Microsoft® Windows™

Version 3.1

Device Driver Adaptation Guide

For the Microsoft Windows Operating System

Microsoft Corporation

Document No. PC29132-0392

Contents

Overview of Windows Drivers

Chapter 1

This chapter describes the purpose and function of Microsoft Windows 3.1 device drivers. You should create a Windows device driver for your device if it is not 100 percent compatible with the devices supported by the Windows 3.1 retail device drivers, or if you want to offer Windows users access to unique features of your device.

1.1 What Is a Device Driver?

A Windows device driver is a dynamic-link library (DLL) that Windows uses to interact with a hardware device such as a display or a keyboard. Rather than access devices directly, Windows loads device drivers and calls functions in the drivers to carry out actions on the device. Each device driver exports a set of functions; Windows calls these functions to complete an action, such as drawing a circle or translating a keyboard scan code. The driver functions also contain the device-specific code needed to carry out actions on the device.

Windows requires device drivers for the display, keyboard, and communication ports. Mouse, network, and printer drivers are required if the user adds these optional devices to the system. The following is a brief description of each type of driver.

Driver	Description
Communications	Supports serial and parallel device communications. Windows loads and enables this driver, checking the COMM.DRV setting in the SYSTEM.INI file to determine the filename of the driver to load. The communications driver must provide functions to enable and disable the communication device, to get and set the device status, and read and write data through the device. The USER module provides a general interface for Windows applications to call, and translates these calls into appropriate calls to the driver. The module name for the communications driver is COMM.
Display	Supports the system display and cursor for pointing devices. Windows loads and enables the display driver, checking the DISPLAY.DRV setting in the SYSTEM.INI file to determine the filename of the driver to load. The display driver must provide functions to enable and disable the device, get information about the capabilities of the device, carry out graphics operations such as drawing lines and transferring bitmaps, and to show and hide a cursor. Windows and Windows-based applications call functions in the GDI module to carry out graphics operations on the display, and GDI translates these calls into corresponding calls to the driver. Depending on the capabilities of the display device, GDI may generate many calls to the driver from a single call from an application. The module name for the display driver is DISPLAY.

Driver	Description
Grabber	Supports the management of non-Windows applications. Although not technically a driver, a display grabber plays a similar role as a device driver in helping the WINOLDAP module manage non-Windows applications. A display grabber provides the support Windows needs to share the display device with non-Windows applications. WINOLDAP loads the display grabbers and calls grabber functions to carry out tasks such as capturing the contents of the screen, or managing output from a non-Windows application. Windows requires unique display grabbers for standard and 386 enhanced modes.
Keyboard	Supports keyboard input. Windows loads and enables the keyboard driver, checking the KEYBOARD.DRV setting in the SYSTEM.INI file to determine the filename of the driver to load. The keyboard driver must provide functions to enable and disable the keyboard, and to translate keyboard scan codes into character values and virtual-key codes. A keyboard driver also replaces the MS-DOS keyboard-interrupt handler with its own. When the driver is enabled, the USER module provides the address of a callback function that the driver calls whenever an event occurs, such a keystroke. The module name for the keyboard driver is KEYBOARD.
Mouse	Supports mouse or other pointing device input. Because a mouse is optional, Windows checks the MOUSE.DRV setting in the SYSTEM.INI file to determine whether to load a driver. A mouse driver must provide functions to enable and disable the mouse, retrieve information about the mouse, and allow users to modify the operation of the mouse through Control Panel. When the driver is enabled, the USER module provides the address of a callback function that the driver calls whenever an event occurs, such as a mouse movement. The module name for the mouse driver is MOUSE.
Network	Supports networks. Because a network is optional, Windows checks the NETWORK.DRV setting in the SYSTEM.INI file to determine whether to load a driver. A network driver must provide functions to retrieve information about the network, redirect local drives, and add jobs to a network print queue. The network driver may use MS-DOS functions, NetBIOS routines, and network software to complete these network requests. Windows does not require a specific module name for the network driver.

Driver	Description
Printer	Supports printer output. Windows applications indirectly load printer drivers by calling the **CreateDC** function in the GDI module. A printer driver must provide functions to enable and disable the printer, to get information about the capabilities of the printer, to carry out graphics operations such as drawing lines and transferring bitmaps, and to display dialog boxes to let the user change printer settings. Windows and Windows-based applications call functions in the GDI module to carry out graphics operations on the printer, and GDI translates these calls into corresponding calls to the driver. Windows does not require a specific module name for a printer driver.

Since the network and printer drivers are optional, their module names are not reserved. However, you should name your driver to represent your device appropriately. For example, you could use PSCRIPT for a PostScript® printer driver or MSNET for an MS®-Network driver.

1.2 Creating a Device Driver

You create a device driver either by adapting a sample driver, or writing a driver from scratch. You can write Windows device drivers in assembly language or in a high-level language such as the C language. Assembly language programmers can use the CMACROS assembly-language macro package.

To create a device driver, you need to:

1. Read the chapter in this manual that describes the driver for your type of device.
2. Write the required driver functions.
3. Create and compile the required resources.

   Every device driver must have at least a **VERSIONINFO** resource that contains the version stamp for the driver. Setup and Control Panel both look for this resource when installing drivers. For more information about **VERSIONINFO** and other resources, see the *Microsoft Windows Programmer's Reference.*
4. Create a module-definition file that identifies the appropriate module name for your driver, and exports the required functions.
5. Assemble and link your driver.
6. Test your driver using the debugging version of Windows.
7. Create an installation file (OEMSETUP.INF) for your driver and related files.
8. Create your final distribution disk or disks.

1.3 Guidelines for Designing and Writing a Driver

When designing and writing your device driver, follow these guidelines:

- Make every effort to make your device driver as small as possible; reserve system memory for applications.
- Use multiple, discardable code segments to help reduce the amount of driver code needed in memory at any given time.
- Use an automatic data segment only if necessary.
- Make resources discardable, and lock them in memory only when needed.
- Use the stack sparingly. Because device drivers use the stack of the application that initiated the call to the driver, there is no way for the driver to determine how much available space is on the stack.
- Check for NULL pointers to avoid a general protection faults from using an invalid selector.
- Check the segment limits when reading from or writing to allocated segments to avoid a general protection fault from attempting to access data beyond the end of a segment.
- Use the **__ahincr** constant when creating selectors for huge memory (allocated memory greater than 64 kilobytes). Other methods of selector arithmetic can create invalid selectors and cause general protection faults.
- Create code-segment aliases for any code to be executed from data segments. Attempting to call or jump to a data segment address generates a general protection fault.

1.4 Windows Calling Conventions

This manual presents the syntax of most functions in C-language notation. All such functions are assumed to be declared as FAR PASCAL functions, and Windows will call these functions as such. In general, exported functions in a device driver must execute the standard Windows prolog on entry and epilog on exit. For more information about the prolog and epilog, see the *Microsoft Windows Programmer's Reference*.

The following list highlights the calling conventions:

- Set the DS register to the selector of the driver's automatic data segment.
- Save and restore the following registers if used: SS, SP, BP, SI, DI, and DS.
- Clear the direction flag if it has been set or modified.

- Place 16-bit return values in the AX register; 32-bit values in the DX:AX register pair.
- Execute a FAR return.

Windows pushes all parameters on the stack in a left to right order (the last parameter shown in the function syntax is closest to the stack pointer). Windows also passes pointers parameters as 32-bit quantities, pushing the selector first then the offset. This allows exported functions to use the **lds** or **les** instructions to retrieve pointers from the stack.

1.5 Header Files

When writing assembly-language drivers, you may need to use the following header files.

File	Description
CMACROS.INC	Contains a set of assembly-language macros that provide a simplified interface to the function and segment conventions of high-level languages, such as C and Pascal.
GDIDEFS.INC	Contains definitions for symbolic constants and structures. To shorten the assembly time and cross-reference lists, you can selectively include parts of GDIDEFS.INC by defining equates that tell the assembler which parts to include.

Equate	Description
incLogical equ 1	Includes logical pen, brush, and font definitions.
incDevice equ 1	Includes the symbolic names for **GDIINFO** definitions.
incFont equ 1	Includes the **FONTINFO** and **TEXTXFORM** definitions.
incDrawMode equ 1	Includes the **DRAWMODE** data structure definitions.
incOutput equ 1	Includes the output style constants.
incControl equ 1	Includes the escape number definitions.

WINDEFS.INC	Contains definitions for symbolic constants and structures used with Windows functions.

Display Drivers

Chapter 2

The Microsoft Windows display driver manages all screen output for Windows applications. A display driver provides a set of functions that Windows uses to enable the display hardware, retrieve information about the display, and draw text and graphics.

2.1 About the Display Driver

The display driver is a dynamic-link library that consist of a set of graphics functions for a particular display device. These functions translate device-independent graphics commands from the graphic-device interface (GDI) into the commands and actions the display device needs to draw graphics on the screen. The functions also give information to Windows and Windows applications about color resolution, screen size and resolution, graphics capabilities, and other advanced features that may be available on the hardware. Applications use this information to create the desired screen output.

Although Windows requires only a few functions to start, each Windows application the user starts can potentially use any GDI functions. This means a display driver should provide as complete support for GDI as possible.

2.1.1 Display-Driver Initialization

Display-driver initialization occurs when Windows creates the original-device context for the Windows desktop. To create the device context, Windows loads the display driver and calls the driver's initialization routine.

Although the initialization routine can carry out any task, many drivers do the following:

- Determine whether 386 enhanced-mode Windows screen switching is required
- Initialize display hardware
- Determine whether mouse trails support is required
- Install any modal functions

Modal functions have implementations based on CPU type, hardware configuration, or Windows mode of operation. For example, the **ExtTextOut** function for a 80386 CPU may use 32-bit registers but the same function for a 80826 uses 16-bit registers. In another example, a driver may install cursor functions for a hardware cursor in one hardware configuration and install functions for a software cursor in another.

Although a display driver may carry out some hardware initialization in its initialization routine, it should wait until GDI calls the driver's **Enable** function for a second time before fully initializing the video hardware.

The initialization routine returns to Windows if the initialization was successful. Otherwise, it returns zero and Windows immediately terminates.

2.1.2 GDI Information Structure

Every display driver has a **GDIINFO** structure that specifies the display's capabilities and characteristics. GDI uses this information to determine what the display driver can do and what GDI must simulate. The GDI information can be classified as follows:

- Driver management
- Driver capabilities
- Device dimensions

The driver-management information specifies the version of Windows for which the driver was written, and the type of technology the display uses to generate output. Additionally, the driver-management information also specifies the size in bytes of the **PDEVICE** structure, and number of device contexts the driver can manage at the same time. The version number specifies a Windows version (not the display driver version). For example, a display driver written for Windows 3.1 should set the **dpVersion** member to 0x30A.

The driver-capabilities information specifies the capabilities of the display device, such as whether the display hardware can draw polygons and ellipses, scale text, or clip output. Driver capabilities also specify the number of brushes, pens, fonts, and colors available on the display and whether the display can handle bitmaps and color palettes.

The device-dimension information specifies the maximum width and height of the screen in both millimeters and device units, the number of color bits or planes, the aspect ratio, the minimum length of a dot in a styled line, and the number of device units (or pixels per inch).

The subsequent sections of this chapter describe the **GDIINFO** structure more fully. Each section describes the capabilities associated with given members and explains how to determine what capabilities a display driver can support.

2.1.3 Enabling and Disabling the Physical Device

GDI enables the display driver by calling the **Enable** function and directing the driver to initialize a physical device for subsequent graphics output. A physical device is a **PDEVICE** structure that represents the display and its current operating state. A display driver uses the physical device information to determine how to carry out specific tasks, such as which display mode to use. The display driver initializes the physical device by copying information to the **PDEVICE** structure.

In 386 enhanced-mode Windows, GDI calls the **Enable** function only when the display driver is first loaded. In standard-mode Windows, GDI calls the **Enable** function when first loaded and whenever the user switches back to Windows from a non-Windows application.

GDI calls **Enable** twice: Once to retrieve a copy of the driver's **GDIINFO** structure, and a second time to initialize the **PDEVICE** structure. After the first call, GDI uses the **dpDEVICEsize** member in the **GDIINFO** structure to determine the size of the driver's **PDEVICE** structure. GDI then allocates memory for the structure and calls **Enable** for the second time, passing a pointer to structure. At this point, the driver initializes the display hardware and the structure.

When Windows switches back from a non-Windows application, GDI calls **Enable** once. The driver reinitializes the display hardware and the **PDEVICE** structure, reinitializing any screen data that may have been discarded when Windows switched to the non-Windows application.

Although only the display driver initializes and uses the **PDEVICE** structure, it is GDI that allocates memory for the structure, determines when to pass it to the driver's output functions, and deletes the structure when it is no longer needed. Except for the first two bytes (16 bits) of the **PDEVICE** structure, the content and format of the structure depends entirely on the display driver. Typically, the driver includes all the information that the output functions need to generate appropriate graphics commands. The first two bytes, on the other hand, must be set to a non-zero value. GDI reserves zero to indicate a **PBITMAP** structure. GDI creates and uses **PBITMAP** structures in place of **PDEVICE** structures when an application creates a memory-device context.

GDI disables the display driver by calling the **Disable** function when Windows quits. GDI expects the driver to free any resources associated with the physical device and to restore the display hardware to the state before Windows started. After the driver returns from the **Disable** function, GDI frees the memory it allocated for the **PDEVICE** structure and frees the driver, removing any driver code and data from memory.

In standard-mode Windows, GDI also calls **Disable** when the user switches to a non-Windows application. In this case, GDI temporarily disables the physical device, expecting the driver to select a text mode for the display hardware so that the non-Windows application has a nongraphics mode in which to start. Although Windows saves the display driver's data segment when it switches, it discards all other segments. Therefore, the display driver should save any data that may be discarded so that the data can be restored when Windows switches back from the non-Windows application.

2.1.4 Hardware Initialization

A display driver sets the display mode and registers for the display hardware whenever GDI calls the **Enable** and **Disable** functions. When running under 386 enhanced-mode Windows (or other operating systems featuring pre-emptive multitasking), the driver is also responsible for saving and restoring the display mode and registers whenever Windows is switching between Windows applications and non-Windows applications.

2.1.4.1 Enabling and Disabling the Display Hardware

A display driver prepares the display hardware for Windows whenever GDI calls the **Enable** function. To prepare the hardware, the display driver saves the current display mode, then sets the display hardware to graphics mode, initializing hardware registers as needed. Although GDI calls the **BitBlt** function to clear the screen as Windows starts, many display drivers eliminate the possibility of the user seeing any random data by also clearing the screen as they initialize the display hardware.

A display driver restores the display hardware to its original state whenever GDI calls the **Disable** function. To restore the display hardware, the driver sets the display hardware to a text mode, and restores the original number of lines. If possible, the driver should use the same text mode as before Windows started.

2.1.4.2 Screen Switching

In a pre-emptive multitasking environment, such as 386 enhanced-mode Windows, the display driver should save and restore the display hardware whenever the environment switches Windows to or from the foreground. 386 enhanced-mode Windows switches Windows to and from the foreground whenever the user switches to or from non-Windows applications. Saving and restoring the display hardware ensures that the display driver is not affected by changes non-Windows applications make to the display hardware.

To detect screen switches, the display driver hooks Interrupt 2Fh and checks for the Notify Background Switch and Notify Foreground Switch functions (Interrupt 2Fh Functions 4001h and 4002h). When the driver detects one of these functions, it either saves or restores the display mode and registers accordingly. When the driver detects a switch to the foreground, the driver also calls the **RepaintScreen** function (USER.275) to direct Windows to restore the entire contents of the screen by repainting it.

The display driver should hook Interrupt 2Fh whenever GDI calls the **Enable** function. The driver hooks the interrupt by using the MS-DOS functions Get Interrupt Vector (Interrupt 21h Function 35h) and Set Interrupt Vector (Interrupt 21h Function 25h).

A display driver can temporarily disable screen switching by calling the Enter Critical Section function (Interrupt 2Fh Function 4003h). This function prevents 386 enhanced-mode Windows from proceeding with a switch until the display driver calls Exit Critical Section (Interrupt 2Fh Function 4004h).

If a display driver uses portions of video memory that are not used by the current display mode (for example, the driver uses extra video RAM for saving screen bitmaps), the driver should call the Enable VM-Assisted Save/Restore function (Interrupt 2Fh Function 4000h) to pass 386 enhanced-mode Windows the address of save and restore flags. Windows sets or clears these flags depending on whether it used the extra video memory while a non-Windows application was in the foreground. In such cases, the driver must call the Disable VM-Assisted Save/Restore function (Interrupt 2Fh Function 4007h) to disable this feature whenever GDI calls the **Disable** function.

Although the display driver is responsible for repainting the screen when Windows is switched to the foreground, occasionally the driver must postpone repainting because Windows is in a critical section and is not ready to process the repainting. In such cases, Windows calls the **UserRepaintDisable** function in the display driver directing the driver to postpone repainting. Windows will call **UserRepaintDisable** a second time when it completes the critical section.

2.1.5 Physical Objects

Physical objects define the attributes (such as color, width, and style) of lines, patterns, and characters drawn by a display driver. Physical objects correspond to the logical pens, brushes, and fonts that Windows applications create but contain device-dependent information that the display driver needs to generate output. A display driver creates physical objects when GDI calls the **RealizeObject** function. The driver uses physical objects when GDI calls output functions such as **Output**, **BitBlt**, and **ExtTextOut**.

The **dpNumPens**, **dpNumBrushes**, and **dpNumFonts** members in the **GDIINFO** structure specify the number of pens, brushes, and fonts a display driver supports. A display driver must supply information about these objects whenever GDI requests it. GDI calls the **EnumObj** function to request information about pens and brushes; it calls the **EnumDFonts** function to request information about device fonts. For each pen, brush, or font, the display driver calls the callback function. GDI pass a **LPEN**, **LBRUSH**, or **LFONT** structure to the callback function. These structures specify the attributes of the object.

Although a display driver can use display hardware to support objects, the driver must be able to generate the same output both on the screen and in memory bitmaps.

2.1.5.1 Physical Pens

A physical pen specifies the color, style, and width of polylines and borders drawn by a display driver. A display driver realizes a physical pen by filling a **PPEN** structure with information about the pen. The content and format of the **PPEN** structure depends entirely on the display driver. In general, the driver copies all the information the **Output** function need to draw lines.

A display driver should support the standard GDI pen styles: Solid, dashed, dotted, dot-dashed, dash-dot-dotted, and empty. A display driver must support the empty style although it is not required to supply information about it when GDI calls **EnumObj**. When drawing with a empty pen, the pen itself does not contribute to the output, but the driver may still draw a line if, for example, the raster operation combines the destination with itself using the XOR operator (DDx) or inverts the destination (Dn).

A display driver that supports wide and styled lines must use the same drawing algorithms for lines drawn on the screen and in memory bitmaps. Because GDI efficiently synthesizes both wide and styled lines, some display drivers do not support them.

Under certain conditions, GDI may pass the display driver a request to realize a wide or styled line even though the display driver has specified that it does not support them. In such cases, the driver should realize a solid, one-pixel wide (or nominal) pen. GDI will use this pen to simulate styled and wide lines.

2.1.5.2 Physical Brushes

A physical brush specifies the color and style of patterns used to fill figures drawn by the **Output** function and to combine with bitmaps drawn by the **BitBlt** function. A display driver realizes a physical brush by filling a **PBRUSH** structure with information about the brush. The content and format of the **PBRUSH** structure depends entirely on the display driver.

A display driver should support the standard GDI brush styles: solid, hatched, patterned, and hollow. A display driver must support hollow brushes although it is not required to provide information about it when GDI calls the **EnumObj** function. When drawing with a hollow brush, the brush itself does not contribute to the output, but other factors, such as a raster operation that combines the destination with itself or inverts the destination, can cause the driver to generate output.

The display driver can dither solid brushes if the specified color does not exactly match a physical color. Otherwise, it should choose the closest available color for the brush.

The display driver should support the standard hatched brush styles: horizontal, vertical, forward diagonal, backward diagonal, cross, and diagonal crosshatch. On raster displays, a driver typically implements these styles as predefined, 8-by-8 bit patterns.

On color displays, the display driver should support both monochrome and color bitmaps for patterned brushes. When drawing monochrome bitmaps, the driver sets 1 bit to the current text color and 0 bits to the current background color. The text and background colors are specified in the **DRAWMODE** structure passed to these functions. A display driver is not required to provide information about patterned brushes when GDI calls the **EnumObj** function.

2.1.6 Physical Colors

A display driver is responsible for translating logical colors (RGB values) into physical colors that are appropriate to the display hardware. Similarly, it must translate physical colors to logical colors. GDI calls the **ColorInfo** function whenever it needs a translated color.

If the display device provides a color palette, the driver converts colors to palette indexes. To indicate a palette index, the driver always sets the high byte of the index to 0xFF.

To indicate an RGB color value, the display driver sets the high byte to zero.

2.1.7 Screen Metrics

The screen metrics, specified by the **GDIINFO** structure, define such items as width and height in millimeters, screen resolution, aspect ratio, and mapping modes. GDI uses screen metrics to generate coordinate data that is appropriate for the display hardware.

2.1.7.1 Logical Pixels Per Inch

A display driver sets the **dpLogPixelsX** and **dpLogPixelsY** members to specify the number of pixels per logical inch along horizontal and vertical lines on the screen. A display driver uses logical inches (about 40 percent larger than physical inches) for readability reasons.

The GDI font mapper uses these values to determine which screen fonts to use with the display. The display driver should make sure the **dpLogPixelsX** and **dpLogPixelsY** members match an existing font. If these members do not match one of the default screen fonts, an appropriate font must be provided with the display driver.

2.1.7.2 Screen Resolution and Size

A display driver sets the **dpHorzRes** and **dpVertRes** members to specify the width and height of the screen in pixels, and sets the **dpHorzSize** and **dpVertSize** members to specify the width and height of the screen in millimeters. These values must have the following relationships:

```
dpHorzSize = (dpHorzRes/dpLogPixelsX) * 25.4

dpVertSize = (dpVertRes/dpLogPixelsY) * 25.4
```

In these equations, 25.4 represents the number of millimeters per inch.

2.1.7.3 Aspect Ratios

The aspect ratio defines the relative dimensions of the display's pixels. The ratio consists of three values: an *x*-, *y*-, and an *xy*-aspect. These represent the relative width, height, and diagonal length (or hypotenuse) of a pixel. GDI uses the aspect ratio to determine how to draw squares and circles as well as drawing lines at an angle.

The aspect values have the following relationship:

```
dpAspectXY ** 2 == (dpAspectX ** 2) + (dpAspectY ** 2)
```

Since the dimensions are given as relative values, they may be scaled as needed to get accurate integer values. They should be kept under 1000 for numerical stability in GDI calculations. For example, a device with a 1:1 aspect ratio (such as a VGA) can use 100 for **dpAspectX** and **dpAspectY** and 141 (100 * 1.41421...) for **dpAspectXY**.

2.1.7.4 Styled-Line Length

The styled-line length (**dpStyleLen**) specifies the length of the smallest line segment the display driver uses to build the dots and dashes of a styled line. GDI uses this number when it draws into bitmaps and on displays. To ensure consistency between displays and printers, the styled line segment length is always two times the value of the **dpAspectXY** member.

2.1.7.5 Standard Mapping Modes

Some Windows application programs rely on standard mapping modes to produce printer output with spacing that is proportional to the screen. By using the standard mapping modes, an application can show a border or graphic picture that is proportionately the same size on the printer as it is on the screen.

All standard mapping-mode ratios must be the same because it might be preferable for an application to use the metric system rather than the inches/feet (English) system for its calculations. For example, Windows Write allows the user to choose whether to express the border widths in millimeters or inches. Therefore, it is up to the display driver to provide the correct numbers.

Standard mapping modes are expressed as two coordinate pairs: the width and height (in logical units) of a "window" and the width and height (in physical units, that is, pixels) of a "viewport" that maps onto that window. The driver for a VGA adapter, for example, might set the coordinates pairs for the low-resolution metric mapping mode (tenths-of-millimeters) to (254,254) and (96,96). These coordinates map an 1-inch by 1-inch window (25.4 millimeters equals 1 inch) to a 96-pixel by 96-pixel viewport. These coordinate pairs define a set of equations that specify how coordinates in logical space are transformed to coordinates in device space.

The standard mapping-mode members in the **GDIINFO** structure can be set as follows:

```
dpMLoWin.x = dpHorzSize*10;     dpMLoWin.y = dpVertSize *10;
dpMLoVpt.x = dpHorzRes;         dpMLoVpt.y = -dpVertRes;

dpMHiWin.x = dpHorzSize*100;    dpMHiWin.y = dpVertSize *100;
dpMHiVpt.x = dpHorzRes;         dpMHiVpt.y = -dpVertRes;

dpELoWin.x = dpHorzSize*1000;   dpELoWin.y = dpVertSize *1000;
dpELoVpt.x = dpHorzRes * 254;   dpELoVpt.y = -dpVertRes * 254;

dpEHiWin.x = dpHorzSize*10000;  dpEHiWin.y = dpVertSize *10000;
dpEHiVpt.x = dpHorzRes * 254;   dpEHiVpt.y = -dpVertRes * 254;

dpTwpWin.x = dpHorzSize*14400;  dpTwpWin.y = dpVertSize *14400;
dpTwpVpt.x = dpHorzRes * 254;   dpTwpVpt.y = -dpVertRes * 254;
```

A twip is 1/20th of a printer's point (1/72 of an inch).

Windows performs 16-bit, signed calculations on these values, so the numbers must not be greater than 32,768. However, if the screen is larger than just a few inches wide, the values will exceed this limit when calculating the English mapping modes and may even exceed it on the metric mapping modes. Fortunately, the values can be scaled down by dividing by some fixed amount.

2.1.8 Lines, Curves, and Polygons

GDI requires all display drivers to provide an **Output** function that supports a minimum set of line, curve, and polygon drawing capabilities. In particular, a display driver must be able to draw scan lines and polylines. A scan line is a solid or

styled, horizontal line that is exactly one pixel wide. A polyline is a sequence of solid lines, each one pixel wide and each connected at an endpoint to the next line in the sequence.

GDI may call the **Output** function whenever an application calls a function that draws lines, curves, or polygons. If the display driver supports all capabilities, GDI calls the **Output** function for each request. Otherwise, GDI simulates output that the driver does not support by combining scan lines and polylines.

Since GDI can use scan lines and polylines to simulate all other line, curve, and polygon output, many display drivers do not support additional capabilities unless the display hardware can produce faster and higher-quality results than GDI. The only drawback to using display hardware is that the display driver must be able to produce the same results in memory bitmaps as on the screen.

GDI checks the **dpCurve**, **dpLines**, and **dpPolygonals** members of the **GDIINFO** structure to determine what line, curve, and polygon capability the display driver has.

2.1.8.1 Curves

GDI checks the **dpCurve** member to determine whether the display driver supports circles, ellipses, pie wedges, and chord arcs. This member also specifies whether the display driver can draw wide or styled borders for curves and fill the interiors of curves.

If a display driver does not support circles, GDI can use an ellipse to draw a circle. If the display hardware can fill ellipses, the display driver should set the interiors bit. GDI can use an alternate-fill polygon to draw wide borders (both solid and styled) just as efficiently as if the driver supported them correctly.

2.1.8.2 Polylines

GDI checks the **dpLines** member to determine whether the display driver support polylines. This member also specifies whether the display driver can draw wide or styled lines and fill the interiors of wide lines. A display driver must support polylines.

If a display driver supports styled lines, the line segments drawn by the display hardware must be as specified by the **dpStyleLen** member. Although wide, styled lines are used infrequently, the display driver should support them if the display hardware can draw them and the effort to support memory bitmaps is not too great. If necessary, GDI can simulate wide, styled lines.

2.1.8.3 Polygons

GDI checks the **dpPolygonals** member to determine whether the display driver supports rectangles, scan lines, alternate-fill polygons, and winding-number-fill polygons. This member also specifies whether the display driver can draw wide or style borders for polygons, and fill the interiors of polygons. The display driver must support scan lines.

If the display driver does not draw its own wide borders, GDI simulates wide borders using alternate-fill polygons. Some display drivers intentionally take advantage of this by supporting alternate-fill polygons, but not wide borders.

If the display hardware provides support for styled borders, the driver should use the hardware to draw the borders. Although GDI can simulate styled borders, such simulations are always slower than using hardware.

2.1.8.4 Hardware Capabilities

The hardware for some displays supports many of the line, curve, and polygon capabilities that Windows itself supports or expects the display driver to support. A display driver should take advantage of display hardware whenever possible because it often dramatically improves performance.

The only disadvantage to using display hardware is if the hardware cannot produce its output both on the screen and in memory bitmaps. Windows requires that for any figure drawn on the screen, the display driver must also be able to draw it in a memory bitmap. If the hardware cannot access memory bitmaps, the display driver must include code that emulates the algorithms used by the display hardware. Depending on the complexity of the hardware code, this may be a difficult and costly task.

One alternative to emulating the display hardware is to use video memory as a temporary bitmap. The driver copies the memory bitmap from system memory to unused video memory, uses the display hardware to carry out the requested graphic operations, and then copies the results back to system memory. The efficiency of this method depends on the relative speed of the graphics hardware, and the size of the bitmap. This method is not appropriate under 386 enhanced-mode Windows if the display driver does not have full access to the hardware capabilities or to offscreen video memory while running in the background, that is, while the user is running a non-Windows application in text mode. This method is not appropriate if the display driver cannot support monochrome bitmaps. (All display drivers must support output to monochrome bitmaps regardless of whether the display adapter is color or monochrome.) Finally, this method is not appropriate for device-independent bitmaps (DIBs).

2.1.8.5 Partial Support for Capabilities

A display driver does not have to provide complete support for a given capability. Instead, a driver's **Ouput** function can provide support for a few specific cases and return all others to GDI for simulation. In such cases, the display driver sets bits in the **GDIINFO** structure as if it provided complete support, but the driver's **Output** function returns –1 to GDI for all cases that need simulation.

For example, if display hardware can draw polygons with 256 vertices but not with 257, the **Output** function can use the display hardware to draw the smaller polygons and return –1 to let GDI simulate the rest.

GDI does not simulate styled lines. If the display driver specifies support for styled lines, it must provide complete support.

2.1.9 Text

GDI requires all display drivers to provide an **ExtTextOut** function that provides a minimum set of text-drawing capabilities. At the very least, a driver must be able to draw a string of characters at a specified location on the screen and clip any portion of a character that extends beyond the bounding box for the string.

GDI calls the **ExtTextOut** function whenever an application calls a function that draws text or computes text widths. The **ExtTextOut** function receives a string of character values, a count of characters in the string, a starting position, a physical font, and a **DRAWMODE** structure. The function uses these values to create the individual glyph images on the screen.

GDI checks the RC_GDI20_OUTPUT value in the **dpRaster** member of the **GDIINFO** structure to determine whether the driver supports this function. In earlier versions of Windows, the **StrBlt** function supported text drawing, but **StrBlt** is now obsolete. For compatibility with early versions of Windows applications, however, the display driver must provide the **StrBlt** function. In most drivers, **StrBlt** does nothing more than call or fall through to **ExtTextOut**.

GDI checks the **dpText** member of the **GDIINFO** structure to determine which text capabilities the display driver supports. Although Windows requires few text capabilities for a display driver, all display drivers should support as many capabilities as possible so that Windows applications have the greatest flexibility when drawing text. The following are the text capabilities.

Capability	Description
Clipping	Specifies whether the display driver can clip whole or partial characters.
Cosmetics	Specifies whether the display driver can generate bold, italic, underlined, or strikethrough characters from existing characters.

Capability	Description
Fonts	Specifies whether the display driver supports raster and vector fonts.
Output precision	Specifies which font attributes the display driver uses when drawing text.
Rotation	Specifies whether the display driver can rotate characters.
Scaling	Specifies whether the display driver can generate new sizes by scaling an existing size.
Scaling freedom	Specifies whether the display driver can scale independently along the *x*- and *y*-axes.

2.1.9.1 Output Precision

GDI checks the TC_OP_CHARACTER and TC_OP_STROKE values in the **dpText** member to determine the output precision of the driver. Output precision specifies which font attributes the **ExtTextOut** function must use when it draws text. Font attributes include width, height, intercharacter spacing, interword spacing, escapement, orientation, and other attributes specified in the **TEXTXFORM** and **DRAWMODE** structures. These structures are passed to the **ExtTextOut** function.

A display driver sets the TC_OP_STROKE value if it can draw characters as a set of line segments. When drawing with stroke precision, the driver must adhere to all font attributes, and use the current transformations to compute the starting point of the string.

A display driver sets the TC_OP_CHARACTER value if it can draw characters at any given escapement. Character precision ensures the placement of individual characters without guaranteeing the exact size requested. With character precision, the driver must:

- Use the width and height to determine a "best-fit" character size. If no font matches exactly, the driver should use either the largest font that does not exceed the requested size or the smallest available font.
- Use the current transformations to compute the starting point of the string.
- Use the current intercharacter and interword spacing to position the individual characters in the string.
- Use the current escapement.

The driver can ignore all other attributes. If possible, the display driver should use the display's character generation hardware to draw individual characters.

If a display driver does not set the TC_OP_STROKE and TC_OP_CHARACTER values, GDI requires the driver to support string precision. String precision is identical to character precision except that the current escapement can be ignored.

Output precision does not affect the bold, italic, underline, or strikethrough capabilities. If a driver registers these abilities, it must perform them when requested.

2.1.9.2 Clipping

GDI checks the TC_CP_STROKE value in the **dpText** member to determine the clip precision of the driver. A display driver sets the TC_CP_STROKE value if it can clip any portion of a character that is outside the clip region and draw the rest of the character. If the driver does not set this value, GDI requires that the driver clips the entire character if any portion of the character is outside the clip region.

A display driver must support stroke clip precision.

2.1.9.3 Rotation

GDI checks the TC_CR_90 and TC_CR_ANY values in the **dpText** member to determine whether the display driver can rotate characters. A display driver sets the TC_CR_ANY value if it can rotate characters to any angle. The driver sets the TC_CR_90 value it can rotate characters at 90 degree increments only. If a display driver does not set either bit, GDI assumes that the driver cannot rotate characters.

2.1.9.4 Scaling and Scaling Freedom

GDI checks the TC_SA_DOUBLE, TC_SA_INTEGER, and TC_SA_CONTIN values to determine whether the display driver can scale characters. GDI also checks the TC_SF_X_YINDEP value to determine whether the driver can scale characters independently on the *x*- and *y*-axes.

A display driver sets the TC_SA_CONTIN value if it can scale existing characters to any size, sets the TC_SA_INTEGER value if it can scale characters by any integer multiple, or sets the TC_SA_DOUBLE value if it can double the size of existing characters. If a display driver sets none of these values, GDI assumes that the driver can not scale characters.

A display driver sets the TC_SF_X_YINDEP value if the driver can scale characters independently in each direction. If this bit is not set but the driver specifies that it can scale characters, GDI assumes that the driver always scales characters the same amount in the each direction.

Whenever a display driver cannot match a requested size exactly, GDI requires the driver to use the largest size available that will not exceed the requested size in either direction.

2.1.9.5 Cosmetics

GDI checks the TC_EA_DOUBLE, TC_IA_ABLE, TC_UA_ABLE, and TC_SO_ABLE values to determine whether the display driver can generate bold, italic, underlined, or strikethrough characters from existing characters.

A display driver sets TC_EA_DOUBLE if it can generate a bold character by doubling the existing character's weight. A typical method is to overstrike the existing character after moving one device unit to the right.

A display driver sets the TC_IA_ABLE value if it can generate italic characters. A typical method is to skew the gylph information, drawing the the character as if it were contained by a parallelogram rather than a rectangle.

A display driver sets the TC_UA_ABLE value if it can generate underlined characters, and sets the TC_SO_ABLE value if it can generate strikethrough characters.

2.1.9.6 Raster and Vector Fonts

GDI checks the TC_RA_ABLE and TC_VA_ABLE values in the **dpText** member to determine whether the display driver supports raster and vector fonts, respectively. A display driver must set at least one of these bits.

2.1.9.7 Orientation and Escapement

Whenever either the character orientation or the difference between the character orientation and the escapement angle is a multiple of 90 degrees, the intercharacter and interword spacing will be the standard intercharacter spacing used for bounding boxes plus the spacing specified by the **CharacterExtra** and **BreakExtra** members in the **DRAWMODE** structure.

The standard intercharacter spacing at a given escapement angle and character orientation is defined as the minimum spacing along the escapement vector, such that the character origins are on the escapement vector, and the character bounding boxes touch. Variable pitch fonts are achieved by using variable width bounding boxes. This model applies to all attribute values. When the sides of the bounding boxes touch, extra space is added in x and, when the tops touch, it is added in y.

In all other escapement and orientation cases, the standard intercharacter spacing is device dependent. The preferred implementation is for the 90-degree cases. In all cases, it is required that all character origins lie on the escapement vector.

It is assumed that arbitrary escapement angles can be achieved, if by no other means than, by placing each character as a separate entity. Many devices are able to do arbitrary character rotation only if the character orientation matches the

escapement angle. For such devices, it is assumed that the driver will place each character individually at the proper orientation and escapement, when escapement and character orientation do not match.

2.1.10 Fonts

A font is a collection of glyphs that define the size and appearance of individual characters in a character set. A display driver uses physical fonts when it draws text. A physical font is a structure that contains all the information the driver needs to draw the glyphs on the screen. GDI supplies these physical fonts whenever it calls the **ExtTextOut** function.

GDI supports a variety of font types but lets each display driver determine which fonts it supports. A display driver can choose to support any combination of the fonts.

Type	Description
Device	A font supplied by the display device. The display driver must provide complete support for device fonts, including realizing the fonts and using display hardware to draw the fonts.
Raster	A font containing glyph bitmaps that is intended to be used by raster device. GDI supplies a variety of raster fonts, but Windows applications and even display drivers can supply additional fonts. GDI realizes raster fonts as needed, but display drivers that support the fonts must be able to draw the fonts using the data in the physical font format.
TrueType	A font containing sophisticated glyph definitions that is intended to be used in conjunction with a rasterizer to produce a corresponding raster font. GDI supplies a variety of TrueType fonts as well as the TrueType rasterizer.
Vector	A font containing glyph definitions that is intended to be used by a vector device. GDI supplies a variety of vector fonts, but Windows applications and even display drivers can supply additional fonts. GDI realizes vector fonts as needed, but display drivers that support the fonts must be able to draw the fonts using the data in the physical font.

