertically), and works only for integer stretching (×1, ×2, and so on).

DDFXCAPS_BLTFILTER
Driver can do surface-reconstruction filtering for warped blits.

DDFXCAPS_BLTMIRRORLEFTRIGHT
Supports mirroring left to right in a blit operation.

DDFXCAPS_BLTMIRRORUPDOWN
Supports mirroring top to bottom in a blit operation.

DDFXCAPS_BLTROTATION
Supports arbitrary rotation in a blit operation.

DDFXCAPS_BLTROTATION90
Supports 90-degree rotations in a blit operation.

DDFXCAPS_BLTSHRINKX
Supports arbitrary shrinking of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSHRINKXN
Supports integer shrinking (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSHRINKY
Supports arbitrary shrinking of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_BLTSHRINKYN
Supports integer shrinking (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHX
Supports arbitrary stretching of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHXN
Supports integer stretching (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHY
Supports arbitrary stretching of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_BLTSTRETCHYN
Supports integer stretching (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for blit operations.

DDFXCAPS_BLTTRANSFORM
Supports geometric transformations (or warps) for blitted sprites. Transformations are not currently supported for explicit blit operations.

DDFXCAPS_OVERLAYALPHA
Supports alpha blending for overlay surfaces.

DDFXCAPS_OVERLAYARITHSTRETCHY
Supports integer stretching (×1, ×2, and so on) of an overlay surface along the y-axis (vertically).

DDFXCAPS_OVERLAYARITHSTRETCHYN
Supports arbitrary stretching of a surface along the x-axis (horizontal) for overlays.

DDFXCAPS_OVERLAYFILTER
Supports surface-reconstruction filtering for warped overlay sprites. Filtering is not currently supported for explicitly displayed overlay surfaces (those displayed with calls to **DirectDrawSurface4.UpdateOverlay**).

DDFXCAPS_OVERLAYMIRRORLEFTRIGHT
Supports mirroring of overlays across the vertical axis.

DDFXCAPS_OVERLAYMIRRORUPDOWN
Supports mirroring of overlays across the horizontal axis.

DDFXCAPS_OVERLAYSHRINKX
Supports arbitrary shrinking of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKXN
Supports integer shrinking (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKY
Supports arbitrary shrinking of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKYN
Supports integer shrinking (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSTRETCHX
Supports arbitrary stretching of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHXN
Supports integer stretching (×1, ×2, and so on) of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This

flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHY
Supports arbitrary stretching of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHYN
Supports integer stretching (×1, ×2, and so on) of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYTRANSFORM
Supports geometric transformations (or warps) for overlay sprites. Transformations are not currently supported for explicitly displayed overlay surfaces (those displayed with calls to **DirectDrawSurface4.UpdateOverlay**).

CONST_DDGBSFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_DDGBSFLAGS** enumeration is used by the *flags* parameter of the **DirectDrawSurface4.GetBltStatus** method to specify what type of status to obtain.

```
Enum CONST_DDGBSFLAGS
   DDGBS_CANBLT = 1
   DDGBS_ISBLTDONE = 2
End Enum
```

DDGBS_CANBLT
Inquires whether a blit involving this surface can occur immediately, and returns DD_OK if the blit can be completed.

DDGBS_ISBLTDONE
Inquires whether the blit is done, and returns DD_OK if the last blit on this surface has completed.

CONST_DDGFSFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_DDGFSFLAGS** enumeration is used by the *flags* parameter of the **DirectDrawSurface4.GetFlipStatus** method to specify the type of flip status to obtain.

```
Enum CONST_DDGFSFLAGS
```

IDH__dx_CONST_DDGBSFLAGS_ddraw_vb
IDH__dx_CONST_DDGFSFLAGS_ddraw_vb

```
    DDGFS_CANFLIP = 1
    DDGFS_ISFLIPDONE = 2
End Enum
```

DDGFS_CANFLIP
: Inquires whether this surface can be flipped immediately and returns DD_OK if the flip can be completed.

DDGFS_ISFLIPDONE
: Inquires whether the flip has finished and returns DD_OK if the last flip on this surface has completed.

CONST_DDLOCKFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDLOCKFLAGS** enumeration is used by the *flags* parameter of both the **DirectDrawSurface4.Lock** and **Direct3DVertexBuffer.Lock** method to indicate the how the lock is to be performed.

```
Enum CONST_DDLOCKFLAGS
    DDLOCK_EVENT = 2
    DDLOCK_NOSYSLOCK = 2048
    DDLOCK_READONLY = 16
    DDLOCK_SURFACEMEMORYPTR = 0
    DDLOCK_WAIT = 1
    DDLOCK_WRITEONLY = 32
End Enum
```

DDLOCK_EVENT
: This flag is not currently implemented.

DDLOCK_NOSYSLOCK
: If possible, do not take the Win16Mutex (also known as Win16Lock). This flag is ignored when locking the primary surface.

DDLOCK_READONLY
: Indicates that the surface being locked will only be read.

DDLOCK_SURFACEMEMORYPTR
: Indicates that a valid memory pointer to the top of the specified rectangle should be returned. If no rectangle is specified, a pointer to the top of the surface is returned. This is the default.

DDLOCK_WAIT
: If a lock cannot be obtained because a blit operation is in progress, the method retries until a lock is obtained or another error occurs, such as DDERR_SURFACEBUSY.

DDLOCK_WRITEONLY

[#] IDH__dx_CONST_DDLOCKFLAGS_ddraw_vb

Indicates that the surface being locked will be write enabled.

CONST_DDOVERFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDOVERFLAGS** enumeration is used in the *flags* parameter of the **DirectDrawSurface4.UpdateOverlay** method to specify how the overlay should be updated.

```
Enum CONST_DDOVERFLAGS
    DDOVER_ADDDIRTYRECT = 32768
    DDOVER_ALPHADEST = 1
    DDOVER_ALPHADESTCONSTOVERRIDE = 2
    DDOVER_ALPHADESTNEG = 4
    DDOVER_ALPHADESTSURFACEOVERRIDE = 8
    DDOVER_ALPHAEDGEBLEND = 16
    DDOVER_ALPHASRC = 32
    DDOVER_ALPHASRCCONSTOVERRIDE = 64
    DDOVER_ALPHASRCNEG = 128
    DDOVER_ALPHASRCSURFACEOVERRIDE = 256
    DDOVER_AUTOFLIP = 1048576
    DDOVER_BOB = 2097152
    DDOVER_DDFX = 524288
    DDOVER_HIDE = 512
    DDOVER_INTERLEAVED = 8388608
    DDOVER_KEYDEST = 1024
    DDOVER_KEYDESTOVERRIDE = 2048
    DDOVER_KEYSRC = 4096
    DDOVER_KEYSRCOVERRIDE = 8192
    DDOVER_OVERRIDEBOBWEAVE = 4194304
    DDOVER_REFRESHALL = 131072
    DDOVER_REFRESHDIRTYRECTS = 65536
    DDOVER_SHOW = 16384
End Enum
```

DDOVER_ADDDIRTYRECT

Adds a dirty rectangle to an emulated overlay surface.‹???: Is this flag supported, overlay surfaces are not supported in the HEL.›

DDOVER_ALPHADEST

Uses either the alpha information in pixel format or the alpha channel surface attached to the destination surface as the alpha channel for this overlay.

DDOVER_ALPHADESTNEG

[#] IDH__dx_CONST_DDOVERFLAGS_ddraw_vb

Indicates that the destination surface becomes more transparent as the alpha value increases (0 is opaque).

DDOVER_ALPHASRC
Uses either the alpha information in pixel format or the alpha channel surface attached to the source surface as the source alpha channel for this overlay.

DDOVER_ALPHASRCNEG
Indicates that the source surface becomes more transparent as the alpha value increases (0 is opaque).

DDOVER_AUTOFLIP
Automatically flip to the next surface in the flip chain each time a video port VSYNC occurs.

DDOVER_BOB
Display each field individually of the interlaced video stream without causing any artifacts.

DDOVER_BOBHARDWARE
Indicates that bob operations will be performed using hardware rather than software or emulated. This flag must be used with the DDOVER_BOB flag.

DDOVER_HIDE
Turns off this overlay.

DDOVER_INTERLEAVED
Indicates that the surface memory is composed of interleaved fields.

DDOVER_KEYDEST
Uses the color key associated with the destination surface.

DDOVER_KEYSRC
Uses the color key associated with the source surface.

DDOVER_OVERRIDEBOBWEAVE
Indicates that bob/weave decisions should not be overridden by other classes.

DDOVER_REFRESHALL

DDOVER_REFRESHDIRTYRECTS

DDOVER_SHOW
Turns on this overlay.

CONST_DDOVERLAYFXFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDOVERLAYFXFLAGS** enumeration is used by the **lDDFX** member of the **DDOVERLAYFX** type to specify the overlay FX.

```
Enum CONST_DDOVERLAYFXFLAGS
    DDOVERFX_ARITHSTRETCHY = 1
```

[#] IDH__dx_CONST_DDOVERLAYFXFLAGS_ddraw_vb

```
    DDOVERFX_MIRRORLEFTRIGHT = 2
    DDOVERFX_MIRRORUPDOWN = 4
End Enum
```

DDOVERFX_ARITHSTRETCHY
: If stretching, use arithmetic stretching along the y-axis for this overlay.