Note Display drivers that support raster fonts can also use TrueType fonts.

Although display drivers generally do not use device fonts, a display driver can realize device fonts if the display hardware support them. Most display drivers return zero when GDI calls the **RealizeObject** function requesting a realized font. Returning zero directs GDI to realize the font using existing raster or vector fonts.

Fonts, like other graphics objects, must be realized before the display driver can use them. The format of the realized font depends on the font type. For more information about raster- and vector-font format, see Chapter 13, "Font Files."

For more information about characters within a specific character set, see Appendix B, "Character Tables."

2.1.10.1 Raster Fonts

GDI checks the TC_RA_ABLE value in the **dpText** member to determine whether a display driver supports raster fonts. A raster font is a set of glyph bitmaps, each defining the size and appearance of a character in the font. A display driver that supports raster fonts uses the glyph bitmap to generate the character image on the screen.

Windows provides the following raster fonts.

Font	Description
Courier®	A fixed-width font with serifs in the ANSI character set.
Fixedsys	The Windows 2.*x* fixed-width system font in the ANSI character set.
MS Sans Serif	A proportional-width font without (sans) serifs in the ANSI character set.
MS Serif	A proportional-width font with serifs in the ANSI character set.
Symbol	A representation of math symbols in the Symbol character set.
System	A proportional-width font without serifs in the ANSI character set.
Small	A set of raster fonts used for displaying characters of 8 points or below with greater readability.
Terminal	A fixed-width font with serifs in the OEM character set.

2.1.10.2 Vector Fonts

GDI checks the TC_VA_ABLE value in the **dpText** member to determine whether a display driver supports vector fonts. A vector font is a set of glyph definitions, each containing a sequence of points respresenting the start and endpoints of the line segments that define the appearance of a character in the font. A display driver that supports vector fonts uses the glyph definitions to generate the character image on the screen.

2.1.10.3 Big Fonts

GDI checks the RC_BIGFONT value in the **dpRaster** member to determine whether the display driver supports big fonts (also called Windows 3.*x* fonts). A big font is any font in which the font and glyph information can exceed 64K bytes. Big fonts are primarily designed for use on systems with more than average memory and a microprocessor (such as an 80386) that has instructions that use 32-bit address offsets.

When GDI realizes a font for a driver supporting big fonts, GDI includes additional members (**dfFlags** through **dfReserved1**) in the the physical font's **FONTINFO** structure. Furthermore, since font and glyph information may exceed 64K, the display driver must use 32-bit offsets to access this information. This means the driver should make use of the extended registers of the 80386, such as ESI and EDI.

All display drivers must support standard fonts whether or not they support big fonts.

For big fonts, GDI sets the **dfVersion** member in the font structure to 0x0300. Standard fonts are set to 0x0200. Also, GDI never mixes fonts for a device. If the display driver registers RC_BIGFONT capability, GDI will always give the driver big fonts—not a mixture of standard and big fonts.

2.1.10.4 TrueType Fonts

Display drivers that handle raster fonts can also handle TrueType fonts without modification. GDI supports TrueType in display drivers by building **FONTINFO** structures that contain rasterized glyph bitmaps. GDI passes a pointer to this structure to the driver's **ExtTextOut** or **StrBlt** function along with the string to be displayed.

An important difference between TrueType and standard fonts is that TrueType fonts are "sparse" in their glyph definitions, that is, the **FONTINFO** structure may only contain those characters in the string to be displayed. For this reason, display drivers cannot cache the font in private memory and later reference glyphs not previously displayed. The display driver, however, can build up a cache of glyphs as they are displayed. Such a "glyph cache" can boost performance on some display adapters that have hardware text and blit support.

2.1.10.5 Overlapping Glyphs

Display drivers can handle text in TrueType fonts almost identically to text in regular raster fonts. One important difference, however, is that TrueType glyphs are designed to be more readable by minimizing the gaps between glyphs—in fact, some glyphs overlap. To take full advantage of the design and maintain performance, display drivers may need some modification to allow for overlapping glyphs.

TrueType fonts may affect the performance of display drivers since many TrueType glyphs are designed to overlap neighboring glyphs. Overlapping makes text more readable, but complicates the process used by a display driver to draw the text. A display driver can improve its performance by handling overlapping glyphs using the technique described in this section.

The most important performance improvement is to revise a driver's code to handle background opaquing (in the case of opaque text) and overlapping glyphs in a single pass. If a display driver sets the RC_OP_DX_OUTPUT value in the **dpRaster** member of the **GDIINFO** structure, GDI assumes that the **ExtTextOut** function can do background opaquing at the same time the glyphs are drawn. If this value is not set, GDI will split the text output operation into two steps, with the first step being an opaque rectangle that fills the text bounding box. The second operation will then draw the text transparently on top of the previously filled rectangle.

GDI passes the text string and a width array to either the **ExtTextOut** or **StrBlt** function in the driver. The width array contains $n-1$ entries for a string of n characters. Each entry contains a pixel offset from the origin of one character to the origin of the next character. For example, if the third width element is 20, the fourth character should be drawn 20 pixels to the right of the third character.

Handling kerning (glyph overlap) efficiently can be a problem on display adapters with relatively slow video memory access times (such as the EGA and VGA). The algorithm described below is appropriate for such displays. A different approach may be necessary for other display architectures.

The idea behind this algorithm is to never access a video memory location more than once, and, where possible, do word-aligned, 16-bit memory accesses. Ideally, this algorithm should be small, simple, and efficient for text strings with or without kerning.

Note The following discussion assumes some familiarity with the implementation of the **ExtTextOut** function in the sample EGA/VGA driver provided with the DDK. For complete details, please see the EGA/VGA source code.

For each character that is partially or completely within the string's clipping rectangle, the stack builder pushes a phase, width, and glyph pointer onto the stack. If kerning occurs in the text string, the stack builder will recognize this and push a "backup" sentinel along with a word that contains the number of pixels to backup before rendering subsequent stack entries. When the entire string has been processed, the stack builder pushes a sentinel word on the stack to mark the end of the entries.

The format of a stack entry is a word containing the phase of the bit pattern (high byte) and the width of the bit pattern (low byte). Following this word, a pointer to a glyph pattern is stored on the stack. For fonts that are less than 64K bytes in size, this is a word value representing an offset from the base of the font segment to the bit patterns for a given glyph. For fonts greater than 64K bytes, this glyph pointer is 2 words which forms a 32-bit offset from the current font segment base.

As an example:

```
Stack Entries      Meaning
--------------     ------
0308, Glyph_ptr    8-bit wide column, starting 3 bits into a byte.
0308, Glyph_ptr    Next column, 8 bits wide, phase is again 3.
8000, FFFE         Backup by 2 pixels.
0108, Glyph_ptr    Next column (overlaps previous column), phase is 1.
FFFF               End of stack entries.
```

After the stack has been built, control is passed to a routine which unstacks each entry and composes an image of the string into a memory buffer. This string image is called a "SuperGlyph" because the format of the memory buffer is identical to the glyph format in the raster font (that is, column major ordering). Each stack entry is processed by fetching up to 4 bytes worth of glyph bit pattern from the font structure, phase aligning with respect to the final destination, and ORing into the compose buffer. This process continues until all stack entries have been processed.

Once complete, the SuperGlyph is then passed to a destination specific output routine which copies the SuperGlyph to the final destination (either video memory, a color bitmap or a monochrome bitmap). Each output routine is optimized for word-aligned, 16-bit accesses to the final destination.

In some cases, the size of the compose buffer may not be big enough to hold a complete image of the string. The VGA/EGA code handles this case with restart logic that composes as much of the SuperGlyph as possible, and then outputs it to the destination. The compose buffer is then cleared and the process repeats until the entire string has been rendered.

2.1.10.6 Font Caching for TrueType Fonts

Display drivers that do their own font caching may encounter conflicts with the new TrueType font technology provided with Windows 3.1. Display drivers that use glyph caching rather than caching the entire character set should work without problems.

2.1.11 Clipping

A display driver or display hardware can clip if it can discard output that would be outside of a specified rectangle or region if drawn on the screen. GDI checks the **dpClip** member in the **GDIINFO** structure to determine whether the display driver can clip output generated by the **Output** function. If a display driver can clip, it should set the bits in this member to specify clipping for rectangles or regions. If the display driver does not support clipping, GDI will simulate clipping.

Although **dpClip** only applies to output generated by the driver's **Output** function, other members in the **GDIINFO** structure specify clipping for other graphics output. For example, the **dpText** member specifies whether the display driver clips text.

2.1.12 Bitmaps

GDI requires a display driver to support bit-block transfers by providing the **BitBlt** and related bitmap functions. At the very least, a display driver should provide a **BitBlt** function that can carry out bitmap operations on the screen and in monochrome bitmaps. The brute functions provided by GDI and used by many printer drivers require monochrome bitmap support.

GDI checks the **dpRaster** member in the **GDIINFO** structure to determine whether the display driver supports bitmaps. If the driver does, GDI checks the **dpBitsPixel** and **dpPlanes** members to determine the format of the screen bitmap.

2.1.12.1 Bitmap Format

Most display devices are either planar or packed-pixel devices. A planar device stores screen bits in separate bitmap planes, each representing a distinct color. A packed-pixel device stores screen bits in a single plane, but each pixel on the screen has a set of two or more corresponding bits that define the pixel color.

A display driver specifies the type of display device by setting the **dpBitsPixel** and **dpPlanes** members to appropriate values. For a planar device, **dpBitsPixel** is 1 and **dpPlanes** specifies the number of planes. For a packed-pixel device, **dpPlanes** is 1 and **dpBitsPixel** specifies the number of bits per pixel.

These values also help determine the number of nondithered colors available for the screen. A device with 4 planes is capable of 16 colors. A device with 8 bits per pixel is capable of 256 colors. However, the exact color specified by a combination of bits depends on whether the device supports a color palette.

2.1.12.2 Pixel Output

A display driver must support setting and getting individual pixels by providing the **Pixel** function. GDI uses **Pixel** to support a variety of simulations. It also calls the function whenever an application calls **SetPixel** (GDI.31) and **GetPixel** (GDI.83).

2.1.12.3 Bit-Block Transfers

A display driver can support bit-block transfers by providing a **BitBlt** function. GDI uses **BitBlt** to copy a rectangular block of bits from bitmap to bitmap, possibly applying a raster operation to the source and destination bits as it copies. GDI checks the RC_BITBLT value in the **dpRaster** member to determine whether the display driver supports the function. If it does, GDI calls **BitBlt** whenever an application calls GDI functions such as **BitBlt** (GDI.34) and **PatBlt** (GDI.29).

A display driver can provide additional bit-block transfers by providing a **StretchBlt** function. GDI uses **StretchBlt** to stretch or compress a block of bits to fit a given rectangle in a bitmap. Stretching and compressing requires either adding or removing bits as defined by a stretching mode. GDI checks the RC_STRETCHBLT value in **dpRaster** to determine whether the driver provides the **StretchBlt** function. GDI calls the function whenever an application calls **StretchBlt** (GDI.35).

If a display driver can carry out bit-block transfers on bitmaps that are larger than 64K, it should set the RC_BITMAP64 value in the **dpRaster** member. GDI checks this bit to determine whether large bitmaps are permitted.

2.1.12.4 Transparent-Block Transfers

In Windows 3.1, display drivers can indicate that they support transparent-block transfers by setting the C1_TRANSPARENT value in the **dpCaps1** member of the **GDIINFO** structure. In a transparent-block transfer, a driver excludes source and brush pixels from a **BitBlt** or **StretchBlt** operation if those pixels have the same color as the current background color for the destination device.

If a display driver supports transparent block transfers, the **BitBlt** function must check the **bkMode** member of the **DRAWMODE** structure as well as the *Rop3* parameter of the **BitBlt** function to determine how to carry out the transfer. If the **bkMode** member specifies the background mode TRANSPARENT1, **BitBlt** must not transfer source and brush bits that have the same color as the destination's background color (as specified by the **bkColor** member of the **DRAWMODE** structure pointed to by the *lpDrawMode* parameter). In other words, the corresponding destination bits must be left unchanged. Other background modes do not affect the transfer.

Although a display driver may support transparent-block transfers, GDI does not currently provide access to this support for Windows applications.

2.1.12.5 Fast Borders

A display driver can support fast-border drawing by providing a **FastBorder** function. Windows uses the function to quickly draw borders for windows and dialog boxes. GDI checks the RC_GDI20_OUTPUT value in the **dpRaster** member to determine whether a display driver provides the **FastBorder** function. If it does, GDI calls the function to draw the borders. Otherwise, it returns an error value to direct Windows to use some other means to draw the borders.

A display driver also uses the RC_GDI20_OUTPUT value to specify whether it supports the **ExtTextOut** function. In some cases, a display driver may support **ExtTextOut** but not **FastBorder**. To account for this, the display driver must provide a **FastBorder** function, but the function can immediately return an error value to direct Windows to use some other means to draw borders.

2.1.12.6 Saved Bitmaps

A display driver can permit Windows to temporarily save bitmaps in off-screen video memory by providing the **SaveScreenBitmap** function. This function allows the driver to take advantage of unused video memory and to speed up drawing operations that require restoring a portion of the screen that was previously overwritten.

GDI checks the RC_SAVEBITMAP value in the **dpRaster** member to determine whether the driver supports **SaveScreenBitmap**.

2.1.12.7 Flood Fill

A display driver can support flood-fill operations by providing a **FloodFill** function. GDI uses the function to quickly fill a region on the screen or in a bitmap with a specified color. GDI calls the function whenever an application calls **FloodFill** (GDI.25).

GDI checks the RC_FLOODFILL value in **dpRaster** to determine whether the driver provides the function.

2.1.13 Device-Independent Bitmaps

A device-independent bitmap (DIB) is a color bitmap in a format that eliminates the problems that occur when transferring device-dependent bitmaps to devices having difference bitmap formats. DIBs provide color and bitmap information that any display or printer driver can translate into the proper format for its corresponding device.

A display driver can support DIBs by providing the following device-independent bitmap functions.

Function	Description
DeviceBitmapBits	Copies a DIB to a device-dependent bitmap or a device-dependent bitmap to a DIB.
SetDIBitsToDevice	Copies any portion of a DIB to the screen. This function copies the bits directly without applying a raster operation.
StretchDIBits	Moves a source rectangle into a destination rectangle, stretching or compressing the bitmap if necessary to fit the dimensions of the destination rectangle.

These functions receive and operate on **BITMAPINFO**, **BITMAPINFOHEADER**, and **RGBQUAD** structures.

A display driver that supports DIBs must provide the **CreateDIBitmap** function as well. The function should do nothing more than return zero indicating that the creation of a DIB is not supported at the driver level.

If a display driver supports DIBs, it must set one or more of the RC_DI_BITMAP, RC_DIBTODEV, and RC_STRETCHDIB values in the **dpRaster** member of the **GDIINFO** structure. If a driver does not set the RC_DI_BITMAP value, GDI simulates DIB conversions using monochrome bitmaps.

2.1.13.1 Logical-Color Tables

A display driver creates a logical-color table when translating a device-specific bitmap into a DIB. The table resides in the DIB header block. The device driver, if it is not a palette device, can fill up the table with whatever color it supports and then, use the corresponding indexes in the bitmap. The driver must also set the number of colors it is using in the **biClrUsed** member of the header block.

Consider an example in which the display device is a 4-plane EGA device, and in which the DIB has 8 bits-per-pixel. The logical color table for the DIB has provisions for 256 colors, but the 4-plane driver can deal with only 16 colors. The driver would prepare a color table for the DIB that looked like the following:

```
RGBQUAD ColorTable[] = {
    {0,     0,   0, 0},
    {128,   0,   0, 0},
    {0,   128,   0, 0},
    {128, 128,   0, 0},
    {0,     0, 128, 0},
    {128,   0, 128, 0},
    {0,   128, 128, 0},
    {128, 128, 128, 0},
```

```
#ifdef VGA
    {196, 196, 196, 0},
#else /* EGA */
    {64,   64,  64, 0},
#endif
    {255,   0,   0, 0},
    {0,   255,   0, 0},
    {255, 255,   0, 0},
    {0,     0, 255, 0},
    {255,   0, 255, 0},
    {0,   255, 255, 0},
    {255, 255, 255, 0}};
```

The device driver may fill in just 16 colors and set **biClrUsed** to 16, or it may fill up entries 16 through 255 with zeros and set **biClrUsed** to 0.

The color-mapping tables for each of the DIB formats can be predefined for a particular driver and should be copied into the DIB header during the format-specific initialization.

2.1.13.2 DIB to Device

A display driver can copy a DIB directly to the screen using the **SetDIBitsToDevice** function. This function saves the trouble of first converting the DIB into the device-dependent format and then, transferring it onto the screen. However, only a direct copy of the DIB is provided. For other raster operations that **BitBlt** supports, the driver must first convert the DIB into the internal format. Moreover, only one direction of copy (DIB to screen) is provided.

The process of copying the bits is similar to the one adopted in the **SetDeviceBitmapBits** function except that for some devices, such as the EGA/VGA, the nature of the hardware might make it advantageous to copy one pixel at a time. The concept of having a color-translation table and format-specific initialization remains the same.

2.1.14 Device Bitmaps

A device bitmap is any bitmap whose bitmap bits are stored in device memory (such as RAM on a display adapter) instead of main memory. Device bitmaps can significantly increase the performance of a graphics driver as well as free main memory for other uses. To realize these benefits, the corresponding graphics device must have ample video memory in addition to the video memory used to generate the current display. The device should also have efficient routines for copying bits to video memory.

Graphics drivers that set the RC_DEVBITS value in the **dpRaster** member of the **GDIINFO** structure support device bitmaps. GDI checks this bit to determine how to carry out requests to create and select bitmaps. If a driver sets the RC_DEVBITS value, it must export the following functions:

- **BitmapBits**
- **RealizeObject**
- **SelectBitmap**

BitmapBits copies bitmap data to and from device bitmaps. GDI calls this function when initializing the bitmap bits after creating the bitmap. It also calls the function when an application calls the functions such as **GetBitmapBits** (GDI.74) and **SetBitmapBits** (GDI.106).

RealizeObject creates or deletes a device bitmap. GDI calls this function when creating the bitmap, specifying the OBJ_BITMAP style. **RealizeObject** is responsible for allocating memory to the device as well as filling a physical **PBITMAP** structure that GDI will use to identify the device bitmap. If the bitmap is to be deleted, **RealizeObject** must free the device memory.

SelectBitmap associates a device bitmap with the given **PDEVICE** structure. GDI passes the physical **PBITMAP** structures of both the currently selected bitmap and the new bitmap so that **SelectBitmap** can carry out any special processing to enable or disable access to the device bitmaps.

Device bitmaps cannot be monochrome bitmaps. GDI intercepts all requests to create monochrome bitmaps and creates main memory bitmaps instead. This means a graphics driver that supports device bitmaps must also be able to support main memory bitmaps.

Note If the display driver supports device bitmaps, it must not fail any calls to the **SelectBitmap**, **BitmapBits**, or **RealizeObject** functions. If the driver cannot allocate more video memory for a requested bitmap, it must use **GlobalAlloc** to allocate system memory instead.

2.1.15 Color Palettes

Display devices that are capable of displaying 256 or more simultaneous colors using a palette need to provide support for color palettes. A display driver specifies that it has palette support by setting the RC_PALETTE value in the **dpRaster** member in the **GDIINFO** structure. The display driver also must set the **dpPalColors**, **dpPalReserved**, and **dpPalResolution** members.

The number of reserved colors on the palette is always 20, with 16 corresponding to the VGA colors and 4 special colors. Half of the reserved palette colors are placed at the beginning and half at the end of the palette.

If a display driver supports color palettes, it must export the **SetPalette**, **GetPalette**, **SetPalTrans**, and **GetPalTrans** functions.

2.1.15.1 Hardware-Palette Initialization

A display driver should initialize the hardware palette when the driver initializes the rest of the display hardware. The driver initializes the palette for static and nonstatic colors.

Static colors are system colors that remain in the palette at all times and are used for old-style RGB support. The number of static colors to be used is specified in the **dpPalReserved** member in the **GDIINFO** structure. This number must always be even. The colors are placed in the hardware palette so that the first half is in the lowest entries, and the second half is in the highest entries.

For Windows 3.*x*, 20 static colors should be set as follows:

```
; Lowest ten palette entries

db      0,   0,   0,   0        ; 0
db      080h,0,   0,   0        ; 1
db      0,   080h,0,   0        ; 2
db      080h,080h,0,   0        ; 3
db      0,   0,   080h,0        ; 4
db      080h,0,   080h,0        ; 5
db      0,   080h,080h,0        ; 6
db      0C0h,0C0h,0C0h,0        ; 7
db      192, 220, 192, 0        ; 8
db      166, 202, 240, 0        ; 9

; Highest ten palette entries

db      255, 251, 240, 0        ; 10
db      160, 160, 164, 0        ; 11
db      080h,080h,080h,0        ; 12
db      0FFh,0,   0,   0        ; 13
db      0,   0FFh,0,   0        ; 14
db      0FFh,0FFh,0,   0        ; 15
db      0,   0,   0FFh,0        ; 16
db      0FFh,0,   0FFh,0        ; 17
db      0,   0FFh,0FFh,0        ; 18
db      0FFh,0FFh,0FFh,0        ; 19
```

These colors consist of the 16 standard VGA colors and four other special colors used by Windows.

The nonstatic colors included in the hardware palette need not be initialized. GDI manages these and initializes them when needed.

The RGBs for an 8bpp system palette should exactly match those reported to GDI by the 8514/a driver. Using these RGBs ensures that colors will be preserved when an image created with solid colors is viewed on a 4bpp display.

An 8bpp display driver should not necessarily program the DAC with the exact RGBs as in the GDI system palette. Instead, DAC values should be used that result in on-screen system colors reasonably close to those shown by the Windows 8514/a display driver. This accounts for differences in color representation by DACs on different display adapters.

2.1.15.2 Palette-Translation Table

The driver has to maintain a palette-translation table to translate the logical color indexes, passed to it by GDI, into the actual physical color indexes. The translation has to occur before any raster operation (ROP) is performed. ROPs are always applied to physical colors.

Whenever a display driver function receives a **DRAWMODE**, **LPEN**, **PBRUSH**, or **PBITMAP** structure, the driver may need to translate the logical colors in these structures to physical colors before using the colors.

An application has to perform color translation only when the physical device is involved. In other words, if a line is drawn into a memory bitmap or a bitmap is block transferred into another memory device, no color translation is required. On the other hand, if a bitmap is transfered to or from the screen into a memory bitmap or a line is drawn directly onto the screen, color translation is required. In the case of a block transfer from the screen to the screen (where the physical device is both the source and destination of the block transfer), color translation is not needed since all the color indexes are already translated into physical indexes.

Color specifications are passed to display drivers as either color indexes or RGB values. A color index is a 32-bit value in which the high 16-bits is set to 0xFF00 and the low 16-bits is the actual index. An RGB value is a 32-bit value as specified by the **RBGQUAD** structure. When an RGB value is specified, it should match this color as closely as possible among the 20 reserved colors. In the case of a brush, the color may be dithered with the 16 VGA colors.

A palette-translation table is an array of 16-bit indexes, each mapping a logical color index to a physical color index. A display driver uses the translation table to translate color indexes in physical pens and brushes and in the **DRAWMODE** structure to the actual color indexes used by the hardware palette.

GDI calls the **GetPalTrans** and **SetPalTrans** functions to get and set the translation table. The translation table has the number of elements specified by the **dpPalColors** member. In constructing the inverse table, the driver may come across ambiguities because different logical colors can map to the same physical color. It is up to the driver to decide how to resolve these cases since the result will look

the same no matter how such ambiguities have been resolved. The driver can also set accelerator bits to bypass the various translations. For **BitBlt**, bypassing color translation results in substantial performance improvements.

2.1.15.3 UpdateColors Function

If a display driver supports color palettes, it must include the **UpdateColors** function. GDI calls this function to direct the driver to redraw a region on screen using the translation table passed to the function. For each pixel in the region, the function retrieves the pixel's color index, translates the index, and writes the translated index back to the given pixel.

2.1.15.4 Black-and-White Palette Entries

Display drivers that support color palettes must make sure that the index for the palette entry that corresponds to black must be the one's complement of the index for the palette entry for white. Black and white must be static palette entries, meaning the driver sets the indexes for these colors during initialization and does not change the indexes.

2.1.16 DIBs with Color-Palette Management

The color table for a device-independent bitmap (DIB) consists of 16-bit indexes used as the colors for the bitmap. For the **SetDIBitsToDevice** function, they are physical indexes; for the **DeviceBitmapBits** function, they are logical indexes.

In the **DeviceBitmapBits** and **SetDIBitsToDevice** functions, the final parameter, *lpTranslate*, provides information that is useful only for palette-capable devices. However, all devices need to include this parameter.

When **DeviceBitmapBits** is retrieving bits on a 256-color device, *lpTranslate* is a pointer to a translation table with the following results.

Bitcount	Result
1	A palette-sized array of bytes, each one either 0x00H or 0xFFH. The array is used to determine if the index in the bitmap corresponds to a zero or a 1 in the DIB.
4	A palette-sized array of bytes, each one containing a value between 0x00H and 0xFFH. Each index in the bitmap will map to the corresponding 4-bit values in the DIB.
8	*lpTranslate* will equal an identity table that can be ignored.
24	A palette-sized array of 32-bit values, each one containing the RGB value (and unused high byte) corresponding to the index in the bitmap.

On a palette-capable device when the source bitmap is a monochrome bitmap, **DeviceBitmapBits** passes in a translation table that has only two entries, instead of 256. Entry 0 has the color-table index for black and entry 1 has the color table index for white.

When **DeviceBitmapBits** sets bitmap bits, GDI passes a pointer only if the bit-count is 24. The parameter points to data maintained by GDI. For every RGB value in the DIB, the display driver calls the **DeviceColorMatch** function (GDI.449) to match an RGB value with an index. This function returns an index to represent that color in the device-dependent bitmap.

2.1.17 Stub Functions

Although the **SetAttribute** and **DeviceBitmap** functions are not currently supported in GDI, a display driver must provide these functions. In all drivers, these functions are implemented as stub functions that do nothing more than always return a –1.

2.1.18 Cursors

The USER module requires all display drivers to provide cursor support. A display driver must provide the following functions.

Function	Description
CheckCursor	Draws the cursor if drawing is not disabled.
InquireCursor	Retrieves information about the cursor.
MoveCursor	Moves the cursor to a specified position on the display.
SetCursor	Sets the cursor shape.

When Windows first starts, USER calls the **InquireCursor** function to retrieve information about the cursor. It then sets a system timer to call the **CheckCursor** function on each timer interrupt and enables the mouse driver, allowing the Windows mouse-event routine to call **MoveCursor** at each mouse interrupt occurrence. USER and Windows applications subsequently set the shape of the cursor using **SetCursor**.

Because USER calls **MoveCursor** on each mouse interrupt, **MoveCursor** should set a semaphore to prevent the function from being called before it can complete the current call. The function should set the semaphore using a noninterruptable instruction such as the **xchg** instruction. Once the semaphore is set, **MoveCursor** should retrieve the *x* and *y* coordinates of the current call, set a flag to indicate that the cursor is being redrawn, and clear the semaphore. Whenever **MoveCursor** is called, it should check this flag before continuing the request.

The **CheckCursor** function is called on each timer interrupt. The function should determine whether the cursor needs redrawing and whether drawing is enabled. If so, the function should redraw the cursor.

For displays that do not have hardware cursor, the display driver should exclude the cursor from the screen before carrying out drawing operations in functions such as **BitBlt**, **Output**, and **ExtTextOut**. The driver should check the current *x* and *y* coordinates and exclude the cursor if any part of the cursor lies within the drawing region as specified in the following.

Operation	Region
Bit-block transfer	Destination rectangle
Ellipse	Bounding rectangle
Other line drawing	Bounding rectangle
Polygon	Clip rectangle
Polyline	Clip rectangle
Scan line	Whole scan line
Text	Bounding rectangle and/or opaquing rectangle

2.1.19 Display-Driver Escapes

A display driver provides the **Control** function to give applications direct access the display driver. Most display drivers only provide support for the following escapes:

- GETCOLORTABLE
- MOUSETRAILS
- QUERYESCSUPPORT

Applications call the QUERYESCSUPPORT escape to determine whether the display driver supports a given escape.

2.1.20 Mouse Trails

In Windows 3.1, display drivers can improve mouse cursor visibility (especially on liquid crystal displays (LCD)) by supporting mouse trails. A mouse trail is a sequence of two or more cursor images that mark current and previous mouse cursor positions. A display driver creates a mouse trail by drawing a cursor at each new mouse position and leaving additional cursors at previous positions. The driver delays erasing the previous cursors until a specified number of cursors are visible.

A display driver provides support for mouse trails by processing the MOUSETRAILS escape in its **Control** function. The MOUSETRAILS escape enables or disables mouse trails. It also sets the maximum number of cursors to be displayed in the trail. The display driver must also process the QUERYESCSUPPORT escape, returning the status of the mouse trails if the requested escape is MOUSETRAILS.

The display driver draws the mouse trail when it processes the **MoveCursor** function. Any trailing cursors must have the same shape as the current cursor. The display driver should provide mouse trails for system cursors, such as the pointer and hourglass, and for application-specific cursors as well. Currently, support is not provided for mouse trails for XOR-only cursors, such as the I-beam and the crosshair cursor used by PaintBrush.

The display driver is responsible for recording the mouse trail status by maintaining the **MouseTrails** setting in the [Windows] section of the WIN.INI file. The driver switches this setting to a positive value to indicate that mouse trails capability is enabled. The value also specifies the number of cursors to display. The driver switches this setting to a zero or a negative value when mouse trails capability is disabled. The absolute value specifies the number of cursors to display prior to being disabled.

Users can turn mouse trails on or off by using Windows 3.1 Control Panel. The mouse settings dialog box displays a Mouse Trails option.

Applications should not enable or disable mouse trails. However, third-party mouse configuration programs should.

2.1.21 Multiple-Resolution Drivers

A multiple-resolution display driver provides the code and resources needed to support all resolutions of a given display device. In previous versions of Windows, separate display drivers were required for each resolution. In Windows 3.1, a single display driver can handle all resolutions.

A multiple-resolution driver must provide the following:

- **GetDriverResourceID** function
- Icons and cursors for each supported resolution
- Installation information

2.1.21.1 Resources and Resource Mapping

In many cases, the only difference for the driver between the device resolutions is the size of the resources and the horizontal and vertical dimensions. A driver can support multiple resolutions as long as it can determine which resolution Windows expects it to use.

To support multiple resolutions in a single display driver, Windows checks for the **GetDriverResourceID** function in the driver and calls it before loading the driver's resources and dimensions. This gives the driver a chance to check the SYSTEM.INI file using the **GetPrivateProfileInt** (KERNEL.127) or **GetPrivate-ProfileString** (KERNEL.128) functions for information specifying the desired resolution. The **GetDriverResourceID** function can then map the request resource identifier to the identifier of the corresponding resource for the desired resolution. In this way, the driver makes sure Windows loads the appropriate resources for the resolution to be used.

When Windows calls the **GetDriverResourceID** function, the driver should at least check for the **aspect** entry in the [Boot.description] section of the SYSTEM.INI file. This entry specifies the resolution (in dots-per-inch) of the display. Drivers use this value to determine which fonts to use as well as to determine other resource that depend on the number of dots-per-inch the display generates.

Display drivers that support multiple resolutions must export the **GetDriver-ResourceID** function and provide a set of appropriate resources for each resolution.

2.1.21.2 Installation Information

Users select the desired resolution for the display using the Setup program. To display each screen resolution, Setup checks the [Display] section in the SETUP.INF or OEMSETUP.INF file. There must be one line for each resolution. For example, if a video adapter named "ZGA" has three resolutions (640x480, 800x600, and 1024x768), the [Display] section should look something like this:

```
[display]
zga1=1:zga.drv,"ZGA(640x480)","100,96,96",3:zgacolor.2gr,... ,zgalo
zga2=1:zga.drv,"ZGA(800x600)","100,96,96",3:zgacolor.2gr,... ,zgamed
zga3=1:zga.drv,"ZGA(1024x768)","100,120,120",3:zgacolor.2gr,... ,zgahi
```

Setup installs the same driver (ZGA.DRV), grabbers, virtual-display device, and other related components. However, the last field in the line specifies an additional section in the SETUP.INF or OEMSETUP.INF file that contains resolution-specific information.

The resolution-specific information allows Windows Setup to copy files and write profile data for later use by the display driver. Setup can write the information to the SYSTEM.INI file in the section and format the driver understands. When a display driver is initialized, it can use the Windows functions **GetPrivateProfileInt** or **GetPrivateProfileString** to read the user-specified screen resolution from the appropriate section of SYSTEM.INI. This will let the driver decide what resource ID to return to Windows by the **GetDriverResourceID** function.

The following examples show how resolution-specific information sections might look in the SETUP.INF or OEMSETUP.INF file:

```
[zgalo]
4:zgalo.dll,0:system,,,,

[zgamed]
,,system.ini,zga.drv,,"ZGAres=800x600"

[zgahi]
4:zgahi.dll,0:system,system.ini,zga.drv,"ZGAres=800x600","ZGAMode=Hi"
```

For more information about the resolution-specific information sections, see the [Display] section.

2.1.22 Microsoft Windows for Pen-Computing Extensions

Display drivers that support Pen Extensions provide a set of functions and resources that permit the display drivers to carry out inking. Pen Windows is a version of Windows in which character-recognition software (Recognition Context (RC) Manager) allows a pen device to be used in place of a keyboard. Inking is drawing done by a display driver in response to input from the RC Manager.

2.1.22.1 Inking Functions

Display drivers that support inking must be prepared to process inking requests whenever pen input generates an interrupt. The drivers must export the **GetLPDevice** and **InkReady** functions.

If the RC Manager requires inking, it calls the **GetLPDevice** function to retrieve a pointer to the display driver's **PDEVICE** structure. The RC Manager uses this pointer in subsequent calls to the display driver's **Output** function to complete the inking. The color and width of the ink is set by the RC Manager.

Before calling **Output**, the RC Manager calls the **InkReady** function to notify the display driver that it is ready to ink. The display driver must determine whether any other drawing operation is under way. If so, the display driver must wait until

the current operation is complete before completing the inking. In any case, the display driver calls a callback function supplied with the call to **InkReady** to complete the inking.

2.1.22.2 Inking Resources

Display drivers that support inking must provide the following cursor resources.

Value	Meaning
IDC_NEPEN (32630)	Pen points to northeast.
IDC_NWPEN (32631)	Pen points to northwest.
IDC_SEPEN (32632)	Pen points to southeast.
IDC_PEN (32633)	Default pen is same as IDC_SWPEN.
IDC_SWPEN (32633)	Pen points to southwest.

The pen cursors must be added to the driver in the same way as the standard Windows cursors.

2.2 About Display-Driver Resources

Display drivers contain most of the cursors, icons, and bitmaps that use Windows. They are supplied by the display driver to account for the aspect ratio and resolution of the display adapter. Also, the definitions of certain system parameters (for example, default colors and border widths) are supplied by the display driver. All of this information is supplied as resources that are added to the driver by the resource compiler (RC.EXE).

A display driver requires the following resources.

Resource	Description
Stock fonts	The stock-font resource defines the characteristics of the stock-font objects used by Windows.
Configuration	The configuration resource contains information about default system colors, line widths, cursor sizes, icon sizes, and so on.
Color table	The color table resource contains the color table for Control Panel's Color dialog box.
Icons, cursors, bitmaps	The icon, cursor, and bitmap resources define the shape and appearance of various elements of the Windows desktop.

2.2.1 Stock-Fonts Resource

The stock-font resource (defined by the FONTS.BIN file) defines the characteristics of the stock-font objects used by Windows. Windows requires three stock fonts:

- OEM font (the Terminal font)
- ANSI fixed-pitch font
- ANSI variable-pitch font

Windows supplies a wide variety of stock-font files at various aspect ratios, so most display drivers do not have to provide the actual font files to support these stock fonts. Windows provides the following screen and system fonts.

Font name	Aspect ratio	Logical pixels per inch X	Logical pixels per inch Y
COURB	EGA, 1.33 to 1	96	72
EGASYS	EGA, 1.33 to 1	96	72
SERIFB	EGA, 1.33 to 1	96	72
SMALLB	EGA, 1.33 to 1	96	72
SSERIFB	EGA, 1.33 to 1	96	72
COURE	VGA, 1 to 1	96	96
SERIFE	VGA, 1 to 1	96	96
SMALLE	VGA, 1 to 1	96	96
SSERIFE	VGA, 1 to 1	96	96
VGAFIX	VGA, 1 to 1	96	96
VGASYS	VGA, 1 to 1	96	96
8514FIX	8514/A, 1 to 1	120	120
8514SYS	8514/A, 1 to 1	120	120
COURF	8514/A, 1 to 1	120	120
SERIFF	8514/A, 1 to 1	120	120
SMALLF	8514/A, 1 to 1	120	120
SSERIFF	8514/A, 1 to 1	120	120

All the filenames have the same .FON extension.

2.2.2 Configuration Resource

The configuration resource (defined by the CONFIG.BIN file) contains information about default system values, such as, desktop colors, horizontal and vertical line widths, scroll bar "thumb" sizes, and cursor and icon compression ratios.

2.2.3 Color-Table Resource

The color-table resource (defined by the COLORTAB.BIN file which contains the **COLORTABLE** structure) contains a list of the colors that are to appear in the Control Panel's Color dialog box. This table should contain all the solid colors that are representable as RGB values as well any patterned colors that are reasonably close to the colors of the 8514/a display under Windows 3.1. The table may contain up to 48 RGB values. It is not necessary for RGBs to be exact across all drivers, but it is important for color representations to be similar. This accounts for variations in dithering algorithms of different display drivers.