DDOVERFX_MIRRORLEFTRIGHT
: Mirror the overlay around the vertical axis.

DDOVERFX_MIRRORUPDOWN
: Mirror the overlay around the horizontal axis.

CONST_DDOVERZFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDOVERZFLAGS** enumeration is used in the *flags* parameter of the **DirectDrawSurface4.UpdateOverlayZOrder** method to specify how the z-order of an overlay should be updated.

```
Enum CONST_DDOVERZFLAGS
    DDOVERZ_INSERTINBACKOF = 5
    DDOVERZ_INSERTINFRONTOF = 4
    DDOVERZ_MOVEBACKWARD = 3
    DDOVERZ_MOVEFORWARD = 2
    DDOVERZ_SENDTOBACK = 1
    DDOVERZ_SENDTOFRONT = 0
End Enum
```

DDOVERZ_INSERTINBACKOF
: Inserts this overlay in the overlay chain behind the reference overlay.

DDOVERZ_INSERTINFRONTOF
: Inserts this overlay in the overlay chain in front of the reference overlay.

DDOVERZ_MOVEBACKWARD
: Moves this overlay one position backward in the overlay chain.

DDOVERZ_MOVEFORWARD
: Moves this overlay one position forward in the overlay chain.

DDOVERZ_SENDTOBACK
: Moves this overlay to the back of the overlay chain.

DDOVERZ_SENDTOFRONT
: Moves this overlay to the front of the overlay chain.

[#] IDH__dx_CONST_DDOVERZFLAGS_ddraw_vb

CONST_DDPCAPSFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_DDPCAPSFLAGS** enumeration is used by the *flags* parameter of **DirectDraw4.CreatePalette**, as a return value for **DirectDrawPalette.GetCaps** and by the **lPalCaps** member of **DDCAPS** to describe the capabilities of the palette.

```
Enum CONST_DDPCAPSFLAGS
    DDPCAPS_1BIT = 256
    DDPCAPS_2BIT = 512
    DDPCAPS_4BIT = 1
    DDPCAPS_8BIT = 4
    DDPCAPS_8BITENTRIES = 2
    DDPCAPS_ALLOW256 = 64
    DDPCAPS_ALPHA = 1024
    DDPCAPS_INITIALIZE = 8
    DDPCAPS_PRIMARYSURFACE = 16
    DDPCAPS_PRIMARYSURFACELEFT = 32
    DDPCAPS_VSYNC = 128
End Enum
```

DDPCAPS_1BIT
: Indicates that the index is 1 bit. There are two entries in the color table.

DDPCAPS_2BIT
: Indicates that the index is 2 bits. There are four entries in the color table.

DDPCAPS_4BIT
: Indicates that the index is 4 bits. There are 16 entries in the color table.

DDPCAPS_8BIT
: Indicates that the index is 8 bits. There are 256 entries in the color table_dx_color_table_glos.

DDPCAPS_8BITENTRIES
: Indicates that the index refers to an 8-bit color index. This flag is valid only when used with the DDPCAPS_1BIT, DDPCAPS_2BIT, or DDPCAPS_4BIT flag, and when the target surface is in 8 bpp. Each color entry is 1 byte long and is an index to a destination surface's 8-bpp palette.

DDPCAPS_ALPHA
: Indicates that the **flags** member of the associated **PALETTEENTRY** type is to be interpreted as a single 8-bit alpha value. A palette created with this flag can only be attached to a texture (a surface created with the DDSCAPS_TEXTURE capability flag).

DDPCAPS_ALLOW256
: Indicates that this palette can have all 256 entries defined.

DDPCAPS_INITIALIZE

IDH__dx_CONST_DDPCAPSFLAGS_ddraw_vb

This flag is obsolete and ignored by DirectDraw.

DDPCAPS_PRIMARYSURFACE
This palette is attached to the primary surface. Changing this palette's color table immediately affects the display unless DDPSETPAL_VSYNC is specified and supported.

DDPCAPS_PRIMARYSURFACELEFT
This palette is the one attached to the left eye primary surface. Changing this palette's color table immediately affects the left eye display unless DDPSETPAL_VSYNC is specified and supported.

DDPCAPS_VSYNC
This palette can have modifications to it synced with the monitors refresh rate.

CONST_DDPIXELFORMATFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDPIXELFORMATFLAGS** enumeration is

```
Enum CONST_DDPIXELFORMATFLAGS
    DDPF_ALPHA = 2
    DDPF_ALPHAPIXELS = 1
    DDPF_ALPHAPREMULT = 32768
    DDPF_BUMPDUDV = 524288
    DDPF_BUMPLUMINANCE = 262144
    DDPF_COMPRESSED = 128
    DDPF_FOURCC = 4
    DDPF_LUMINANCE = 131072
    DDPF_PALETTEINDEXED1 = 2048
    DDPF_PALETTEINDEXED2 = 4096
    DDPF_PALETTEINDEXED4 = 8
    DDPF_PALETTEINDEXED8 = 32
    DDPF_PALETTEINDEXEDTO8 = 16
    DDPF_RGB = 64
    DDPF_RGBTOYUV = 256
    DDPF_STENCILBUFFER = 16384
    DDPF_YUV = 512
    DDPF_ZBUFFER = 1024
    DDPF_ZPIXELS = 8192
End Enum
```

DDPF_ALPHA
The pixel format describes an alpha-only surface.

[#] IDH__dx_CONST_DDPIXELFORMATFLAGS_ddraw_vb

DDPF_ALPHAPIXELS
The surface has alpha channel information in the pixel format.

DDPF_ALPHAPREMULT
The surface uses the premultiplied alpha format. That is, the color components in each pixel are premultiplied by the alpha component.

DDPF_BUMPDUDV
Bump-map data in the pixel format is valid. Bump-map information is in the **lBumpBitCount**, **lBumpDuBitMask**, **lBumpDvBitMask**, and **lBumpLuminanceBitMask** members.

DDPF_BUMPLUMINANCE
The luminance data in the pixel format is valid, and the **lLuminanceBitMask** member descibes valid luminance bits for a luminance-only or luminance-alpha surface.

DDPF_COMPRESSED
The surface will accept pixel data in the specified format and compress it during the write operation.

DDPF_FOURCC
The **lFourCC** member is valid and contains a FOURCC code describing a non-RGB pixel format.

DDPF_LUMINANCE
The pixel format describes a luminance-only or luminance-alpha surface.

DDPF_PALETTEINDEXED1
DDPF_PALETTEINDEXED2
DDPF_PALETTEINDEXED4
DDPF_PALETTEINDEXED8
The surface is 1-, 2-, 4-, or 8-bit color indexed.

DDPF_PALETTEINDEXEDTO8
The surface is 1-, 2-, or 4-bit color indexed to an 8-bit palette.

DDPF_RGB
The RGB data in the pixel format type is valid.

DDPF_RGBTOYUV
The surface will accept RGB data and translate it during the write operation to YUV data. The format of the data to be written will be contained in the pixel format type. The DDPF_RGB flag will be set.

DDPF_STENCILBUFFER
The surface encodes stencil and depth information in each pixel of the z-buffer.

DDPF_YUV
The YUV data in the pixel format type is valid.

DDPF_ZBUFFER
The pixel format describes a z-buffer-only surface.

DDPF_ZPIXELS
The surface contains z information in the pixels.

CONST_DDRAW

[This is preliminary documentation and subject to change.]

The **CONST_DDRAW** enumeration contains constants used throughout DirectDraw.

```
Enum CONST_DDRAW
   DD_ROP_SPACE = 8
End Enum
```

CONST_DDSCLFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_DDSCLFLAGS** enumeration is used in the *flags* parameter of the **DirectDraw4.SetCooperativeLevel** method to determine the top-level behavior of the application.

```
Enum CONST_DDSCLFLAGS
   DDSCL_ALLOWMODEX = 64
   DDSCL_ALLOWREBOOT = 2
   DDSCL_CREATEDEVICEWINDOW = 512
   DDSCL_EXCLUSIVE = 16
   DDSCL_FULLSCREEN = 1
   DDSCL_MULTITHREADED = 1024
   DDSCL_NORMAL = 8
   DDSCL_NOWINDOWCHANGES = 4
   DDSCL_SETDEVICEWINDOW = 256
   DDSCL_SETFOCUSWINDOW = 128
End Enum
```

DDSCL_ALLOWMODEX
: Allows the use of Mode X display modes. This flag can only be used if the DDSCL_EXCLUSIVE and DDSCL_FULLSCREEN flags are present.

DDSCL_ALLOWREBOOT
: Allows CTRL+ALT+DEL to function while in exclusive (full-screen) mode.

DDSCL_CREATEDEVICEWINDOW
: **This flag is supported in Windows 98 and Windows 2000 only.** Indicates that DirectDraw is to create and manage a default device window for this DirectDraw object. For more information, see Focus and Device Windows.

DDSCL_EXCLUSIVE
: Requests the exclusive level. This flag must be used with the DDSCL_FULLSCREEN flag.

IDH__dx_CONST_DDRAW_ddraw_vb
IDH__dx_CONST_DDSCLFLAGS_ddraw_vb

DDSCL_FULLSCREEN
Indicates that the exclusive-mode owner will be responsible for the entire primary surface. GDI can be ignored. This flag must be used with the DDSCL_EXCLUSIVE flag.

DDSCL_MULTITHREADED
Requests multithread-safe DirectDraw behavior. This causes Direct3D to take the global critical section more frequently.

DDSCL_NORMAL
Indicates that the application will function as a regular Windows application. This flag cannot be used with the DDSCL_ALLOWMODEX, DDSCL_EXCLUSIVE, or DDSCL_FULLSCREEN flags.

DDSCL_NOWINDOWCHANGES
Indicates that DirectDraw is not allowed to minimize or restore the application window on activation.

DDSCL_SETDEVICEWINDOW
This flag is supported in Windows 98 and Windows 2000 only. Indicates that the *hdl* argument is the window handle of the device window for this DirectDraw object. This flag cannot be used with the DDSCL_SETFOCUSWINDOW flag.

DDSCL_SETFOCUSWINDOW
This flag is supported in Windows 98 and Windows 2000 only. Indicates that the *hdl* argument is the window handle of the focus window for this DirectDraw object. This flag cannot be used with the DDSCL_SETDEVICEWINDOW flag.

CONST_DDSDMFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDSDMFLAGS** enumeration is used in the *flags* parameter of the **DirectDraw4.SetDisplayMode** method to set the mode of the display-device hardware.

```
Enum CONST_DDSDMFLAGS
   DDSDM_DEFAULT = 0
   DDSDM_STANDARDVGAMODE = 1
End Enum
```

DDSDM_STANDARDVGAMODE
Causes the method to set Mode 13 instead of Mode X 320x200x8 mode. If you are setting another resolution, bit depth, or a Mode X mode, do not use this flag and set the argument to 0.

[#] IDH__dx_CONST_DDSDMFLAGS_ddraw_vb

CONST_DDSGRFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_DDSGRFLAGS** enumeration is used in the flags parameter of the **DirectDrawGammaControl.SetGammaRamp** method to indicate that gamma calibration is desired.

```
Enum CONST_DDSGRFLAGS
   DDSGR_CALIBRATE = 1
   DDSGR_DEFAULT = 0
End Enum
```

DDSGR_CALIBRATE
: Requests that the calibrator adjust the gamma ramp according to the physical properties of the display, making the result identical on all systems. If calibration is not needed, set this argument to 0.

CONST_DDSTEREOCAPSFLAGS

[This is preliminary documentation and subject to change.]

The **CONST_DDSTEREOCAPSFLAGS** enumeration is used by the **lSVCaps** member of the **DDCAPS** type to describe stereo vision capabilities.

```
Enum CONST_DDSTEREOCAPSFLAGS
   DDSVCAPS_ENIGMA = 1
   DDSVCAPS_FLICKER = 2
   DDSVCAPS_REDBLUE = 4
   DDSVCAPS_SPLIT = 8
End Enum
```

DDSVCAPS_ENIGMA
: Indicates that the stereo view is accomplished using Enigma encoding.

DDSVCAPS_FLICKER
: Indicates that the stereo view is accomplished using high-frequency flickering.

DDSVCAPS_REDBLUE
: Indicates that the stereo view is accomplished when the viewer looks at the image through red and blue filters placed over the left and right eyes. All images must adapt their color spaces for this process.

DDSVCAPS_SPLIT
: Indicates that the stereo view is accomplished with split-screen technology.

IDH__dx_CONST_DDSGRFLAGS_ddraw_vb
IDH__dx_CONST_DDSTEREOCAPSFLAGS_ddraw_vb

CONST_DDSURFACECAPS2FLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDSURFACECAPS2FLAGS** enumeration is used by the **lCaps2** member of **DDSCAPS2** type to describe additional surface capabilities.

```
Enum CONST_DDSURFACECAPS2FLAGS
    DDSCAPS2_HARDWAREDEINTERLACE = 2
    DDSCAPS2_HINTANTIALIASING = 256
    DDSCAPS2_HINTDYNAMIC = 4
    DDSCAPS2_HINTSTATIC = 8
    DDSCAPS2_OPAQUE = 128
    DDSCAPS2_TEXTUREMANAGE = 16
End Enum
```

DDSCAPS2_HARDWAREDEINTERLACE
: Indicates that this surface will receive data from a video port using the de-interlacing hardware. This allows the driver to allocate memory for any extra buffers that may be required. The DDSCAPS_VIDEOPORT and DDSCAPS_OVERLAY flags must also be set.

DDSCAPS2_HINTANTIALIASING
: Indicates that the application intends to use antialiasing. Only valid if DDSCAPS_3DDEVICE is also set.

DDSCAPS2_HINTDYNAMIC
: Indicates to the driver that this surface will be locked very frequently (for procedural textures, dynamic lightmaps, etc). This flag can only be used for texture surfaces (DDSCAPS_SYSTEMMEMORY flag set in the **lCaps** member). This flag cannot be used with the DDSCAPS2_HINTSTATIC or DDSCAPS2_OPAQUE flags.

DDSCAPS2_HINTSTATIC
: Indicates to the driver that this surface can be reordered or retiled on load. This operation will not change the size of the texture. It is relatively fast and symmetrical, since the application may lock these bits (although it will take a performance hit when doing so). This flag can only be used for texture surfaces (DDSCAPS_SYSTEMMEMORY flag set in the **lCaps** member). This flag cannot be used with the DDSCAPS2_HINTDYNAMIC or DDSCAPS2_OPAQUE flags.

DDSCAPS2_OPAQUE
: Indicates to the driver that this surface will never be locked again. The driver is free to optimize this surface by retiling and actual compression. Such a surface cannot be locked or used in blit operations, attempts to lock or blit a surface

---

[#] IDH__dx_CONST_DDSURFACECAPS2FLAGS_ddraw_vb

with this capability will fail. This flag can only be used for texture surfaces (DDSCAPS_SYSTEMMEMORY flag set in the **lCaps** member). This flag cannot be used with the DDSCAPS2_HINTDYNAMIC or DDSCAPS2_HINTSTATIC flags.

DDSCAPS2_TEXTUREMANAGE
Indicates that the client would like this texture surface to be managed by DirectDraw and Direct3D. This flag can only be used for texture surfaces (DDSCAPS_TEXTURE flag set in the **lCaps** member). For more information, see Automatic Texture Management in the Direct3D Immediate Mode documentation.

CONST_DDSURFACECAPSFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDSURFACECAPSFLAGS** enumeration is used in the **lCaps** member of the **DDSCAPS2** type to describe the capabilities of the surface.

```
Enum CONST_DDSURFACECAPSFLAGS
    DDSCAPS_3DDEVICE = 8192
    DDSCAPS_ALLOCONLOAD = 67108864
    DDSCAPS_ALPHA = 2
    DDSCAPS_BACKBUFFER = 4
    DDSCAPS_COMPLEX = 8
    DDSCAPS_FLIP = 16
    DDSCAPS_FRONTBUFFER = 32
    DDSCAPS_HWCODEC = 1048576
    DDSCAPS_LIVEVIDEO = 524288
    DDSCAPS_LOCALVIDMEM = 268435456
    DDSCAPS_MIPMAP = 4194304
    DDSCAPS_MODEX = 2097152
    DDSCAPS_NONLOCALVIDMEM = 536870912
    DDSCAPS_OFFSCREENPLAIN = 64
    DDSCAPS_OPTIMIZED = -2147483648
    DDSCAPS_OVERLAY = 128
    DDSCAPS_OWNDC = 262144
    DDSCAPS_PALETTE = 256
    DDSCAPS_PRIMARYSURFACE = 512
    DDSCAPS_PRIMARYSURFACELEFT = 1024
    DDSCAPS_RESERVED2 = 8388608
    DDSCAPS_STANDARDVGAMODE = 1073741824
    DDSCAPS_SYSTEMMEMORY = 2048
    DDSCAPS_TEXTURE = 4096
```

[#] IDH__dx_CONST_DDSURFACECAPSFLAGS_ddraw_vb

```
    DDSCAPS_VIDEOMEMORY = 16384
    DDSCAPS_VIDEOPORT = 134217728
    DDSCAPS_VISIBLE = 32768
    DDSCAPS_WRITEONLY = 65536
    DDSCAPS_ZBUFFER = 131072
End Enum
```

DDSCAPS_3DDEVICE
: Indicates that this surface can be used for 3-D rendering. Applications can use this flag to ensure that a device that can only render to a certain heap has off-screen surfaces allocated from the correct heap. If this flag is set for a heap, the surface is not allocated from that heap.

DDSCAPS_ALLOCONLOAD
: Not used, ignored by DirectDraw and Direct3D.

DDSCAPS_ALPHA
: Indicates that this surface contains alpha-only information.

DDSCAPS_BACKBUFFER
: Indicates that this surface is the back buffer of a surface flipping type. Typically, this capability is set by the **CreateSurface** method when the DDSCAPS_FLIP flag is used. Only the surface that immediately precedes the DDSCAPS_FRONTBUFFER surface has this capability set. The other surfaces are identified as back buffers by the presence of the DDSCAPS_FLIP flag, their attachment order, and the absence of the DDSCAPS_FRONTBUFFER and DDSCAPS_BACKBUFFER capabilities. If this capability is sent to the **CreateSurface** method, a stand-alone back buffer is being created. After this method is called, this surface could be attached to a front buffer, another back buffer, or both to form a flipping surface type. For more information, see **DirectDrawSurface4.AddAttachedSurface**. DirectDraw supports an arbitrary number of surfaces in a flipping type.