For Windows 3.1, the recommended default system color values for 16- and 256-color displays are new. In particular, several system colors that were patterned (dithered) in Windows 3.0 have been changed to solids to improve performance and visibility on interlaced displays. For color drivers intended for use on general purpose color displays, the default desktop colors should always be solid colors. For 4bpp drivers, the default colors should be the same as used by the Windows 3.1 VGA driver. For 8bpp drivers, the default colors should be the same as used by the Windows 3.1 8514/a driver. See CONFIG.BIN for a list of these default colors.

In Windows 3.1, the CONFIG.BIN resource has been expanded to include the **clrInactiveCaptionText** member. This member specifies the color of the text in the title bar of an inactive window.

For a complete description of the default system colors and the **clrInactive-CaptionText** member, see the CONFIG.BIN resource.

2.2.4 Icon, Cursor, and Bitmap Resources

The icon, cursor, and bitmap resources define the appearance of elements of the Windows desktop, such as the minimize and maximum buttons, scroll boxes, menu icons, and others. The USER module extracts these resources from the display driver when Windows starts, then uses **BitBlt** to copy the resources to the screen when drawing windows. Windows provides a wide variety of icon, cursor, and bitmap resources at various resolutions and aspect ratios.

2.2.4.1 Cursors

The following table provides a list of required cursor resources. The last four cursors in the list are no longer used by Windows, but must be provided for compatibility with existing Windows applications.

Resource	Appearance
CROSS.CUR	An upright cross used as a selection marker.*
IBEAM.CUR	An I-beam shaped cursor used in edit control windows.
ICON.CUR	An empty box formerly used when the mouse was in the icon area.*
NORMAL.CUR	An upward diagonal arrow used as the default mouse cursor.
SIZE.CUR	A box shape formerly used when sizing tiled windows.*
SIZENESW.CUR	A two-headed arrow used when sizing windows. Arrows point northeast and southwest.
SIZENS.CUR	A two-headed arrow used when sizing windows. Arrows point north and south.
SIZEWE.CUR	A two-headed arrow used when sizing windows. Arrows point west and east.
SIZENWSE.CUR	A two-headed arrow used when sizing windows. Arrows point northwest and southeast.
UP.CUR	An upward arrow.*
WAIT.CUR	An hourglass that is used while carrying out lengthy operations.

* These resources are only for compatibility for versions of Windows prior to 3.0.

2.2.4.2 Icons

The following table provides a list of required icons resources.

Resource	Appearance
BANG.ICO	An exclamation mark used to emphasize the consequences of an operation.
HAND.ICO	A stop sign used to indicate an error condition that halts operation.
NOTE.ICO	An asterisk used to indicate noncritical situations.
QUES.ICO	A question mark used when querying for a reply.
SAMPLE.ICO	The default icon used when no other icon to an operation can be found.

2.2.4.3 Bitmaps

The following table provides a list of required bitmap resources. The first seven bitmaps have two forms (up and down) used to create a 3-D effect when pushing

in a button. The last eleven bitmaps are no longer used by Windows, but must be supplied for compatibility with applications that expect them to be available.

Resource	Appearance
COMBO.BMP	An arrow used in combo boxes.
DOWN.BMP	Up down-pointing arrow for scroll bars.
DOWND.BMP	Down down-pointing arrow for scroll bars.
LEFT.BMP	Up left-pointing arrow for scroll bars.
LEFTD.BMP	Down left-pointing arrow for scroll bars.
MAX.BMP	Up maximize button on the title bar.
MAXD.BMP	Down maximize button on the title bar.
MIN.BMP	Up minimize button on the title bar.
MIND.BMP	Down minimize button on the title bar.
MNARROW.BMP	An arrow used in multilevel menus.
OBTNCORN.BMP	A circle formerly used to draw round-cornered buttons.
OBTSIZE.BMP	A size box used at the intersection of vertical and horizontal scroll bars.*
OBUTTON.BMP	A box used for check boxes in dialog boxes.
OCHECK.BMP	A check mark used to check menu items.
OCLOSE.BMP	The system-menu bitmaps used for Windows 2.*x*.*
ODOWN.BMP	A down-arrow bitmap used for Windows 2.*x*.*
OLEFT.BMP	A left-arrow bitmap used for Windows 2.*x*.*
ORED.BMP	This resource minimizes the bitmap used for Windows 2.*x*.*
OREST.BMP	Restores the bitmap used for Windows 2.*x*.*
ORIGHT.BMP	A right-arrow bitmap used for Windows 2.*x*.*
OSIZE.BMP	A size box formerly used on tiled windows.*
OUP.BMP	An up-arrow bitmap used for Windows 2.*x*.*
OZOOM.BMP	Maximizes the bitmap used for Windows 2.*x*.*
RESTORE.BMP	Up restore button on the title bar.
RESTORED.BMP	Down restore button on the title bar.
RIGHT.BMP	Up right-pointing arrow for scroll bars.
RIGHTD.BMP	Down right-pointing arrow for scroll bars.
SYSMENU.BMP	A double-wide image that contains system menu shapes for both main windows and multiple document interface (MDI) windows.
UP.BMP	Up up-pointing arrow for scroll bars.
UPD.BMP	Down up-pointing arrow for scroll bars.

* These resources are only for compatibility for versions of Windows prior to 3.0.

2.2.5 Large Icons and Cursors

In Windows 3.1, display drivers can use icons larger than 64-by-64 bits and cursors larger than 32-by-32 bits. Large icons and cursors can improve screen readability for high-resolution graphics adapters.

Display drivers specify icon and cursor size in the **CONFIG_BIN** structure in the **IconXRatio**, **IconYRatio**, **CurXRatio**, and **CurYRatio** members of the CONFIG.BIN resource (that is, the resource having identifier 1 and type OEMBIN). In Windows 3.1, these members specify either a width and height in pixels or a compression ratio. In Windows 3.0, these members only specify compression ratios. For a complete description of the members, see the CONFIG.BIN resource.

In all cases, each icon or cursor must have the same width and height in pixels.

Display drivers which specify actual widths and heights in the OEMBIN resource can not be used with Windows 3.0. Drivers which specify compression factors work with both Windows 3.0 and 3.1. However, drivers cannot simultaneously specify compression factors and icons or cursors larger than 32-by-32.

2.2.6 Optimizing Performance

Whenever possible, display drivers should use the features of the CPU to optimize their performance. This is particularly important for drivers when used in multimedia versions of Windows.

A display driver can determine what CPU is present and what mode Windows is operating in by examining the **__WinFlags** variable or by calling the **GetWinFlags** function. If the CPU is at least a 386, the display driver should take advantage of the CPU's 32-bit registers to manipulate data and to index huge arrays.

2.2.6.1 Tips for Writing Transparent Text

Drivers generating VGA text output can make two significant speedups for transparent text: don't draw empty bytes or words and use write mode 3.

While drawing the character to video memory, avoid copying the "empty" part of a glyph's bitmap. For example, there is no need to copy the space character to video memory or the ascender on letters like the lowercase 'p'. Since accessing video memory is nearly 5 times slower than accessing system memory, it is

cheaper to check for and ignore empty bytes or words in a glyph bitmap, than it is to store nothing to video memory. In the following example, the code gives a significant performance increase for many video adapters:

```
    mov  ax,glyphbits
    or   ax,ax
    jz   around_it
    xchg es:[di],ax
around_it:
```

Use write mode 3 on VGA hardware. This mode simplifies the output of transparent text by eliminating the extra step of setting the bitmask register for each store to video memory. For example, the VGA portion of the following EGA/VGA code uses write mode 3 for the VGA adapter and is considerably faster than the equivalent EGA code.

```
        mov     ax,glyphbits
if _EGA
        push    dx
        mov     dx,3cfh
        out     dx,al
        xchg    es:[di],al              ;write ax to screen.
        mov     al,ah
        out     dx,al
        xchg    es:[di+1],ah            ;write ax to screen.
        pop     dx
else
        xchg    es:[di],al              ;write ax to screen.
        xchg    es:[di+1],ah            ;write ax to screen.
endif
```

2.2.6.2 Tips for Using the Interrupt Flag

Display drivers should avoid needless clearing and setting of the interrupt flag using the **cli** and **sti** instructions. Since 386 enhanced-mode Windows traps these instructions as part of its management of the CPU's virtual mode, these instructions each take about 600 CPU clocks to execute.

2.3 Function Reference

The following is an alphabetical listing of graphics functions that are specific to display drivers. For a complete description of the graphics functions that are common to other graphics drivers, see Chapter 10, "Graphics-Driver Functions."

CheckCursor

void CheckCursor(*void*)

The **CheckCursor** function is called on every timer interrupt. It allows the cursor to be displayed if it is no longer excluded.

Parameters This function has no parameters.

Return Value This function has no return value.

Comments The export ordinal for this function is 104.

FastBorder

WORD FastBorder(*lpRect*, *wHorizBorderThick*, *wVertBorderThick*, *dwRasterOp*, *lpDestDev*, *lpPBrush*, *lpDrawMode*, *lpClipRect*)
LPRECT *lpRect*;
WORD *wHorizBorderThick*;
WORD *wVertBorderThick*;
DWORD *dwRasterOp*;
LPDEVICE *lpDestDev*;
LPBRUSH *lpPBrush*;
LPDRAWMODE *lpDrawMode*;
LPRECT *lpClipRect*;

The **FastBorder** function draws a rectangle with a border on the screen. However, the size is subject to the limits imposed by the specified clipping rectangle. The border is drawn within the boundaries of the specified rectangle.

Parameters

lpRect
Points to a **RECT** structure specifying the rectangle to be framed.

wHorizBorderThick
Specifies the width in pixels of the left and right borders.

wVertBorderThick
Specifies the width in pixels of the top and bottom borders.

dwRasterOp
Specifies the raster operation to be used.

lpDestDev
Points to a **PDEVICE** structure that identifies the device to receive the output.

lpPBrush
Points to a **PBRUSH** structure.

lpDrawMode
Points to a **DRAWMODE** structure that includes the current text color, background mode, background color, text justification, and character spacing. See the **DRAWMODE** data structure description in Chapter 12, "Graphics-Driver Types and Structures," for a description of text justification and character spacing.

lpClipRect
Points to a **RECT** structure specifying the clipping rectangle.

Return Value

The return value is AX = 1 if successful. Otherwise, AX = 0.

Comments

The export ordinal for this function is 17.

The specified rectangle should be given as (UpperLeftCorner, LowerRightCorner). If it is specified incorrectly, the sample function will draw the borders outside of the specified rectangle, instead of correctly drawing them inside.

The function is optional for display drivers. It is required at the GDI level but not at the display level.

The raster operation to be used will never have a source operand within it.

The *lpDrawMode* parameter is simply a long pointer to the **DRAWMODE** data structure. It is included only for compatibility with earlier versions and is not crucial. The only field that you may use from there is **BackgroundColor**.

Inquire

WORD Inquire(*lpCursorInfo*)
LPCURSORINFO *lpCursorInfo*;

The **Inquire** function returns the mouse's mickey-to-pixel ratio for your screen.

Parameters

lpCursorInfo
Points to a **CURSORINFO** structure containing a device information that is filled in by the device driver. The first word is the X mickey-to-pixel ratio, and the second word is the Y mickey-to-pixel ratio.

Return Value

The return value is the number of bytes (4) in the AX register actually written into the data structure.

Comments

The export ordinal for this function is 101.

This function is called once per initialization before the **Enable** function.

MOUSETRAILS

#define MOUSETRAILS 39

WORD Control(*lpDestDevice***, MOUSETRAILS,** *lpTrailSize***, NULL)**
LPPDEVICE *lpDestDevice***;**
LPINT *lpTrailSize***;**

The MOUSETRAILS escape enables or disables mouse trails for the display driver.

Parameters

lpDestDevice
Points to a **PDEVICE** structure specifying the destination device.

lpTrailSize
Points to a 16-bit variable containing a value specifying the action to take, and the number of mouse cursor images to display (trail size). The variable can be one of the following values.

Value	Meaning
1 through 7	Enables mouse trails and sets the trail size to the specified number. A value of 1 requests a single mouse cursor; 2 requests that one extra mouse cursor be drawn behind the current mouse cursor, and so on, up to a maximum of 7 total cursor images. The function sets the **MouseTrails** setting in the WIN.INI file to the given value. The function then returns the new trail size.
0	Disables mouse trails. The function sets the **MouseTrails** setting to the negative value of the current trail size (if positive). The function then returns the negative value.
–1	Enables mouse trails. The display driver reads the **MouseTrails** setting from the [Windows] section of the WIN.INI file. If the setting value is positive, the function sets the trail size to the given value. If the setting value is negative, the function sets trail size to the setting's absolute value and writes the positive value back to WIN.INI. If the **MouseTrails** setting is not found, the function sets the trail size to 7 and writes a new **MouseTrails** setting to the WIN.INI file, setting its value to 7. The function then returns the new trail size.

Value	Meaning
–2	Disables mouse trails but does not cause the driver to update the WIN.INI file.
–3	Enables mouse trails but does not cause the driver to update the WIN.INI file.

Return Value

The return value specifies the size of the escape if successful. Otherwise, it is zero if the escape is not supported.

Comments

An application can retrieve the current trail size by calling the QUERYESCSUPPORT escape as shown in the following example:

```
Escape(hDC, QUERYESCSUPPORT, sizeof(int), MOUSETRAILS, NULL);
```

The QUERYESCSUPPORT escape returns the current trail size without changing to the current setting. If the escape returns zero, mouse trails are not supported by the display driver. If the escape returns a positive value, mouse trails are enabled and the value specifies the current trail size. If the escape returns a negative value, mouse trails are currently disabled and the value is the same as the negative **MouseTrails** value specified in the WIN.INI file.

MoveCursor

void MoveCursor(*wAbsX*, *wAbsY*)
WORD *wAbsX*;
WORD *wAbsY*;

The **MoveCursor** function moves the cursor to the given screen coordinates. If the cursor is a composite of screen and cursor bitmaps (that is, not a hardware cursor), this function must ensure that screen bits under the current cursor position are restored and the bits under the new position are saved. The function must move the cursor, even if the cursor is not currently displayed.

Parameters

wAbsX
Specifies the absolute *x*-coordinate of the new cursor position.

wAbsY
Specifies the absolute *y*-coordinate of the new cursor position.

Return Value

This function has no return value.

Comments

The export ordinal for this function is 103.

Microsoft Windows may specify a position at which the cursor shape would lie partially outside of the display bitmap. The OEM function is responsible for clipping the cursor shape to the display boundary.

The **MoveCursor** function is called at mouse-interrupt time, outside of the main thread of Windows processing. Since **MoveCursor** may even interrupt its own processing, the device driver should disable interrupts while reading the *wAbsX* and *wAbsY* coordinates by using the **EnterCrit** and **LeaveCrit** macros in the WINDEFS.INC file. Do not use **sti** and **cli** instructions in the driver.

SaveScreenBitmap

WORD SaveScreenBitmap(*lpRect*, *wCommand*)
LPRECT *lpRect*;
WORD *wCommand*;

The **SaveScreenBitmap** function saves a single bitmap from the display or restores a single (previously stored) bitmap to the display. It is used, for example, when a menu is pulled down, to store the part of the screen that is "behind" the menu until the menu is closed.

Parameters

lpRect
Points to a **RECT** structure containing the rectangle to use.

wCommand
Specifies the action to take. It can be one of the following values:

Value	Meaning
0	Saves the rectangle.
1	Restores the rectangle.
2	Discards previous save, if there was one.

Return Value

The return value is AX=1 if successful. Otherwise, AX=0 for any of the following error conditions:

- "Shadow memory" does not exist (save, restore, ignore).
- "Shadow memory" is already in use (save).

- "Shadow memory" is not in use (restore).
- "Shadow memory" has been stolen or trashed (restore).

Comments

The export ordinal for this function is 92.

Because **SaveScreenBitmap** can save only one bitmap at a time, the device driver must maintain a record of whether or not the save area is currently in use.

The bitmap is stored in "shadow memory" (that is, memory for which the device has control of allocation). Therefore, the device can save the bitmap in whatever form is most convenient for it, without the rest of Windows worrying about where it goes.

SetCursor

void SetCursor(*lpCursorShape*)
LPCURSORSHAPE *lpCursorShape*;

The **SetCursor** function sets the cursor bitmap that defines the cursor shape. Each call replaces the previous bitmap with that pointed to by *lpCursorShape*. If *lpCursorShape* is NULL, the cursor has no shape and its image is removed from the display screen.

Parameters

lpCursorShape
Points to a **CURSORSHAPE** structure that specifies the appearance of the cursor for the specified device.

Return Value

This function has no return value.

Comments

The export ordinal for this function is 102.

The cursor bitmap is actually two bitmaps. The first bitmap is ANDed with the contents of the screen, and the second is XORed with the result. This helps to preserve the appearance of the screen as the cursor is replaced and ensures that at least some of the cursor is visible on all the potential backgrounds.

2.4 Windows for Pen Computing Function Reference

The following is an alphabetical listing of the Pen computing functions for display drivers.

GetLPDevice

LPPDEVICE GetLPDevice(VOID);

The **GetLPDevice** function retrieves a pointer to the display driver's physical device structure (**PDEVICE**). The Recognition Context Manager (RC) for Pen computing calls this function when preparing to ink.

Display drivers that support Pen computing must export the **GetLPDevice** function.

Parameters

This function has no parameters.

Return Value

The return value is a 32-bit pointer to the physical-device structure (**PDEVICE**).

Comments

The export ordinal for this function is 601.

The RC Manager uses the pointer in subsequent calls to the display driver's **Output** function. When the RC Manager has pen strokes to draw, it calls **Output** using the OS_POLYLINE parameter.

See Also

InkReady, **Output**

InkReady

BOOL InkReady(*lpfn*)
LPFN *lpfn*; /* pointer to callback function */

The **InkReady** function notifies the display driver that the Recognition Context Manager (RC) is ready to ink a path drawn by the pen. The RC Manager for Pen computing calls this function when the user stroked a path using the pen. The display driver completes the inking by calling the callback function pointed to by the *lpfn* parameter. Depending on whether any other drawing operation is under way, the driver may call the callback function immediately or wait for the current drawing operation to complete.

Display drivers that support Pen computing must export the **InkReady** function.

Parameter

lpfn
Points to the callback function that completes the inking.

Return Value

The return value is TRUE if it completed the inking (called the callback function) immediately. Otherwise, it is FALSE to indicate that the display driver will complete the inking as soon as the current drawing operation is complete.

Comments

The export ordinal for this function is 600.

The **InkReady** function determines whether any other drawing operation is under way. If not, the function should call the callback function immediately, and then return TRUE. If a drawing operation is under way, **InkReady** must save the callback function address and return FALSE. The display driver is then responsible for calling the callback function as soon as the current drawing operation is complete.

The callback function pointed to by *lpfn* takes no parameters and returns no value.

See Also

GetLPDevice

2.5 File-Format Reference

The following is an alphabetical listing of the file formats that are specific to display drivers.

CONFIG.BIN

```
typedef struct tagCONFIG_BIN {

    /* machine-dependent parameters */

    short   VertThumHeight; /* vertical thumb height (in pixels)  */
    short   HorizThumWidth; /* horizontal thumb width (in pixels) */
    short   IconXRatio;     /* icon width (in pixels)             */
    short   IconYRatio;     /* icon height (in pixels)            */
    short   CurXRatio;      /* cursor width (in pixels)           */
    short   CurYRatio;      /* cursor height (in pixels)          */
    short   Reserved;       /* reserved                           */
    short   XBorder;        /* vertical-line width                */
    short   YBorder;        /* horizontal-line width              */

    /* default-system color values */

    RGBQUAD clrScrollbar;
    RGBQUAD clrDesktop;
    RGBQUAD clrActiveCaption;
    RGBQUAD clrInactiveCaption;
    RGBQUAD clrMenu;
    RGBQUAD clrWindow;
    RGBQUAD clrWindowFrame;
    RGBQUAD clrMenuText;
    RGBQUAD clrWindowText;
    RGBQUAD clrCaptionText;
    RGBQUAD clrActiveBorder;
    RGBQUAD clrInactiveBorder;
    RGBQUAD clrAppWorkspace;
    RGBQUAD clrHiliteBk;
    RGBQUAD clrHiliteText;
    RGBQUAD clrBtnFace;
    RGBQUAD clrBtnShadow;
    RGBQUAD clrGrayText;
    RGBQUAD clrBtnText;
    RGBQUAD clrInactiveCaptionText;
} CONFIG_BIN;
```

The CONFIG.BIN resource defines the default values for system colors, line widths (both horizontal and vertical), scroll bar "thumb" sizes, and cursor and icon widths or compression ratios.

The CONFIG.BIN resource is required for display drivers.

Members

VertThumHeight
Specifies the height in pixels of the vertical scroll-bar thumb.

HorizThumWidth
Specifies the width in pixels of the horizontal scroll-bar thumb.

IconXRatio
Specifies either the icon width (in pixels) or the compression ratio. If a width is specified, it must be greater than 10. The icon resources in the driver must have the specified width. If a compression ratio is specified, it must be either 1 or 2, and all icon resources in the display driver must be 64-by-64-bits.

IconYRatio
Specifies either the icon height (in pixels) or the compression ratio. If a width is specified, it must be greater than 10. The icon resources in the driver must have the specified width. If a compression ratio is specified, it must be either 1 or 2, and all icon resources in the display driver must be 64-by-64-bits.

CurXRatio
Specifies either the cursor width (in pixels) or the compression ratio. If a width is specified, it must be greater than 10. The cursor resources in the driver must have the specified width. If a compression ratio is specified, it must be either 1 or 2, and all cursor resources in the display driver must be 32-by-32-bits.

CurYRatio
Specifies either the cursor height (in pixels) or the compression ratio. If a height is specified, it must be greater than 10. The cursor resources in the driver must have the specified height. If a compression ratio is specified, it must be either 1 or 2, and all cursor resources in the display driver must be 32-by-32-bits.

Reserved
Reserved; must be zero.

XBorder
Specifies the thickness in pixels of vertical lines.

YBorder
Specifies the thickness in pixels of horizontal lines.

clrScrollbar
Specifies the color of the scroll bar.

clrDesktop
Specifies the color of the Windows background.

clrActiveCaption
Specifies the color of the caption in the active window.

clrInactiveCaption
Specifies the color of the caption in an inactive window.

clrMenu
Specifies the color of the menu background.

clrWindow
Specifies the color of a window's background.

clrWindowFrame
Specifies the color of the window frame.

clrMenuText
Specifies the color of the text in a menu.

clrWindowText
Specifies the color of the text in a window.

clrCaptionText
Specifies the color of the text in an active caption.

clrActiveBorder
Specifies the default color of the text in an active border.

clrInactiveBorder
Specifies the color of the text in an inactive border.

clrAppWorkspace
Specifies the color of the application workspace (MDI background).

clrHiliteBk
Specifies the highlight color used in menus, edit controls, list boxes, and so on.

clrHiliteText
Specifies the text color for highlighted text.

clrBtnFace
Specifies the color of the 3-D button face shading.

clrBtnShadow
Specifies the color of the 3-D button edge shadow.

clrGrayText
Specifies the color of solid gray to be used for drawing disabled items. This member must be set to RGB(0,0,0) if no solid gray is available.

clrBtnText
Specifies the color of button text.

clrInactiveCaptionText
Specifies the color of the text in an inactive caption.

Comments

The resource identifier for this structure is 1; the resource type is OEMBIN.

The recommended default system colors depend on the type of display. There are the following recommended values.

Member	16-Color	256-Color
clrActiveBorder	192,192,192	192,192,192
clrActiveCaption	000,000,128	164,200,240
clrAppWorkspace	255,255,255	255,251,240
clrBtnFace	192,192,192	192,192,192
clrBtnShadow	128,128,128	128,128,128
clrBtnText	000,000,000	000,000,000
clrCaptionText	255,255,255	000,000,000
clrDesktop	128,128,000	160,160,164
clrGrayText	192,192,192	192,192,192
clrHiliteBk	000,000,128	164,200,240
clrHiliteText	255,255,255	000,000,000
clrInactiveBorder	192,192,192	192,192,192
clrInactiveCaption	255,255,255	255,255,255
clrInactiveCaptionText	000,000,000	000,000,000
clrMenu	255,255,255	255,255,255
clrMenuText	000,000,000	000,000,000
clrScrollbar	192,192,192	192,192,192
clrWindow	255,255,255	255,255,255
clrWindowFrame	000,000,000	000,000,000
clrWindowText	000,000,000	000,000,000

DISPLAY

DESCRIPTION 'DISPLAY : *AspectXY*, *LogPixelsX*, *LogPixelsY* : *Description*'

The **DESCRIPTION** statement in a display driver's module-definition file identifies the driver DLL as a a display driver, specifies the aspect-ratio values associated with the display device, and provides a description for the driver.

Parameters

AspectXY

Specifies the aspect ratio for the display. This parameter must be set to the same value as the driver's **dpAspectXY** member in the **GDIINFO** structure.

LogPixelsX
Specifies the horizontal pixels-per-inch for the display. This parameter must be set to the same value as the driver's **dpLogPixelsX** member in the **GDIINFO** structure.

LogPixelsY
Specifies the vertical pixels-per-inch for the display. This parameter must be set to the same value as the driver's **dpLogPixelsX** member in the **GDIINFO** structure.

Description
Specifies the name the display models supported by this driver. Although more than one name can be given, the names must not be separated commas (,).

Comments

The **DISPLAY** keyword must be capitalized.

Example

The following example shows the **DESCRIPTION** statement for the VGA display driver:

```
DESCRIPTION 'DISPLAY : 100,96,96 : VGA Color Display'
```

The Microsoft Windows printer drivers manage all printer output for Windows applications. Each driver provides a set of functions that Windows uses to initialize the printer, retrieve information about the printer, print text and graphics, and allow users to modify the operation of the printer.

4.1 About the Printer Driver

A printer driver is a dynamic-link library (DLL) that consists of a set of graphics functions for a particular printer device. These graphics functions translate device-independent graphics commands into a stream of device-dependent commands and data. A printer driver receives the graphics commands from the graphics-device interface (GDI) portion of Windows and sends device commands either directly or indirectly to the printer.

Each printer driver supports a specific printer technology. Printer technologies include raster devices (for example, dot-matrix printers) and vector devices (for example, plotters), and devices with higher-level languages, such as PostScript page-description language (PDL). However, a printer driver can support any number of models and manufacturers as long as the printers share the same basic technology.

Unlike most Windows device drivers, a printer driver is generally not responsible for hardware communication with the printer. Instead, the driver uses existing Windows functions to open and write to a printing queue, to a file on disk, or directly to the printer through a communications port.

4.1.1 Printer-Driver Initialization

Printer-driver initialization occurs whenever Windows or a Windows application loads the printer driver using the **LoadLibrary** function (KERNEL.95). Windows loads a printer driver whenever an application uses the **CreateDC** function (GDI.53) to create a device context for the printer. Windows applications load the driver to prepare for subsequent calls to the printer driver's **DeviceMode**, **ExtDeviceMode**, or **DeviceCapabilities** function.

As with other dynamic-link libraries, Windows calls the printer driver's initialization routine (if any) when it loads the driver. The routines of most drivers do little more than initialize the heap and load resources, such as the name of the driver's Help file. Although a driver may require additional initialization, it typically waits until GDI provides the additional information needed for this initialization when it calls the driver's **Enable** function.

If a printer driver allocates global resources or alters the state of the system, the driver is responsible for freeing these resources and restoring the previous state before the driver terminates. Because every driver includes a **WEP** function that Windows calls as the driver is quitting, the printer driver can use this function to free resources and restore the system.

4.1.2 GDI Information Structure

Every printer driver must have a **GDIINFO** structure that specifies the printer's capabilities and characteristics. GDI uses this information to determine what the printer can do and how much GDI must do to support graphics output to the printer. The GDI information can be classified as follows:

- Driver management
- Driver capabilities
- Device dimensions

The driver-management information specifies the version of Windows for which the driver was written, the type of technology the printer uses to generate output, the size in bytes of the printer's **PDEVICE** structure, and number of device contexts the driver can manage at the same time.

The driver-capabilities information specifies the capabilities of the device, such as whether the printer can draw polygons and ellipses, scale text, and clip output. Driver capabilities also specify the number of device brushes, pens, fonts, and colors available on the printer and whether the printer can handle bitmaps and color palettes.

The device-dimension information specifies the maximum width and height of the printable area in both millimeters and device units, the number of color bits or planes, the aspect ratio, the minimum length of a dot in a styled line, and the number of device units or pixels per inch.

4.1.3 Enabling and Disabling Physical Devices

GDI enables operation of the printer driver by calling the driver's **Enable** function and directing the driver to initialize a physical device for subsequent graphics output. A physical device is a **PDEVICE** structure that represents a printer and its current operating state. A printer driver uses the physical-device information to determine how to carry out specific tasks, such as which device-dependent graphics commands to use, and which output port to send the commands. The printer driver initializes the physical device by copying information to the **PDEVICE** structure.

GDI calls the **Enable** function whenever an application calls the **CreateDC** function (GDI.53) to create a device context (DC) for the printer. GDI calls the function twice: once to retrieve a copy of the driver's **GDIINFO** structure, and a second time to initialize the **PDEVICE** structure. After the first call, GDI uses the **dpDEVICEsize** member in the **GDIINFO** structure to determine the size of the driver's **PDEVICE** structure. GDI then allocates memory for the structure and calls **Enable** for the second time, passing a pointer to the structure. With this call, the driver initializes the structure.

To initialize the **PDEVICE** structure, the driver typically examines the names of the printer model and output device or file passed to **Enable** by GDI. It may also examine any printer environment passed to **Enable**. The driver then fills the **PDEVICE** structure with all the information that the output functions need to generate appropriate graphics commands for the given printer model and to send the commands to the given device or file.

Although only the printer driver initializes and uses the **PDEVICE** structure, GDI allocates memory for the structure, determines when to pass it to the driver's output functions, and deletes the structure when it is no longer needed. Except for the first 16 bits of the **PDEVICE** structure, the content and format of the structure depends entirely on the printer driver. The first 16 bits, on the other hand, must be set to a nonzero value. GDI reserves zero to indicate a **BITMAP** structure. GDI creates and uses a **BITMAP** structure in place of a **PDEVICE** structure when an application creates a memory-device context.

A printer driver can allocate additional structures and store their addresses in the **PDEVICE** structure. Because GDI may direct a printer driver to create a large number of physical devices, the printer driver should not allocate additional structures in the limited space of the driver's automatic data segment, especially if the driver allows multiple device contexts.

GDI disables the physical device and possibly frees the printer driver whenever an application calls the **DeleteDC** function (GDI.68). GDI disables the physical device by calling the driver's **Disable** function. It expects the driver to complete any outstanding printing job and free any resources associated with the physical device. After the driver returns from the **Disable** function, GDI frees the memory it allocated for the **PDEVICE** structure. If there are no other device contexts for this printer driver, Windows frees the driver, removing any driver code and data from memory.

4.1.4 Device-Context Management

Since Windows is a multitasking environment, more than one application may create a device context for a printer at the same time. This means GDI may direct the printer driver to initialize more than one physical device. The printer driver sets

the **dpDCManage** member in its **GDIINFO** structure to specify how it will manage these requests. The driver can specify one of the following methods:

- Driver allows multiple device contexts. It creates a new **PDEVICE** for each device context that specifies a new device and filename pair, but uses the same **PDEVICE** for any subsequent device contexts that specify the same device and filename pair.
- Driver allows multiple device contexts, but it creates a new **PDEVICE** for each device context regardless of whether the device and filename pairs are the same.
- Driver allows multiple device contexts but only if all device contexts have unique device and filename pairs. The driver creates a **PDEVICE** for each device context. The driver returns an error on any attempt to create a second device context with an existing device and filename pair.
- Driver allows multiple device contexts, but only creates one **PDEVICE**. All device contexts share the same **PDEVICE** regardless of the device and filename pairs.
- Driver allows only one device context. The driver returns an error on any attempt to create a second device context.

Printer drivers normally allow multiple device contexts but create new **PDEVICE** structures for each device context. With separate **PDEVICE** structures, the driver can maintain information about multiple print jobs without one job affecting the other.

4.1.5 Dimensions and Mapping Modes

A printer driver provides values in its **GDIINFO** structure specifying the aspect ratio, logical pixels-per-inch, and mapping modes for the printer. The following sections detail these modes.

4.1.5.1 Aspect Ratio

The aspect ratio defines the relative dimensions of the printer's pixels. The ratio consists of three values: an *x*-, *y*-, and *xy*-aspect. These represent the relative width, height, and diagonal length (or hypotenuse) of a pixel. GDI uses the aspect ratio to determine how to draw squares and circles as well as drawing lines at an angle.

The aspect values have the following relationship:

```
(dpAspectXY * dpAspectXY) == (dpAspectX * dpAspectX) +
(dpAspectY * dpAspectY)
```

Because the dimensions are given as relative values, they may be scaled as needed to obtain accurate integer values. Keep the values under 1000 for numerical stability in GDI calculations. For example, a device with a 1:1 aspect ratio (such as a 300 dpi laser printer) can use 100 for the **dpAspectX** member and the **dpAspectY** member and 141 (100 * 1.41421...) for **dpAspectXY**.

4.1.5.2 Logical Pixels-Per-Inch

Printer drivers should always use real inches. A 300 dpi laser printer puts 300 in both members.

4.1.5.3 Mapping Modes

The printer driver provides viewport and window extents for the standard mapping modes: MM_LOENGLISH, MM_HIENGLISH, MM_LOMETRIC, MM_HIMETRIC, and MM_TWIPS.

Place the device resolution in pixels-per-inch in the viewport extents and the number of logical units-per-inch in the window extent. The *y*-coordinate of the viewport is negated to reflect the fact that the *x*-axis is along the top of the paper in the default mapping mode (**MM_TEXT**, which specifies device coordinates) with *y* increasing while going down the page; whereas in the other mapping modes, the *x*-axis is along the bottom edge of the page.

For example, on a 300 dpi laser printer, the **MM_TWIPS** mapping mode will require that the **dpTwpWin** member be set to (1440,1440) and the **dpTwpVpt** member be set to (300, –300).

4.1.6 Printer-Driver Environment

The printer-driver environment consists of information about the printer, such as font cartridges, paper trays and sizes, printer orientation, graphics capabilities, color, and other advanced features. Windows applications use this information to create printed output that takes full advantage of the printer environment.

Printers normally have a large number of options from which the user can select. This information can come from any of four sources:

- The driver's default setup.
- The driver's WIN.INI section of user options. The WIN.INI should maintain at least one such section so that modified printer setups can be retained from session to session. This information is edited by the driver's Setup dialog box.

- The driver may call GDI to retain the driver's environment from device context to device context on a port-by-port basis. This allows faster initialization of the driver and avoids the time-consuming process of reading options from the WIN.INI file.
- The application can pass the environment to the driver in a buffer pointed to by the *lpInitData* parameter of the **Enable** function.

Upon device initialization (that is, during the pair of **Enable** functions), this information is used to set up information in the **GDIINFO** and **PDEVICE** structures. For example, the paper size selection will affect the height and width fields. Also, a printer that allows multiple graphics densities will modify the various resolution fields.

4.1.7 Device-Mode Dialog Boxes

The **DEVMODE** structure is used for the environment and the initialization data (which are the same). By convention, all drivers place the device name in the first 32 bytes of **DEVMODE** as a null-terminated string. All the other data is device dependent.

For Windows 3.*x*, a new convention has been adopted that defines an additional number of members. These members enable the application to perform device-independent manipulations of the device environment.

When **Enable** is called, the device driver should first check the *lpInitData* parameter to see if the application has supplied valid initialization data. If it is valid, then the driver should use that environment to initialize the **GDIINFO** and **PDEVICE** structures and not use or modify the default environment information.

If the environment cannot be found or if the data obtained is invalid or intended for another device, the device driver should extract user settings from the WIN.INI file, by using the profile string functions, such as **GetProfileInt**.

However, the driver should contain useful defaults for all strings, so that it can create a valid environment even if the WIN.INI file is empty.

The driver should use the device name string at the beginning of the **DEVMODE** structure to determine whether or not the environment obtained from **GetEnvironment** is correct.

A driver may also maintain additional information in its **DEVMODE** structure to determine validity if the device name matches one the driver supports.

The printer driver can set or delete the environment by using the **SetEnvironment** function (GDI.132). It can retrieve the current environment using the **GetEnvironment** function (GDI.133).

The driver should always set up the default environment if it is not present, except when the driver is initialized with a non-default environment (that is, the *lpInitData* parameter to **Enable** points to application-supplied data).

4.1.7.1 New Members in the DEVMODE Structure

The **dmYResolution** and **dmTTOption** members in the **DEVMODE** structure are new for Windows 3.1. The **DEVMODE** structure returned by the **ExtDeviceMode** function contains one or both of these members if the **dmField** member includes the following values:

```
DM_YRESOLUTION  0x0002000L
DM_TTOPTION     0x0004000L
```

The **dmYResolution** member specifies the vertical resolution of the printer in dots per inch. In this case, the **dmPrintQuality** member specifies the horizontal resolution in dots per inch. If the DM_YRESOLUTION bit is not set in **dmFields**, **dmYResolution** is not used and **dmPrintQuality** retains the original meaning.

The **dmTTOption** member specifies how TrueType fonts should be printed. The member can be set to one of the following values.

Value	Meaning
DMTT_BITMAP (1)	Prints TrueType fonts as graphics.
DMTT_DOWNLOAD (2)	Downloads TrueType fonts as soft fonts.
DMTT_SUBDEV (3)	Substitutes device fonts for TrueType.

Note Before calling the the **CreateDC** or **CreateIC** functions, an application should call the **DeviceCapabilities** function with the DC_TRUETYPE index to retrieve the driver's TrueType capabilities. The application can use the value returned by the driver to set the **dmTTOption** member to the appropriate value.

The default action is to download TrueType as soft fonts for Hewlett-Packard printers that use PCL; substitute device fonts for TrueType for PostScript printers; and to print TrueType fonts as graphics for dot-matrix printers.

4.1.8 Printer-Device Modes

All printer drivers are required to export the **DeviceMode** function, which displays a dialog box to edit the default environment. This function sets the profile strings in the WIN.INI file for the options chosen by the user. It should also set the environment using the **SetEnvironment** function. The **DeviceMode** function uses USER functions, such as **DialogBox** (USER.240), to display dialog boxes.

The most common way to call **DeviceMode** is with Control Panel. However, other applications that make heavy use of printer output, such as Microsoft Write or Microsoft Excel, may also provide a means for calling the printer driver's **DeviceMode** function.

Windows 3.*x* drivers also export two environment-related functions, **ExtDeviceMode** and **DeviceCapabilities**. These functions are designed to allow greater application control over the printer environment. All printer drivers should implement these functions.

ExtDeviceMode enables the application to call the driver to obtain device initialization data either from the user or from the application's modifications to the default environment. GDI then calls the **Enable** function with a pointer to this information, allowing the driver to preset its **GDIINFO** and **PDEVICE** structures according to the application's options, rather than the defaults. That way, the application can store different printer settings for itself and its documents or even request specific setup properties, such as orientation.

DeviceCapabilities enables the application to get information about such things as what **DeviceMode** fields the driver uses and what limits, ranges, or selections are valid.

4.1.8.1 Indexes for the DeviceCapabilities Function

Printer drivers must process requests from applications for new **DeviceCapabilities** indexes. A driver's **DeviceCapabilities** function must check for and process the following values.

Value	Meaning
DC_ENUMRESOLUTIONS (13)	Retrieves a list of resolutions supported by the model.
DC_FILEDEPENDENCIES (14)	Retrieves a list of filenames which also need to be installed when the driver is installed.
DC_TRUETYPE (15)	Retrieves the driver's capabilities with regard to printing TrueType fonts.
DC_PAPERNAMES (16)	Enumerates the actual string describing the paper names.
DC_ORIENTATION (17)	Retrieves the relationship between Portrait and Landscape orientations in terms of the number of degrees that portrait orientation is to be rotated counterclock-wise to get landscape orientation.
DC_COPIES (18)	Retrieves the maximum number of copies the device can print.

The implementation of one or all of the indices is optional. If a driver does not implement a given index, the **DeviceCapabilities** function should return –1. For more information about the new indices, see the **DeviceCapabilities** function.

4.1.8.2 New Paper Sizes

There are 21 new predefined paper sizes. The include file, PRINT.H, contains a complete list of all supported paper sizes. Applications typically call the **DeviceCapabilities** function to retrieve names and sizes of the supported paper sizes, so printer drivers must be ready to process the following **DeviceCapabilities** index values:

Value	Meaning
DC_PAPER	Retrieves the identifiers of the listed paper sizes.
DC_PAPERNAMES	Retrieves the names of the listed paper sizes.
DC_PAPERSIZES	Retrieves the width and height of the listed paper sizes.

4.1.9 Per-Page Printer Settings

Applications can now modify printer settings (that is, **DEVMODE** values) while printing a document. The **ResetDC** function lets applications update the printer device context at each new page. This means applications can print documents containing a mixture of page orientations, paper sources, and other printing options without having to divide the document into smaller documents.

To support **ResetDC**, a printer driver must provide code for the RESETDEVICE escape. This escape, processed by the driver's **Control** function, copies printer output state information from one physical device context (**PDEVICE**) to another. GDI uses the escape to replace existing device contexts with new ones.

When an application calls **ResetDC**, GDI first creates a new **PDEVICE** by calling the driver's **Enable** function. This new **PDEVICE** receives the new printer settings as specified by the application. GDI next uses the RESETDEVICE escape to copy output state information from the existing **PDEVICE** to the new one. The output state includes information such as job number and list of downloaded fonts. The escape must copy the state information without changing the printer settings in the new **PDEVICE**. Finally, GDI substitutes the new **PDEVICE** for the old one in the application's device context, discarding the old **PDEVICE** by calling the **Disable** function.

When the printer driver receives the RESETDEVICE request, it must expect a subsequent call to **Disable** with no intervening call to the ENDDOC escape. The printer driver must also ensure that the new printer settings take affect starting on the next printed page. If a printer driver allocated bitmap or working buffers when

it received the STARTDOC escape, the driver should free the buffers for the old **PDEVICE** and allocate new buffers for the new **PDEVICE** (an option is to reallocate the buffers).

The **ResetDC** function cannot be used to change the driver name, device name, or the output port. When the user physically changes port connection or the device name, the application must delete the original device context and recreate a device context with the new device name and output port. Also the application must cancel the selection of all objects in its device context before calling **ResetDC**.

4.1.10 Printer-Model Names

To make printer driver installation easier and more intuitive, printer-model names have been removed from the **DeviceMode** dialog box. Users have found it confusing to select a printer model when installing the driver only to have to reselect the printer model again within the printer's **DeviceMode** dialog box. For Windows 3.1, the **DeviceMode** dialog box displays the name of the printer model (selected during installation by the user) in the title bar of the dialog box.

To support this change, printer drivers must now have corresponding SETUP.INF or OEMSETUP.INF entries that list all the printer-model names. Control Panel uses the entries to determine which printer models are available. If the entries aren't available, Control Panel obtains the printer-model name from the **DESCRIPTION** statement in the driver's executable file header.

4.1.11 Standard Print Dialogs

Windows 3.1 includes a set of standard print dialogs for use by Windows applications. These dialogs provide a standard interface for users to select and change printer settings, such as page range, number of copies, and print quality.

To support the standard print dialogs, a printer driver must export the **ExtDeviceMode** and **DeviceCapabilities** functions. The dialogs call these functions to get the necessary settings for the standard dialog boxes. If a printer driver does not export these functions, no printer settings can be set from within an application using the standard dialog. In such cases, the dialog boxes are displayed, but the printer settings fields are disabled (greyed out) and the driver is forced to use the system defaults that were previously set in Control Panel.

A printer driver can extend the capabilities of the standard print dialogs by exporting the **AdvancedSetUpDialog** function. If this function is available, the standard print dialogs display a More button. If the user chooses the button, the dialog box calls the **AdvancedSetUpDialog** function allowing the driver to display its own dialog box containing fields for advanced printer settings, such as color, duplex printing, and screen parameters.

4.1.12 Printer Entries in the WIN.INI File

Printer drivers that support more than one model should now record current printer settings for each model even if the models use the same port. A driver records the printer settings for a given model by adding to the WIN.INI file a section having the following form:

[*ModelName*, *Port*]

The *ModelName* specifies the name of the model and *Port* specifies the output port current associated with the printer as shown in the following example:

```
[HP LaserJet IIID, LPT2]
```

This section must include all printer settings except printer fonts. To record font information, the driver must add a section having the following form:

[*DeviceName*, *Port*]

The *DeviceName* specifies the name of the device and the port specifies the name of the output port associated with the device as shown in the following example:

```
[HPPCL5A, LPT1]
```

This section only lists fonts, including those in external cartridges. Also this section ensures backward compatibility with third-party font packages that use this section to add fonts.

4.1.13 Physical Objects

Physical objects are device-dependent representations of the logical pens, brushes, and fonts that Windows applications use to perform output. GDI directs a printer driver to create physical objects whenever an application selects a logical object for subsequent drawing. The process of converting a logical object into a physical object is called realizing the object, and every printer driver provides a **RealizeObject** function to carry out this task.

GDI calls the **RealizeObject** function when an application calls the **SelectObject** function (GDI.45) to select a logical object for a device context. GDI calls the function twice: once to retrieve the size of the physical object, and a second time to realize the object. Although the printer driver realizes an object, GDI manages it, allocating memory for the object, passing the object to the driver to perform output, and deleting the object when it is no longer needed.

GDI realizes a physical object when it is selected to avoid the overhead of realizing the object each time it is used for drawing. The contents and organization of the structure defining a physical object are specific to the driver. Usually, the structure includes the logical object plus any other information that the driver needs.

4.1.14 Device Pens and Brushes

Most printer drivers maintain and manipulate device objects. Device objects are structures that represent pens and brushes that the device supports.

To determine what device objects are available, GDI calls the **EnumObj** function and expects the driver to enumerate all the pens and brushes that a device supports. The function translates physical descriptions of the objects into logical descriptions and returns these descriptions to GDI. All logical objects created this way must be unique, that is, when translated into physical objects and used for drawing, they should produce different output.

The driver enumerates all the styles and colors of pens and brushes. Since pens are defined to be only pure colors, only logical colors that translate to pure physical colors will be enumerated. For devices that support many colors, only a subset of all the colors will be enumerated.

4.1.15 Device Fonts

The most common type of device object is a device font. Most printers are capable of printing some set of built-in fonts. The concept of device fonts enables drivers and applications to take advantage of a device's ability to render fonts. Device fonts are also expected to print faster and look better than GDI fonts.

To determine what device fonts are available, GDI calls the **EnumDFonts** function and expects the printer driver to enumerate the fonts. Typically, GDI first calls **EnumDFonts** passing an empty string for the font name. This indicates that the driver should enumerate each font name that it supports. On subsequent calls, GDI passes one of enumerated font names and expects the driver to enumerate all the sizes of that font.

Printer drivers may also wish to support GDI raster and vector fonts. For banding devices, it is usually not difficult to support GDI raster fonts because GDI contains support functions to render raster fonts into monochrome-band bitmaps. GDI raster fonts are most useful for devices (such as lower-resolution dot-matrix printers) with resolutions near those of the display.

For nonbanding devices, supporting GDI raster fonts is not as easy. In fact, the PostScript driver (a nonbanding device) does not support GDI raster fonts.

Supporting vector fonts is also optional. If a driver does not support vector fonts, GDI will simulate them by drawing line segments.

4.1.16 Color

A printer driver must provide support for translating logical colors (RGB values) to physical colors. For printers with color palettes, the printer driver must provide additional information in its **GDIINFO** structure.

4.1.16.1 Color Objects

GDI calls the **ColorInfo** function to translate physical and logical color representations. This function translates in both directions, that is, from physical to logical and from logical to physical. When given a logical color, the function returns the nearest physical color. When given a physical color, the function returns the logical color that best describes that color.

This function supports the GDI function **GetNearestColor**.

4.1.16.2 Color Palettes

GDI ignores the color palette members if the RC_PALETTE value is not set in the **dpRaster** member. However, they must be present (and accounted for in the length returned by **Enable**) if the driver is version 3.*x*.

4.1.17 Output

A printer driver provides a variety of graphics output from lines to text. The driver supports each type of output with a specific output function. When an application calls GDI to carry out a graphics operation, GDI calls the corresponding output function in the driver. The following lists output functions and associated graphics output.

Type of Output	Functions
Bitmaps	**BitBlt**, **StretchBlt**
Device-independent bitmaps	**SetDIBitsToDevice**, **StretchDIBits**
Floodfills	**ScanLR**
Lines and figures	**Output**
Pixels	**Pixel**
Text	**ExtTextOut**, **StrBlt**

Some output functions are optional. If a printer driver does not include an optional function, GDI simulates the output using other output functions.

When GDI calls the output functions, it passes parameters that specify the output as well as the physical device and physical objects to be used to generate the output.

4.1.18 Text

A printer driver provides supporting functions and values for text output, such as maximum text width and height, character widths, and character clipping.

4.1.18.1 Maximum Text Width and Height

The maximum width and length of text that can be printed on a page is determined by choosing either the Portrait or Landscape orientation. In Portrait orientation, the page is taller than wide, when viewing the text upright. In Landscape orientation, the page is wider than tall, when viewing the text upright.

4.1.18.2 Character Widths

Use the **GetCharWidth** function to determine character widths for variable-width fonts. It is important that the values returned by this function match and that the actual widths of the characters displayed. Any differences will produce misalignments, and any text formatting or justification will not work as intended.

4.1.18.3 Clipping Character Strings

Character strings should be clipped pixel-by-pixel like all other graphics, whenever possible. Some types of printers, however, do not allow pixel-precision clipping of device-based fonts. When this is the case, the driver should clip character strings on a per-character basis.

A driver should clip a character if any part of the character is outside the left or right edge of the clipping rectangle. If any part of the character is outside the top or bottom edge of the clipping rectangle, the driver should print the character.

4.1.19 Fonts

A printer driver provides support for the various fonts provided or supported by GDI.

4.1.19.1 Big Fonts

Some printer drivers use a new format for fonts when running in protected mode (with Windows running in either standard or 386 enhanced mode) on an 80386 or 80486-based computer. This format allows for fonts that can exceed 64K bytes of data, as well as some other options. In this situation, GDI will convert all the 2.*x* format fonts to the new format so that only one font file format is used in memory.

If a printer driver prints GDI bitmap fonts directly, it should be modified to handle big fonts. In many cases, a driver will not need to be updated if it is a banding monochrome driver that uses only the brute functions. The brute functions are already aware of the new format. For more information about brute functions, see Section 4.1.25 "The Brute Functions," later in this chapter.

If a printer driver uses the color library, the library supports the big font format, as well as the old 2.*x* format.

4.1.19.2 TrueType

GDI provides TrueType support for any driver that supports GDI raster fonts (that is, it supports drivers that set the DT_RASPRINTER value in the **dpTechnology** member of the **GDIINFO** structure). To support TrueType on these printers, GDI generates a raster font from the TrueType outlines and passes the font to the driver just as with any other raster font. This means many Windows 3.0 printer drivers, especially drivers for dot-matrix printers, currently support TrueType—no code changes in the driver are required.

Printer drivers that do not support raster fonts must be modified to support TrueType. Typically, the modification consists of adding code to convert TrueType fonts into a downloadable font format that the printer accepts. To simplify the task of adding TrueType support to nonraster printer drivers, GDI provides several TrueType service functions.

4.1.19.3 TrueType Service Functions

Printer drivers can use TrueType service functions to retrieve information about TrueType fonts, to retrieve bitmaps of individual glyphs in the fonts, and to realize a complete font, retrieving both information and bitmaps. Following are the service functions and brief descriptions.

Function	Description
EngineDeleteFont	Deletes a realized TrueType font.
EngineEnumerateFont	Enumerates TrueType fonts.
EngineGetCharWidth	Retrieves character widths for TrueType fonts.
EngineGetGlyphBmp	Retrieves the bitmap for a rasterized glyph.
EngineRealizeFont	Realizes a TrueType font.
EngineSetFontContext	Sets the context for rasterizing glyphs.
GetRasterizerCaps	Specifies whether TrueType is available.

A printer driver can determine whether TrueType is currently available by calling the **GetRasterizerCaps** function. Since users can turn TrueType off, it is important that a driver check for TrueType before generating output.

If TrueType is on, the driver can call the **EngineEnumFonts** function whenever GDI calls the driver's **EnumDFonts** function. This gives the font engine the chance to enumerate the TrueType fonts having the specified font. The driver can call the **EngineRealizeFont** function whenever GDI calls the driver's **RealizeObject** function. This gives the TrueType font engine an opportunity to generate a physical font that matches the specified logical font. When the engine realizes a font, it fills a **FONTINFO** structure with information about the font as well as the bitmap data for the individual glyphs. A driver can use this information to create a downloadable font in the format recognized by the printer.

In general, a printer driver should call the **EngineRealizeFont** and **EngineEnumerateFont** functions before processing its own device fonts (if any). If GDI requests that the driver delete a realized TrueType font (by calling **RealizeObject**), the driver must call the **EngineDeleteFont** function to delete the font.

A printer driver can retrieve character width information for a realized TrueType font by using the **EngineGetCharWidth** function. It can retrieve bitmaps for individual glyphs in the font by using the **EngineSetFontContext** and **EngineGetGlyphBmp** functions. The driver must call **EngineSetFontContext** first to set the font before calling **EngineGetGlyphBmp**. Drivers for printers that accept individual glyph definitions (as opposed to full font definitions) can use the character width information and bitmap data to download individual glyphs.

4.1.19.4 Specifying the Spot Size

The spot size is a set of values that help the TrueType rasterizer create the best glyph bitmaps for a given printer. All printer drivers can benefit by setting the spot size in their **GDIINFO** structure regardless of whether the driver uses TrueType service functions. The spot size should be specified in the **dpSpotSizeX** and **dpSpotSizeY** members of the driver's **GDIINFO** structure. If these members are not zero, GDI passes the values to the TrueType engine to help it rasterize glyphs.

4.1.20 Device-Independent Bitmaps

Device-independent bitmaps (DIBs) are bitmaps in a new format that was designed to provide a device-independent way for applications to transfer bitmap images to a variety of output devices. Besides the bitmap bits, these bitmaps contain color-table information and additional dimension information.

A printer driver should include support for DIBs, especially in color devices. If a driver does not support DIBs, GDI can convert DIBs into the standard monochrome bitmap format, but the quality of DIB output will rarely be satisfactory in such situations.

If a printer driver can do more with the bitmap, it should attempt to support DIBs handling, especially if it can perform its own half-toning or coloring of bitmaps. To enable such functionality, the driver should support the **SetDIBitsToDevice** and **StretchDIBits** functions, and possibly support the **SetDIBits** and **GetDIBits** functions if the driver deals with GDI bitmaps.

4.1.21 Print Jobs

A printer driver provides print job support by handling printer-specific escapes. GDI passes escapes to a printer driver's **Control** function whenever an application calls the **Escape** function (GDI.38).

4.1.21.1 The QUERYESCSUPPORT Escape

All drivers are required to implement the QUERYESCSUPPORT escape. For this escape, the *lpInData* parameter points to a 16-bit value that contains the index of another escape.

The driver returns a positive number if the driver implements that escape, or zero if the escape is unimplemented. The driver always returns nonzero if the escape queried is QUERYESCSUPPORT.

4.1.21.2 The SETABORTPROC Escape

SETABORTPROC is the first escape an application calls when printing. An application passes a pointer to a callback function in the *lpInData* parameter when it calls the SETABORTPROC escape. This callback function is used to check for user actions such as cancelling the print job. The printer driver, however, is not responsible for the callback function; GDI modifies the SETABORTPROC escape so that *lpInData* points to the application's device context handle.

The *hDC* parameter given to the driver by this escape should be used with the **OpenJob** function to enable the output functions in GDI to call the application's callback function. Printer drivers generally save this handle in the **PDEVICE** structure. If the application does not use SETABORTPROC, NULL should be passed to the **OpenJob** function.

4.1.21.3 The STARTDOC Escape

Usually, STARTDOC is the escape an application calls. STARTDOC indicates to GDI and the device driver that the application is printing.

This escape also supplies a Print Manager job title in a NULL-terminated string pointed to by *lpInData*. The *lpOutData* parameter is unused. This supplies the title used by the **OpenJob** function.

Together, with the port name supplied as a parameter to **Enable** and the *hDC* supplied by the SETABORTPROC escape, the driver now has all the data necessary to call the **OpenJob** function.

4.1.21.4 RESETDEVICE and STARTDOC Escapes

To support the **ResetDC** and **StartDoc** functions, which are new to Windows 3.1, printer drivers must process the RESETDEVICE and STARTDOC escapes in their **Control** functions.

Although the STARTDOC escape was available in Windows 3.0, the *lpInData* and *lpOutData* parameters have changed. Specifically, *lpInData* points to a null-terminated string specifying the name of the document, and *lpOutData* points to a **DOCINFO** structure specifying the output port or file as well as the document name. The structure has the following form:

```
typedef struct {
    short   cbSize;
    LPSTR   lpszDocName;
    LPSTR   lpszOutput;         // output port name
} DOCINFO, FAR * LPDOCINFO;
```

The **lpszOutput** is the name of the output file to use. If either *lpOutData* or **lpszOutput** is NULL, the output port given to **CreateDC** should be used.

The RESETDEVICE escape, corresponding to the new **ResetDC** function, allows the driver to move a printer's output state from an old physical device structure to a new one. This allows applications to change the printer setup, such as orientation, with creating a new print job. For more information about this escape, see Section 4.1.23, "Other Escapes."

4.1.21.5 The NEWFRAME Escape

An application calls the NEWFRAME escape when a new page is to be printed. The printer driver completes output for the given page and advances to the next page. NEWFRAME does not use the *lpInData* or *lpOutData* parameters.

4.1.21.6 The ENDDOC and ABORTDOC Escapes

When an application has completed printing all output, it calls the ENDDOC escape. ENDDOC does not use either the *lpInData* or *lpOutData* parameters. At this point, the driver may call the **CloseJob** function.

Another common escape is ABORTDOC, which is also called ABORTPIC in older documentation or applications, and has the same number assigned. This escape allows GDI or the application to cancel a print job. Generally, if the job is valid, the driver will clean up and call the **DeleteJob** function.

4.1.22 Banding Drivers

How the output functions are implemented depends on whether or not the device uses banding. Banding devices have their output stored in a metafile. This metafile is replayed for every band that is rendered (either by GDI or applications that wish to implement banding). Therefore, output coordinates must be mapped into the current band, and output outside of the band must be clipped.

Nonbanding devices perform output to the device in one pass. Therefore, the device must have access to the entire display surface. Drivers must be able to perform all the output functions to both the display surface and to memory bitmaps. This restriction would make it very difficult for devices that supported complex drawing primitives if it were not for the help that GDI and the display driver supply.

In the **GDIINFO** structure, banding color drivers should use the same bits-per-pixel and planes values used for band bitmaps.

Devices choose numbers that match how they output color. Color printers that use the color dot-matrix libraries specify the same values as the bitmap format. The color library in this DDK uses one bit-per-pixel in three planes. The driver developer must modify the library to use another format.

4.1.22.1 Raster vs. Vector Devices

Many printers (such as dot-matrix and most laser printers) are raster printers, that is, they print out text and graphics as bitmaps or raster lines. Other devices (such as plotters and PostScript-based printers) are vector devices, which draw text and graphics as a sequence of vectors or lines. (Although PostScript printers are based on raster engines, the language itself is vector oriented except where bitmaps are concerned.)

Raster devices usually have constraints that cause problems for implementing the full GDI device model. Raster devices, for example, do not implement any vector graphics operations. Therefore, all vector graphics must be drawn as a bitmap before printing. Some devices, such as dot-matrix printers, do not allow the driver to print anywhere on the page. They require that text and graphics be output in the order of the print direction position on the page.

These bitmaps can be enormous for a device such as a 300 dpi laser printer. In such cases, the driver breaks up the page into smaller rectangles that are printed individually. For each of the rectangles, GDI or the application will draw all the graphics that fit in each rectangle into a bitmap and then print each individual bitmap.

These rectangles are called bands, and the printing process that uses these bands is called banding. It is usually necessary to band raster printers; however, banding is not necessary for vector devices.

For vector devices (that is, nonbanding devices), the application calls GDI graphics functions, which are translated into device-driver graphics primitives. After each page is printed, the application uses the NEWFRAME escape to eject the page.

An application can either treat the driver as if it were a nonbanding device by calling the GDI functions and ending each page with the NEWFRAME escape, in which case GDI performs the banding, or it can handle the banding itself.

4.1.22.2 The NEXTBAND Escape

Before any graphics are drawn, the driver is called upon to perform the NEXTBAND escape. When the **Control** function is called for the NEXTBAND escape, *lpInData* points to a **POINT** structure, and *lpOutData* points to a **RECT** structure.

The driver should initialize its band bitmap and set the **RECT** structure to the size, in device coordinates, of the rectangle that the band represents on the page.

GDI adds the **POINT** structure to determine the scaling factor for graphics output. Some devices support the use of graphics at a lower resolution than text to allow for faster output. The *x*-coordinate of the **POINT** corresponds to horizontal scaling and the *y*-coordinate to vertical scaling.

The value in the structure corresponds to a shift count. A point of (0,0) specifies graphics at the same density as text, whereas a point of (1,1) specifies half-density graphics in both directions, for example, a 300 dpi laser printer printing bitmaps at 150 dpi.

GDI then calls the driver's **Output** function to draw text or graphics in the band bitmap. When all drawing for the band is finished, GDI calls the driver with another NEXTBAND escape. The driver draws the band in the band bitmap, reinitializes the bitmap, sets a new rectangle, and continues with the next band as it did with the first.

When all the bands on the page are exhausted, and the driver receives a NEXTBAND escape, it should output the last graphics band and then set the rectangle pointed to by the *lpOutData* parameter to an empty rectangle to indicate that there are no more bands on the page. It should also perform all the processing necessary to eject the completed page. The next NEXTBAND escape will correspond to the first band of the next page.

If the application performs banding, it will call the **Escape** function to get the band rectangles. If GDI is handling banding for an application, then GDI collects all the graphics functions on a page into a metafile, that is, a temporary file containing a list of the graphics functions and their parameters. When the application calls **Escape** to carry out the NEWFRAME escape, GDI turns this escape into a sequence of NEXTBAND calls to the **Control** function. GDI sets the clip region for the actual printer device context to the band rectangle and then plays back the metafile, which recreates all of the application's output in the band bitmap. GDI does this for each band until the band rectangle returned by the driver is empty.

Some devices, such as raster laser printers, allow text to be placed anywhere on the page at any time. Furthermore, these printers do not place text into the band bitmap, since all the device fonts exist in printer or cartridge memory. To optimize text output, their drivers use a single, full-page band for all the text output and a sequence of smaller bands for bitmapped graphics.

As an optimization, some of these drivers maintain a flag to detect whether or not any output, other than text, is attempted during the first, full-page band. If not, the driver skips the graphics bands.

4.1.22.3 The BANDINFO Escape

Some devices, such as laser printers, can print text and graphics anywhere on the page but still require banding support for vector graphics operations. Since these devices usually use their own internal device fonts, they can greatly improve their text printing performance by using a single, full-page band for text only as the first band. The driver ignores graphics calls during this band and handles only the **ExtTextOut** or **StrBlt** functions. Graphics are printed on subsequent, smaller bands.

An application that is aware of this process can speed up its printing operation by determining whether text or graphics will be printed on the current band. It may do so using the BANDINFO escape. The application can also use BANDINFO to optimize the banding process.

4.1.23 Other Escapes

Few applications use the QUERYESCSUPPORT escape to look for the SETABORTPROC, STARTDOC, NEWFRAME, ENDDOC, or ABORTDOC escapes. Therefore, a printer driver should handle all of these escapes.

In addition, there are a few applications that perform banding without verifying that banding is required either by using QUERYESCSUPPORT or the **GetDeviceCaps** function (which examines the **GDIINFO** structure). A nonbanding driver can easily support such an application by returning the full page as the band rectangle on the first NEXTBAND escape and returning an empty rectangle for the next NEXTBAND escape and ejecting the page.

There are a large number of other escapes that may or may not be appropriate to a specific driver. They are all listed alphabetically and described in detail in Chapter 11, "Graphics-Driver Escapes."

4.1.24 Print Manager Support

In most cases, printer drivers are not responsible for sending bytes directly to the output port. Instead, printer drivers call special GDI functions to carry out the output. Depending on the options selected by the user, those functions will route the output to a specific port, to a disk file, across a network connection, or to a temporary file for later output by Print Manager.

GDI contains functions a device driver can call to perform output. The driver does not need to know if output is being queued or written directly to the port. The following lists these functions, and provides a brief description of each.

Function	Description
CloseJob	Closes a print job, and enables it for printing.
DeleteJob	Deletes a open job, removing it from the print queue.
EndSpoolPage	Marks the end of a spooled page.
OpenJob	Opens a print job, returning a handle that the driver can use to write output to the job.
StartSpoolPage	Marks the start of a spooled page. Print Manager manages print jobs by printing one spooled page at a time.
WriteDialog	Displays a dialog box directing the user to carry out some action to permit printing to continue.
WriteSpool	Writes data to an open print job.

4.1.25 The Brute Functions

The brute functions is a set of graphics-support functions that a printer driver can use to carry out certain graphics operations. The brute functions permit a printer driver to use the resources of GDI rather than provide its own support to complete some graphics operations.

Brute functions primarily provide support for memory bitmaps. On each call to a printer-driver function, the driver checks the **PDEVICE** structure and determines if it represents a memory bitmap. If such cases, the driver calls a corresponding brute function, passing on all the parameters, to carry out the graphics operation.

For banding drivers, the driver calls the brute functions for the main output device as well as for the memory bitmaps.

4.1.25.1 Brute-Information Functions

The following brute functions take the same parameters and return values as the corresponding driver functions and actually call the display driver to manipulate monochrome memory bitmaps:

- **dmRealizeObject**
- **dmEnumDFonts**
- **dmEnumObject**
- **dmColorInfo**

For graphics, most raster drivers call the **dmColorInfo** function to map colors. Display drivers then add together the three color components (R, G, and B). If the weighted, average color value of these components is equal to or greater than 128, then the color maps to white. Otherwise, it maps to black.

The IBM Color Printer driver does the same thing in monochrome mode. However, in color mode, each color is compared individually to the 128 value. Hewlett-Packard printers that use the PCL driver are only adjustable for text.

Notice that the following colors all map to white on EGA, VGA, and 8514/a displays:

- Light grey
- Green
- Yellow
- Magenta
- Cyan

4.1.25.2 Brute-Output Functions

Nonbanding drivers (such as PostScript or a plotter driver) use the brute functions to support memory bitmaps. Whenever a pointer to the driver's **PDEVICE** structure is passed to the driver, GDI may substitute a pointer to a **BITMAP** structure. The driver can differentiate between the two cases because the first member (**bmType**) of a **BITMAP** structure must be zero, whereas the first member (often called **epType**) of a **PDEVICE** structure must be nonzero. If this first member is zero, the driver simply passes the same arguments through to the corresponding brute function and returns its return value.

Banding drivers operate by using a memory bitmap to simulate the display surface. Therefore, a banding driver calls the brute function with a pointer to a **BITMAP** structure that defines the band bitmap. The driver also has to translate some coordinate parameters from device coordinates to bitmap coordinates since, in general, there will be many bands in different positions on a page of output.

The brute functions, however, always use coordinates relative to the bitmap, that is, (0,0) to (**bmWidth**, **bmHeight**).

Therefore, many output functions may take the following form:

```
Function(LPPDEVICE lpDevice, ... )
{
    if (!lpDevice-> epType)
      {
        /* output to memory bitmap */
        return dmFunction(lpDevice, ... );
      }
    /* if this is a nonbanding driver, perform
     * device-specific output. Otherwise, for a
     * banding driver,
     */

    /* transform coordinates according to band position
     */
    somerandomxcoordinate -= lpDevice-> xBandPosition;
    somerandomycoordinate -= lpDevice-> yBandPosition;

    /* assume that a BITMAP structure is stored somewhere in
     * the PDEVICE for the band bitmap
     */
    lpDevice = (LPPDEVICE)&lpDevice-> epBandBmpHdr;

    return dmFunction(lpDevice, ... );
}
```

The brute functions operate by calling the corresponding display driver function to manipulate a memory bitmap. Therefore, they have exactly the same parameters as the corresponding driver functions, with the exception that the *lpDevice* parameters are always assumed to point to **BITMAP** structures.

Since driver capabilities and bitmap formats vary from display to display, the printer driver should use the brute functions only for monochrome (not color) bitmaps. Also, since the scan line and polyline support is required for all display drivers, the printer driver can assume that this support is in the brute functions.

The following brute functions are available for output:

- **dmBitBlt**
- **dmOutput**
- **dmPixel**
- **dmStrBlt**
- **dmScanLR**

There is currently no **dmExtTextOut** function; the driver calls the **ExtTextOut** function.

4.1.25.3 Color-Library Functions

Although the dot-matrix (brute) library functions in GDI (such as **dmBitBlt** and **dmOutput**) only support monochrome printers, a color printer driver can call corresponding functions in the the color library which do implement color. The color library supports all the dot-matrix (**dm**) functions except **dmTranspose**. The **dmTranspose** function does not depend on color format. The arguments and return values of these functions are the same as those for the GDI monochrome versions of these functions.

The library implements color using a 3-plane RGB (Red, Green, Blue) banding bitmap, which is converted to CMYK (Cyan, Magenta, Yellow, Black) when the bitmap is sent to the printer. If a printer requires a different format, the color library must be modified.

Both **dmBitBlt** and **dmOutput** compile short, efficient functions into the stack segment and then call them to perform the actual operation. In protected mode, this requires creating a code segment alias for the stack segment. A selector must be allocated for these two functions to operate. It is stored in the global variable **ScratchSelector**, which is external to the library and which must be supplied by the driver. In the sample IBM Color Printer driver, the selector is allocated and freed in **Enable** and **Disable**, respectively.

4.1.26 GDI Priority Queues

The GDI library provides the priority-queue data type that is used with device-specific fonts to sort output strings into the correct order on the page. Priority queues are accessed through a two-byte value, known as the key. Each key can also have two bytes of associates information, called a tag.

The following lists the priority-queue functions, and provides a brief description of each.

Function	Description
CreatePQ	Creates a priority queue.
DeletePQ	Deletes a priority queue.
InsertPQ	Inserts a key in a priority queue.
MinPQ	Returns the tag having the smallest key in the queue.
SizePQ	Sets the size of the priority queue.
ExtractPQ	Extracts a key from a priority queue.

4.1.27 Stub Functions

Because printer drivers are dynamic-link libraries that GDI loads using the **LoadLibrary** function, they must also export the **WEP** function. This function indicates that Windows is shutting down or the printer driver is being unloaded from the system.

Printer drivers must include the **SetAttribute** and **DeviceBitmap** functions.

4.1.28 Installing Over Previous Versions

The Windows 3.1 Setup program will automatically update any installed printer drivers with new drivers provided with Windows 3.1. As part of this process, Setup will replace:

```
generic printer name=driver, port
```

in the [Devices] and [PrinterPorts] sections with:

```
specific model name=new driver, port
```

Setup will not update the [*DriverName*, *Port*] section. This will be done by the individual driver when Setup calls the **DevInstall** function.

4.2 Function Reference

The following is an alphabetical listing of graphics functions that are specific to printer drivers. For a complete description of the graphics functions that are common to other graphics drivers, see Chapter 10, "Graphics-Driver Functions."

AdvancedSetUpDialog

WORD AdvancedSetUpDialog(*hWnd*, *hDriver*, *lpDevModeIn*, *lpDevModeOut*)
HWND *hWnd*;
HANDLE *hDriver*;
LPDEVMODE *lpDevModeIn*;
LPDEVMODE *lpDevModeOut*;

The **AdvancedSetUpDialog** function displays a dialog box with which the user sets advanced print-job options. Applications call this function indirectly

whenever the user chooses the Options button in the Print Setup dialog box. **AdvancedSetUpDialog** lets the user specify print-job options that the driver supports, such as color, duplex printing, and text quality, but that are not available through the Print Setup dialog boxes.

Printer drivers that do not export **AdvancedSetUpDialog** must export the **ExtDeviceMode** function.

Parameters

hWnd
Identifies the parent window. **AdvancedSetUpDialog** uses this handle as the parent window handle when it creates the dialog box.

hDriver
Identifies the module instance of the device driver. **AdvancedSetUpDialog** uses this handle as the module instance handle when it creates the dialog box.

lpDevModeIn
Points to a **DEVMODE** structure specifying the initial values for the advanced printer options in the dialog box. If the *lpDevModeIn* parameter is NULL or the **dmSpecVersion** member in this **DEVMODE** structure is less than 0x0300, **AdvancedSetUpDialog** must return –1 without displaying the dialog box.

lpDevModeOut
Points to a **DEVMODE** structure that receives the final values for the advanced printer options. The final values include any input from the user. If the user cancels the dialog box, **AdvancedSetUpDialog** must at least copy the initial values (supplied in the *lpDevModeIn* parameter) to the *lpDevModeOut* parameter. The *lpDevModeOut* parameter must specify all print settings, not just those changed by the user.

Return Value

The return value is IDOK if the user chose the OK button to exit the dialog box, or IDCANCEL if the user chose the Cancel button. In either case, the structure pointed to by *lpDevModeOut* contains final values for the advanced pint-job options. If an error occurs, the function returns –1.

Comments

The export ordinal for this function is 93.

AdvancedSetUpDialog creates an application-modal dialog box using the **DialogBox** function (USER.240). The dialog box must provide options for the user to set the various advanced operating modes of the device. If **DialogBox** returns IDOK, **AdvancedSetUpDialog** copies the complete print-job information including all the user's input to *lpDevModeOut*. Otherwise, it must copy the complete contents of *lpDevModeIn* to *lpDevModeOut*.

Although **AdvancedSetUpDialog** and **ExtDeviceMode** may display similar dialog boxes, these functions are not identical. In particular, **AdvancedSetUpDialog** must not update or modify the current environment or WIN.INI settings.

See Also **ExtDeviceMode**

DeviceCapabilities

#include <drivinit.h>

DWORD DeviceCapabilities(*lpDeviceName*, *lpPort*, *nIndex*, *lpOutput*, *lpDevMode*)

```
LPSTR lpDeviceName;     /* pointer to device-name string          */
LPSTR lpPort;           /* pointer to port-name string            */
WORD nIndex;            /* device capability to query             */
LPSTR lpOutput;         /* pointer to the output                  */
LPDEVMODE lpDevMode;    /* pointer to structure with device data  */
```

The **DeviceCapabilities** function retrieves the capabilities of the printer device driver and is recommended for all printer drivers.

Parameters

lpDeviceName
Points to a null-terminated string that contains the name of the printer device, such as Hewlett-Packard LaserJet that uses PCL.

lpPort
Points to a null-terminated string that contains the name of the port to which the device is connected, such as LPT1.

nIndex
Specifies the capabilities to query. It can be any one of the following values.

Value	Meaning
DC_BINNAMES	Copies an array containing a list of the names of the paper bins. This array is in the form **char** *PaperNames*[*cBinMax*][*cchBinName*] where *cchBinName* is 24. If the *lpszOutput* parameter is NULL, the return value is the number of bin entries required. Otherwise, the return value is the number of bins copied. To work properly with the common dialog box library (COMMDLG), a printer driver for Windows 3.1 must support the DC_BINNAMES index.