DDSCAPS_COMPLEX
: Indicates that a complex surface is being described. A complex surface results in the creation of more than one surface. The additional surfaces are attached to the root surface. The complex type can be destroyed only by destroying the root.

DDSCAPS_FLIP
: Indicates that this surface is a part of a surface flipping type. When this capability is passed to the **CreateSurface** method, a front buffer and one or more back buffers are created. DirectDraw sets the DDSCAPS_FRONTBUFFER bit on the front-buffer surface and the DDSCAPS_BACKBUFFER bit on the surface adjacent to the front-buffer surface. The **lBackBufferCount** member of the **DDSURFACEDESC2** type must be set to at least 1 in order for the method call to succeed. The DDSCAPS_COMPLEX capability must always be set when creating multiple surfaces by using the **CreateSurface** method.

DDSCAPS_FRONTBUFFER
: Indicates that this surface is the front buffer of a surface flipping type. This flag is typically set by the **CreateSurface** method when the DDSCAPS_FLIP

capability is set. If this capability is sent to the **CreateSurface** method, a stand-alone front buffer is created. This surface will not have the DDSCAPS_FLIP capability. It can be attached to other back buffers to form a flipping type by using **DirectDrawSurface4.AddAttachedSurface**.

DDSCAPS_HWCODEC
Indicates that this surface should be able to have a stream decompressed to it by the hardware.

DDSCAPS_LIVEVIDEO
Indicates that this surface should be able to receive live video.

DDSCAPS_LOCALVIDMEM
Indicates that this surface exists in true, local video memory rather than non-local video memory. If this flag is specified then DDSCAPS_VIDEOMEMORY must be specified as well. This flag cannot be used with the DDSCAPS_NONLOCALVIDMEM flag.

DDSCAPS_MIPMAP
Indicates that this surface is one level of a mipmap. This surface will be attached to other DDSCAPS_MIPMAP surfaces to form the mipmap. This can be done explicitly by creating a number of surfaces and attaching them by using the **DirectDrawSurface4.AddAttachedSurface** method, or implicitly by the **CreateSurface** method. If this capability is set, DDSCAPS_TEXTURE must also be set.

DDSCAPS_MODEX
Indicates that this surface is a 320×200 or 320×240 Mode X surface.

DDSCAPS_NONLOCALVIDMEM
Indicates that this surface exists in non-local video memory rather than true, local video memory. If this flag is specified, then DDSCAPS_VIDEOMEMORY flag must be specified as well. This cannot be used with the DDSCAPS_LOCALVIDMEM flag.

DDSCAPS_OFFSCREENPLAIN
Indicates that this surface is any off-screen surface that is not an overlay, texture, z-buffer, front-buffer, back-buffer, or alpha surface. It is used to identify plain surfaces.

DDSCAPS_OPTIMIZED
Not currently implemented.

DDSCAPS_OVERLAY
Indicates that this surface is an overlay. It may or may not be directly visible depending on whether it is currently being overlaid onto the primary surface. DDSCAPS_VISIBLE can be used to determine if it is being overlaid at the moment.

DDSCAPS_OWNDC
Indicates that this surface will have a device context (DC) association for a long period.

DDSCAPS_PALETTE
Indicates that this device driver allows unique DirectDrawPalette objects to be created and attached to this surface.

DDSCAPS_PRIMARYSURFACE
Indicates the surface is the primary surface. It represents what is visible to the user at the moment.

DDSCAPS_PRIMARYSURFACELEFT
Indicates that this surface is the primary surface for the left eye. It represents what is visible to the user's left eye at the moment. When this surface is created, the surface with the DDSCAPS_PRIMARYSURFACE capability represents what is seen by the user's right eye.

DDSCAPS_RESERVED2
Reserved for future use.

DDSCAPS_STANDARDVGAMODE
Indicates that this surface is a standard VGA mode surface, and not a Mode X surface. This flag cannot be used in combination with the DDSCAPS_MODEX flag.

DDSCAPS_SYSTEMMEMORY
Indicates that this surface memory was allocated in system memory.

DDSCAPS_TEXTURE
Indicates that this surface can be used as a 3-D texture. It does not indicate whether the surface is being used for that purpose.

DDSCAPS_VIDEOMEMORY
Indicates that this surface exists in display memory.

DDSCAPS_VIDEOPORT
Indicates that this surface can receive data from a video port.

DDSCAPS_VISIBLE
Indicates that changes made to this surface are immediately visible. It is always set for the primary surface, as well as for overlays while they are being overlaid and texture maps while they are being textured.

DDSCAPS_WRITEONLY
Indicates that only write access is permitted to the surface. Read access from the surface may generate a general protection (GP) fault, but the read results from this surface will not be meaningful.

DDSCAPS_ZBUFFER
Indicates that this surface is the z-buffer. The z-buffer contains information that cannot be displayed. Instead, it contains bit-depth information that is used to determine which pixels are visible and which are obscured.

CONST_DDSURFACEDESCFLAGS

[#] [This is preliminary documentation and subject to change.]

[#] IDH__dx_CONST_DDSURFACEDESCFLAGS_ddraw_vb

The **CONST_DDSURFACEDESCFLAGS** enumeration is used in the **lFlags** member of the **DDSURFACEDESC2** type to specify which members of **DDSURFACEDESC2** contain valid data.

```
Enum CONST_DDSURFACEDESCFLAGS
    DDSD_ALL = 1047022
    DDSD_ALPHABITDEPTH = 128
    DDSD_BACKBUFFERCOUNT = 32
    DDSD_CAPS = 1
    DDSD_CKDESTBLT = 16384
    DDSD_CKDESTOVERLAY = 8192
    DDSD_CKSRCBLT = 65536
    DDSD_CKSRCOVERLAY = 32768
    DDSD_HEIGHT = 2
    DDSD_LINEARSIZE = 524288
    DDSD_LPSURFACE = 2048
    DDSD_MIPMAPCOUNT = 131072
    DDSD_PITCH = 8
    DDSD_PIXELFORMAT = 4096
    DDSD_REFRESHRATE = 262144
    DDSD_TEXTURESTAGE = 1048576
    DDSD_WIDTH = 4
    DDSD_ZBUFFERBITDEPTH = 64
End Enum
```

DDSD_ALL
: Indicates that all input members are valid.

DDSD_ALPHABITDEPTH
: Indicates that the **lAlphaBitDepth** member is valid.

DDSD_BACKBUFFERCOUNT
: Indicates that the **lBackBufferCount** member is valid.

DDSD_CAPS
: Indicates that the **ddsCaps** member is valid.

DDSD_CKDESTBLT
: Indicates that the **ddckCKDestBlt** member is valid.

DDSD_CKDESTOVERLAY
: Indicates that the **ddckCKDestOverlay** member is valid.

DDSD_CKSRCBLT
: Indicates that the **ddckCKSrcBlt** member is valid.

DDSD_CKSRCOVERLAY
: Indicates that the **ddckCKSrcOverlay** member is valid.

DDSD_HEIGHT
: Indicates that the **lHeight** member is valid.

DDSD_LINEARSIZE

Not used.‹This will be implemented sometime after DX6 and beyond Indicates that lLinearSize member is valid.›

DDSD_LPSURFACE
Indicates that the **lpSurface** member is valid.

DDSD_MIPMAPCOUNT
Indicates that the **lMipMapCount** member is valid.

DDSD_PITCH
Indicates that the **lPitch** member is valid.

DDSD_PIXELFORMAT
Indicates that the **ddpfPixelFormat** member is valid.

DDSD_REFRESHRATE
Indicates that the **lRefreshRate** member is valid.

DDSD_TEXTURESTAGE
Indicates that the **lTextureStage** member is valid.

DDSD_WIDTH
Indicates that the **lWidth** member is valid.

DDSD_ZBUFFERBITDEPTH
Indicates that the **lZBufferBitDepth** member is valid.

CONST_DDWAITVBFLAGS

[#] [This is preliminary documentation and subject to change.]

The **CONST_DDWAITVBFLAGS** enumeration is used by the *flags* parameter of the **DirectDraw4.WaitForVerticalBlank** method to specify how long to wait for the vertical blank.

```
Enum CONST_DDWAITVBFLAGS
    DDWAITVB_BLOCKBEGIN = 1
    DDWAITVB_BLOCKBEGINEVENT = 2
    DDWAITVB_BLOCKEND = 4
End Enum
```

DDWAITVB_BLOCKBEGIN
Returns when the vertical-blank interval begins.

DDWAITVB_BLOCKBEGINEVENT
Triggers an event when the vertical blank begins. This value is not currently supported.

DDWAITVB_BLOCKEND
Returns when the vertical-blank interval ends and the display begins.

[#] IDH__dx_CONST_DDWAITVBFLAGS_ddraw_vb

Error Codes

[This is preliminary documentation and subject to change.]

Errors are represented by negative values and cannot be combined. This table lists the values that can be returned by all methods of the **DirectDraw4**, **DirectDrawSurface4**, **DirectDrawPalette**, and **DirectDrawClipper** objects. For a list of the error codes that each method can return, see the method description.

DD_OK
The request completed successfully.

DDERR_ALREADYINITIALIZED
The object has already been initialized.

DDERR_BLTFASTCANTCLIP
A DirectDrawClipper object is attached to a source surface that has passed into a call to the **DirectDrawSurface4.BltFast** method.