Value	Meaning
DC_BINS	Retrieves a list of available bins. The function copies the list to *lpOutput* as a **WORD** array. If *lpOutput* is NULL, the function returns the number of supported bins to allow the application the opportunity to allocate a buffer with the correct size. See the description of the **dmDefaultSource** member of the **DEVMODE** structure for information on these values. An application can determine the name of device-specific bins by using the ENUMPAPERBINS escape.
DC_COPIES	Returns the maximum number of copies the device can produce.
DC_DRIVER	Returns the printer-driver version number.
DC_DUPLEX	Returns the level of duplex support. The function returns 1 if the printer is capable of duplex printing. Otherwise, the return value is zero.
DC_ENUMRESOLUTIONNAMES	Retrieves a list of resolution names supported by the model. The application should allocate a buffer to hold one or more arrays each containing CCHPAPERNAME (64) bytes. If *lpOutput* is NULL, the function returns the number of resolutions supported by the model. If *lpOutput* is not NULL, the buffer is filled in.
DC_ENUMRESOLUTIONS	Returns a list of available resolutions. If *lpOutput* is NULL, the function returns the number of available resolution configurations. Resolutions are represented by pairs of **LONG** integers representing the horizontal and vertical resolutions.
DC_EXTRA	Returns the number of bytes required for the device-specific portion of the **DEVMODE** structure for the printer driver.
DC_FIELDS	Returns the **dmFields** member of the printer driver's **DEVMODE** data structure. The **dmFields** member indicates which member in the device-independent portion of the structure are supported by the printer driver.

Value	Meaning
DC_FILEDEPENDENCIES	Returns a list of files which also need to be loaded when a driver is installed. If *lpOutput* is NULL, the function returns the number of files. If *lpOutput* is not NULL, it is a pointer to an array of filenames. Each element in the array is exactly 64 characters long.
DC_MAXEXTENT	Returns a **POINT** structure containing the maximum paper size that the **dmPaperLength** and **dmPaperWidth** members of the printer driver's **DEVMODE** structure can specify.
DC_MINEXTENT	Returns a **POINT** structure containing the minimum paper size that the **dmPaperLength** and **dmPaperWidth** members of the printer driver's **DEVMODE** structure can specify.
DC_ORIENTATION	Retrieves the relationship between portrait and landscape orientations in terms of the number of degrees that portrait orientation is to be rotated counterclock-wise to get landscape orientation. It can be one of the following values.

Value	Meaning
0	No landscape orientation.
90	Portrait is rotated 90 degrees to produce landscapes. (For example, PCL.)
270	Portrait is rotated 270 degrees to produce landscape. (For example, dot-matrix printers.)

DC_PAPERNAMES	Retrieves a list of the nonstandard paper names supported by the model. The application should allocate a buffer to hold one or more arrays each containing CCHPAPERNAME (64) bytes. If *lpOutput* is NULL, the function returns the number of non-standard paper sizes supported by the model. If *lpOutput* is not NULL, the buffer is filled in.

DC_PAPERS

Retrieves a list of supported paper sizes. The function copies the list to *lpOutput* as a **WORD** array and returns the number of entries in the array. If *lpOutput* is NULL, the function returns the number of supported paper sizes to allow the application the opportunity to allocate a buffer with the correct size. See the description of the **dmPaperSize** member of the **DEVMODE** data structure for information on these values.

DC_PAPERSIZE

Copies the dimensions of supported paper sizes in tenths of a millimeter to an array of **POINT** structures pointed to by *lpOutput*. This allows an application to obtain information about nonstandard paper sizes.

DC_SIZE

Returns the **dmSize** member of the printer driver's **DEVMODE** data structure.

DC_TRUETYPE

Retrieves the driver's capabilities with regard to printing TrueType fonts. The return value can be one or more of the following capability flags.

Value	Meaning
DCTT_BITMAP	(0x0000001L) Device is capable of printing TrueType fonts as graphics.
DCTT_DOWNLOAD	(0x0000002L) Device is capable of downloading TrueType fonts.
DCTT_SUBDEV	(0x0000004L) Device is capable of substituting device fonts for TrueType.

In this case, the *lpOutput* parameter is not used, and should be NULL.

DC_VERSION

Returns the specification version to which the printer driver conforms.

lpOutput
Points to an array of bytes. The actual format of the array depends on the setting of the *nIndex* parameter. If set to zero, **DeviceCapabilities** returns the number of bytes required for the output data.

lpDevMode
Points to a **DEVMODE** structure. If the *lpDevMode* parameter is NULL, this function retrieves the current default initialization values for the specified printer driver. Otherwise, the function retrieves the values contained in the structure to which *lpDevMode* points.

The **DEVMODE** structure has the following form:

```
typedef struct _devicemode {      /* dm */
    char  dmDeviceName[CCHDEVICENAME];
    WORD  dmSpecVersion;
    WORD  dmDriverVersion;
    WORD  dmSize;
    WORD  dmDriverExtra;
    DWORD dmFields;
    short dmOrientation;
    short dmPaperSize;
    short dmPaperLength;
    short dmPaperWidth;
    short dmScale;
    short dmCopies;
    short dmDefaultSource;
    short dmPrintQuality;
    short dmColor;
    short dmDuplex;
    short dmYResolution;
    short dmTTOption;
} DEVMODE;
```

Return Value

The return value depends on the setting of the *nIndex* parameter if successful. Otherwise, the return value is –1 if the function fails.

Comments

The export ordinal for this function is 91.

See Also

ENUMPAPERBINS, **ExtDeviceMode**

DevInstall

WORD DevInstall(*hWnd*, *lpModelName*, *OldPort*, *NewPort*)
HWND *hWnd*;
LPSTR *lpModelName*;
LPSTR *OldPort*;
LPSTR *NewPort*;

The **DevInstall** function changes port connections, and installs and removes printers. Control Panel calls this function whenever the user switches the port for a printer model.

Parameters

hWnd
Identifies the parent window to use for any dialog boxes the function creates.

lpModelName
Points to a null-terminated string specifying the name of the current printer model.

OldPort
Points to a null-terminated string specifying the name of the port being changed. If the *OldPort* parameter is NULL, the function installs the new printer model.

NewPort
Points to a null-terminated string specifying the name of the port to be changed to. If the *NewPort* parameter is NULL, the function removes printer model.

Return Value

The return value is one of the following values.

Value	Meaning
1	Succeeded
0	Doesn't support this function
–1	Failed for unknown reason

Comments

When the function changes a port, it must change the port in the [ModelName, Port] section of the WIN.INI file. Also, the function should check for a font section with valid entries for the old port. If there are fonts, the function should warn the user with a message informing them that there are printer fonts installed for the old port. The user can install them using the Fonts button from the Printer Setup dialog box.

When installing a new printer, the function must scan the WIN.INI file for a [DriverName, NewPort] section. If one is found and *ModelName* matches the printer index listed there, the function creates a new section [ModelName,

NewPort] and moves all nonfont-related information from the [DriverName, Newport] section to this new [ModelName, NewPort] section. After this operation, the [DriverName, NewPort] section contains only the printer fonts listing. If the function doesn't find a [DeviceName, NewPort] section, no action is required.

When removing a printer, the function must remove the [ModelName, OldPort] section including all settings there, but it must not remove the [DriverName, OldPort] section. Other models may be using fonts installed on this port.

4.3 Printer Environment Function Reference

The following is an alphabetical listing of printer-environment functions.

GetEnvironment

int GetEnvironment(*lpszPort*, *lpvEnviron*, *cbMaxCopy*)
LPCSTR *lpszPort*; /* address of port name */
void FAR* *lpvEnviron*; /* address of buffer for environment */
UINT *cbMaxCopy*; /* maximum number of bytes to copy */

The **GetEnvironment** function retrieves the current environment that is associated with the device attached to the specified system port and copies it into the specified **DEVMODE** structure. The environment, maintained by graphics device interface (GDI), contains binary data that GDI uses whenever a device context is created for the device on the given port.

Parameters

lpszPort
Points to the null-terminated string that specifies the name of the desired port.

lpvEnviron
Points to the **DEVMODE** structure that will receive the environment. The **DEVMODE** structure has the following form:

```
typedef struct _devicemode {      /* dm */
    char  dmDeviceName[CCHDEVICENAME];
    WORD  dmSpecVersion;
    WORD  dmDriverVersion;
    WORD  dmSize;
    WORD  dmDriverExtra;
    DWORD dmFields;
    short dmOrientation;
    short dmPaperSize;
    short dmPaperLength;
```

```
    short dmPaperWidth;
    short dmScale;
    short dmCopies;
    short dmDefaultSource;
    short dmPrintQuality;
    short dmColor;
    short dmDuplex;
    short dmYResolution;
    short dmTTOption;
} DEVMODE;
```

cbMaxCopy
Specifies the maximum number of bytes to be copied to the structure.

Return Value

The return value specifies the number of bytes copied to the **DEVMODE** structure pointed to by the *lpvEnviron* parameter, if the function is successful. If *lpvEnviron* is NULL, the return value is the number of bytes required to hold the environment. It is zero if the environment cannot be found.

Comments

The **GetEnvironment** function is used by drivers only.

The **GetEnvironment** function fails if there is no environment for the given port.

An application can call this function with the *lpvEnviron* parameter set to NULL to determine the size of the structure required to hold the environment. It can then allocate the required number of bytes and call the **GetEnvironment** function a second time to retrieve the environment. The first member in the block pointed to by the *lpEnviron* parameter should be an atom that the printer driver has added to the global atom table.

See Also

SetEnvironment

SetEnvironment

int SetEnvironment(*lpszPort*, *lpvEnviron*, *cbMaxCopy*)

```
LPCSTR lpszPort;          /* address of port name                    */
const void FAR* lpvEnviron;  /* address of buffer for new environment  */
UINT cbMaxCopy;           /* maximum number of bytes to copy         */
```

The **SetEnvironment** function copies the contents of the specified buffer into the environment associated with the device attached to the specified system port. The environment, maintained by GDI, contains binary data used by GDI whenever a device context is created for the device on the given port.

Parameters

lpszPort
Points to a null-terminated string that specifies the name of the port.

lpvEnviron
Points to the buffer that contains the new environment. This buffer is in the form of a **DEVMODE** structure. The **DEVMODE** structure has the following form:

```
typedef struct _devicemode {      /* dm */
    char  dmDeviceName[CCHDEVICENAME];
    WORD  dmSpecVersion;
    WORD  dmDriverVersion;
    WORD  dmSize;
    WORD  dmDriverExtra;
    DWORD dmFields;
    short dmOrientation;
    short dmPaperSize;
    short dmPaperLength;
    short dmPaperWidth;
    short dmScale;
    short dmCopies;
    short dmDefaultSource;
    short dmPrintQuality;
    short dmColor;
    short dmDuplex;
    short dmYResolution;
    short dmTTOption;
} DEVMODE;
```

cbMaxCopy
Specifies the maximum number of bytes to copy to the buffer.

Return Value

The return value is the number of bytes copied to the environment, if the function is successful. It is zero if there is an error or 1 if the environment is deleted.

Comments

The **SetEnvironment** function is used by device drivers.

The **SetEnvironment** function deletes any existing environment. If there is no environment for the given port, **SetEnvironment** creates one. If the *cbMaxCopy* parameter is zero, the existing environment is deleted and not replaced.

The first member of the **DEVMODE** structure pointed to by the *lpvEnviron* parameter must be the same as that passed in the *lpDestDevType* parameter of the **Enable** function. If *lpszPort* specifies a null port, the **dmDeviceName** member is used to locate the environment. The first member in the block pointed to by the *lpEnviron* parameter should be an atom that the printer driver has added to the global atom table.

See Also

GetEnvironment

4.4 Priority-Queue Function Reference

The following is an alphabetical listing of the priority-queue functions.

CreatePQ

HPQ CreatePQ(*size*)
int *size*;

The **CreatePQ** function creates a priority queue.

Parameters

size
Specifies the maximum number of items to be inserted into this priority queue.

Return Value

The return value is a handle to the priority queue if the function is successful. Otherwise, it is zero.

See Also

DeletePQ

DeletePQ

int DeletePQ(*hPQ*)
HPQ *hPQ*;

The **DeletePQ** function deletes a priority queue.

Parameter

hPQ
Identifies a priority queue.

Return Value

The return value is TRUE if the queue is deleted. Otherwise, it is −1.

ExtractPQ

int ExtractPQ(*hPQ*)
HPQ *hPQ*;

The **ExtractPQ** function returns the tag associated with the key having the smallest value in the priority queue and removes the key from the queue.

Parameter

hPQ
Identifies a priority queue.

Return Value

The return value is a tag associated with the key in the priority queue.

InsertPQ

BOOL InsertPQ(*hPQ*, *tag*, *key*)
HPQ *hPQ*;
int *tag*;
int *key*;

The **InsertPQ** function inserts the key and its associated tag into the priority queue.

Parameters

hPQ
Identifies a priority queue.

tag
Specifies a tag associated with the key.

key
Specifies a key.

Return Value

The return value is TRUE if the insertion is successful. Otherwise, it is –1.

MinPQ

int MinPQ(*hPQ*)
HPQ *hPQ*;

The **MinPQ** function returns the tag associated with the key having the smallest value in the priority queue, without removing this element from the queue.

Parameter

hPQ
Identifies a priority queue.

Return Value

The return value is the tag associated with the key in the priority queue.

See Also

ExtractPQ

SizePQ

int SizePQ(*hPQ*, *sizechange*)
HPQ *hPQ*;
int *sizechange*;

The **SizePQ** function increases or decreases the size of the priority queue.

Parameters

hPQ
Identifies a priority queue.

sizechange
Specifies the number of entries to be added or removed.

Return Value

The return value is the number of entries that can be accommodated by the resized priority queue. The return value is –1 if the resulting size is smaller than the actual number of elements in the priority queue.

4.5 Print Manager Function Reference

The following is an alphabetical listing of Print Manager functions.

CloseJob

int FAR PASCAL CloseJob(*hJob***)**
HANDLE ***hJob*;**

The **CloseJob** function closes the print job identified by the given handle.

Parameters

hJob
Identifies the print job to close. The handle must have been previously opened using the **OpenJob** function.

Return Value

The return value is positive if the function is successful. Otherwise, it is one of the following error values.

Value	Meaning
SP_ERROR (–1)	A general error condition or general error in banding occurred.
SP_APPABORT (–2)	The job was stopped because the application's callback function returned FALSE (0).
SP_USERABORT (–3)	The user stopped the print job by choosing the Delete button from Print Manager.
SP_OUTOFDISK (–4)	A lack of disk space caused the job to stop. There is not enough disk space to create or extend the Print Manager temporary file.
SP_OUTOFMEMORY (–5)	A lack of memory caused the job to stop.

See Also

OpenJob

DeleteJob

int FAR PASCAL DeleteJob(*hJob*, *wDummy*)
HANDLE *hJob*;
WORD *wDummy*;

The **DeleteJob** function deletes the given print job from the printing queue. A driver should call this function if it detects an error condition, or is asked to quit a job by the application.

Parameters

hJob
Identifies the print job to delete. The handle must have been previously opened using the **OpenJob** function.

wDummy
Reserved; must be set to zero.

Return Value

The return value is positive if the function is successful. Otherwise, it is one of the following error values.

Value	Meaning
SP_ERROR (–1)	A general error condition or general error in banding occurred.
SP_APPABORT (–2)	The job was stopped because the application's callback function returned FALSE (0).
SP_USERABORT (–3)	The user stopped the print job by choosing the Delete button from Print Manager.
SP_OUTOFDISK (–4)	A lack of disk space caused the job to stop. There is not enough disk space to create or extend the Print Manager temporary file.
SP_OUTOFMEMORY (–5)	A lack of memory caused the job to stop.

See Also

OpenJob

EndSpoolPage

int FAR PASCAL EndSpoolPage(*hJob*)
HANDLE *hJob*;

The **EndSpoolPage** function marks the end of a spooled page. A driver uses this function, in conjunction with the **StartSpoolPage** function, to divide printer output into pages. Each page is stored in a temporary file on the machine's hard disk

when Print Manager is running. Dividing a print job into pages allows Print Manager to begin printing one page while the driver is still generating output on subsequent pages. A Print Manager page does not need to correspond to a physical page of printed output; the division is the driver's decision.

Parameters

hJob
Identifies the print job. The handle must have been previously opened using the **OpenJob** function.

Return Value

The return value is positive if the function is successful. Otherwise, it is one of the following error values.

Value	Meaning
SP_ERROR (–1)	A general error condition or general error in banding occurred.
SP_APPABORT (–2)	The job was stopped because the application's callback function returned FALSE (0).
SP_USERABORT (–3)	The user stopped the print job by choosing the Delete button from Print Manager.
SP_OUTOFDISK (–4)	A lack of disk space caused the job to stop. There is not enough disk space to create or extend the Print Manager temporary file.
SP_OUTOFMEMORY (–5)	A lack of memory caused the job to stop.

Comments

When Print Manager is not running, page division is not very important because temporary files are not involved. However, starting and ending at least one Print Manager page is still required.

Calls to **StartSpoolPage** and **EndSpoolPage** can occur at any point during the output. Some drivers use one spool page per physical page. Others use one page for the whole job. The printing of a particular page by the Print Manager application does not begin until it receives the corresponding **EndSpoolPage** function.

A driver can perform output at any point between these two calls. When **EndSpoolPage** is called and Print Manager is loaded, the page's temporary file is submitted to Windows Print Manager.

See Also

OpenJob, StartSpoolPage

OpenJob

HANDLE FAR PASCAL OpenJob(*lpOutput*, *lpTitle*, *hdc*)
LPSTR *lpOutput*;
LPSTR *lpTitle*;
HDC *hdc*;

The **OpenJob** function creates a print job and returns a handle identifying the job. A driver uses the handle in subsequent functions to write output to the print job as well as control the job.

Parameters

lpOutput
Points to a null-terminated string specifying the port or file to receive the output. A driver typically supplies the same filename as pointed to by the *lpOutputFile* parameter when GDI calls the driver's **Enable** function.

lpTitle
Points to a null-terminated string specifying the title of the document to print. This parameter must be supplied by the application when it calls the STARTDOC escape. This title appears in the Print Manager display.

hdc
Identifies the application's device context. This parameter must be supplied by the application when it calls the STARTDOC escape.

Return Value

The return value is a handle identifying the print job if the function is successful. Otherwise, it is one of the following error values.

Value	Meaning
SP_ERROR (–1)	A general error condition or general error in banding occurred.
SP_APPABORT (–2)	The job was stopped because the application's callback function returned FALSE (0).
SP_USERABORT (–3)	The user stopped the print job by choosing the Delete button from Print Manager.
SP_OUTOFDISK (–4)	A lack of disk space caused the job to stop. There is not enough disk space to create or extend the Print Manager temporary file.
SP_OUTOFMEMORY (–5)	A lack of memory caused the job to stop.

See Also

CloseJob, **Enable**, STARTDOC

StartSpoolPage

int FAR PASCAL StartSpoolPage(*hJob*)
HANDLE *hJob*;

The **StartSpoolPage** function marks the start of a spooled page. A driver uses this function, in conjunction with the **EndSpoolPage** function, to divide printer output into pages. Each page is stored in a temporary file on the machine's hard disk when Print Manager is running. Dividing a print job into pages allows Print Manager to begin printing one page while the driver is still generating output on subsequent pages. A Print Manager page does not need to correspond to a physical page of printed output; the division is the driver's decision.

Parameters

hJob
Identifies the print job. The handle must have been previously opened using the **OpenJob** function.

Return Value

The return value is positive if the function is successful. Otherwise, it is one of the following error values.

Value	Meaning
SP_ERROR (–1)	A general error condition or general error in banding occurred.
SP_APPABORT (–2)	The job was stopped because the application's callback function returned FALSE (0).
SP_USERABORT (–3)	The user stopped the print job by choosing the Delete button from Print Manager.
SP_OUTOFDISK (–4)	A lack of disk space caused the job to stop. There is not enough disk space to create or extend the Print Manager temporary file.
SP_OUTOFMEMORY (–5)	A lack of memory caused the job to stop.

Comments

When Print Manager is not running, page division is not very important because temporary files are not involved. However, starting and ending at least one Print Manager page is still required.

Calls to **StartSpoolPage** and **EndSpoolPage** can occur at any point during the output. Some drivers use one spool page per physical page. Others use one page for the whole job. The printing of a particular page by the Print Manager application does not begin until it receives the corresponding **EndSpoolPage** function.

A driver can perform output at any point between these two calls. When **EndSpoolPage** is called and Print Manager is loaded, the page's temporary file is submitted to Windows Print Manager.

See Also **EndSpoolPage, OpenJob**

WriteDialog

int FAR PASCAL WriteDialog(*hJob*, *lpMsg*, *cch*)
HANDLE *hJob*;
LPSTR *lpMsg*;
WORD *cch*;

The **WriteDialog** function displays a message box containing the specified message. A driver uses this function to inform the user of a possible printing problem. For example, a driver for a printer using manual-paper loading can call **WriteDialog** to ask the user to place a new sheet in the printer. The print job will not continue printing until the user chooses the OK button in the message box. The user may also choose a Cancel button to cancel the print job.

Parameters

hJob
Identifies the print job. The handle must have been previously opened using the **OpenJob** function.

lpMsg
Points to a null-terminated string containing the message to be displayed.

cch
Specifies the number of bytes in the message pointed to by the *lpMsg* parameter.

Return Value The return value is IDOK if the function is successful. Otherwise, it is IDCANCEL.

See Also **OpenJob**

WriteSpool

int FAR PASCAL WriteSpool(*hJob*, *lpData*, *cch*)
HANDLE *hJob*;
LPSTR *lpData*;
WORD *cch*;

The **WriteSpool** function writes printer output to the port or file associated with the print job. A driver must call this function after calling **StartSpoolPage** and before calling **EndSpoolJob**.

Parameters

hJob
Identifies the print job. The handle must have been previously opened using the **OpenJob** function.

lpData
Points to the device-dependent data to write.

cch
Specifies the number of bytes to write.

Return Value

The return value is positive if the function is successful. Otherwise, it is one of the following error values.

Value	Meaning
SP_ERROR (–1)	A general error condition or general error in banding occurred.
SP_APPABORT (–2)	The job was stopped because the application's callback function returned FALSE (0).
SP_USERABORT (–3)	The user stopped the print job by choosing the Delete button from Print Manager.
SP_OUTOFDISK (–4)	A lack of disk space caused the job to stop. There is not enough disk space to create or extend the Print Manager temporary file.
SP_OUTOFMEMORY (–5)	A lack of memory caused the job to stop.

See Also

EndSpoolPage, **OpenJob**, **StartSpoolPage**

4.6 TrueType Structure Reference

The following is an alphabetical listing of the structures that are specific to TrueType support for printer drivers.

BITMAPMETRICS

```
typedef struct _BITMAPMETRICS {
    SIZEL sizlExtent;
    POINTFX pfxOrigin;
    POINTFX pfxCharInc;
} BITMAPETRICS;
```

The **BITMAPMETRICS** structure defines the character cell that corresponds to a given glyph bitmap. The structure gives the width and height of the cell, the position of the bitmap relative to the origin of the cell, and the horizontal and vertical increments.

Members

sizlExtent
Specifies the width and height of the bitmap. Since scan lines are aligned on 32-bit boundaries, the byte width of a scan line is the number of pixels rounded to the next multiple of 32 and divided by 8.

pfxOrigin
Specifies the position of the upper-left corner of the bitmap relative to the character origin.

pfxCharInc
Specifies the increment to the next character. In this case, PFXCHARINC.X is the increment along the baseline.

Comments

The increment and origin of a character may be such that consecutive characters overlap. That is, the origin may be negative or the increment may be smaller that the actual width. The device driver is responsible for drawing overlapping glyphs without overwriting characters.

When calculating string widths, a device driver uses the increment as the width.

FIXED

```
typedef DWORD FIXED;
```

The **FIXED** type specifies a 32-bit, fixed point number. The type consists of 16-bit fields, representing an integer and a fraction as follows:

Bits	Meaning
0–15	Specifies the fractional part of the fixed point number. The fraction is always a positive value representing the numerator *n* of the expression *n*/65536.
16–31	Specifies the integer part of the fixed point number. The integer is a signed value in the range –32768 to 32767.

POINTFX

```
typedef struc _POINTFX {
    FIXED x;
    FIXED y;
} POINTFX;
```

The **POINTFX** structure specifies the *x*- and *y*-coordinates of a point. The coordinates are expressed as 32-bit fixed point numbers.

Members

x
Specifies a width or *x*-coordinate.

y
Specifies a height or *y*-coordinate.

SIZEL

```
typedef struct _SIZEL {
    DWORD x;
    DWORD y;
} SIZEL;
```

The **SIZEL** structure contains information about the size or location of a object specified as two 32-bit values.

Members

x
Specifies a width or *x*-coordinate.

y
Specifies a height or *y*-coordinate.

TTINFO

```
typedef struct tagTTINFO {
    WORD cbInfo;
    WORD fFlags;
} TTINFO;
```

The **TTINFO** structure contains information specifying whether TrueType is operating and whether TrueType fonts have been installed.

Members

cbInfo
Specifies the number of bytes in the structure.

fFlags
Specifies the state of TrueType and TrueType fonts. This field can be a combination of the following values:

Value	Meaning
0x0001	At least one TrueType has been installed.
0x0002	TrueType rasterizer is operating.

4.7 File-Format Reference

The following is an alphabetical listing of the file formats that are specific to printer drivers. For a complete description of the file formats that are common to other graphics drivers, see Chapter 13, "Font Files."

DDRV

DESCRIPTION 'DDRV *Description:AspectXY,LogPixelsX,LogPixelsY'*

The **DDRV** statement in the printer driver's module-definition file names the printer models supported by the printer driver, and specifies the aspect ratio and logical pixels-per-inch values of the printer.

Every printer driver must provide a **DDRV** statement. Control Panel uses the statement to display the name of the printer to the user and to choose matching screen fonts for the printer.

Parameters

Description
Specifies the name or names of the printer models supported by this driver. Although more than one name can be given, the names must not be separated by commas (,).

AspectXY
Specifies the aspect ratio for the printer. This parameter must be set to the same value as the driver's **dpAspectXY** member in the **GDIINFO** structure.

LogPixelsX
Specifies the horizontal pixels-per-inch for the printer. This parameter must be set to the same value as the driver's **dpLogPixelsX** member in the **GDIINFO** structure.

LogPixelsY
Specifies the vertical pixels-per-inch for the printer. This parameter must be set to the same value as the driver's **dpLogPixelsX** member in the **GDIINFO** structure.

Comments

The **DDRV** keyword must be capitalized. At least one character (typically a space) must follow the **DDRV** keyword; Control Panel always ignores this first character

Control Panel uses the *Description* (all characters up to the colon) parameter to create new settings for the [Devices] and [PrinterPorts] sections in the WIN.INI file. The settings have the following form:

Description=Port[,*Data*]

Examples

The following example shows a **DDRV** description for a printer driver supporting a single model:

```
DESCRIPTION 'DDRV PCL / HP LaserJet:100,300,300'
```

The following example shows support for multiple printer models:

```
DESCRIPTION 'DDRV Printer 1/Printer2:100,300,300'
```

Graphics-Driver Escapes

Chapter 11

The Microsoft Windows graphics escapes provide graphics support that is otherwise not available through graphics-device interface (GDI). Applications use graphics escapes to perform device-dependent operations that are not supported by GDI. Applications call the **Escape** function (GDI.38) to initiate an escape and GDI calls the device driver's **Control** function to complete the escape.

11.1 About the Graphics-Driver Escapes

Graphics drivers should support all escapes that are reasonable for a given device. In particular, printer drivers should be prepared applications that assume that the printer-setting escapes, such as STARTDOC and ENDDOC, are always available on high-end devices. All drivers should support the QUERYESCSUPPORT escape which identifies which graphics escapes the driver supports.

Display drivers should support the following escapes:

- QUERYESCSUPPORT
- GETCOLORTABLE
- SETCOLORTABLE

Printer drivers should support at least the following escapes:

- QUERYESCSUPPORT
- SETABORTPROC
- STARTDOC
- NEWFRAME
- ENDDOC
- ABORTDOC
- NEXTBAND

11.2 Obsolete Escapes

The EXTTEXTOUT and SELECTPAPERSOURCE escapes are now obsolete. EXTTEXTOUT has been replaced by the **ExtTextOut** function, and SELECTPAPERSOURCE has been superseded by the GETSETPAPERBINS escape.

11.3 Escape Reference

The following is an alphabetical listing of the graphics-driver escapes.

ABORTDOC

#define ABORTDOC 2

short Control(*lpDevice***, ABORTDOC, NULL, NULL)**
LPPDEVICE *lpDevice*;

The ABORTDOC escape cancels and deletes the job using the **DeleteJob** function.

The ABORTDOC escape should be used for printing operations that do not specify a stopping function with the SETABORTPROC escape, and to stop printing operations that have not yet reached their first NEWFRAME or NEXTBAND call.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

Return Value

The return value is positive if the escape is successful. Otherwise, it is negative.

Comments

GDI calls this escape when a banding error occurs. It is also called by an application when an error occurs or when the application wants to cancel the print job.

See Also

DeleteJob, ENDDOC, NEWFRAME, NEXTBAND, SETABORTPROC

BANDINFO

#define BANDINFO 24

short Control(*lpDevice***, BANDINFO,** *lpInData*, *lpOutData*)
LPPDEVICE *lpDevice*;
LPBANDINFOSTRUCT *lpInData*;
LPBANDINFOSTRUCT *lpOutData*;

The BANDINFO escape copies information about a device with banding capabilities to a structure pointed at by the *lpInData* parameter.

Banding is a property of an output device that allows a page of output to be stored in a metafile and divided into bands, each of which is sent to the device to create a complete page. Use banding with devices that cannot scroll backwards.

The information copied to the structure pointed at by *lpInData* includes a flag indicating whether or not there are graphics in the next band, a flag indicating whether or not there is text on the page, and a rectangle structure that contains a bounding rectangle for all graphics on the page.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a **BANDINFOSTRUCT** structure containing information about the graphics band. The **BANDINFOSTRUCT** structure has the following form:

```
typedef struct _BANDINFOSTRUCT {
    BOOL fGraphics;
    BOOL fText;
    RECT rcGraphics;
} BANDINFOSTRUCT;
```

lpOutData
Points to a **BANDINFOSTRUCT** structure containing information about the graphics band.

Return Value

The return value is 1 if the escape is successful. Otherwise, it is 0.

Comments

This escape should only be implemented for devices that use banding. It should be called immediately after each call to the NEXTBAND escape. If the *lpOutData* parameter is not NULL and graphics will be printed in the current band, the driver will set the **fGraphics** member in the output structure. If text will be printed, the **fText** member will be nonzero. The **rcGraphics** member is not used for output.

Therefore, on the first band, the driver would set the rectangle returned by NEXTBAND to the whole page. If it receives a BANDINFO escape, it will set the **fText** member and clear **fGraphics**.

On subsequent bands, it will band the page in small rectangles and handle only graphics calls. Additionally, if the application calls BANDINFO, clears the **fText** member and sets **fGraphics**.

The application can also optimize the banding process somewhat by describing the page with the structure passed by *lpInData*. The application sets the **fGraphics** member, if there are any graphics on the page, and the **fText** member if there is any text. If there are no graphics, the driver may be able to skip the graphics bands. The application should also set **rcGraphics** to the rectangle bounding all

nontext graphics on the page. The driver has the option of banding only the specified graphics rectangle rather than the whole page.

Vector fonts complicate the process somewhat. Since vector devices using banding generally cannot print vector fonts, these fonts are simulated using polylines or scan lines. Therefore, they appear to the driver to be graphics in the text band. Since vector fonts can appear anywhere on the page and require graphics banding support, the driver must band graphics on the whole page even if the **BANDINFOSTRUCT** passed by the application specifies otherwise.

If the application never calls BANDINFO, the driver can decide whether or not to band graphics by maintaining a flag that is set if any graphics calls are seen during the text band.

See Also NEXTBAND

BEGIN_PATH

#define BEGIN_PATH 4096

short Control(*lpDevice***, BEGIN_PATH, NULL, NULL)**
LPPDEVICE *lpDevice***;**

The BEGIN_PATH escape opens a path. A path is a connected sequence of primitives drawn in succession to form a single polyline or polygon. Paths enable applications to draw complex borders, filled shapes, and clipping areas by supplying a collection of other primitives that define the desired shape.

Printer escapes that support paths enable applications to render images on sophisticated devices such as PostScript printers without generating huge polygons to simulate them.

To draw a path, an application first issues the BEGIN_PATH escape. It then draws the primitives defining the border of the desired shape, and issues an END_PATH and EXT_DEVICE_CAPS escape. The END_PATH escape includes a parameter specifying how the path is to be rendered.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

Return Value

The return value is a short integer value specifying the current path nesting level. If the escape is successful, the number of BEGIN_PATH calls without a corresponding END_PATH call is the result. Otherwise, the result is zero.

Comments

You may open a path within another path. A path drawn within another path is treated exactly like a polygon (if the subpath is closed) or a polyline (if the subpath is open).

You may use the CLIP_TO_PATH escape to define a clipping area corresponding to the interior or exterior of the currently open path.

Device drivers that implement this escape must also implement the END_PATH and EXT_DEVICE_CAPS escapes and should also implement the SET_ARC_DIRECTION escape.

See Also

CLIP_TO_PATH, END_PATH, EXT_DEVICE_CAPS, SET_ARC_DIRECTION

CLIP_TO_PATH

#define CLIP_TO_PATH 4097

short Control(*lpDevice***, CLIP_TO_PATH,** *lpClipMode***, NULL)**
LPPDEVICE *lpDevice*;
LPINT *lpClipMode*;

The CLIP_TO_PATH escape defines a clipping area bounded by the currently open path. It enables the application to save and restore the current clipping area and to set up an inclusive or exclusive clipping area bounded by the currently open path.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpClipMode
Points to a 32-bit variable specifying the clipping mode. It may be one of the following values.

Value	Meaning
CLIP_SAVE(0)	Saves the current clipping area.
CLIP_RESTORE(1)	Restores the previous clipping area.
CLIP_INCLUSIVE(2)	Sets a clipping area in such a way that portions of primitives falling outside the interior bounded by the current path are clipped.

Value	Meaning
CLIP_EXCLUSIVE(3)	Sets a clipping area in such a way that portions of primitives falling inside the interior bounded by the current path should be clipped.

The high-order 16 bits specifies the interior mode. It may be either ALTERNATIVE or WINDING.

Return Value The return value is a nonzero value if the escape is successful. Otherwise, the return value is zero.

Comments Device drivers implementing the CLIP_TO_PATH escape must also implement the BEGIN_PATH, END_PATH, and EXT_DEVICE_CAPS escapes. Device drivers should also implement the SET_ARC_DIRECTION escape if they support elliptical arcs.

See Also BEGIN_PATH, END_PATH, EXT_DEVICE_CAPS, SET_ARC_DIRECTION

DRAFTMODE

#define DRAFTMODE 7

short Control(*lpDevice***, DRAFTMODE,** *lpDraftMode***, NULL)**
LPPDEVICE *lpDevice*;
LPINT *lpDraftMode*;

The DRAFTMODE escape turns draft mode off or on. Turning draft mode on instructs the device driver to print faster and with lower quality (if necessary). The draft mode can only be changed at page boundaries, for example, after a NEWFRAME escape.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpDraftMode
Points to a 16-bit variable containing a value specifying the draft mode. If it is 1, the escape turns on draft mode; if 0, the escape turns off draft mode.

Return Value The return value is positive if the escape is successful. Otherwise, it is negative.

Comments The default draft mode is off.

See Also NEWFRAME

DRAWPATTERNRECT

#define DRAWPATTERNRECT 25

short Control(*lpDevice***, DRAWPATTERNRECT,** *lpInData***, NULL)**
LPPDEVICE *lpDevice*;
LPPOINT *lpInData*;

The DRAWPATTERNRECT escape creates a pattern, gray scale, or solid black rectangle using the pattern or rule capabilities of PCL printers. With the Hewlett-Packard LaserJet IIP, this escape can also create a solid white rectangle. A gray scale is a gray pattern that contains a specific mixture of black and white pixels. A PCL printer is an HP LaserJet or LaserJet-compatible printer.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a **PATTERNRECT** structure containing information about the rectangle to create. The structure has the following form:

```
typedef struct tagPATTERNRECT {
    POINT prPosition;
    POINT prSize;
    WORD prStyle;
    WORD prPattern;
} PATTERNRECT;
```

Return Value

The return value is nonzero if the escape is successful. Otherwise, it is zero.

Comments

An application should use the QUERYESCSUPPORT escape to determine whether a device is capable of drawing patterns and rules before implementing this escape. If a printer is capable of outputting a white rule, the return value for QUERYESCSUPPORT is 2.

The effect of a white rule is to erase any text or other pattern rules already written in the specified area.

The driver sends all text and rules in the first band before any GDI bitmap graphics are sent. Therefore, it is not possible to erase bitmap graphics with white rules.

If an application uses the BANDINFO escape, the drive should send all patterns and rectangles specified by the DRAWPATTERNRECT escape in the first band.

Patterns and rules created with this escape may not be erased by placing opaque objects over them unless you have white rule capability. An application should use the function calls provided in GDI to obtain this effect.

See Also BANDINFO, QUERYESCSUPPORT

ENABLEDUPLEX

#define ENABLEDUPLEX 28

short Control(*lpDevice***, ENABLEDUPLEX,** *lpInData***, NULL)**
LPPDEVICE *lpDevice*;
LPWORD *lpInData*;

The ENABLEDUPLEX escape enables the duplex printing capability of a printer. A device that has duplex printing capability is able to print on both sides of the output medium.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a 16-bit variable that contains one of the following values.

Value	Meaning
0	Simplex
1	Duplex with vertical binding
2	Duplex with horizontal binding

Return Value The return value is 1 if the escape is successful. Otherwise, it is 0.

Comments An application should use the QUERYESCSUPPORT escape to determine whether or not an output device is capable of creating duplex output. If QUERYESCSUPPORT returns a nonzero value, the application should send the ENABLEDUPLEX escape even if simplex printing is desired. This

guarantees the overriding of any values set in the driver-specific dialog. If duplex printing is enabled and an uneven number of NEWFRAME escapes is sent to the driver prior to the ENDDOC escape, the driver will add one page eject before ending the print job.