DDERR_CANNOTATTACHSURFACE
A surface cannot be attached to another requested surface.

DDERR_CANNOTDETACHSURFACE
A surface cannot be detached from another requested surface.

DDERR_CANTCREATEDC
Windows can not create any more device contexts (DCs), or a DC was requested for a palette-indexed surface when the surface had no palette and the display mode was not palette-indexed (in this case DirectDraw cannot select a proper palette into the DC).

DDERR_CANTDUPLICATE
Primary and 3-D surfaces, or surfaces that are implicitly created, cannot be duplicated.

DDERR_CANTLOCKSURFACE
Access to this surface is refused because an attempt was made to lock the primary surface without DCI support.

DDERR_CANTPAGELOCK
An attempt to page lock a surface failed. Page lock will not work on a display-memory surface or an emulated primary surface.

DDERR_CANTPAGEUNLOCK
An attempt to page unlock a surface failed. Page unlock will not work on a display-memory surface or an emulated primary surface.

DDERR_CLIPPERISUSINGHWND
An attempt was made to set a clip list for a DirectDrawClipper object that is already monitoring a window handle.

DDERR_COLORKEYNOTSET
No source color key is specified for this operation.

DDERR_CURRENTLYNOTAVAIL
No support is currently available.

DDERR_DCALREADYCREATED

A device context (DC) has already been returned for this surface. Only one DC can be retrieved for each surface.

DDERR_DEVICEDOESNTOWNSURFACE
Surfaces created by one DirectDraw device cannot be used directly by another DirectDraw device.

DDERR_DIRECTDRAWALREADYCREATED
A DirectDraw object representing this driver has already been created for this process.

DDERR_EXCEPTION
An exception was encountered while performing the requested operation.

DDERR_EXCLUSIVEMODEALREADYSET
An attempt was made to set the cooperative level when it was already set to exclusive.

DDERR_EXPIRED
The data has expired and is therefore no longer valid.

DDERR_GENERIC
There is an undefined error condition.

DDERR_HEIGHTALIGN
The height of the provided rectangle is not a multiple of the required alignment.

DDERR_HWNDALREADYSET
The DirectDraw cooperative level window handle has already been set. It cannot be reset while the process has surfaces or palettes created.

DDERR_HWNDSUBCLASSED
DirectDraw is prevented from restoring state because the DirectDraw cooperative level window handle has been subclassed.

DDERR_IMPLICITLYCREATED
The surface cannot be restored because it is an implicitly created surface.

DDERR_INCOMPATIBLEPRIMARY
The primary surface creation request does not match with the existing primary surface.

DDERR_INVALIDCAPS
One or more of the capability bits passed to the callback function are incorrect.

DDERR_INVALIDCLIPLIST
DirectDraw does not support the provided clip list.

DDERR_INVALIDDIRECTDRAWGUID
The globally unique identifier (GUID) passed to the **DirectX7.DirectDrawCreate** function is not a valid DirectDraw driver identifier.

DDERR_INVALIDMODE
DirectDraw does not support the requested mode.

DDERR_INVALIDOBJECT

DirectDraw received a pointer that was an invalid DirectDraw object.

DDERR_INVALIDPARAMS
One or more of the parameters passed to the method are incorrect.

DDERR_INVALIDPIXELFORMAT
The pixel format was invalid as specified.

DDERR_INVALIDPOSITION
The position of the overlay on the destination is no longer legal.

DDERR_INVALIDRECT
The provided rectangle was invalid.

DDERR_INVALIDSTREAM
The specified stream contains invalid data.

DDERR_INVALIDSURFACETYPE
The requested operation could not be performed because the surface was of the wrong type.

DDERR_LOCKEDSURFACES
One or more surfaces are locked, causing the failure of the requested operation.

DDERR_MOREDATA
There is more data available than the specified buffer size can hold.

DDERR_NO3D
No 3-D hardware or emulation is present.

DDERR_NOALPHAHW
No alpha acceleration hardware is present or available, causing the failure of the requested operation.

DDERR_NOBLTHW
No blitter hardware is present.

DDERR_NOCLIPLIST
No clip list is available.

DDERR_NOCLIPPERATTACHED
No DirectDrawClipper object is attached to the surface object.

DDERR_NOCOLORCONVHW
The operation cannot be carried out because no color-conversion hardware is present or available.

DDERR_NOCOLORKEY
The surface does not currently have a color key.

DDERR_NOCOLORKEYHW
The operation cannot be carried out because there is no hardware support for the destination color key.

DDERR_NOCOOPERATIVELEVELSET
A create function is called without the **DirectDraw4.SetCooperativeLevel** method being called.

DDERR_NODC
No DC has ever been created for this surface.

DDERR_NODDROPSHW
No DirectDraw raster operation (ROP) hardware is available.

DDERR_NODIRECTDRAWHW
Hardware-only DirectDraw object creation is not possible; the driver does not support any hardware.

DDERR_NODIRECTDRAWSUPPORT
DirectDraw support is not possible with the current display driver.

DDERR_NOEMULATION
Software emulation is not available.

DDERR_NOEXCLUSIVEMODE
The operation requires the application to have exclusive mode, but the application does not have exclusive mode.

DDERR_NOFLIPHW
Flipping visible surfaces is not supported.

DDERR_NOFOCUSWINDOW
An attempt was made to create or set a device window without first setting the focus window.

DDERR_NOGDI
No GDI is present.

DDERR_NOHWND
Clipper notification requires a window handle, or no window handle has been previously set as the cooperative level window handle.

DDERR_NOMIPMAPHW
The operation cannot be carried out because no mipmap capable texture mapping hardware is present or available.

DDERR_NOMIRRORHW
The operation cannot be carried out because no mirroring hardware is present or available.

DDERR_NONONLOCALVIDMEM
An attempt was made to allocate non-local video memory from a device that does not support non-local video memory.

DDERR_NOOPTIMIZEHW
The device does not support optimized surfaces.

DDERR_NOOVERLAYDEST
The **DirectDrawSurface4.UpdateOverlay** method has not been called on to establish a destination.

DDERR_NOOVERLAYHW
The operation cannot be carried out because no overlay hardware is present or available.

DDERR_NOPALETTEATTACHED
No palette object is attached to this surface.

DDERR_NOPALETTEHW
There is no hardware support for 16- or 256-color palettes.

DDERR_NORASTEROPHW
The operation cannot be carried out because no appropriate raster operation hardware is present or available.

DDERR_NOROTATIONHW
The operation cannot be carried out because no rotation hardware is present or available.

DDERR_NOSTRETCHHW
The operation cannot be carried out because there is no hardware support for stretching.

DDERR_NOT4BITCOLOR
The DirectDrawSurface object is not using a 4-bit color palette and the requested operation requires a 4-bit color palette.

DDERR_NOT4BITCOLORINDEX
The DirectDrawSurface object is not using a 4-bit color index palette and the requested operation requires a 4-bit color index palette.

DDERR_NOT8BITCOLOR
The DirectDrawSurface object is not using an 8-bit color palette and the requested operation requires an 8-bit color palette.

DDERR_NOTAOVERLAYSURFACE
An overlay component is called for a non-overlay surface.

DDERR_NOTEXTUREHW
The operation cannot be carried out because no texture-mapping hardware is present or available.

DDERR_NOTFLIPPABLE
An attempt has been made to flip a surface that cannot be flipped.

DDERR_NOTFOUND
The requested item was not found.

DDERR_NOTINITIALIZED
An attempt was made to call an interface method of a DirectDraw object created by **CoCreateInstance** before the object was initialized.

DDERR_NOTLOADED
The surface is an optimized surface, but it has not yet been allocated any memory.

DDERR_NOTLOCKED
An attempt is made to unlock a surface that was not locked.

DDERR_NOTPAGELOCKED
An attempt is made to page unlock a surface with no outstanding page locks.

DDERR_NOTPALETTIZED
The surface being used is not a palette-based surface.

DDERR_NOVSYNCHW
The operation cannot be carried out because there is no hardware support for vertical blank synchronized operations.

DDERR_NOZBUFFERHW

The operation to create a z-buffer in display memory or to perform a blit using a z-buffer cannot be carried out because there is no hardware support for z-buffers.

DDERR_NOZOVERLAYHW
The overlay surfaces cannot be z-layered based on the z-order because the hardware does not support z-ordering of overlays.

DDERR_OUTOFCAPS
The hardware needed for the requested operation has already been allocated.

DDERR_OUTOFMEMORY
DirectDraw does not have enough memory to perform the operation.

DDERR_OUTOFVIDEOMEMORY
DirectDraw does not have enough display memory to perform the operation.

DDERR_OVERLAPPINGRECTS
Operation could not be carried out because the source and destination rectangles are on the same surface and overlap each other.

DDERR_OVERLAYCANTCLIP
The hardware does not support clipped overlays.

DDERR_OVERLAYCOLORKEYONLYONEACTIVE
An attempt was made to have more than one color key active on an overlay.

DDERR_OVERLAYNOTVISIBLE
The method is called on a hidden overlay.

DDERR_PALETTEBUSY
Access to this palette is refused because the palette is locked by another thread.

DDERR_PRIMARYSURFACEALREADYEXISTS
This process has already created a primary surface.

DDERR_REGIONTOOSMALL
The region passed to the **DirectDrawClipper.GetClipList** method is too small.

DDERR_SURFACEALREADYATTACHED
An attempt was made to attach a surface to another surface to which it is already attached.

DDERR_SURFACEALREADYDEPENDENT
An attempt was made to make a surface a dependency of another surface to which it is already dependent.

DDERR_SURFACEBUSY
Access to the surface is refused because the surface is locked by another thread.

DDERR_SURFACEISOBSCURED
Access to the surface is refused because the surface is obscured.

DDERR_SURFACELOST

Access to the surface is refused because the surface memory is gone. Call the **DirectDrawSurface4.Restore** method on this surface to restore the memory associated with it.

DDERR_SURFACENOTATTACHED
The requested surface is not attached.

DDERR_TOOBIGHEIGHT
The height requested by DirectDraw is too large.

DDERR_TOOBIGSIZE
The size requested by DirectDraw is too large. However, the individual height and width are valid sizes.

DDERR_TOOBIGWIDTH
The width requested by DirectDraw is too large.

DDERR_UNSUPPORTED
The operation is not supported.

DDERR_UNSUPPORTEDFORMAT
The FourCC format requested is not supported by DirectDraw.

DDERR_UNSUPPORTEDMASK
The bitmask in the pixel format requested is not supported by DirectDraw.

DDERR_UNSUPPORTEDMODE
The display is currently in an unsupported mode.

DDERR_VERTICALBLANKINPROGRESS
A vertical blank is in progress.

DDERR_VIDEONOTACTIVE
The video port is not active.

DDERR_WASSTILLDRAWING
The previous blit operation that is transferring information to or from this surface is incomplete.

DDERR_WRONGMODE
This surface cannot be restored because it was created in a different mode.

DDERR_XALIGN
The provided rectangle was not horizontally aligned on a required boundary.

E_INVALIDINTERFACE
The specified interface is invalid or does not exist.

E_OUTOFMEMORY
Not enough free memory to complete the method.

Pixel Format Masks

[This is preliminary documentation and subject to change.]

This section contains information about the pixel formats supported by the hardware-emulation layer (HEL). The following topics are discussed:

- Texture Map Formats

- Off-Screen Surface Formats

Texture Map Formats

[This is preliminary documentation and subject to change.]

A wide range of texture pixel formats are supported by the HEL. The following table shows these formats. The Masks column contains the red, green, blue, and alpha masks for each set of pixel format flags and bit depths.

Pixel format flags	Bit depth	Masks
DDPF_RGB | DDPF_PALETTEINDEXED1	1	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED1 | DDPF_PALETTEINDEXEDTO8	1	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED2	2	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED2 | DDPF_PALETTEINDEXEDTO8	2	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED4	4	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED4 | DDPF_PALETTEINDEXEDTO8	4	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB |	8	R: 0x00000000

DDPF_PALETTEINDEXED8		G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB	8	R: 0x000000E0 G: 0x0000001C B: 0x00000003 A: 0x00000000
DDPF_RGB | DDPF_ALPHAPIXELS	16	R: 0x00000F00 G: 0x000000F0 B: 0x0000000F A: 0x0000F000
DDPF_RGB	16	R: 0x0000F800 G: 0x000007E0 B: 0x0000001F A: 0x00000000
DDPF_RGB	16	R: 0x0000001F G: 0x000007E0 B: 0x0000F800 A: 0x00000000
DDPF_RGB	16	R: 0x00007C00 G: 0x000003E0 B: 0x0000001F A: 0x00000000
DDPF_RGB | DDPF_ALPHAPIXELS	16	R: 0x00007C00 G: 0x000003E0 B: 0x0000001F A: 0x00008000
DDPF_RGB	24	R: 0x00FF0000 G: 0x0000FF00 B: 0x000000FF A: 0x00000000
DDPF_RGB	24	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000 A: 0x00000000

DDPF_RGB	32	R: 0x00FF0000 G: 0x0000FF00 B: 0x000000FF A: 0x00000000
DDPF_RGB	32	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000 A: 0x00000000
DDPF_RGB | DDPF_ALPHAPIXELS	32	R: 0x00FF0000 G: 0x0000FF00 B: 0x000000FF A: 0xFF000000
DDPF_RGB | DDPF_ALPHAPIXELS	32	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000 A: 0xFF000000

The HEL can create these formats in system memory. The DirectDraw device driver for a 3-D–accelerated display card may create textures of other formats in display memory. Such a driver exports the DDSCAPS_TEXTURE flag to indicate that it can create textures.

Off-Screen Surface Formats

[This is preliminary documentation and subject to change.]

The following table shows the pixel formats for off-screen plain surfaces supported by the DirectX® 5 HEL. The Masks column contains the red, green, blue, and alpha masks for each set of pixel format flags and bit depths.

Pixel format flags	**Bit depth**	**Masks**
DDPF_RGB | DDPF_PALETTEINDEXED1	1	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED2	2	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000

DDPF_RGB | DDPF_PALETTEINDEXED4	4	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED8	8	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB	16	R: 0x0000F800 G: 0x000007E0 B: 0x0000001F A: 0x00000000
DDPF_RGB	16	R: 0x00007C00 G: 0x000003E0 B: 0x0000001F A: 0x00000000
DDPF_RGB	24	R: 0x00FF0000 G: 0x0000FF00 B: 0x000000FF A: 0x00000000
DDPF_RGB	24	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000 A: 0x00000000
DDPF_RGB	32	R: 0x00FF0000 G: 0x0000FF00 B: 0x000000FF A: 0x00000000
DDPF_RGB	32	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000 A: 0x00000000
DDPF_RGB |	32	R: 0x0000F800

DDPF_ZPIXELS		G: 0x000007E0 B: 0x0000001F Z: 0xFFFF0000
DDPF_RGB |	32	R: 0x00007C00
DDPF_ZPIXELS		G: 0x000003E0 B: 0x0000001F Z: 0xFFFF0000

In addition to supporting a wide range of off-screen surface formats, the HEL also supports surfaces intended for use by Direct3D, or other 3-D renderers.

Four Character Codes (FOURCC)

[This is preliminary documentation and subject to change.]

DirectDraw utilizes a special set of codes that are four characters in length. These codes, called four character codes or FOURCCs, are stored in file headers of files containing multimedia data such as bitmap images, sound, or video. FOURCCs describe the software technology that was used to produce multimedia data. By implication, they also describe the format of the data itself.

DirectDraw applications use FOURCCs for image color and format conversion. If an application calls the **DirectDrawSurface4.GetPixelFormat** method to request the pixel format of a surface whose format is not RGB, the **lFourCC** member of the **DDPIXELFORMAT** type identifies the FOURCC when the method returns. For more information, see Converting Color and Format.

In addition, the **biCompression** member of the **BITMAPINFOHEADER** type can be set to a FOURCC to indicate the codec used to compress or decompress an image.

FOURCCs are registered with Microsoft by the vendors of the respective multimedia software technologies. Some common FOURCCs appear in the following list.

FOURCC	Company	Technology Name
AUR2	AuraVision Corporation	AuraVision Aura 2: YUV 422
AURA	AuraVision Corporation	AuraVision Aura 1: YUV 411
CHAM	Winnov, Inc.	MM_WINNOV_CAVIARA_CHAMPAGNE
CVID	Supermac	Cinepak by Supermac
CYUV	Creative Labs, Inc	Creative Labs YUV
FVF1	Iterated Systems, Inc.	Fractal Video Frame
IF09	Intel Corporation	Intel Intermediate YUV9
IV31	Intel Corporation	Indeo 3.1
JPEG	Microsoft Corporation	Still Image JPEG DIB
MJPG	Microsoft Corporation	Motion JPEG Dib Format

MRLE	Microsoft Corporation	Run Length Encoding
MSVC	Microsoft Corporation	Video 1
PHMO	IBM Corporation	Photomotion
RT21	Intel Corporation	Indeo 2.1
ULTI	IBM Corporation	Ultimotion
V422	Vitec Multimedia	24 bit YUV 4:2:2
V655	Vitec Multimedia	16 bit YUV 4:2:2
VDCT	Vitec Multimedia	Video Maker Pro DIB
VIDS	Vitec Multimedia	YUV 4:2:2 CCIR 601 for V422
YU92	Intel Corporation	YUV
YUV8	Winnov, Inc.	MM_WINNOV_CAVIAR_YUV8
YUV9	Intel Corporation	YUV9
YUYV	Canopus, Co., Ltd.	BI_YUYV, Canopus
ZPEG	Metheus	Video Zipper

DirectDraw Samples

[This is preliminary documentation and subject to change.]

This section gives brief descriptions of sample applications in the DirectX Programmer's Reference primarily intended to demonstrate the DirectDraw component. The following sample programs are included:

- DDEnum Sample
- DDEx1 Sample
- DDEx2 Sample
- DDEx3 Sample
- DDEx4 Sample
- DDEx5 Sample
- DDOverlay Sample
- Donut Sample
- Flip2D Sample
- FSWindow Sample
- Font Sample
- Memtime Sample
- Mosquito Sample

- Multimonitor Space Donuts Sample
- Space Donuts Sample
- Stretch Sample
- Stretch2 Sample
- Stretch3 Sample
- Switcher Sample
- Wormhole Sample

Although DirectX samples include Microsoft® Visual C++® project workspace files, you might need to verify other settings in your development environment to ensure that the samples compile properly. For more information, see Compiling DirectX Samples and Other DirectX Applications.