See Also ENDDOC, NEWFRAME, QUERYESCSUPPORT

ENABLEPAIRKERNING

#define ENABLEPAIRKERNING 769

short Control(*lpDevice***, ENABLEPAIRKERNING,** *lpInData***,** *lpOutData***)**
LPPDEVICE *lpDevice*;
LPINT *lpInData*;
LPINT *lpOutData*;

The ENABLEPAIRKERNING escape enables or disables the driver's ability to automatically kern character pairs. When it is enabled, the driver automatically kerns those pairs of characters that are listed in the font's character-pair kerning table. The driver reflects this kerning both in the printer and in calls to the **GetTextExtent** (GDI.91) function.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a 16-bit variable that specifies whether to enable or disable automatic pair kerning. If it is 1, kerning is enabled; if 0, kerning is disabled.

lpOutData
Points to a 16-bit variable variable that receives the previous automatic pair-kerning member.

Return Value

The return value is 1 if the escape is successful. Otherwise, the return value is 0 if not successful, or if the escape is not implemented.

Comments

The default state of this capability is zero; that is, automatic character-pair kerning is disabled.

A driver does not have to support this escape just because it supplies the character-pair kerning table to the application through the GETPAIRKERNTABLE escape. When the GETPAIRKERNTABLE escape is supported but the ENABLEPAIRKERNING escape is not, it is the application's responsibility to properly space the kerned characters on the output device.

See Also GETPAIRKERNTABLE

ENABLERELATIVEWIDTHS

#define ENABLERELATIVEWIDTHS 768

short Control(*lpDevice***, ENABLERELATIVEWIDTHS,** *lpInData***,** *lpOutData***)**
LPPDEVICE *lpDevice*;
LPINT *lpInData*;
LPINT *lpOutData*;

The ENABLERELATIVEWIDTHS escape enables or disables relative character widths. When it is disabled (the default setting), each character's width can be expressed as an integer number of device units. This expression guarantees that the extent of a string will equal the sum of the extents of the characters that make up the string. Such behavior enables applications to build an extent table manually using one-character calls to the **GetTextExtent** (GDI.91) function. When it is enabled, the width of a string may or may not equal the sum of the widths of the characters in the string. Applications that enable this feature are expected to retrieve the font's extent table and compute relatively-scaled string widths themselves.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a 16-bit variable that specifies whether to enable or disable relative widths. If it is 1, relative widths are enabled; if 0, relative widths are disabled.

lpOutData
Points to a 16-bit variable that receives the previous relative character-width member.

Return Value The return value is 1 if the escape is successful. Otherwise, the return value is 0 if the escape is not successful, or if the escape is not implemented.

Comments

The default state of this capability is zero; that is, relative character widths are disabled.

Enabling this feature creates values that are specified as "font units" and accepted and returned by the escapes described in this chapter to be returned in the relative units of the font.

It is assumed that only linear scaling devices will be dealt with in a relative mode. Nonlinear scaling devices should not implement this escape.

END_PATH

#define END_PATH 4098

short Control(*lpDevice***, END_PATH,** *lpInfo***, NULL)**
LPPDEVICE *lpDevice***;**
LPPATH_INFO *lpInfo***;**

The END_PATH escape ends a path. A path is a connected sequence of primitives drawn in succession to form a single polyline or polygon. Paths enable applications to draw complex borders, filled shapes, and clipping areas by supplying a collection of other primitives defining the desired shape.

Printer escapes that support paths enable applications to render images on sophisticated devices such as PostScript printers without generating huge polygons to simulate them.

To draw a path, an application first issues the BEGIN_PATH escape. It then draws the primitives defining the border of the desired shape, and issues an END_PATH escape.

The END_PATH escape takes a pointer to a structure as a parameter, specifying the manner in which the path is to be rendered. The structure specifies whether or not the path is to be drawn and whether or not it is open or closed. Open paths define polylines, and closed paths define polygons that can be filled.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInfo
Points to a **PATH_INFO** structure. The **PATH_INFO** structure has the following form:

```
typedef struct tagPATH_INFO {
    short       RenderMode;
    BYTE        FillMode;
    BYTE        BkMode;
    LPEN        Pen;
    LBRUSH      Brush;
    DWORD       BkColor;
} PATH_INFO;
```

Return Value

This escape returns a short integer value specifying the current path nesting level. If the escape is successful, the number of BEGIN_PATH calls without a corresponding END_PATH call is the result. Otherwise, –1 is the result.

Comments

You may draw a path within another path. A path drawn within another path is treated exactly like a polygon (if the subpath is closed) or a polyline (if the subpath is open).

You may use the CLIP_TO_PATH escape to define a clipping area corresponding to the interior or exterior of the currently open path.

Device drivers that implement this escape must also implement the BEGIN_PATH and EXT_DEVICE_CAPS escapes and should also implement the SET_ARC_DIRECTION escape.

See Also

BEGIN_PATH, CLIP_TO_PATH, EXT_DEVICE_CAPS, SET_ARC_DIRECTION

ENDDOC

#define ENDDOC 11

short Control(*lpDevice***, ENDDOC, NULL, NULL)**
LPPDEVICE *lpDevice*;

The ENDDOC escape ends a print job that is started by a STARTDOC escape and that is to be ended in a standard way, instead of stopping the job in the middle of the process.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

Return Value

The return value is positive if the escape is successful. Otherwise, it is negative.

Comments

When a printing error occurs, the ENDDOC escape should not be used to terminate the printing operation.

See Also

STARTDOC

ENUMPAPERBINS

#define ENUMPAPERBINS 31

short Control(*lpDevice***, ENUMPAPERBINS,** *lpInData***,** *lpOutData***)**
LPPDEVICE *lpDevice*;
LPINT *lpInData*;
LPINT *lpOutData*;

The ENUMPAPERBINS escape retrieves attribute information about a specified number of paper bins. The GETSETPAPERBINS escape retrieves the number of bins available on a printer.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a 16-bit variable that specifies the number of bins for which information is to be retrieved.

lpOutData
Points to a **BINNAMES** structure to which information about the paper bins is copied. The size of the structure depends on the number of bins for which information was requested. The **BINNAMES** structure has the following form:

```
typedef struct tagBINNAMES {
    short BinList[CBINMAX];
    char PaperNames[CBINMAX][CCHBINNAME]
} BINNAMES;
```

Return Value

It is 1 if the escape is successful. Otherwise, it is 0 if the escape is not successful or not implemented.

See Also

GETSETPAPERBINS

ENUMPAPERMETRICS

#define ENUMPAPERMETRICS 34

short Control(*lpDevice***, ENUMPAPERMETRICS,** *lpInData***,** *lpOutData***)**
LPPDEVICE *lpDevice***;**
LPINT *lpInData***;**
LPRECT *lpOutData***;**

The ENUMPAPERMETRICS escape either retrieves the number of paper types supported by the driver, or fills an array of **RECT** structures with the dimensions of each paper type.

The **ExtDeviceMode** function achieves the same results.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to an 16-bit variable that specifies what action to take. If the variable is zero, the escape returns the number of paper types. If zero, the escape fills an array of **RECT** structures with paper dimensions.

lpOutData
Points to an array of **RECT** structures that receive the coordinates of the imageable area of the page. The top-left corner of the rectangle specifies the page margins, and the bottom-right corner specifies the sum of the page margins and the width and height of the imageable area. The units are device coordinates. The orientation returned is always portrait.

Return Value

The return value is positive, if successful. Otherwise, it is zero if the escape is not implemented, and negative if an error occurs.

Comments

The following example illustrates the required actions:

```
#define ENUMPAPERMETRICS    34
#define INFORM              0
#define PERFORM             1

int cPaperTypes = CPAPERTYPES;
RECT arectPage[CPAPERTYPES] = { ... };

short Control(lpDevice, wFunction, lpInData, lpOutData)
LPPDEVICE lpDevice;
WORD wFunction;
LPVOID lpInData;
LPVOID lpOutData;
```

```
{
    LPRECT arect;

    switch (wFunction) {

    case ENUMPAPERMETRICS:
        switch (*((LPINT)lpInData)) {
        case INFORM:
            return cPaperTypes;

        case PERFORM:
            arect = (LPRECT)lpOutData;
            for (i=0; i<cPaperTypes; arect++, i++)
                CopyRect(arect, arectPage[i]);
            return cPaperTypes;

        default:
            return 0;
        }
    .
    .
    .
}
```

See Also **ExtDeviceMode**

EPSPRINTING

#define EPSPRINTING 33

short Control(*lpDevice***, EPSPRINTING,** *lpBool***, NULL)**
LPPDEVICE *lpDevice*;
LPBOOL *lpBool*;

The EPSPRINTING escape only controls the downloading of the control portions of the PostScript prolog. It sets up the portrait versus landscape orientation and leaves the printer in the default 72 dpi user space.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpBool
Points to a 16-bit variable specifying whether to enable or disable encapsulated PostScript (EPS) printing. If it is TRUE, EPS printing is enabled. If FALSE, EPS printing is disabled.

Return Value

The return value is positive, if successful. Otherwise, it is zero if the escape is not implemented, and negative if an error occurs.

Comments

This escape is used to suppress the output of the Windows PostScript header control section, which is about 10K. If it is used, no GDI calls are allowed.

EXT_DEVICE_CAPS

#define EXT_DEVICE_CAPS 4099

short Control(*lpDevice***, EXT_DEVICE_CAPS,** *lpIndex***,** *lpCaps***)**
LPPDEVICE *lpDevice*;
LPINT *lpIndex*;
LPLONG *lpCaps*;

The EXT_DEVICE_CAPS escape retrieves information about device-specific capabilities. It serves as a supplement to the **GetDeviceCaps** (GDI.80) function.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpIndex
Points to a 16-bit variable specifying the index of the capability to be retrieved. It can be one of the following values.

Value	Meaning
R2_CAPS (1)	Specifies which of the 16-bit binary-raster operations the device driver supports. A unique bit is set for each supported raster operation. For example, the following code fragment sets the bit for support of the R2_XORPEN raster operation: `Caps = Caps | (1<<R2_XORPEN);` For more information about the binary-raster operations, see the **DRAWMODE** structure.

Value	Meaning
PATTERN_CAPS (2)	Specifies the maximum dimensions of a pattern brush bitmap. The low-order 16 bits of the capability value contains the maximum width of a pattern brush bitmap; the high-order 16 bits contains the maximum height.
PATH_CAPS (3)	Specifies whether the device is capable of creating paths using alternate and winding interiors, and whether the device can do exclusive or inclusive clipping to path interiors. The path capabilities is a combination of two of the following: PATH_ALTERNATE (1) PATH_WINDING (2) PATH_INCLUSIVE (3) PATH_EXCLUSIVE (4)
POLYGON_CAPS(4)	Specifies the maximum number of polygon points supported by the device. The capability value is an unsigned value specifying the maximum number of points.
PATTERN_COLOR_CAPS (5)	Specifies whether the device can convert monochrome pattern bitmaps to color. The capability value is one if the device can do pattern bitmap color conversions and zero if it cannot.
R2_TEXT_CAPS (6)	Specifies whether the device is capable of performing binary raster operations on text. The low-order 16 bits of the capability value specifies which raster operations are supported on text. A bit is set for each supported raster operation, as in the R2_CAPS escape. The high-order 16 bits specifies to which type of text the raster capabilities apply. It can be a combination of the the following values: RASTER_TEXT (1) DEVICE_TEXT (2) VECTOR_TEXT (3)
POLYMODE_CAPS (7)	Specifies the polygon modes supported by the device driver. The capability value is obtained by setting a bit in the corresponding position for each polygon mode supported. For example, if a device supports the PM_POLYSCANLINE and PM_BEZIER polygon modes, the capability value would be set as follows:

```
Caps = Caps | (1<<PM_POLYSCANLINE) | (1<<PM_BEZIER);
```

lpCaps
Points to a 32-bit variable that receives the specified capability.

Return Value The return value is a nonzero value if the specified extended capability is supported. Otherwise, it is zero if the capability is not supported.

Comments A device driver implementing this escape must not modify the value of the 32-bit integer described by the *lpCaps* parameter unless it returns a valid value for the capability.

See Also SET_POLY_MODE

FLUSHOUTPUT

#define FLUSHOUTPUT 6

short Control(*lpDevice***, FLUSHOUTPUT, NULL, NULL)**
LPPDEVICE *lpDevice*;

The FLUSHOUTPUT escape flushes output in the device's buffer.

Parameters *lpDevice*
Points to a **PDEVICE** structure specifying the destination device.

Return Value The return value is positive if the escape is successful. Otherwise, it is negative.

Comments This escape is intended for banding printer drivers. When called, they should reinitialize the banding bitmap (that is, eliminate anything in the bitmap that is only partially drawn).

GETCOLORTABLE

#define GETCOLORTABLE 5

short Control(*lpDevice***, GETCOLORTABLE,** *lpIndex***,** *lpColor***)**
LPPDEVICE *lpDevice*;
LPINT *lpIndex*;
LPLONG *lpColor*;

The GETCOLORTABLE escape retrieves an RGB color-table entry and copies it to the location specified by the *lpColor* parameter.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpIndex
Points to a 16-bit a variable specifying the index of a color-table entry. Color-table indexes start at zero for the first table entry.

lpColor
Points to 32-bit variable that receives the RGB color value for the given entry.

Return Value

The return value is positive if the escape is successful. Otherwise, it is negative.

See Also

SETCOLORTABLE

GETEXTENDEDTEXTMETRICS

#define GETEXTENDEDTEXTMETRICS 256

short Control(*lpDevice***, GETEXTENDEDTEXTMETRICS,** *lpInData***,** *lpOutData***)**
LPPDEVICE *lpDevice*;
LPEXTTEXTDATA *lpInData*;
LPEXTTEXTMETRIC *lpOutData*;

The GETEXTENDEDTEXTMETRICS escape fills the buffer pointed to by the *lpOutData* parameter with the extended text metrics for the currently selected font.

Parameters

lpDevice
A long pointer to a **PDEVICE** structure, which is the destination device bitmap.

lpInData
Points to a **EXTTEXTDATA** structure containing information The **EXTTEXTDATA** structure has the following form:

```
typedef struct tagEXTTEXTDATA {
    short             nSize;
    LPAPPEXTTEXTDATA  lpInData;
    LPFONTINFO        lpFont;
    LPTEXTXFORM       lpXForm;
    LPDRAWMODE        lpDrawMode;
} EXTTEXTDATA;
```

lpOutData
Points to a **EXTTEXTMETRIC** structure. The **EXTTEXTMETRIC** structure has the following form:

```
typedef struct tagEXTTEXTMETRIC {
    short  etmSize;
    short  etmPointSize;
    short  etmOrientation;
    short  etmMasterHeight;
    short  etmMinScale;
    short  etmMaxScale;
    short  etmMasterUnits;
    short  etmCapHeight;
    short  etmXHeight;
    short  etmLowerCaseAscent;
    short  etmUpperCaseDescent;
    short  etmSlant;
    short  etmSuperScript;
    short  etmSubScript;
    short  etmSuperScriptSize;
    short  etmSubScriptSize;
    short  etmUnderlineOffset;
    short  etmUnderlineWidth;
    short  etmDoubleUpperUnderlineOffset;
    short  etmDoubleLowerUnderlineOffset;
    short  etmDoubleUpperUnderlineWidth;
    short  etmDoubleLowerUnderlineWidth;
    short  etmStrikeOutOffset;
    short  etmStrikeOutWidth;
    WORD   etmKernPairs;
    WORD   etmKernTracks;
} EXTTEXTMETRIC;
```

Return Value

The return value is the number of bytes copied to the buffer pointed to by the *lpOutData* parameter. This value will never exceed the size specified by the **etmSize** member in the **EXTTEXTMETRIC** structure. Otherwise, the return value is zero if the escape fails or is not implemented.

Comments The values returned in many of the fields of the **EXTTEXTMETRIC** structure are affected by whether relative character widths are enabled or disabled.

See Also ENABLERELATIVEWIDTHS

GETEXTENTTABLE

#define GETEXTENTTABLE 257

short Control(*lpDevice***, GETEXTENTTABLE,** *lpInData***,** *lpOutData***)**
LPPDEVICE *lpDevice*;
LPBYTE *lpInData*;
LPINT *lpOutData*;

The GETEXTENTTABLE escape returns the width (extent) of individual characters from a group of consecutive characters in the selected font's character set. The first and last character (from the group of consecutive characters) are function arguments.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a **CHARRANGE** structure containing the character codes for the first and last characters in the range. The **CHARRANGE** structure has the following form:

```
typedef struct tagCHARRANGE {
    BYTE chFirst;
    BYTE chLast;
} CHARRANGE;
```

lpOutData
Points to an array of integer values. The size of the array must be at least one more than the difference between the **chLast** and **chFirst** members of the structure pointed to by the *lpInData* parameter.

Return Value The return value is 1 if the escape is successful. Otherwise, the return value is 0 if it is not successful, or if the escape is not implemented.

Comments The values returned are affected by whether relative character widths are enabled or disabled.

See Also ENABLERELATIVEWIDTHS

GETFACENAME

#define GETFACENAME 513

short Control(*lpDevice***, GETFACENAME, NULL,** *lpFaceName***)**
LPPDEVICE *lpDevice*;
LPSTR *lpFaceName*;

The GETFACENAME escape gets the name of the current physical font.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpFaceName
Points to the buffer that receives the name.

Return Value

The return value is positive if successful. It is zero if the escape is not implemented, and negative if an error occurs.

GETPAIRKERNTABLE

#define GETPAIRKERNTABLE 258

short Control(*lpDevice***, GETPAIRKERNTABLE, NULL,** *lpOutData***)**
LPPDEVICE *lpDevice*;
LPKERNPAIR *lpOutData*;

The GETPAIRKERNTABLE escape fills the buffer pointed to by the *lpOutData* parameter with the character pair-kerning table for the currently selected font.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpOutData
Points to an array of **KERNPAIR** structures. This array must be large enough to accommodate the font's entire character pair-kerning table. The number of character-kerning pairs in the font can be obtained from the **EXTTEXTMETRIC** structure which is returned by the GETEXTENDED-TEXTMETRICS escape. The **KERNPAIR** structure has the following form:

```
typedef struct tagKERNPAIR {
    union {
        BYTE each [2];
        WORD both;
    } kpPair;
    short  kpKernAmount;
} KERNPAIR;
```

Return Value

The return value is the number of **KERNPAIR** structures copied to the buffer. Otherwise, the return value is zero if the font does not have kerning pairs defined, the escape fails, or the escape is not implemented.

Comments

The values returned in the **KERNPAIR** structures are affected by whether relative character widths are enabled or disabled.

See Also

ENABLERELATIVEWIDTHS

GETPHYSPAGESIZE

#define GETPHYSPAGESIZE 12

short Control(*lpDevice***, GETPHYSPAGESIZE, NULL,** *lpDimensions***)**
LPPDEVICE *lpDevice*;
LPPOINT *lpDimensions*;

The GETPHYSPAGESIZE escape retrieves the physical page size in device units (that is, how many pixels wide by how many scan lines high) and copies it to the location pointed to by the *lpDimensions* parameter.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpDimensions
Points to a **POINT** structure that receives the physical page dimensions. The **POINT** structure has the following form:

```
typedef struct tagPOINT {
    short   x;
    short   y;
} POINT;
```

Return Value

The return value is positive if the escape is successful. Otherwise, it is negative.

GETPRINTINGOFFSET

#define GETPRINTINGOFFSET 13

short Control(*lpDevice***, GETPRINTINGOFFSET, NULL,** *lpOffset***)**
LPPDEVICE *lpDevice*;
LPPOINT *lpOffset*;

The GETPRINTINGOFFSET escape retrieves the offset from location 0, 0 (the upper-left corner of the physical page), which is the point at which the actual printing or drawing begins.

This escape function is not generally useful for devices that allow the user to set the origin by hand.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpOffset
Points to a **POINT** structure that receives the horizontal and vertical coordinates (in device units) of the printing offset. The **POINT** structure has the following form:

```
typedef struct tagPOINT {
    short   x;
    short   y;
} POINT;
```

Return Value

The return value is positive if the escape is successful. Otherwise, it is negative.

GETSCALINGFACTOR

#define GETSCALINGFACTOR 14

short Control(*lpDevice***, GETSCALINGFACTOR, NULL,** *lpFactors***)**
LPPDEVICE *lpDevice*;
LPPOINT *lpFactors*;

The GETSCALINGFACTOR escape retrieves the scaling factors for the *x* and *y* axes of a printing device. For each scaling factor, the escape copies an exponent of two to the location pointed to by the *lpFactors* parameter. For example, the value three is copied to *lpFactors* for a scaling factor of eight.

Scaling factors are used by printing devices that cannot provide the same resolution as the device resolution. This escape communicates to GDI the factor by which it needs to stretch bitmaps when sending them to the printer.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpFactors
Points to a **POINT** structure that receives the horizontal and vertical scaling factors in the **x** and **y** members, respectively. The **POINT** structure has the following form:

```
typedef struct tagPOINT {
    short   x;
    short   y;
} POINT;
```

Return Value

The return value is positive if the escape is successful. Otherwise, it is negative.

GETSETPAPERBINS

#define GETSETPAPERBINS 29

short Control(*lpDevice***, GETSETPAPERBINS,** *lpInData***,** *lpOutData***)**
LPPDEVICE *lpDevice***;**
LPBININFO *lpInData***;**
LPBININFO *lpOutData***;**

The GETSETPAPERBINS escape retrieves the number of paper bins available on a printer and sets the current paper bin.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a **BININFO** structure that specifies the new paper bin. This parameter may be set to NULL. The **BININFO** structure has the following form:

```
typedef struct tagBININFO {
    short   BinNumber;
    short   NbrofBins;
    short   Reserved[4];
} BININFO;
```

lpOutData
Points to a **BININFO** structure that receives information about the current or previous paper bin and the number of bins available. This parameter may be set to NULL.

Return Value

The return value is TRUE, if successful. Otherwise, it is FALSE.

Comments

If the *lpInData* parameter is NULL and the *lpOutData* parameter points to a **BININFO** structure, the escape retrieves the number of bins and the number of the current bin.

If both *lpInData* and *lpOutData* point to **BININFO** structures, the escape sets the current bin to the number specified in the **BinNumber** member of the structure pointed to by *lpInData* and retrieves the number of the previous bin.

If *lpInData* points to a **BININFO** structure but *lpOutData* is NULL, the escape sets the current bin to the number specified in the **BinNumber** member of the structure pointed to by *lpInData*.

When setting the paper bin with GETSETPAPERBINS, the bin selected is set for the current job by setting the bit 15 of the bin index. If this bit is not set, the selected paper bin becomes the default for later print jobs, and the current job's selection is unchanged. Setting bit 15 enables an application to change bins in the middle of a printing job.

GETSETPAPERMETRICS

#define GETSETPAPERMETRICS 35

short Control(*lpDevice*, **GETSETPAPERMETRICS**, *lpNewPaper*, *lpOrigPaper*)
LPPDEVICE *lpDevice*;
LPRECT *lpNewPaper*;
LPRECT *lpOrigPaper*;

The GETSETPAPERMETRICS escape sets the paper type according to the given paper-metrics information. It also gets the current printer's paper-metrics information. However, this escape is needed now only for backward compatibility with earlier applications. The **ExtDeviceMode** function achieves the same results.

This escape expects a **RECT** structure, representing the imageable area of the physical page, and assumes the origin is in the upper-left corner.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpNewPaper
Points to a **RECT** structure that contains the new imageable area. The **RECT** structure has the following form:

```
typedef struct tagRECT {
    short left;
    short top;
    short right;
    short bottom;
} RECT;
```

The coordinates are measured in device units. The orientation is set to match this parameter.

lpOrigPaper
Points to a **RECT** structure that receives the original value.

The coordinates are measured in device units.

Return Value

The return value is positive if successful. Otherwise, the return value is zero if the escape is not implemented, and negative if an error occurs.

See Also

ExtDeviceMode

GETSETPRINTORIENT

#define GETSETPRINTORIENT 30

short Control(*lpDevice***, GETSETPRINTORIENT,** *lpInData***, NULL)**
LPPDEVICE *lpDevice*;
LPORIENT *lpInData*;

The GETSETPRINTORIENT escape returns or sets the current paper orientation. However, this escape is needed now only for backward compatibility with earlier applications. The **ExtDeviceMode** function achieves the same results.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to an **ORIENT** structure that specifies the new paper orientation. If this parameter is NULL, the GETSETPRINTORIENT escape returns the current paper orientation. The **ORIENT** structure has the following form:

```
typedef struct tagORIENT {
    short Orientation;
    short Reserved[4];
} ORIENT;
```

Return Value

The return value specifies the current orientation, if the *lpInData* parameter is NULL. Otherwise, it is the previous orientation, or –1 if the escape failed.

Comments

The new orientation will take effect for the next device context created for the device on this port.

See Also

ExtDeviceMode

GETSETSCREENPARAMS

short Escape(*hdc***, GETSETSCREENPARAMS, sizeof(SCREENPARAMS),** *lpInData***,** *lpOutData***)**

The GETSETSCREENPARAMS printer escape retrieves or sets the current screen information for rendering halftones.

Parameters

hdc
HDC Identifies the device context.

lpInData
SCREENPARAMS FAR * Points to a **SCREENPARAMS** structure that contains the new screen information. For more information about this structure, see the following Comments section. This parameter may be NULL.

lpOutData
SCREENPARAMS FAR * Points to a **SCREENPARAMS** structure that retrieves the previous screen information. For more information about this structure, see the following Comments section. This parameter may be NULL.

Return Value

The return value specifies the outcome of the escape. This value is positive if the escape is successful. Otherwise, it is negative.

Comments

This escape affects how device-independent bitmaps (DIBs) are rendered and how color objects are filled.

The **SCREENPARAMS** structure has the following form:

```
typedef struct tagSCREENPARAMS {
  int   angle;
  int   frequency;
} SCREENPARAMS;
```

Following are the members of the **SCREENPARAMS** structure:

angle
Specifies, in degrees, the angle of the halftone screen.

frequency
Specifies, in dots per inch, the screen frequency.

GETTECHNOLOGY

#define GETTECHNOLOGY 20

short Control(*lpDevice***, GETTECHNOLOGY, NULL,** *lpTechnology***)**
LPPDEVICE *lpDevice*;
LPSTR *lpTechnology*;

The GETTECHNOLOGY escape retrieves the general technology type for a printer. This allows an application to perform technology-specific actions.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpTechnology
Points to a buffer to which the driver copies a null-terminated string containing the printer technology type, such as "PostScript."

Return Value

The return value is 1 if the escape is successful. Otherwise, it is 0 if the escape is not successful or not implemented.

GETTRACKKERNTABLE

#define GETTRACKKERNTABLE 259

short Control(*lpDevice***, GETTRACKKERNTABLE, NULL,** *lpOutData***)**
LPPDEVICE *lpDevice***;**
LPKERNTRACK *lpOutData***;**

The GETTRACKKERNTABLE escape fills the buffer pointed to by the *lpOutData* parameter with the track-kerning table for the currently selected font.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpOutData
Points to an array of **KERNTRACK** structures. This array must be large enough to accommodate all the font's kerning tracks. The number of tracks in the font can be obtained from the **EXTTEXTMETRIC** structure returned by the GETEXTENDEDTEXTMETRICS escape. If *lpOutData* is NULL, GETTRACKKERNTABLE returns the number of table entries. The **KERNTRACK** structure has the following form:

```
typedef struct tagKERNTRACK {
    short ktDegree;
    short ktMinSize;
    short ktMinAmount;
    short ktMaxSize;
    short ktMaxAmount;
} KERNTRACK;
```

Return Value

The return value is the number of **KERNTRACK** structures copied to the buffer. The return value is zero if the font does not have kerning tracks defined, if the function fails, or if the escape is not implemented.

Comments

The values returned in the **KERNTRACK** structures are affected by whether relative character widths are enabled or disabled.

See Also

ENABLERELATIVEWIDTHS, GETEXTENDEDTEXTMETRICS

GETVECTORBRUSHSIZE

#define GETVECTORBRUSHSIZE 27

short Control(*lpDevice***, GETVECTORBRUSHSIZE,** *lpInData***,** *lpOutData***)**
LPPDEVICE *lpDevice***;**
LPLBRUSH *lpInData***;**
LPPOINT *lpOutData***;**

The GETVECTORBRUSHSIZE escape retrieves the size in device units of a plotter pen used to fill closed figures. GDI uses this information to prevent the filling of closed figures (for example, rectangles and ellipses) from overwriting the borders of the figure.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a **LBRUSH** structure that specifies the brush for which data is to be returned. The **LBRUSH** structure has the following form:

```
typedef struct tagLBRUSH {
    short   lbStyle;
    long    lbColor;
    short   lbHatch;
    long    lbBkColor;
} LBRUSH;
```

lpOutData
Points to a **POINT** structure that receives the width of the pen in device units. The escape copies the width to the **y** member. The **POINT** structure has the following form:

```
typedef struct tagPOINT {
    short   x;
    short   y;
} POINT;
```

Return Value

The return value is 1 if the escape is successful. Otherwise, it is 0 if the escape is not successful or not implemented.

GETVECTORPENSIZE

#define GETVECTORPENSIZE 26

short Control(*lpDevice***, GETVECTORPENSIZE,** *lpInData*, *lpOutData*)
LPPDEVICE *lpDevice*;
LPPEN *lpInData*;
LPPOINT *lpOutData*;

The GETVECTORPENSIZE escape retrieves the size in device units of a plotter pen. GDI uses this information to prevent hatched brush patterns from overwriting the border of a closed figure.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a **LPEN** structure that specifies the pen for which the width is to be retrieved. The **LPEN** structure has the following form:

```
typedef struct tagLPEN {
    long  lopnStyle;
    POINT lopnWidth;
    long  lopnColor;
} LPEN;
```

lpOutData
Points to a **POINT** structure that receives the pen width in device units. The escape copies the pen width to the **y** member. The **POINT** structure has the following form:

```
typedef struct tagPOINT {
    short   x;
    short   y;
} POINT;
```

Return Value

The return value is 1 if the escape is successful. Otherwise, it is 0 if the escape is not successful or not implemented.

NEWFRAME

#define NEWFRAME 1

short Control(*lpDevice***, NEWFRAME, NULL, NULL)**
LPPDEVICE *lpDevice*;

The NEWFRAME escape informs the device that the application has finished writing to a page. This escape is used typically with a printer to direct the device driver to advance to a new page by performing a page-break algorithm or form feed.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

Return Value

The return value is positive if the escape is successful. Otherwise, it is one of the following values.

Value	Meaning
SP_ERROR (–1)	A general error condition or general error in banding occurred.
SP_APPABORT (–2)	The job was stopped because the application's callback function returned FALSE (0).
SP_USERABORT (–3)	The user stopped the print job by choosing the Delete button from Print Manager.
SP_OUTOFDISK (–4)	A lack of disk space caused the job to stop. There is not enough disk space to create or extend the Print Manager temporary file.
SP_OUTOFMEMORY (–5)	A lack of memory caused the job to stop.

NEXTBAND

#define NEXTBAND 3

short Control(*lpDevice***, NEXTBAND,** *lpInData*, *lpBandRect*)
LPPDEVICE *lpDevice*;
LPPOINT *lpInData*;
LPRECT *lpBandRect*;

The NEXTBAND escape informs the device driver that the application has finished writing to a band. The device driver then sends the band to the printer and returns the coordinates of the next band.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a **POINT** structure that receives the horizontal and vertical scaling factors in the **x** and **y** members respectively. The **POINT** structure has the following form:

```
typedef struct tagPOINT [
    short   x;
    short   y;
] POINT;
```

The shift count in the **POINT** structure pointed to by the *lpInData* parameter is used for devices such as laser printers that support graphics at a lower resolution than text.

lpBandRect
Points to a **RECT** structure that receives the next band coordinates. The device driver copies the device coordinates of the next band into this structure.

Return Value

The return value is positive if the escape is successful. Otherwise, it is one of the following values.

Value	Meaning
SP_ERROR (–1)	A general error condition or general error in banding occurred.
SP_APPABORT (–2)	The job was stopped because the application's callback function returned FALSE (0).
SP_USERABORT (–3)	The user stopped the print job by choosing the Delete button from Print Manager.
SP_OUTOFDISK (–4)	A lack of disk space caused the job to stop. There is not enough disk space to create or extend the Print Manager temporary file.
SP_OUTOFMEMORY (–5)	A lack of memory caused the job to stop.

PASSTHROUGH

#define PASSTHROUGH 19

short Control(*lpDevice***, PASSTHROUGH,** *lpInData***, NULL)**
LPPDEVICE *lpDevice*;
LPWORD *lpInData*;

The PASSTHROUGH escape enables the application to send data directly to the printer, bypassing the standard printer-driver code.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a buffer in which the first 16 bits specifies the number of remaining bytes in the buffer. The escape sends the remaining bytes in the buffer to the printer.

Return Value

The return value is the number of bytes transferred to the printer if the escape is successful. Otherwise, the return value is zero if the escape is not successful, or if the escape is not implemented. If the return value is nonzero but less than the size of the data, an error prohibited transmission of the entire data block.

Comments

There may be restrictions on the kinds of device data an application may send to the device without interfering with the operation of the driver should not send commands that reset the printer or print the current page. Additionally, applications are strongly discouraged from performing functions that consume printer memory, such as downloading a font or a macro.

The driver should invalidate its internal state variables such as "current position" and "current line style" when carrying out this escape. The application must be able to issue multiple, sequential PASSTHROUGH escapes without intervening "saves" and "restores" being inserted by the driver.

This escape is also known as DEVICEDATA.

QUERYESCSUPPORT

#define QUERYESCSUPPORT 8

short Control(*lpDevice***, QUERYESCSUPPORT,** *lpEscNum***, NULL)**
LPPDEVICE *lpDevice*;
LPINT *lpEscNum*;

The QUERYESCSUPPORT escape determines if a particular escape is implemented by the device driver.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpEscNum
Points to a 16-bit variable specifying the escape function to be checked.

Return Value

The return value is nonzero for implemented escapes. Otherwise, it is zero for unimplemented escapes. All device drivers must return success if queried about whether they support the QUERYESCSUPPORT escape.

RESETDEVICE

#define RESETDEVICE 128

short Control(*lpDevice***, RESETDEVICE,** *lpDeviceOld***, NULL)**
LPPDEVICE *lpDevice*;
LPPDEVICE *lpDeviceOld*;

The RESETDEVICE escape resets the device context by copying information that is specific to the current print job from the original physical device structure to the new one. GDI calls this escape whenever an application calls the **ResetDC** function (GDI.376).

Printer drivers that can reset print-job options (such as orientation and paper source) during a single print job must support the RESETDEVICE escape.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the new physical device.

lpDeviceOld
Points to a **PDEVICE** structure containing the device-specific settings to be copied to the new physical device structure.

Return Value The return value is TRUE if the function is successful. Otherwise, it is FALSE.

Comments The RESETDEVICE escape must copy all information required to continue the current print job to the new physical device structure. This includes information such as the print-job number and record of downloaded resources, but does not include information about the environment (such as orientation and paper source). GDI calls the **Enable** function to set the environment for the new physical device structure before calling this escape.

After the RESETDEVICE escape returns, GDI immediately calls the **Disable** function with the old physical device structure. RESETDEVICE must ensure that resources copied to the new physical device structure are not deleted on the subsequent call to **Disable**. For example, if the **Disable** function frees any working buffers allocated by the driver at the start of a print job, RESETDEVICE must either allocate new buffers for the new physical device structure or remove all pointers to these buffers from the old physical device structure.

See Also **Disable**, **Enable**

RESTORE_CTM

#define RESTORE_CTM 4100

short Control(*lpDevice***, RESTORE_CTM, NULL, NULL)**
LPPDEVICE *lpDevice*;

The RESTORE_CTM escape restores the current, previously saved transformation matrix (CTM). The CTM controls the manner in which coordinates are translated, rotated, and scaled by the device. By using matrixes, you can combine these operations in any order to produce the desired mapping for a particular picture.

Parameters *lpDevice*
Points to a **PDEVICE** structure specifying the destination device.

Return Value The return value is the number of SAVE_CTM escapes without a corresponding RESTORE_CTM escape. The return value is –1 if the escape is unsuccessful.

Comments Applications should not make any assumptions about the initial contents of the CTM.

When a driver transforms a primitive using a transformation matrix modified by the application, it should ignore the clipping rectangle specified by GDI. Applications should specify the desired clipping rectangle using the SET_CLIP_BOX escape.

Drivers supporting this escape must also implement the SET_CLIP_BOX, SAVE_CTM, and TRANSFORM_CTM escapes.

See Also SAVE_CTM, SET_CLIP_BOX, TRANSFORM_CTM

SAVE_CTM

#define SAVE_CTM 4101

short Control(*lpDevice***, SAVE_CTM, NULL, NULL)**
LPPDEVICE *lpDevice*;

The SAVE_CTM escape saves the current transformation matrix (CTM). The CTM controls the manner in which coordinates are translated, rotated, and scaled by the device. By using matrixes, you can combine these operations in any order to produce the desired mapping for a particular image.

You can restore the matrix by using the RESTORE_CTM escape.

An application typically saves the CTM before changing it. This enables the application to restore the previous state upon completion of a particular operation.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

Return Value The return value is the number of SAVE_CTM escapes without a corresponding RESTORE_CTM escape. The return value is zero if the escape is unsuccessful.

Comments Applications should not make any assumptions about the initial contents of the CTM and are expected to restore the contents of the CTM.

When a driver transforms a primitive using a transformation matrix modified by the application, it should ignore the clipping rectangle specified by GDI. Applications should specify the desired clipping rectangle using the SET_CLIP_BOX escape.

Drivers supporting this escape must also implement the SET_CLIP_BOX, RESTORE_CTM, and TRANSFORM_CTM escapes.

See Also RESTORE_CTM, SET_CLIP_BOX, TRANSFORM_CTM

SET_ARC_DIRECTION

#define SET_ARC_DIRECTION 4102

short Control(*lpDevice***, SET_ARC_DIRECTION,** *lpDirection***, NULL)**
LPPDEVICE *lpDevice*;
LPINT *lpDirection*;

The SET_ARC_DIRECTION escape specifies the direction in which elliptical arcs are drawn using the **Arc** (GDI.23) function.