DDEnum Sample

[This is preliminary documentation and subject to change.]

Description

This sample shows how to enumerate the current DirectDraw devices and how to get information using the **IDirectDraw4::GetDeviceIdentifier** method.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\DDEnum

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

The user interface is a simple dialog box. Two buttons, **Prev** and **Next**, display information for the previous or next device that was enumerated. Click the **Close** button to exit the application.

DDEx1 Sample

[This is preliminary documentation and subject to change.]

Description

DDEx1 demonstrates the tasks required to initialize and run a DirectDraw application.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\Ddex1

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

DDEx1 requires no user input. Press the F12 key or the ESC key to quit the program.

Programming Notes

This program shows how to initialize DirectDraw and create a DirectDraw surface. It creates a back buffer and uses page flipping to alternately display the contents of the front and back buffers. Other techniques demonstrated include color fills and using GDI functions on a DirectDraw surface.

DDEx2 Sample

[This is preliminary documentation and subject to change.]

Description

The DDEx2 program is an extension of DDEx1 that adds a bitmap.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\Ddex2

Executable: *(SDK root)*\Samples\Multimedia\DDraw\bin

User's Guide

DDEx2 requires no user input. Press F12 or ESC to quit the program.

Programming Notes

DDEx2 shows how to set a palettized video mode. Routines in DDutil.cpp load a bitmap file and copy it to a DirectDraw surface.

DDEx3 Sample

[This is preliminary documentation and subject to change.]

Description

The DDEx3 program is an extension of DDEx2. This example demonstrates the use of off-screen surfaces.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\Ddex3

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

DDEx3 requires no user input. Press F12 or ESC to quit the program.

The program requires at least 1.2 MB of video RAM.

Programming Notes

In addition to the front and back buffers, the program creates two off-screen surfaces and loads bitmaps into them. It calls the **IDirectDrawSurface4::BltFast** method to copy the contents of an off-screen surface to the back buffer, alternating the source surface on each frame. After it blits the bitmap to the back buffer, DDEx3 flips the front and back buffers.

DDEx4 Sample

[This is preliminary documentation and subject to change.]

Description

The DDEx4 program is an extension of DDEx3. It demonstrates a simple animation technique.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\Ddex4

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

DDEx4 requires no user input. Press F12 or ESC to quit the program.

This program requires at least 1.2 MB of video RAM.

Programming Notes

Unlike DDEx3, the DDEx4 program creates only one off-screen surface. It loads a bitmap containing a series of animation images onto this surface. To create the animation, it blits portions of the off-screen surface to the back buffer, then flips the front and back buffers.

The blitting routines illustrate the use of a source color key to create a sprite with a transparent background.

DDEx5 Sample

[This is preliminary documentation and subject to change.]

Description

The DDEx5 program is an extension of DDEx4. It demonstrates a simple palette manipulation.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\Ddex5

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

DDEx4 requires no user input. Press F12 or ESC to quit the program.

This program requires at least 1.2 MB of video RAM.

Programming Notes

The program uses **IDirectDrawPalette::GetEntries** to read a palette, modifies the entries, and then uses **IDirectDrawPalette::SetEntries** to update the palette.

DDOverlay Sample

[This is preliminary documentation and subject to change.]

Description

This sample application demonstrates the use of overlays in a windowed DirectDraw application.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\DDovrly

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

Your hardware must support overlays in order for the program to run.

Try moving, resizing, and minimizing and restoring the window. Press ALT+F4 or click the **Close** button to quit.

Programming Notes

The program checks for overlay support, loads a bitmap into an overlay surface, and updates the window from the overlay surface in response to Windows messages.

Donut Sample

[This is preliminary documentation and subject to change.]

Description

The Donut program uses DirectDraw to display an animated sprite directly on the screen. In non-exclusive mode, the sprite appears on top of the desktop and any windows. In exclusive mode, it appears on an otherwise blank screen.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\Donut

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

The Donut application requires no user input. Press F12 or ESC to quit. Note that because the program does not run in a window, you may have to switch to it by using the taskbar or ALT+TAB before you can close it.

You can specify various command line switches to modify the operational characteristics of this program. Each command line switch consists of one character, and need not be preceded with a hyphen or slash. Alphabetical characters must be capitals. The switches are as follows:

0	Default. Display the donut in the left position.
1	Display the donut in the middle position.
2	Display the donut in the right position.
X	Use exclusive mode. The default is non-exclusive mode.
A	Switch to 640x480x8 resolution and use exclusive mode.
B	Switch to 800x600x8 resolution and use exclusive mode.
C	Switch to 1024x768x8 resolution and use exclusive mode.
D	Switch to 1280x1024x8 resolution and use exclusive mode.

The switches can be combined. If you specify two or more command line switches that contradict each other, the last switch is used.

If you run the program in non-exclusive mode, it attempts to continue to run even when it loses focus. If you run it in exclusive mode, it does not attempt to modify the screen when it doesn't have focus.

Programming Notes

The Donut program creates a primary DirectDraw surface and two off-screen surfaces. Animation images are blitted from the off-screen surfaces directly to the primary surface during each frame.

The program demonstrates how DirectX applications can set the video mode based on user input. It is also useful for testing multiple exclusive mode applications interacting with multiple non-exclusive mode applications.

Flip2D Sample

[This is preliminary documentation and subject to change.]

Description

This sample program demonstrates DirectDraw animation using surface flipping.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\Flip2d

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

The Flip2D program displays and animates a cube. You can change the screen mode by pressing the F10 key and selecting a resolution and color depth value from the **Modes** menu. Use the **Cube** menu or the keyboard shortcuts to increase (F7) or decrease (F8) the displayed size of the cube, or to switch to GDI drawing on the primary surface (F9).

Programming Notes

Normally the cube is rendered on the back buffer and then the surface is flipped to the front. You can compare the results with those from rendering directly on to the front surface by choosing GDI drawing.

The palette that the Flip2D program uses for 8-bpp modes is red, green and blue wash. There isn't enough room in the palette for yellow, orange, and purple. In 8-bpp modes the Flip2D sample application maps yellow, orange and purple to red. If you select **DrawWithGDI** from the **Cube** menu, yellow, orange, purple are drawn without very many shades. This is because the program is using system colors, and there are only a few shades of yellow, orange, and purple available in the system palette.

Font Sample

[This is preliminary documentation and subject to change.]

Description

This sample program shows how to directly lock and access video memory, using text generated from a GDI font. There are much better ways to draw text into a DirectDrawSurface, and the only point of this sample is to show exactly how to lock and access the video memory directly.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\Font

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

The program repeatedly updates a text string in the Arial font and moves it randomly about the screen. No user input is required. Quit by closing the window.

Programming Notes

The program creates a font in a memory device context and a DIB section, and uses them to get access to the pixels once GDI has drawn them. The text bitmap is then put on the primary surface with a straight memory copy. It could easily be moved instead to an off-screen surface which could then be blitted as needed, transparently or not.

FSWindow Sample

[This is preliminary documentation and subject to change.]

Description

This sample shows how you can bring up a dialog box, or any other type of window, while your application is running in DirectDraw's full-screen exclusive mode. Even devices that are non-GDI are detected and supported by creating a bitmap from their window contents and then blitting that to the device.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\FSWindow

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

If you have more than one device, the sample starts by displaying a dialog box so that you can select which device to run the sample on. It then switches to full-screen exclusive mode and displays a dialog box and the mouse cursor.

Click on the **Cancel** or **OK** button to close the dialog and hide the mouse cursor. Press **F1** to bring the dialog and cursor back. Press **Esc** to exit the program when the dialog is no longer displayed.

Programming Notes

Most of the important code is in the Fswindow.cpp file.

The sample uses just the dynamic content mode, which constantly refreshes the dialog box to show which controls have focus, text in the edit field, and so on. The static content mode consumes less CPU time and would be good for pop-up windows that display help messages, for example.

It is important to understand that the content window can be any type of window. It could be an HTML Help window, or a window with a rich edit control to display formatted text. The window does not need to have a square clipping region; it could have a complex clipping region that fits the shape of the window you want to display.

Memtime Sample

[This is preliminary documentation and subject to change.]

Description

The Memtime sample program is intended to be a demonstration of the advantages and disadvantages of various drawing methods. It runs a series of tests of memory bandwidth and displays a report.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\MemTime

Executable: None.

User's Guide

Choose **Time All** from the main menu to begin the tests. Note that your screen will change resolution during the tests. It will also go blank for short periods of time.

Programming Notes

The sample is not intended as an example of particular techniques in DirectDraw programming. Of more interest are the results of the memory tests.

Mosquito Sample

[This is preliminary documentation and subject to change.]

Description

This program demonstrates DirectDraw animation using overlays.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\Mosquito

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

To run the Mosquito application, you must have a display adapter that supports overlays. On a computer with overlay support, the program creates a large mosquito that flies around the screen. If your display adapter card doesn't support source color keying for overlays, you'll see an ugly, black, rectangular background around the mosquito.

Some cards have better overlay support in certain resolutions that others. If you know your card has overlay support through DirectDraw, but the Mosquito program is having problems creating or displaying the overlay, try switching to a lower screen resolution or color depth and restarting the application.

Programming Notes

The program creates a complex overlay surface and animates by flipping.

Multimonitor Space Donuts Sample

[This is preliminary documentation and subject to change.]

Description

This version of Space Donuts demonstrates the multimonitor capabilities of DirectX.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\MultiNut

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

In order to run the program, you must have Windows 2000 or Windows 98 configured for multiple monitors.

The program will run on up to nine monitors. The play area is determined by the monitor configuration of your Windows desktop, which may be adjusted on the **Settings** tab of the Display Control Panel. The amount of video RAM you have on each video card will determine the maximum resolutions at which you will experience a reasonable frame rate. If the frame rate is slow, try exiting, lowering the resolution or color depth of your windows desktop, and restart.

Multimonitor Space Donuts supports two command-line switches:

-w#	Put monitor number # in windowed mode
-s	Turn sound off

For information on the keyboard interface, see Space Donuts.

Space Donuts Sample

[This is preliminary documentation and subject to change.]

Description

This simple game shows how to combine DirectDraw, DirectInput, and DirectSound. Although it demonstrates other DirectX components, it is primarily intended to show how to animate multiple sprites.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\Donuts

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

Input is from the keyboard by default, but you can select a joystick from the **Game** menu.

The commands are listed on the opening screen. All numbers must be entered from the numeric keypad. "Joy" refers to a joystick button.

Key	Command

ESC, F12	Quit
4	Turn left
6	Turn right
5 (Joy 3)	Stop
8	Accelerate forward
2	Accelerate backward
7 (Joy 2)	Shield
SPACEBAR (Joy 1)	Fire
ENTER	Start game
F1	Toggle trailing afterimage effect on/off
F3	Turn audio on/off
F5	Toggle frame rate display on/off
F10	Main menu

Space Donuts defaults to 640x480 at 256 colors. You can specify a different resolution and pixel depth on the command line.

The game uses the following command line switches, which are case-sensitive:

e	Use software emulation, not hardware acceleration
t	Test mode, no input required
x	Stress mode. Never stop if you can help it
S	Turn sound off/on

These switches may be followed by three option numbers representing x-resolution, y-resolution, and bits per pixel. For example:

```
donuts -S 800 600 16
```

Programming Notes

This game demonstrates many of the features of DirectDraw. It takes advantage of hardware acceleration if it is supported by the driver.

The program requires less than 1 MB of video RAM.

The sound code is deliberately designed to stress the DirectSound API. It is not intended to be a demonstration of how to use DirectSound API efficiently. For example, each bullet on the screen uses a different sound buffer. Space Donuts creates over 70 sound buffers (including duplicates), and between 20 to 25 may be playing at any time.

The sounds are implemented using the helper functions in Dsutil.h and Dsutil.c (found in the Sdk\Samples\Misc directory). These functions might help you to add sound to your application quickly and easily.

Stretch Sample

[This is preliminary documentation and subject to change.]

Description

The Stretch sample application program illustrates stretching and clipping while blitting a bitmap image.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\Stretch

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

Stretch must be run in a video mode that uses 8 bits per pixel. It will not work properly in other video modes.

The program displays a red torus moving in its client window. Control the rotational speed with the **Stop**, **Slow**, and **Fast** options in the **Rotation** menu. Alter the size of the window by selecting items from the **Size** menu, or by resizing the window with the mouse.

Programming Notes

Any time you resize the Stretch program window to a size other than 1x1, you are using the image stretching capabilities of the DirectDraw blitting methods.

The clipper for the primary surface is set to the client window. To demonstrate clipping, partially overlap another window over the Stretch program's window. When Stretch blits the bitmap, the portion of the bitmap that would fall within the other window is clipped.

Stretch2 Sample

[This is preliminary documentation and subject to change.]

Description

Stretch2 is an extension of the Stretch program. In addition to the capabilities of the Stretch example, Stretch2 illustrates how an application can use DirectDraw on multiple monitors.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\Stretch2

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

Resize the window to see the bitmap stretch. If you have more than one monitor attached to your computer, the window can be dragged from monitor to monitor.

Programming Notes

Look at how the application handles WM_MOVE to detect when the window moves monitors. Also note how it converts from window client coordinates to device coordinates.

Multimon.h contains stub functions that enable the program to run on Windows 95 or Windows NT 4.0.

Stretch3 Sample

[This is preliminary documentation and subject to change.]

Description

The Stretch3 application is an improved version of the Stretch2 program. It demonstrates more complex handling of DirectDraw images on multiple monitors.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\Stretch3

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

Resize the window to see the bitmap stretch. If you have more than one monitor attached to your computer, the window can be dragged from monitor to monitor.

Switcher Sample

[This is preliminary documentation and subject to change.]

Description

This sample shows how to switch between the normal and exclusive cooperative levels in DirectDraw.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\Switcher

Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

Press ALT+ENTER to switch between full-screen and windowed mode. Quit the program by pressing ESC.

Programming Notes

In normal (windowed) mode, the sample assigns a clipper, shows the mouse cursor, and handles window moves, WM_PAINT messages, and pausing caused by losing focus to other applications. In exclusive (full-screen) mode it uses page flipping rather than blitting to update the scene.

Wormhole Sample

[This is preliminary documentation and subject to change.]

Description

This sample program shows how palette changes can create an animated effect.

Path

Source: *(SDK root)*\Samples\Multimedia\DDraw\Src\Wormhole

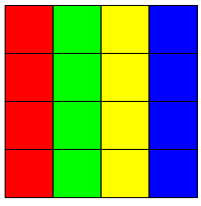
Executable: *(SDK root)*\Samples\Multimedia\DDraw\Bin

User's Guide

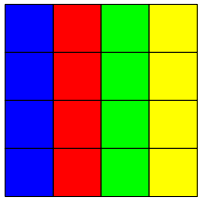
Press F12 or ESC to quit the program.

Programming Notes

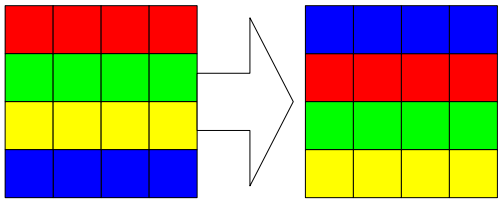
Imagine a 4x4 display using 4 colors. We could set the colors up to look something like this:



Now we can cycle all of the colors in each row to the right. The one on the right will wrap-around to the left.

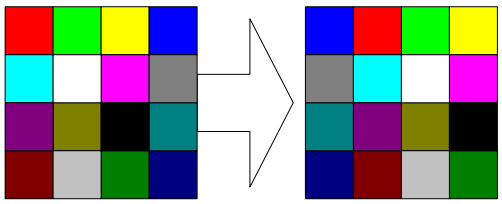


If we continue this cycling we would get animated lines moving to the right. The same can be done to animate the lines going down:

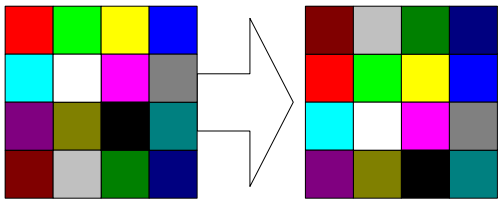


Now if we expand our palette to 16 color we can combine moving down and right at the same time.

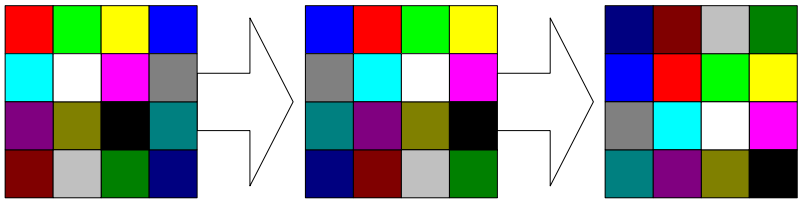
Move right:



Move down:



Move right and down:



Now if you tile these 4x4 blocks end to end and cycle the colors as above, you get a moving checkerboard. Wormhole does the same thing, except that it uses 15x15 blocks (225 colors) and instead of tiling the blocks end to end on a flat plane, it tiles them in 3-D converging at the center of the wormhole.

The following code will generate the 3-D wormhole using the aforementioned 15x15 grids:

```
//Do all the work!
//convert r,theta,z to x,y,x to screen x,y
//plot the point
//z=-1.0+(log(2.0*j/DIVS) is the line that sets the math eqn for plot
//Feel free to try other functions!
//Cylindrical coordinates, e.g. z=f(r,theta)

#define STRETCH 25
#define PI      3.14159265358979323846
#define XCENTER 160
#define YCENTER 50
#define DIVS    1200
#define SPOKES  2400

void transarray(void)
   {
   float x,y,z;
   int i,j,color;
   for(j=1;j<DIVS+1;j++)
      for(i=0;i<SPOKES;i++)
         {
         z=-1.0+(log(2.0*j/DIVS));
         x=(320.0*j/DIVS*cos(2*PI*i/SPOKES));
         y=(240.0*j/DIVS*sin(2*PI*i/SPOKES));
         y=y-STRETCH*z;
         x+=XCENTER;
         y+=YCENTER;
         color=((i/8)%15)+15*((j/6)%15)+1;
         if ((x>=0)&&(x<=320)&&(y>=0)&&(y<=200))
            plot((int) x,(int) y,color);
         }
   }
```

After loading the bitmap to a DirectDraw surface, all that is left to do is rotate the colors and you have a wormhole.