By convention, elliptical arcs are drawn counterclockwise by GDI. This escape enables an application to draw paths containing arcs drawn clockwise.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpDirection
Points to a 16-bit variable specifying the arc direction. It may be either COUNTERCLOCKWISE(0) or CLOCKWISE(1).

Return Value The return value is the previous arc direction.

Comments The default arc direction is COUNTERCLOCKWISE.

Device drivers that implement the BEGIN_PATH and END_PATH escapes should also implement this escape.

This escape maps to the PostScript page-description language elements and is intended for PostScript line devices.

See Also BEGIN_PATH, END_PATH

SET_BACKGROUND_COLOR

#define SET_BACKGROUND_COLOR 4103

short Control(*lpDevice***, SET_BACKGROUND_COLOR,** *lpNewColor***,** *lpOldColor***)**
LPPDEVICE *lpDevice*;
LPLONG *lpNewColor*;
LPLONG *lpOldColor*;

The SET_BACKGROUND_COLOR escape enables an application to set and retrieve the current background color for the device. The background color is the color of the display surface before an application draws anything on the device. This escape is particularly useful for color printers and film recorders.

This escape should be sent before the application draws anything on the current page.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpNewColor
Points to a 32-bit variable specifying the desired background color. This parameter can be NULL if the application merely wants to retrieve the current background color.

lpOldColor
Points to a 32-bit variable that receives the previous background color. This parameter can be NULL if the application wants to ignore the previous background color.

Return Value

The return value is TRUE if the escape is successful. Otherwise, it is FALSE.

Comments

The default background color is white.

The background color is reset to the default color when the device driver receives an ENDDOC or ABORTDOC escape.

See Also

ABORTDOC, ENDDOC

SET_BOUNDS

#define SET_BOUNDS 4109

short Control(*lpDevice***, SET_BOUNDS,** *lpBounds***, NULL)**
LPPDEVICE *lpDevice*;
LPRECT *lpBounds*;

The SET_BOUNDS escape sets the bounding rectangle for the image being output by the device driver when creating images in a file format such as Encapsulated PostScript (EPS) and Hewlett-Packard Graphics Language (HPGL).

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpBounds
Points to a **RECT** structure that specifies the bounds, in device coordinates, of the image to output.

Return Value

The return value is TRUE if successful. Otherwise, it is FALSE.

Comments

The application should issue this escape before each page. For single-page jobs, this escape should be issued immediately after the STARTDOC escape.

When using coordinate transformation escapes, device drivers may not perform bounding-box calculations correctly. Using this escape saves the driver from the task of calculating the bounding box.

See Also

STARTDOC

SET_CLIP_BOX

#define SET_CLIP_BOX 4108

short Control(*lpDevice***, SET_CLIP_BOX,** *lpClipBox***, NULL)**
LPPDEVICE *lpDevice*;
LPRECT *lpClipBox*;

The SET_CLIP_BOX escape sets the clipping rectangle or restores the previous clipping rectangle. The SET_CLIP_BOX escape is required of any device driver that implements the coordinate transformation escapes.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpClipBox
Points to a **RECT** structure that contains the bounding rectangle of the desired clipping area. If the *lpClipBox* parameter is not NULL, the previous clipping rectangle is saved, and the current clipping rectangle is set to the specified bounds. If *lpClipBox* is NULL, the previous clipping rectangle is restored.

Return Value

The return value is a Boolean value specifying whether or not the clipping rectangle was properly set.

Comments

Drivers that implement the TRANSFORM_CTM, SAVE_CTM, and RESTORE_CTM escapes must also implement this escape.

When an applications calls a GDI output function, GDI calculates a clipping rectangle that bounds the primitive, and then passes both the primitive and the clipping rectangle to the driver. The driver is expected to clip the primitive to the specified bounding rectangle. However, when an application uses the coordinate transformation escapes, the clipping rectangle that was calculated by GDI is generally invalid.

The application can use the SET_CLIP_BOX escape to specify the correct clipping rectangle when coordinate transformations are used.

See Also

RESTORE_CTM, SAVE_CTM, TRANSFORM_CTM

SET_POLY_MODE

#define SET_POLY_MODE 4104

short Control(*lpDevice***, SET_POLY_MODE,** *lpMode***, NULL)**
LPPDEVICE *lpDevice*;
LPINT *lpMode*;

The SET_POLY_MODE escape enables a device driver to draw shapes (such as Bezier curves) that are not supported directly by GDI. This permits applications that draw complex curves to send the curve description directly to a device without having to simulate the curve as a polygon with a large number of points.

The SET_POLY_MODE escape sets the poly mode for the device driver. The poly mode is a state variable indicating how to interpret calls to the **Polygon** (GDI.63) and **Polyline** (GDI.37) functions.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpMode
Points to a 16-bit variable that specifies the desired poly mode. The parameter can be one of the following values.

Value	Meaning
PM_POLYLINE (1)	The points define a conventional polygon or polyline.
PM_BEZIER (2)	The points define a sequence of 4-point Bezier spline curves. The first curve passes through the first four points, with the first and fourth points as end points, and the second and third points as control points. Each subsequent curve in the sequence has the end point of the previous curve as its start point, the next two points as control points, and the third as its end point. The last curve in the sequence is permitted to have fewer than four points. If the curve has only one point, it is considered a point. If it has two points, it is a line segment. If it has three points, it is a parabola defined by drawing a Bezier curve with the end points equal to the first and third points and the two control points equal to the second point.
PM_POLYLINESEGMENT (3)	The points specify a list of coordinate pairs. Line segments are drawn connecting each successive pair of points.

The device driver need not support all the possible modes. It is expected to return zero if it does not support the specified mode.

Return Value

The return value is the previous poly mode if the escape is successful. Otherwise, the return value is zero.

Comments

An application should issue the SET_POLY_MODE escape before it draws a complex curve. It should then call **Polyline** or **Polygon** with the desired control points defining the curve. After drawing the curve, the application should reset the driver to its previous state by reissuing the SET_POLY_MODE escape.

Calls to the **Polyline** function are drawn using the currently selected pen.

Calls to the **Polygon** function are drawn using the currently selected pen and brush. If the start and end points are not equal, a line is drawn from the start point to the end point before filling the polygon (or curve).

Calls to the **Polygon** function using PM_POLYLINESEGMENT mode are treated exactly the same as calls to **Polyline**.

A Bezier curve is defined by four points. The curve is generated by connecting the first and second, second and third, and third and fourth points. The midpoints of these consecutive line segments are then connected. Then the midpoints of the lines connecting the midpoints are connected.

The line segments drawn in this way converge to a curve defined by the following parametric equations, expressed as a function of an independent variable *t*.

```
X(t) = (1-t)**3x1 + 3(1-t)**2tx2 + 3(1-t)t**2x3 + t**3x4

Y(t) = (1-t)**3y1 + 3(1-t)**2ty2 + 3(1-t)t**2y3 + t**3y4
```

The points (x1,y1), (x2,y2), (x3,y3), and (x4,y4) are the control points defining the curve. The independent variable *t* varies from 0 to 1.

A second-degree Bezier curve may be expressed as a third-degree Bezier using the following parameterization:

```
Cx1 = 1/3X1 + 2/3X2     Cy1 = 1/3Y1 + 2/3Y2
Cx2 = 2/3X2 + 1/3X3     Cy2 = 2/3Y2 + 1/3Y3
```

(Cx1, Cy1) and (Cx2, Cy2) are third-degree control points of the second-degree Bezier specified by the points (x1, y1), (x2, y2), and (x3, y3).

Applications are expected to check the return value from this escape to determine whether or not the driver supports the specified poly mode.

See Also **Polygon, Polyline**

SET_SCREEN_ANGLE

#define SET_SCREEN_ANGLE 4105

short Control(*lpDevice***, SET_SCREEN_ANGLE, *lpAngle*, NULL)**
LPPDEVICE *lpDevice*;
LPINT *lpAngle*;

The SET_SCREEN_ANGLE escape sets the current screen angle to the desired angle and enables an application to simulate the tilting of a photographic mask in producing a separation for a particular primary color.

Four-color process separation is the process of separating the colors comprising an image into four component primaries: cyan, magenta, yellow, and black. The image is then reproduced by overprinting each primary. In traditional four-color process printing, half-tone images for each of the four primaries are photographed against a mask tilted to a particular angle. Tilting the mask in this manner minimizes unwanted moire patterns caused by overprinting two or more colors.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpAngle
Points to a 16-bit variable specifying the desired screen angle in tenths of a degree. The angle is measured counterclockwise.

Return Value

This escape returns the previous screen angle.

Comments

The default screen angle is defined by the device driver.

SET_SPREAD

#define SET_SPREAD 4106

short Control(*lpDevice***, SET_SPREAD,** *lpSpread***, NULL)**
LPPDEVICE *lpDevice*;
LPINT *lpSpread*;

The SET_SPREAD escape sets the spread for spot-color separation. The spread is the amount by which all the nonwhite primitives are expanded to provide a slight overlap between primitives to compensate for imperfections in the reproduction process.

Spot-color separation is the process of separating an image into each distinct color used in the image. You then reproduce the image by overprinting each successive color in the image. When reproducing a spot-separated image, the printing equipment must be calibrated to align each page exactly on each pass. However, differences between passes in such factors as temperature and humidity often cause images to align imperfectly on subsequent passes. For this reason, lines in spot separations are often widened (spread) slightly to make up for problems in registering subsequent passes through the printer. This process is called trapping.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpSpread
Points to a 16-bit variable specifying the amount, in device units, by which all the nonwhite primitives are to be expanded.

Return Value

The return value is the previous spread.

Comments

The default spread is zero.

The current spread applies to all the bordered primitives (whether or not the border is visible) and text.

SETABORTPROC

#define SETABORTPROC 9

short Control(*lpDevice***, SETABORTPROC,** *lphDC***, NULL)**
LPPDEVICE *lpDevice*;
LPHANDLE *lphDC*;

The SETABORTPROC escape sets the cancel function for the print job.

If an application wants to allow the print job to be cancelled during spooling, it must set the cancel function before the print job is started with the STARTDOC escape. Print Manager calls the cancel function during spooling to allow the application to cancel the print job or to handle out-of-disk-space conditions. If no cancel function is set, the print job will fail if there is not enough disk space for spooling.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lphDC
Points to a handle to the application's device context for the print job.

Return Value

The return value is positive if the escape is successful. Otherwise, it is negative.

Comments

The driver should pass the application's device context handle to the **OpenJob** (GDI.240) function to allow GDI to call the application's callback function.

See Also

STARTDOC

SETALLJUSTVALUES

#define SETALLJUSTVALUES 771

short Control(*lpDevice***, SETALLJUSTVALUES,** *lpInData***, NULL)**
LPPDEVICE *lpDevice*;
LPINT *lpInData*;

The SETALLJUSTVALUES escape sets all the text justification values that are used for text output. Text justification is the process of inserting extra pixels among break characters in a line of text. The space character is normally used as a break character.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a **JUST_VALUE_STRUCT** structure that contains the justification values. The **JUST_VALUE_STRUCT** structure has the following form:

```
typedef struct tagJUST_VALUE_STRUCT {
    short nCharExtra;
    WORD  nCharCount;
    short nBreakExtra;
    WORD  nBreakCount;
} JUST_VALUE_STRUCT;
```

Return Value

The return value is 1 if the escape is successful. Otherwise, it is 0.

Comments

The units used for the **nCharExtra** and **nBreakExtra** members in the **JUST_VALUE_STRUCT** structure specify font units, and are dependent on whether or not relative character widths were enabled with the ENABLERELATIVEWIDTHS escape.

The values set with this escape will apply to subsequent calls to the **ExtTextOut** function. The driver will stop distributing the **nCharExtra** amount when it has output of **nCharCount** characters. It will also stop distributing the space specified by **nBreakExtra** when it has output of **nBreakCount** characters. A call on the same string to the **GetTextExtent** (GDI.91) function, made immediately after the **ExtTextOut** function, is processed in the same manner.

To reenable justification with the **SetTextJustification** (GDI.10) and **SetTextCharacterExtra** (GDI.8) functions, an application should call SETALLJUSTVALUES and set the members **nCharExtra** and **nBreakExtra** to zero.

See Also

ENABLERELATIVEWIDTHS, EXTTEXTMETRIC

SETCOLORTABLE

#define SETCOLORTABLE 4

short Control(*lpDevice***, SETCOLORTABLE,** *lpColorEntry***,** *lpColor***)**
LPPDEVICE *lpDevice*;
LPWORD *lpColorEntry*;
LPLONG *lpColor*;

The SETCOLORTABLE escape sets an RGB color-table entry. If the device cannot supply the exact color, the function sets the entry to the closest possible approximation of the color.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpColorEntry
Points to a **COLORTABLE_STRUCT** structure. The **COLORTABLE_STRUCT** structure has the following form:

```
typedef struct tagCOLORTABLE_STRUCT {
    WORD Index;
    LONG rgb;
} COLORTABLE_STRUCT;
```

lpColor
Points to a 32-bit variable that receives the RGB color value selected by the device driver to represent the requested color value.

Return Value

The return value is positive if the escape is successful. Otherwise, it is negative.

Comments

A device's color table is a shared resource; changing the system display color for one window changes it for all the windows.

The SETCOLORTABLE escape has no effect on devices with fixed-color tables.

This escape is intended for use by both printer and display drivers. However, the EGA and VGA color drivers do not support it. It should not be used with palette-capable display devices.

It is used by applications that want to change the palette used by the display driver. However, since the driver's color-mapping algorithms will probably no longer work with a different palette, an extension has been added to this escape.

If the color index pointed to by the *lpColorEntry* parameter is 0xFFFFH, the driver is to leave all color-mapping functionality to the calling application. The application will necessarily know the proper color-mapping algorithm and take responsibility for passing the correctly mapped physical color to the driver (instead of the logical RGB color) in functions such as **RealizeObject** and **ColorInfo**.

For example, if the device supports 256 colors with palette indexes of 0 through 255, the application would determine which index contains the color that it wants to use in a certain brush. It would then pass this index in the low byte of the logical color passed to **RealizeObject**. The driver would then use this color exactly as passed instead of performing its usual color-mapping algorithm. If the application wants to reactivate the driver's color-mapping algorithm (that is, if it restores the original palette when switching from its window context), then the color index pointed to by *lpColorEntry* should be 0xFFFEH.

See Also **ColorInfo**, GETCOLORTABLE, **RealizeObject**

SETCOPYCOUNT

#define SETCOPYCOUNT 17

short Control(*lpDevice***, SETCOPYCOUNT,** *lpInData***,** *lpOutData***)**
LPPDEVICE *lpDevice***;**
LPINT *lpInData***;**
LPINT *lpOutData***;**

The SETCOPYCOUNT escape specifies the number of uncollated copies of each page that the printer is to print.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a 16-bit variable containing the number of uncollated copies to print.

lpOutData
Points to a 16-bit variable that receives the number of copies to print. This may be less than the number requested if the requested number is greater than the device's maximum copy count.

Return Value The return value is 1 if the escape is successful. Otherwise it is 0 if it is not, or if the escape is not implemented.

SETKERNTRACK

#define SETKERNTRACK 770

short Control(*lpDevice***, SETKERNTRACK,** *lpInData***,** *lpOutData***)**
LPPDEVICE *lpDevice***;**
LPINT *lpInData***;**
LPINT *lpOutData***;**

The SETKERNTRACK escape specifies which kerning track a driver that supports automatic-track kerning should use. A kerning track of zero disables automatic-track kerning. When this escape is enabled, the driver automatically kerns all characters according to the specified track. The driver reflects this kerning both on the printer and in calls to the **GetTextExtent** (GDI.91) function.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a 16-bit variable that specifies the kerning track to use. A value of zero disables this feature. The **etmKernTracks** member in the **EXTTEXTMETRIC** structure for the driver specifies the maximum permitted value. Each permitted value corresponds to a position in the track-kerning table (using one as the first item in the table).

lpOutData
Points to a 16-bit variable that receives the previous kerning track.

Return Value

The return value is 1 if the escape is successful. Otherwise, it is 0 if it is not, or if the escape is not implemented.

Comments

The default state is zero, which means that automatic-track kerning is disabled.

A driver does not have to support this escape just because it supplies the track-kerning table to the application using the GETTRACKKERNTABLE escape. In the case where GETTRACKKERNTABLE is supported but SETKERNTRACK is not, it is the application's responsibility to properly space the characters on the output device.

See Also

GETEXTENDEDTEXTMETRICS, GETTRACKKERNTABLE

SETLINECAP

#define SETLINECAP 21

short Control(*lpDevice***, SETLINECAP,** *lpInData***,** *lpOutData***)**
LPPDEVICE *lpDevice*;
LPINT *lpInData*;
LPINT *lpOutData*;

The SETLINECAP escape sets the line end cap. An end cap is that portion of a line segment that appears on either end of the segment. The cap may be square or circular; it can extend past, or remain flush with, the specified end points.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a 16-bit variable that specifies the end-cap type. The variable can contain one of the following values.

Value	Meaning
–1	Line segments are drawn by using the default GDI end cap.
0	Line segments are drawn with a squared end point that does not project past the specified segment length.
1	Line segments are drawn with a rounded end point; the diameter of this semicircular arc is equal to the line width.
2	Line segments are drawn with a squared end point that projects past the specified segment length. The projection is equal to half the line width.

lpOutData
Points to a 16-bit variable that receives the previous end-cap setting.

Return Value

The return value is positive if the escape is successful. Otherwise, it is negative.

Comments

The interpretation of this escape varies with page-description languages (PDLs). Consult your PDL documentation for its exact meaning.

This escape is also known as SETENDCAP.

See Also

SETLINEJOIN

SETLINEJOIN

#define SETLINEJOIN 22

short Control(*lpDevice***, SETLINEJOIN,** *lpInData*, *lpOutData*)
LPPDEVICE *lpDevice*;
LPWORD *lpInData*;
LPWORD *lpOutData*;

The SETLINEJOIN escape specifies how to join two lines that meet at an angle.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a 16-bit variable that specifies the newline join type. If this parameter is NULL, the escape does not change the line join. The variable can contain one of the following values.

Value	Meaning
0	Miter join. The outer edges of the strokes for the two segments are extended until they meet at an angle, as in a picture frame. If the segments meet at too sharp of an angle, a bevel join is used instead; this is controlled by the miter limit established by the SETMITERLIMIT escape.
1	Round join. A circular arc with a diameter equal to the line width is drawn around the point where the segments meet, and is filled in, producing a rounded corner. The stroke actually draws a full circle at this point. If path segments shorter than one-half the line width meet at sharp angles, an unintentional "wrong side" of this circle may appear.
2	Bevel join. The meeting path segments are finished with butt-end caps (same as set by the SETLINECAP escape); then, the resulting notch beyond the ends of the segments is filled with a triangle.

lpOutData
Points to a 16-bit variable that receives the previous line-join type. If this parameter is NULL, the escape does not return the previous line-join type.

Return Value

The return value is TRUE if the escape is successful. Otherwise, it is FALSE.

Comments

Join styles are significant only at points at which consecutive segments of a path connect at an angle. Curved lines are actually rendered as sequences of straight line segments, and the current line join is applied to the "corners" between those segments. However, for typical values of the flatness parameter, the corners are so shallow that the difference between join styles is not visible.

See Also

SETLINECAP, SETMITERLIMIT

SETMITERLIMIT

#define SETMITERLIMIT 23

short Control(*lpDevice***, SETMITERLIMIT,** *lpInData***,** *lpOutData***)**
LPPDEVICE *lpDevice***;**
LPWORD *lpInData***;**
LPWORD *lpOutData***;**

The SETMITERLIMIT escape tells the driver how an application wants to clip off miter-type line joins when they become too long. It sets the current miter-limit parameter in the graphics state to a number greater than or equal to one.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpInData
Points to a 16-bit variable that specifies the miter-limit value. If this parameter is NULL, the escape does not change the miter-limit value.

lpOutData
Points to a 16-bit variable that receives the previous or current miter-limit value (if the *lpInData* parameter is NULL). If this parameter is NULL, the escape does not return to the previous limit.

Return Value

The return value is TRUE if the escape is successful. Otherwise, it is FALSE.

Comments

The miter limit controls the stroke operator's treatment of corners when miter joins have been specified. For more information about miter joins, see the SETLINEJOIN escape.

When path segments connect at a sharp angle, a miter join results in a spike that extends well beyond the connection point. The purpose of the miter limit is to cut off such spikes when they become too long.

At any given corner, the miter length is the distance from the point at which the inner edges of the strokes intersect to the point at which the outside edges of the strokes intersect (that is, the diagonal length of the miter). This distance increases as the angle between the segments decreases. If the ratio of the miter length to the line width exceeds the miter-limit parameter, the corner is treated with a bevel join instead of a miter join.

The ratio of miter length to line width is directly related to the angle *alpha* between the segments by the formula:

```
miter-length / line-width = 1 / sin(alpha/2)
```

The following are examples of miter-limit values:

- 1.415 cuts off miters at angles less than 90 degrees, and converts them to bevels.
- 2.0 cuts off miters at angles less than 60 degrees.
- 10.0 cuts off miters at angles less than 11 degrees.

The default value of the miter limit is 10. Setting the miter limit to 1 cuts off miters at all angles so that bevels are always produced even when miters are specified.

See Also SETLINEJOIN

SETPRINTERDC

#define SETPRINTERDC 9

short Control(*lpDevice***, SETPRINTERDC,** *lphdc***, NULL)**
LPPDEVICE *lpDevice*;
LPHANDLE *lphdc*;

The SETPRINTERDC escape saves the given device context for use in a subsequent call to the **OpenJob** function (GDI.240). GDI calls this escape whenever an application calls the **StartDoc** function (GDI.377) or the STARTDOC escape using the **Escape** function (GDI.38).

Printer drivers that call the **OpenJob** function to start a print job must support the SETPRINTERDC escape.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the device.

lphdc
Points to a device context handle for the application starting the print job.

Return Value The return value is TRUE if the escape is successful. Otherwise, it is FALSE.

Comments The SETPRINTERDC escape should save the device context handle in the **PDEVICE** structure and use it when calling the **OpenJob** function as part of processing the STARTDOC escape.

See Also STARTDOC

STARTDOC

#define STARTDOC 10

short Control(*lpDevice***, STARTDOC,** *lpDocName***, NULL)**
LPPDEVICE *lpDevice*;
LPSTR *lpDocName*;

The STARTDOC escape informs the device driver that a new print job is starting and that all subsequent NEWFRAME escapes should be spooled under the same job, until an ENDDOC escape occurs.

This ensures that documents longer than one page will not be interspersed with other jobs.

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpDocName
Points to a null-terminated string specifying the name of the document. The document name is displayed in Print Manager.

Return Value

The return value is positive if successful. Otherwise, it is –1 if an error occurs, such as insufficient memory or an invalid port specification.

See Also

ABORTDOC, ENDDOC, NEWFRAME

TRANSFORM_CTM

#define TRANSFORM_CTM 4107

short Control(*lpDevice***, TRANSFORM_CTM,** *lpMatrix***, NULL)**
LPPDEVICE *lpDevice*;
LPLONG *lpMatrix*;

The TRANSFORM_CTM escape modifies the current transformation matrix (CTM). The CTM controls the manner in which coordinates are translated, rotated, and scaled by the device. By using matrixes, you can combine these operations in any order to produce the desired mapping for a particular image.

The new current transformation matrix will contain the product of the matrix referenced by the *lpMatrix* parameter and the previous CTM (CTM = M * CTM).

Parameters

lpDevice
Points to a **PDEVICE** structure specifying the destination device.

lpMatrix
Points to a three-by-three array of 32-bit fixed values specifying the new transformation matrix.

Return Value

The return value is TRUE if the escape is successful. Otherwise, it is FALSE.

Comments

Applications should not make any assumptions about the initial value of the CTM.

When a driver transforms a primitive using a transformation matrix modified by the application, it should ignore the clipping rectangle specified by GDI. Applications should specify the desired clipping rectangle using the SET_CLIP_BOX escape.

Drivers supporting this escape must also implement the SET_CLIP_BOX, SAVE_CTM, and RESTORE_CTM escapes.

See Also

RESTORE_CTM, SET_CLIP_BOX, SAVE_CTM

Graphics-Driver Types and Structures

Chapter 12

This chapter describes the types and structures used by Microsoft Windows graphics-driver functions and escapes.

12.1 Types

In addition to the standard C-language data types (such as **char**, **int**, **long**, and **void**), the graphics functions and escapes use the following data types.

Type	Description
BOOL	Specifies a 16-bit Boolean value. Its value can be either TRUE or FALSE.
BYTE	Specifies an unsigned, 8-bit integer.
CHAR	Specifies an signed, 8-bit integer.
COLORREF	Specifies a 32-bit RGB color value or a logical color index. For RGB color values, the high-order byte is zero, and bytes 0, 1, and 2 represent the intensity levels of blue, green, and red, respectively. For logical color indexes, the high-order byte is 0xFF, and the low-order, 16 bits represents the index.
DWORD	Specifies a unsigned, 32-bit integer.
FARPROC	Specifies a far pointer to a function that uses the Pascal calling convention.
FIXED	Specifies a fixed, real number in 32 bits. The high-order 16 bits specifies the integer portion, and the low-order 16 bits specifies the fraction expressed as an integer value. To calculate the actual fraction value, divide the low-order 16 bits by 65536.
HANDLE	Specifies a 16-bit handle representing objects such as pens, brushes, bitmaps, and global memory.
HWND	Specifies a 16-bit window handle.
INT	Specifies a signed, 16-bit integer.
LONG	Specifies a signed, 32-bit integer.
LPPDEVICE	Specifies a far pointer to a **PDEVICE** structure.
LPSTR	Specifies a far pointer to an array of bytes.
LPVOID	Specifies a far pointer to an undetermined type. Function parameters having this type must be cast to a specific far pointer type before being used.
PBRUSH	Specifies a integer, an array, or a structure containing device-specific information about a physical brush that a driver can use to fill figures and draw scan lines. The exact size and content of a **PBRUSH** type depends entirely on the driver.
PCOLOR	Specifies a 32-bit integer representing physical-color values. A physical color specifies a given color on the device. The range and meaning of physical color values depends entirely on the driver.

Type	Description
PPEN	Specifies an integer, array, or structure containing device-specific information that a driver can use to draw lines and borders. The exact size and content of a physical pen depend entirely on the driver.
SHORT	Specifies a signed, 16-bit integer.
VOID	Specifies an empty type. This type is typically used with functions that return no value.
WORD	Specifies an unsigned, 16-bit integer.

When **LP** prefix is applied to a data type, the resulting type specifies a far pointer to a variable having the specified data type.

12.2 Structures

The following is an alphabetical listing of the structures used by the graphics drivers and functions. All structures must be packed. This means that you cannot align structure members on 16-bit boundaries by default.

BANDINFOSTRUCT

```
typedef struct _BANDINFOSTRUCT {
    BOOL fGraphics;
    BOOL fText;
    RECT rcGraphics;
} BANDINFOSTRUCT;
```

The **BANDINFOSTRUCT** structure, used by banding drivers, specifies whether graphics and text are on the page.

Members

fGraphics
Specifies whether graphics are on the page. It is nonzero if graphics are on the page; zero if not.

fText
Specifies whether text is on the page. It is nonzero if text is on the page; zero if not.

rcGraphics
Specifies a **RECT** structure that contains the coordinates for the rectangle bounding all nontext graphics on the page.

Comments

A driver receives **BANDINFOSTRUCT** structures from applications that call the BANDINFO escape. Information in the structure helps the driver optimize the banding process. For example, if there are no graphics, the driver may be able to skip the graphics bands. If the bounding rectangle for graphics is smaller than the page, the driver has the option of banding only the specified graphics rectangle rather than the whole page.

See Also

BANDINFO, NEXTBAND

BININFO

```
typedef struct tagBININFO {
    short   BinNumber;
    short   NbrofBins;
    short   Reserved[4];
} BININFO;
```

The **BININFO** structure contains information about a printer's paper bins.

Members

BinNumber
Identifies the current or previous paper bin.

NbrofBins
Specifies the number of paper bins available.

Reserved
Reserved; do not use.

See Also

GETSETPAPERBINS

BINNAMES

```
typedef struct tagBINNAMES {
    short BinList[CBINMAX];
    char PaperNames[CBINMAX][CCHBINNAME]
} BINNAMES;
```

The **BINNAMES** structure contains paper-bin identifiers and names. The structure consists of two arrays: an array of 16-bit values specifying the paper-bin identifiers, and an array of paper-bin names.

Members

BinList
Specifies an array of 16-bit values specifying the paper-bin identifiers. The number of elements in the array (CBINMAX) must be specified by the *lpInData* parameter of the ENUMPAPERBINS escape.

PaperNames
Specifies an array of null-terminated paper-bin names. The number of elements in the array (CBINMAX) must be specified by the *lpInData* parameter of the ENUMPAPERBINS escape. The maximum number of characters in each name (CCHBINNAME) is 24.

See Also

ENUMPAPERBINS

BITMAPINFO

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

The **BITMAPINFO** structure fully defines the dimensions and color information for a Windows 3.*x* device-independent bitmap.

Members

bmiHeader
Specifies a **BITMAPINFOHEADER** structure that contains information about the dimensions and color format of a device-independent bitmap.

bmiColors
Specifies an array of **RGBQUAD** structures that define the colors in the bitmap.

Comments

A Windows 3.*x* device-independent bitmap consists of two distinct parts: a **BITMAPINFO** data structure that describes the dimensions and colors of the bitmap, and an array of bytes that define the pixels of the bitmap. The bits in the array are packed together, but each scan line must be padded with zeros to end on a 32-bit boundary. Segment boundaries can appear anywhere in the bitmap, however. The origin of the bitmap is the lower-left corner.

The **biBitCount** member of the **BITMAPINFOHEADER** structure determines the number of bits which define each pixel and the maximum number of colors in the bitmap. This member may be set to any of the following values.

Value	Meaning
1	The bitmap is monochrome, and the **bmiColors** member must contain two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the **bmiColors** member; if the bit is set, the pixel has the color of the second entry in the table.
4	The bitmap has a maximum of 16 colors, and the **bmiColors** member contains up to 16 entries. Each pixel in the bitmap is represented by a 4-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, then the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the sixteenth table entry.
8	The bitmap has a maximum of 256 colors, and the **bmiColors** member contains up to 256 entries. In this case, each byte in the array represents a single pixel.
24	The bitmap has a maximum of 2**24 colors. The **bmiColors** member is NULL, and each three bytes in the bitmap array represents the relative intensities of blue, green, and red, respectively, of a pixel.

Alternatively, for functions that use device-independent bitmaps, the **bmiColors** member can be an array of 16-bit unsigned integers that specify an index into the currently realized, logical palette instead of explicit RGB values. In this case, an application using the bitmap must call device-independent bitmap functions with the *wColorUse* parameter set to DIB_PAL_COLORS.

See Also **DeviceBitmapBits, SetDIBitsToDevice, StretchDIBits**

BITMAPINFOHEADER

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

The **BITMAPINFOHEADER** structure contains information about the dimensions and color format of a Windows 3.*x* device-independent bitmap.

Members

biSize
Specifies the number of bytes required by the **BITMAPINFOHEADER** structure.

biWidth
Specifies the width of the bitmap in pixels.

biHeight
Specifies the height of the bitmap in pixels.

biPlanes
Specifies the number of planes for the target device and must be set to 1.

biBitCount
Specifies the number of bits per pixel. This value must be 1, 4, 8, or 24.

biCompression
Specifies the type of compression for a compressed bitmap. It can be one of the following values.

Value	Meaning
BI_RGB	Specifies that the bitmap is not compressed.
BI_RLE8	Specifies a run-length encoded (RLE) format for bitmaps with 8 bits per pixel. The compression format is a 2-byte format consisting of a count byte followed by a byte containing a color index.
BI_RLE4	Specifies a run-length encoded format for bitmaps with 4 bits per pixel. The compression format is a two-byte format consisting of a count byte followed by two word-length color indexes.

biSizeImage
Specifies the size in bytes of the image. It is valid to set this member to zero if the bitmap is in the BI_RGB format. The size must then be calculated explicitly.

biXPelsPerMeter
Specifies the horizontal resolution in pixels-per-meter of the target device for the bitmap. An application can use this value to select a bitmap from a resource group that best matches the characteristics of the current device.

biYPelsPerMeter
Specifies the vertical resolution in pixels-per-meter of the target device for the bitmap.

biClrUsed
Specifies the number of color indexes in the color table actually used by the bitmap. If this value is zero, the bitmap uses the maximum number of colors corresponding to the value of the **biBitCount** member.

biClrImportant
Specifies the number of color indexes that are considered important for displaying the bitmap. If this value is zero, then all colors are important.

Comments

The **BITMAPINFO** structure combines the **BITMAPINFOHEADER** structure and a color table to provide a complete definition of the dimensions and colors of a Windows 3.*x* device-independent bitmap.

Bitmap-Compression Formats

Windows supports formats for compressing bitmaps that define their colors with 8 bits per pixel and with 4 bits per pixel. Compression reduces the disk and memory storage required for the bitmap.

When the **biCompression** member is set to BI_RLE8, the bitmap is compressed using a run-length encoding format for an 8-bit bitmap. This format may be compressed in either of two modes:

- Encoded
- Absolute

Both modes can occur anywhere throughout a single bitmap.

Encoded mode consists of two bytes: the first byte specifies the number of consecutive pixels to be drawn using the color index contained in the second byte. In addition, the first byte of the pair can be set to zero to indicate an escape that denotes an end of line, end of bitmap, or a delta. The interpretation of the escape depends on the value of the second byte of the pair. The following list shows the meaning of the second byte.

Value	Meaning
0	End of line.
1	End of bitmap.
2	Delta. The two bytes following the escape contain unsigned values indicating the horizontal and vertical offset of the next pixel from the current position.

Absolute mode is signaled by the first byte set to zero and the second byte set to a value between 03H and 0FFH. In absolute mode, the second byte represents the number of bytes which follow, each of which contains the color index of a single pixel. When the second byte is set to 2 or less, the escape has the same meaning as in encoded mode. In absolute mode, each run must be aligned on a 16-bit boundary.

The following example shows the hexadecimal values of an 8-bit compressed bitmap:

```
03 04 05 06 00 03 45 56 67 00 02 78 00 02 05 01
02 78 00 00 09 1E 00 01
```

This bitmap would expand as follows (two-digit values represent a color index for a single pixel):

```
04 04 04
06 06 06 06 06
45 56 67
78 78
move current position 5 right and 1 down
78 78
end of line
1E 1E 1E 1E 1E 1E 1E 1E 1E
end of RLE bitmap
```

When the **biCompression** member is set to BI_RLE4, the bitmap is compressed using a run-length encoding format for a 4-bit bitmap, which also uses encoded and absolute modes. In encoded mode, the first byte of the pair contains the number of pixels to be drawn using the color indexes in the second byte. The second byte contains two color indexes, one in its high-order nibble (that is, its low-order four bits) and one in its low-order nibble. The first of the pixels is drawn using the color specified by the high-order nibble, the second is drawn using the color in the low-order nibble, the third is drawn with the color in the high-order nibble, and so on, until all the pixels specified by the first byte have been drawn.

In absolute mode, the first byte contains zero, the second byte contains the number of color indexes that follow, and subsequent bytes contain color indexes in their high- and low-order nibbles, one color index for each pixel. In absolute mode, each run must be aligned on a word boundary. The end-of-line, end-of-bitmap, and delta escapes also apply to BI_RLE4.

The following example shows the hexadecimal values of a 4-bit compressed bitmap:

```
03 04 05 06 00 06 45 56 67 00 04 78 00 02 05 01
04 78 00 00 09 1E 00 01
```

This bitmap would expand as follows (single-digit values represent a color index for a single pixel):

```
0 4 0
0 6 0 6 0
4 5 5 6 6 7
7 8 7 8
move current position 5 right and 1 down
7 8 7 8
end of line
1 E 1 E 1 E 1 E 1
end of RLE bitmap
```

CHARRANGE

```
typedef struct tagCHARRANGE {
    BYTE chFirst;
    BYTE chLast;
} CHARRANGE;
```

The **CHARRANGE** structure contains character codes for the first and last characters in a range of characters.

Members

chFirst
Specifies the character code of the first character.

chLast
Specifies the character code of the last character.

See Also GETEXTENTTABLE

COLORTABLE_STRUCT

```
typedef struct tagCOLORTABLE_STRUCT {
    WORD Index;
    LONG rgb;
} COLORTABLE_STRUCT;
```

The **COLORTABLE_STRUCT** structure contains color information for an entry in the color table.

Members

Index
Specifies the color-table index. Color table entries start at zero for the first entry.

rgb
Specifies an RGB color value.

See Also SETCOLORTABLE, GETCOLORTABLE

DEVMODE

```
typedef struct _devicemode {     /* dm */
    char  dmDeviceName[CCHDEVICENAME];
    WORD  dmSpecVersion;
    WORD  dmDriverVersion;
    WORD  dmSize;
    WORD  dmDriverExtra;
    DWORD dmFields;
    short dmOrientation;
    short dmPaperSize;
    short dmPaperLength;
    short dmPaperWidth;
    short dmScale;
    short dmCopies;
    short dmDefaultSource;
    short dmPrintQuality;
    short dmColor;
    short dmDuplex;
    short dmYResolution;
    short dmTTOption;
} DEVMODE;
```

The **DEVMODE** structure contains information about a printer driver's initialization and environment. An application passes this structure to the **DeviceCapabilities** and **ExtDeviceMode** functions.

Members

dmDeviceName
Specifies the name of the device the driver supports—for example, "HP LaserJet III" in the case of the Hewlett-Packard LaserJet III.

dmSpecVersion
Specifies the version number of the **DEVMODE** structure. For Windows version 3.1, this value should be 0x30A.

dmDriverVersion
Specifies the assigned printer driver version number.

dmSize
Specifies the size, in bytes, of the **DEVMODE** structure. (This value does not include the optional **dmDriverData** member for device-specific data, which can follow the structure.) If an application manipulates only the driver-independent portion of the data, it can use this member to find out the length of the structure without having to account for different versions.

dmDriverExtra
Specifies the size, in bytes, of the optional **dmDriverData** member for device-specific data, which can follow the structure. If an application does not use device-specific information, it should set this member to zero.

dmFields
Specifies a value that indicates which of the remaining members in the **DEVMODE** structure have been initialized. It can be any combination (or it can be none) of the following values.

Constant	Value
DM_ORIENTATION	0x0000001L
DM_PAPERSIZE	0x0000002L
DM_PAPERLENGTH	0x0000004L
DM_PAPERWIDTH	0x0000008L
DM_SCALE	0x0000010L
DM_COPIES	0x0000100L
DM_DEFAULTSOURCE	0x0000200L
DM_PRINTQUALITY	0x0000400L
DM_COLOR	0x0000800L
DM_DUPLEX	0x0001000L
DM_YRESOLUTION	0x0002000L
DM_TTOPTION	0x0004000L

A printer driver supports only those members that are appropriate for the printer technology.

dmOrientation
Specifies the orientation of the paper. It can be either DMORIENT_PORTRAIT or DMORIENT_LANDSCAPE.

dmPaperSize
Specifies the size of the paper to print on. This member may be set to zero if the length and width of the paper are specified by the **dmPaperLength** and **dmPaperWidth** members, respectively. Otherwise, the **dmPaperSize** member can be set to one of the following predefined values.

Value	Meaning
DMPAPER_10x14	10 x 14 inches
DMPAPER_11x17	11 x 17 inches
DMPAPER_A3	A3 297 x 420 millimeters
DMPAPER_A4	A4 210 x 297 millimeters
DMPAPER_A4_EXTRA	A4 Extra 9.27 x 12.69 inches
DMPAPER_A4_TRANSVERSE	Transverse 297 x 210 millimeters
DMPAPER_A4SMALL	A4 Small 210 x 297 millimeters
DMPAPER_A5	A5 148 x 210 millimeters
DMPAPER_B4	B4 250 x 354 millimeters
DMPAPER_B5	B5 182 x 257 millimeters

Value	Meaning
DMPAPER_CSHEET	C size sheet
DMPAPER_DSHEET	D size sheet
DMPAPER_ENV_10	Envelope #10 4.125 x 9.5 inches
DMPAPER_ENV_11	Envelope #11 4.5 x 10.375 inches
DMPAPER_ENV_12	Envelope #12 4.75 x 11 inches
DMPAPER_ENV_14	Envelope #14 5 x 11.5 inches
DMPAPER_ENV_9	Envelope #9 3.875 x 8.875 inches
DMPAPER_ENV_C5	Envelope C5 162 x 229 millimeters
DMPAPER_ENV_DL	Envelope DL 110 x 220 millimeters
DMPAPER_ENV_MONARCH	Envelope Monarch 3.875 x 7.5 inches
DMPAPER_ESHEET	E size sheet
DMPAPER_EXECUTIVE	Executive 7.25 x 10.5 inches
DMPAPER_FIRST	Letter 8.5 x 11 inches
DMPAPER_FOLIO	Folio 8.5 x 13 inches
DMPAPER_LAST	Letter Extra Transverse 12 x 9.5 inches
DMPAPER_LEDGER	Ledger 11 x 17 inches
DMPAPER_LEGAL	Legal 8.5 x 14 inches
DMPAPER_LEGAL_EXTRA	Legal Extra 9.5 x 15 inches
DMPAPER_LETTER	Letter 8.5 x 11 inches
DMPAPER_LETTER_EXTRA	Letter Extra 9.5 x 12 inches
DMPAPER_LETTER_EXTRA_TRANSVERSE	Letter Extra Transverse 12 x 9.5 inches
DMPAPER_LETTER_TRANSVERSE	Letter Transverse 11 x 8.5 inches
DMPAPER_LETTERSMALL	Letter Small 8.5 x 11 inches
DMPAPER_NOTE	Note 8.5 x 11 inches
DMPAPER_QUARTO	Quarto 215 x 275 millimeters
DMPAPER_STATEMENT	Statement 5.5 x 8.5 inches
DMPAPER_TABLOID	Tabloid 11 x 17 inches
DMPAPER_TABLOID_EXTRA	Tabloid Extra 11.69 x 18 inches
DMPAPER_USER	User defined

dmPaperLength
Specifies a paper length, in tenths of a millimeter. This member overrides the paper length specified by the **dmPaperSize** member, either for custom paper sizes or for such devices as dot-matrix printers that can print on a variety of page sizes.

dmPaperWidth
Specifies a paper width, in tenths of a millimeter. This member overrides the paper width specified by the **dmPaperSize** member.

dmScale
Specifies the factor by which the printed output is to be scaled. The apparent page size is scaled from the physical page size by a factor of **dmScale**/100. For example, a letter-size paper with a **dmScale** value of 50 would contain as much data as a page of size 17x22 inches because the output text and graphics would be half their original height and width.

dmCopies
Specifies the number of copies printed if the device supports multiple-page copies.

dmDefaultSource
Specifies the default bin from which the paper is fed. The application can override this value by using the GETSETPAPERBINS escape. This member can be one of the following values.

DMBIN_AUTO
DMBIN_CASSETTE
DMBIN_ENVELOPE
DMBIN_ENVMANUAL
DMBIN_FIRST
DMBIN_LARGECAPACITY
DMBIN_LARGEFMT
DMBIN_LAST
DMBIN_LOWER
DMBIN_MANUAL
DMBIN_MIDDLE
DMBIN_ONLYONE
DMBIN_SMALLFMT
DMBIN_TRACTOR
DMBIN_UPPER

A range of values is reserved for device-specific bins. To be consistent with initialization information, the GETSETPAPERBINS and ENUMPAPERBINS escapes use these values.

dmPrintQuality
Specifies the printer resolution. Following are the four predefined device-independent values:

DMRES_HIGH (–4)
DMRES_MEDIUM (–3)
DMRES_LOW (–2)
DMRES_DRAFT (–1)

If a positive value is given, it specifies the number of dots per inch (DPI) and is therefore device dependent.

If the printer initializes the **dmYResolution** member, the **dmPrintQuality** member specifies the *x*-resolution of the printer, in dots per inch.

dmColor
Specifies whether a color printer is to render color or monochrome output. Possible values are:

DMCOLOR_COLOR (1)
DMCOLOR_MONOCHROME (2)

dmDuplex
Specifies duplex (double-sided) printing for printers capable of duplex printing. This member can be one of the following values:

DMDUP_SIMPLEX (1)
DMDUP_HORIZONTAL (2)
DMDUP_VERTICAL (3)

dmYResolution
Specifies the *y*-resolution of the printer, in dots per inch. If the printer initializes this member, the **dmPrintQuality** member specifies the *x*-resolution of the printer, in dots per inch.

dmTTOption
Specifies how TrueType fonts should be printed. It can be one of the following values.

Value	Meaning
DMTT_BITMAP	Prints TrueType fonts as graphics. This is the default action for dot-matrix printers.
DMTT_DOWNLOAD	Downloads TrueType fonts as soft fonts. This is the default action for Hewlett-Packard printers that use Printer Control Language (PCL).
DMTT_SUBDEV	Substitutes device fonts for TrueType fonts. This is the default action for PostScript printers.

Comments

An application can retrieve the paper sizes and names supported by a printer by calling the **DeviceCapabilities** function with the DC_PAPERS, DC_PAPERSIZE, and DC_PAPERNAMES values.

Before setting the value of the **dmTTOption** member, applications should find out how a printer driver can use TrueType fonts by calling the **DeviceCapabilities** function with the DC_TRUETYPE value.

Drivers can add device-specific data immediately following the **DEVMODE** structure.

See Also

DeviceCapabilities, ExtDeviceMode

DRAWMODE

```
typedef struct tagDRAWMODE {
  short    Rop2;       /*binary-raster operations*/
  short    bkMode;     /*background mode*/
  PCOLOR   bkColor;    /*physical background color*/
  PCOLOR   TextColor;  /*physical text (foreground) color*/
  short    TBreakExtra;/*number of extra pixels to add to line*/
  short    BreakExtra; /*pixels per break: TBreakExtra/BreakCount*/
  short    BreakErr;   /*running error term*/
  short    BreakRem;   /*remaining pixels: TBreakExtra%BreakCount*/
  short    BreakCount; /*number of breaks in the line*/
  short    CharExtra;  /*extra pixels for each character*/
  COLORREF LbkColor;   /*logical background color*/
  COLORREF LTextColor; /*logical text (foreground) color*/
} DRAWMODE;
```

The **DRAWMODE** structure contains information used during output, such as drawing lines, filling interiors, and writing text.

Members

Rop2

Specifies a binary-raster operation value. The value, in the range 1 to 16, determines how to combine source and destination colors. This member can be one of the following values.

Value	Meaning
R2_BLACK (1)	Black: 0.
R2_NOTMERGEPEN (2)	Inverse of the bitwise OR of the source and destination colors: NOT (Source OR Dest).
R2_MASKNOTPEN (3)	Bitwise AND of the destination and the inverse of the source: Dest AND (NOT Source).
R2_NOTCOPYPEN (4)	Inverse of the source color: NOT Source.
R2_MASKPENNOT (5)	Bitwise AND of the source and the inverse of the destination: Source AND (NOT Dest).
R2_NOT (6)	Inverse of the destination color: NOT Dest.
R2_XORPEN (7)	Bitwise exclusive OR of the destination and source: Dest XOR Source.
R2_NOTMASKPEN (8)	Inverse of the bitwise AND of the destination and source colors: NOT (Dest AND Source).
R2_MASKPEN 9	Bitwise AND of the destination and source colors: Dest AND Source.
R2_NOTXORPEN (10)	Inverse of the bitwise exclusive OR of the destination and source colors: NOT (Dest XOR Source).
R2_NOP 11	Destination color: Dest.

Value	Meaning
R2_MERGENOTPEN (12)	Bitwise OR of the destination and the inverse of the source: Dest AND (NOT Source).
R2_COPYPEN (13)	Source color: Source.
R2_MERGEPENNOT (14)	Bitwise OR of the source and the inverse of the destination: Source OR (NOT Dest).
R2_MERGEPEN (15)	Bitwise OR of the source and the destination: Source OR Dest.
R2_WHITE (16)	White: 1.

bkMode
Specifies whether the background for styled lines, hatched brushes, brushes used for interiors and scan lines, bitmaps, and text is given the current background color or left unchanged. This member can be one of the following values.

Value	Meaning
TRANSPARENT (1)	Leaves destination background unchanged.
OPAQUE (2)	Replaces destination background with the color specified by the **BackgroundColor** member.
TRANSPARENT1 (4)	Leaves destination background unchanged, but before copying the source to destination, removes pixels from the source that have the current background color.

bkColor
Contains a physical color value specifying the background color.

TextColor
Contains a physical color value specifying the text (foreground) color.

TBreakExtra
Specifies the total amount of space (in pixels) to add to the break characters in a line of text. **TBreakExtra** is set to zero if no justification is required.

BreakExtra
Specifies the amount of space (in pixels) to add to each break character in a line of text. This value is equal to **TBreakExtra** divided by **BreakCount**.

BreakErr
Specifies the running error term: the amount of space (in pixels) not yet added to break characters in a line of text. This member is used in conjunction with the **BreakRem** member to determine which break characters receive the additional pixels specified by **BreakRem**. Initially, **BreakErr** is set to (**BreakCount**)/2+1.

BreakRem
Specifies the amount of space (in pixels) to add to one or more break characters in a line of text. This space is in addition to any extra space specified by the **BreakExtra** member and is intended to be distributed evenly across the line. The **BreakRem** value is equal to the remainder after dividing **TBreakExtra** by **BreakCount**.

BreakCount
Specifies the number of break characters in a line of text. If the **TBreakExtra** member is not zero, each break character must be drawn wider than its normal width. The **BreakExtra** member specifies the extra width (in pixels). The **BreakRem** member may also specify additional pixels for one or more break characters.

CharExtra
Specifies amount of space (in pixels) to add between characters in a line of text.

LbkColor
Contains an **COLORREF** value specifying the logical background color.

LTextColor
Contains an **COLORREF** value specifying the logical text (foreground) color.

See Also

BitBlt, DeviceBitmapBits, ExtTextOut, Output, Pixel, SetDIBitsToDevice, StrBlt, StretchBlt, StretchDIBits

EXTTEXTDATA

```
typedef struct tagEXTTEXTDATA {
    short             nSize;
    LPAPPEXTTEXTDATA  lpInData;
    LPFONTINFO        lpFont;
    LPTEXTXFORM       lpXForm;
    LPDRAWMODE        lpDrawMode;
} EXTTEXTDATA;
```

The **EXTTEXTDATA** structure contains a complete set of information describing the text to be drawn.

Members

nSize
Specifies the size in bytes of the structure.

lpInData
Points to a 16-bit variable that contains the number of bytes pointed to by the *lpOutData* parameter in an GETEXTENDEDTEXTMETRICS escape.

lpFont

Points to a **FONTINFO** structure specifying a physical font. The **FONTINFO** structure has the following form:

```
typedef struct tagFONTINFO {
    short dfType;
    short dfPoints;
    short dfVertRes;
    short dfHorizRes;
    short dfAscent;
    short dfInternalLeading;
    short dfExternalLeading;
    char  dfItalic;
    char  dfUnderline;
    char  dfStrikeOut;
    short dfWeight;
    char  dfCharSet;
    short dfPixWidth;
    short dfPixHeight;
    char  dfPitchAndFamily;
    short dfAvgWidth;
    short dfMaxWidth;
    char  dfFirstChar;
    char  dfLastChar;
    char  dfDefaultChar;
    char  dfBreakChar;
    short dfWidthBytes;
    long  dfDevice;
    long  dfFace;
    long  dfBitsPointer;
    long  dfBitsOffset;
    char  dfReserved;
    /* The following fields present only for Windows 3.x fonts */
    long  dfFlags;
    short dfAspace;
    short dfBspace;
    short dfCspace;
    long  dfColorPointer;
    long  dfReserved1[4];
} FONTINFO;
```

lpXForm

Points to a **TEXTXFORM** structure specifying additional attributes of the text. The **TEXTXFORM** structure has the following form:

```
typedef struct tagTEXTXFORM {
    short  txfHeight;
    short  txfWidth;
    short  txfEscapement;
    short  txfOrientation;
    short  txfWeight;
    char   txfItalic;
```

```
    char   txfUnderline;
    char   txfStrikeOut;
    char   txfOutPrecision;
    char   txfClipPrecision;
    short  txfAccelerator;
    short  txfOverhang;
} TEXTXFORM;
```

lpDrawMode
Points to a **DRAWMODE** structure specifying information used to draw the text. The **DRAWMODE** structure has the following form:

```
typedef struct tagDRAWMODE {
  short    Rop2;       /*binary-raster operations*/
  short    bkMode;     /*background mode*/
  PCOLOR   bkColor;    /*physical background color*/
  PCOLOR   TextColor;  /*physical text (foreground) color*/
  short    TBreakExtra;/*number of extra pixels to add to line*/
  short    BreakExtra; /*pixels per break: TBreakExtra/BreakCount*/
  short    BreakErr;   /*running error term*/
  short    BreakRem;   /*remaining pixels: TBreakExtra%BreakCount*/
  short    BreakCount; /*number of breaks in the line*/
  short    CharExtra;  /*extra pixels for each character*/
  COLORREF LbkColor;   /*logical background color*/
  COLORREF LTextColor; /*logical text (foreground) color*/
} DRAWMODE;
```

See Also **DRAWMODE**, **FONTINFO**, GETEXTENDEDTEXTMETRICS, TEXTXFORM

EXTTEXTMETRIC

```
typedef struct tagEXTTEXTMETRIC {
    short  etmSize;
    short  etmPointSize;
    short  etmOrientation;
    short  etmMasterHeight;
    short  etmMinScale;
    short  etmMaxScale;
    short  etmMasterUnits;
    short  etmCapHeight;
    short  etmXHeight;
    short  etmLowerCaseAscent;
    short  etmUpperCaseDescent;
    short  etmSlant;
    short  etmSuperScript;
    short  etmSubScript;
    short  etmSuperScriptSize;
```

```
    short  etmSubScriptSize;
    short  etmUnderlineOffset;
    short  etmUnderlineWidth;
    short  etmDoubleUpperUnderlineOffset;
    short  etmDoubleLowerUnderlineOffset;
    short  etmDoubleUpperUnderlineWidth;
    short  etmDoubleLowerUnderlineWidth;
    short  etmStrikeOutOffset;
    short  etmStrikeOutWidth;
    WORD   etmKernPairs;
    WORD   etmKernTracks;
} EXTTEXTMETRIC;
```

The **EXTTEXTMETRIC** contains extended information about a font.

Members

etmSize
Specifies the size (in bytes) of the structure.

etmPointSize
Specifies the point size of the font.

etmOrientation
Specifies the orientation.

etmMasterHeight
Specifies the master height.

etmMinScale
Specifies the smallest reasonable scaling factor for the font.

etmMaxScale
Specifies the largest reasonable scaling factor for the font.

etmMasterUnits
Specifies the master units.

etmCapHeight
Specifies the height of capital letters.

etmXHeight
Specifies a representative height for the font.

etmLowerCaseAscent
Specifies the ascent height for lowercase letters.

etmUpperCaseDescent
Specifies the ascent height for uppercase letters.

etmSlant
Specifies the slant of characters in the font.

etmSuperScript
Specifies whether the font supports superscripts.

etmSubScript
Specifies whether the font supports subscripts.

etmSuperScriptSize
Specifies the size of the superscript characters.

etmSubScriptSize
Specifies the size of the subscript characters.

etmUnderlineOffset
Specifies the offset from the baseline to the underline.

etmUnderlineWidth
Specifies the width of an underline.

etmDoubleUpperUnderlineOffset
Specifies the offset from the baseline to the upper portion of a double underline.

etmDoubleLowerUnderlineOffset
Specifies the offset from the baseline to the lower portion of a double underline.

etmDoubleUpperUnderlineWidth
Specifies the width of the upper portion of a double underline.

etmDoubleLowerUnderlineWidth
Specifies the width of the lower portion of a double underline.

etmStrikeOutOffset
Specifies the offset from the baseline of the strikeout line.

etmStrikeOutWidth
Specifies the width of the strikeout line.

etmKernPairs
Specifies the number of kerning pairs.

etmKernTracks
Specifies the number of kerning tracks.

See Also GETEXTENDEDTEXTMETRICS

FONTINFO

```
typedef struct tagFONTINFO {
    short dfType;
    short dfPoints;
    short dfVertRes;
    short dfHorizRes;
    short dfAscent;
    short dfInternalLeading;
    short dfExternalLeading;
    char  dfItalic;
    char  dfUnderline;
```

```
    char  dfStrikeOut;
    short dfWeight;
    char  dfCharSet;
    short dfPixWidth;
    short dfPixHeight;
    char  dfPitchAndFamily;
    short dfAvgWidth;
    short dfMaxWidth;
    char  dfFirstChar;
    char  dfLastChar;
    char  dfDefaultChar;
    char  dfBreakChar;
    short dfWidthBytes;
    long  dfDevice;
    long  dfFace;
    long  dfBitsPointer;
    long  dfBitsOffset;
    char  dfReserved;
    /* The following fields present only for Windows 3.x fonts */
    long  dfFlags;
    short dfAspace;
    short dfBspace;
    short dfCspace;
    long  dfColorPointer;
    long  dfReserved1[4];
} FONTINFO;
```

The **FONTINFO** structure contains information about a physical font. Depending on whether the font is realized by GDI or by a device driver, the **FONTINFO** structure may be immediately followed by a character width table and by font bitmap or vector information.

The **FONTINFO** structure contains optional members (**dfFlags** through **dfReserved1**) that are present only if the font has been designed for Windows 3.*x*. If GDI realizes a font for a driver, the font's corresponding **FONTINFO** structure will not include these optional members unless the RC_BIGFONT bit is set in the **dpRaster** member of the driver's **GDIINFO** structure.

Members

dfType

Specifies font type. The low-order byte, reserved for exclusive GDI use, is a combination of the following values.

Value	Meaning
PF_RASTER_TYPE (0x0000)	Font is a raster font.
PF_VECTOR_TYPE (0x0001)	Font is a vector font.
PF_BITS_IS_ADDRESS (0x0004)	Indicates that the **dfBitsOffset** member specifies the absolute memory address of the font bitmap or vector information.

Value	Meaning
PF_DEVICE_REALIZED (0x0080)	Font has been realized by the device driver.

All other values in the low-order byte are reserved. In particular, the value 0x0008 in the **dfType** member is reserved for use with Asian fonts.

The high-order byte is reserved for device use. GDI never inspects the high byte. If GDI realizes the font, it sets this byte to zero. If the device driver realizes the font, it can set this byte to any value.

dfPoints
Specifies the point size at which this character set looks best.

dfVertRes
Specifies the vertical resolution (dots-per-inch) at which this character set was digitized.

dfHorizRes
Specifies the horizontal resolution (dots-per-inch) at which this character set was digitized.

dfAscent
Specifies the distance from the top of a character definition cell to the baseline of the typographical font. It is useful for aligning the baseline of fonts of different heights.

dfInternalLeading
Specifies the amount of leading inside the bounds set by the **dfPixHeight** member. Accent marks may occur in this area.

dfExternalLeading
Specifies the amount of extra leading that the designer requests the application add between rows. Since this area is outside of the font proper, it contains no marks and will not be altered by text output calls in either the OPAQUE or TRANSPARENT mode.

dfItalic
Specifies whether the character-definition data represents an italic font. The low-order bit is 1 if the flag is set. All other bits are zero.

dfUnderline
Specifies whether the character-definition data represents an underlined font. The low-order bit is 1 if the flag is set. All other bits are zero.

dfStrikeOut
Specifies whether the character definition data represents a struck-out font. The low-order bit is 1 if the flag is set. All other bits are zero.

dfWeight
Specifies the weight of the characters in the character definition data, on a scale from 1–1000. A value of 400 specifies regular weight type; 700 is bold; and so on.

dfCharSet
Specifies the character set defined by this font. It can be one of the following values.

Value	Meaning
0	ANSI character set
2	Symbol character set
255	OEM hardware font

dfPixWidth
Specifies the width of all characters in the font.

For vector fonts, the **dfPixWidth** member is the width of the grid on which the font was digitized.

For raster fonts, **dfPixWidth** is the width (in pixels) of each character bitmap. If that member is zero, the font has variable-width characters and these widths are specified in the character-width table immediately following this structure.

dfPixHeight
Specifies the height of all characters in the font.

For vector fonts, the **dfPixHeight** member is the height of the grid on which the font was digitized.

For raster fonts, **dfPixHeight** is the height (in scan lines) of each character bitmap.

dfPitchAndFamily
Specifies the pitch and font family. The pitch specifies whether the characters in the font have the same width or variable widths. The font family indicates, in a general way, the look of a font.

The **dfPitchAndFamily** member can be a combination of the following values.

Value	Meaning
0x01	Variable-pitch font. If this value is not given, the font is fixed pitch.
FF_ROMAN (0x10)	Proportionally spaced fonts with serifs.
FF_SWISS (0x20)	Proportionally spaced fonts without serifs.
FF_MODERN (0x30)	Fixed-pitch fonts.
FF_SCRIPT (0x40)	Cursive or script fonts.
FF_DECORATIVE (0x50)	Novelty fonts.

If the high-order 4 bits is set to FF_DONTCARE (0x00), the font belongs to no specific family.

dfAvgWidth
Specifies the width of characters in the font. For fixed-pitch fonts, this is the same as **dfPixWidth**. For variable-pitched fonts, this is the width of the character "X."

dfMaxWidth
Specifies the maximum pixel width of any character in the font. For fixed-pitch fonts, this is simply **dfPixWidth**.

dfFirstChar
Specifies the first character code defined by this font. Character definitions are stored only for the characters actually present in a font, so this field should be used when calculating indexes into the character-width table following this structure.

dfLastChar
Specifies the last character code defined by this font. Notice that all the characters with codes between the **dfFirstChar** and **dfLastChar** member must be present in the character-width table.

dfDefaultChar
Specifies the default character. A device driver uses this character as a substitute for any character in a string that is out of the range of the **dfFirstChar** through **dfLastChar** members. The character is given relative to **dfFirstChar** so that the actual value of the default character is the sum of **dfDefaultChar** and **dfFirstChar**. Ideally, the **dfDefaultChar** member should be a visible character in the current font, for example, a period (.).

dfBreakChar
Specifies the word-break character. Applications use this character the separate words when wrapping or justifying lines of text. The character is given relative to **dfFirstChar** so that the actual value of the word-break character is the sum of the **dfBreakChar** and **dfFirstChar** members. In many fonts, **dfBreakChar** is zero and **dfFirstChar** is 32. This means that the word-break character value is 32, an ASCII space.

dfWidthBytes
Specifies the number of bytes in each row of the font bitmap (raster fonts). This field is not used for vector fonts. The **dfWidthBytes** member is always an even quantity so that rows of the bitmap start on 16-bit boundaries.

dfDevice
Specifies the offset from the beginning of the segment containing the **FONTINFO** structure to the null-terminated ASCII string specifying the device name. For a generic font, this value will be NULL.

dfFace
Specifies the offset from the beginning of the segment containing the **FONTINFO** structure to the null-terminated ASCII string specifying the name of the font face.

dfBitsPointer
Specifies the absolute machine address of the bitmap. This is set by GDI. The **dfBitsPointer** member is guaranteed to be even.

dfBitsOffset
Specifies the offset from the beginning of the segment containing the **FONTINFO** structure to the beginning of the bitmap information.

If the PF_BITS_IS_ADDRESS bit is set in **dfType**, **dfBitsOffset** is an absolute address of the bitmap or vector information. For example, this bit is set if the font bitmap or vector information is in ROM.

For raster fonts, **dfBitsOffset** points to a sequence of bytes that make up the bitmaps for each character in the font.

For vector fonts, **dfBitsOffset** points to a string of bytes or words (depending on the size of the grid on which the font was digitized) that specifies the strokes for each character of the font. The **dfBitsOffset** member must be even.

dfReserved
Reserved; do not use. This member is present only for raster fonts. In Windows 2.*x* fonts, this member ensures that the character-width table (which immediately follows this member) starts on a 16-bit boundary.

dfFlags
Specifies the format of the font bitmap information. It can be one of the following values.

Value	Meaning
FSF_FIXED (0x0001)	Font is fixed pitch.
FSF_PROPORTIONAL (0x0002)	Font is proportional pitch
FSF_ABCFIXED (0x0004)	Font is an ABC fixed font. The advance width for each character in the font is the sum of the **dfAspace**, **dfBspace**, and **dfCspace** members.
FSF_ABCPROPORTIONAL (0x0008)	Font is an ABC proportional font.
FSF_1COLOR (0x0010)	Font is one color.
FSF_16COLOR (0x0020)	Font is 16 color.
FSF_256COLOR (0x0040)	Font is 256 color.
FSF_RGBCOLOR (0x0080)	Font is RGB color.

This member is present only for Windows 3.*x* fonts.

dfAspace
Specifies the global A space, if any. The **dfAspace** member is the distance from the current position to the left edge of the bitmap. This member is present only for Windows 3.*x* fonts.

dfBspace
Specifies the global B space, if any. The **dfBspace** member is the width of the character. This member is present only for Windows 3.*x* fonts.

dfCspace
Specifies the global C space, if any. The **dfCspace** member is the distance from the right edge of the bitmap to the new current position. This member is present only for Windows 3.*x* fonts.

dfColorPointer
Specifies the offset to the color table (if any) for color fonts. This member is present only for Windows 3.*x* fonts, however, it is not presently used and should always be set to NULL.

dfReserved1
This member is not used. This member is present only for Windows 3.*x* fonts.

Comments

The **FONTINFO** structure may be immediately followed by one or more of the following items.

Item	Description
Character-width table	Specifies the widths of each character as well as specifies the offset to the corresponding bitmap or vector information.
Bitmaps	Specifies the bits defining the shape of the characters in a raster font. The size of this item is whatever length the total bitmaps occupy. Each row of a raster bitmap must start on a 16-bit boundary. This implies that the end of each row must be padded to an even length.
Vectors	Specifies the set of coordinates that define the shape of the characters in a vector font.
Font name	Specifies a null-terminated ASCII character string specifying the name of the font. The size of this field is the length of the string plus a null character.
Device name	Specifies a null-terminated ASCII character string specifying the name of the device if this font file is for a specific device. The size of this field is the length of the string plus a null character.

All device drivers must support Windows 2.*x* fonts. If a device driver supports Windows 3.*x* fonts, it must set the RC_BIGFONT bit in the **dpRaster** member of its **GDIINFO** structure. Printer drivers can call the **GetDeviceCaps** function (GDI.80), and check for the RC_BIGFONT bit in the raster capabilities to determine whether the display driver uses Windows 3.*x* fonts.

When a device driver realizes a font using the **RealizeObject** function, the **dfFace** and **dfDevice** members must point to valid character strings containing the font and device names.

Windows 2.x Fonts For Windows 2.*x* fonts the character-width table is either an array of integer values or an array of glyph-entry structures. The number of elements in the array is equal to:

```
dfLastChar - dfFirstChar + 2
```

That is, there is always one more element than the number of characters in the font. The extra entry is available for storing the size in bytes of the last character in a vector font. Although this extra entry applies only to vector fonts, it is present for all fonts.

For fixed-pitch vector fonts, the character-width table is an array of integer values. In this case, each element of the array is an offset (relative to the start of the segment containing the **FONTINFO** structure) to the first byte or 16 bits of vector information for the given character. The number of bytes or words for a particular character is calculated by subtracting its character-width table entry from the entry for the next character.

For variable-pitch vector fonts, the character width table is an array of **VECTORGLYPHENTRY** structures. The **VECTORGLYPHENTRY** structure has the following form:

```
typedef struct tagVECTORGLYPHENTRY {
    short vgeOffset;    /* offset to vectors relative to segment start */
    short vgeWidth;     /* width of character in pixels */
} VECTORGLYPHENTRY;
```

The **vgeOffset** member specifies the offset (relative to the start of the segment containing the **FONTINFO** structure) to the first byte or 16 bits of vector information for the given character. The **vgeWidth** member specifies the width for the character.

For raster fonts, the character-width table is an array of **RASTERGLYPHENTRY** structures.

The **rgeWidth** member specifies the width (in pixels) of the bitmap for the given character. The member also specifies the advance width for the given character. The **rgeOffset** member specifies the offset (relative to the start of the segment containing the **FONTINFO** structure) to the first byte of bitmap information for the character.

Windows 2.*x* fonts cannot exceed 64K bytes.

Windows 3.x Fonts Windows 3.*x* fonts are primarily designed for use on systems with more than average memory and a microprocessor (such as an 80386) that has instructions that use 32-bit address offsets.

For Windows 3.*x* fonts, the format of the character-width table is dependent on the value of the **dfFlags** member.

Value	Meaning
DFF_FIXED	Specifies an array of **RASTERGLYPHENTRY** structures.
DFF_PROPORTIONAL	Specifies an array of **RASTERGLYPHENTRY** structures.
DFF_ABCFIXED	Specifies an array of **ABCGLYPHENTRY** structures.
DFF_ABCPROPORTIONAL	Specifies an array of **ABCGLYPHENTRY** structures.
DFF_1COLOR	Specifies an array of **COLORGLYPHENTRY** structures.
DFF_16COLOR	Specifies an array of **COLORGLYPHENTRY** structures.
DFF_256COLOR	Specifies an array of **COLORGLYPHENTRY** structures.
DFF_RGBCOLOR	Specifies an array of **COLORGLYPHENTRY** structures.

Windows presently supports only the DFF_FIXED and DFF_PROPORTIONAL values.

The **rgeWidth** member specifies the width (in pixels) of the bitmap for the given character. The member also specifies the width for the character. The **rgeOffset** member specifies the offset (relative to the start of the segment containing the **FONTINFO** structure) to the first byte of bitmap information for the character.

The **ABCGLYPHENTRY** structure has the following form:

```
typedef struct tagABCGLYPHENTRY {
    short ageWidth;     /* width of character bitmap in pixels */
    long  ageOffset;    /* pointer to the bits */
    FIXED ageAspace;    /* A space in fractional pixels (16.16) */
    FIXED ageBspace;    /* B space in fractional pixels (16.16) */
    FIXED ageCspace;    /* C space in fractional pixels (16.16) */
} ABCGLYPHENTRY;
```

The **ageWidth** member specifies the width (in pixels) of the bitmap for the given character. The **ageOffset** member specifies the offset (relative to the start of the segment containing the **FONTINFO** structure) to the first byte of bitmap information for the character. The sum of the **ageAspace**, **ageBspace**, and **ageCspace** members specify the width of the character.

The **COLORGLYPHENTRY** structure has the following form:

```
typedef struct tagCOLORGLYPHENTRY {
    short cgeWidth;      /* width of character bitmap in pixels */
    long  cgeOffset;     /* pointer to the bits */
    short cgeHeight;     /* height of character bitmap in pixels */
    FIXED cgeAspace;     /* A space in fractional pixels (16.16) */
    FIXED cgeBspace;     /* B space in fractional pixels (16.16) */
    FIXED cgeCspace;     /* C space in fractional pixels (16.16) */
} COLORGLYPHENTRY;
```

The **cgeWidth** member specifies the width (in pixels) of the bitmap for the given character. The **cgeOffset** member specifies the offset (relative to the start of the segment containing the **FONTINFO** structure) to the first byte of bitmap information for the character. The **cgeHeight** member specifies the height (in scan lines) of the bitmap. The sum of the **cgeAspace**, **cgeBspace**, and **cgeCspace** members specify the width of the character.

The number of bits for each pixel in a character bitmap depends on the value of the **dfFlags** member.

Value	Bits per pixel
DFF_1COLOR	Bitmap has 1 bit per pixel.
DFF_16COLOR	Bitmap has 4 bits per pixel.
DFF_256COLOR	Bitmap has 8 bits per pixel.
DFF_RGBCOLOR	Bitmap has 32 bits per pixel (an **RGBQUAD** structure for each pixel).

All other font formats use 1 bit per pixel.

Windows 3.*x* fonts can exceed 64K bytes.

See Also **ExtTextOut**, **RealizeObject**, **StrBlt**

GDIINFO

```
typedef struct tagGDIINFO {
    short  int              dpVersion;
    short  int              dpTechnology;
    short  int              dpHorzSize;
    short  int              dpVertSize;
    short  int              dpHorzRes;
    short  int              dpVertRes;
    short  int              dpBitsPixel;
```

```
    short  int              dpPlanes;
    short  int              dpNumBrushes;
    short  int              dpNumPens;
    short  int              futureuse;
    short  int              dpNumFonts;
    short  int              dpNumColors;
    unsigned  short  int    dpDEVICEsize;
    unsigned  short  int    dpCurves;
    unsigned  short  int    dpLines;
    unsigned  short  int    dpPolygonals
    unsigned  short  int    dpText;
    unsigned  short  int    dpClip;
    unsigned  short  int    dpRaster;
    short  int              dpAspectX;
    short  int              dpAspectY;
    short  int              dpAspectXY;
    short  int              dpStyleLen;
    POINT                   dpMLoWin;
    POINT                   dpMLoVpt;
    POINT                   dpMHiWin;
    POINT                   dpMHiVpt;
    POINT                   dpELoWin;
    POINT                   dpELoVpt;
    POINT                   dpEHiWin;
    POINT                   dpEHiVpt;
    POINT                   dpTwpWin;
    POINT                   dpTwpVpt;
    short  int              dpLogPixelsX;
    short  int              dpLogPixelsY;
    short  int              dpDCManage;
    short  int              dpCaps1;
    long   int              dpSpotSizeX;
    long   int              dpSpotSizeY;
    short  int              dpPalColors;
    short  int              dpPalReserved;
    short  int              dpPalResolution;
} GDIINFO;
```

The **GDIINFO** structure contains information about graphics devices supported by the device driver. GDI retrieves this structure when it loads the driver and uses the information in the structure to initialize the driver.

Members

dpVersion
Specifies the version number. The high-order byte specifies the major version, the low-order byte the minor version. For example, in a device driver developed for Windows 3.1, this member should contain 0x030A.

dpTechnology
Specifies the device technology. It can be one of the following values.

Value	Meaning
DT_PLOTTER (0)	Vector plotter
DT_RASDISPLAY (1)	Raster display
DT_RASPRINTER (2)	Raster printer
DT_RASCAMERA (3)	Raster camera
DT_CHARSTREAM (4)	Character stream, PLP
DT_METAFILE (5)	Metafile, VDM
DT_DISPFILE (6)	Display file

dpHorzSize
Specifies the width of the physical display surface in millimeters.

dpVertSize
Specifies the height of the physical display surface in millimeters.

dpHorzRes
Specifies the width of the display surface in pixels. For nonraster devices, this width is equivalent to the number of vertical grid lines used by the device to plot points on the display surface. In such cases, a pixel is defined to be the smallest mark the device can draw.

dpVertRes
Specifies the height of the display in raster lines. For nonraster devices, this height is equivalent to the number of horizontal grid lines used by the device to plot points on the display surface. In such cases, a raster line is equivalent to a gridline.

dpBitsPixel
Specifies the number of adjacent bits on each plane required to define a single pixel.

dpPlanes
Specifies the number of planes required to define the pixels. For a typical raster device with red, green, and blue bit planes (such as a 3-plane EGA), this member is 3.

dpNumBrushes
Specifies the number of device-specific brushes supported by this device.

dpNumPens
Specifies the number of device-specific pens supported by this device.

futureuse
Reserved; do not use.

dpNumFonts
Specifies the number of device-specific fonts supported by this device.

dpNumColors
Specifies the number of entries in the color table for this device or the number of reserved colors for palette-capable devices.

dpDEVICEsize
Specifies the size (in bytes) of the **PDEVICE** structure for this device. It must be at least two bytes.

dpCurves
Specifies whether the device driver can perform circles, pie wedges, chord arcs, and ellipses. The **dpCurves** member also specifies whether the interior of those figures that can be handled can be brushed in, and whether the borders of those figures that can be handled can be drawn with wide lines, styled lines, or lines that are both wide and styled. The **dpCurves** member can be a combination of the following values.

Value	Meaning
CC_NONE (0x0000)	Curves not supported.
CC_CIRCLES (0x0001)	Can perform circles.
CC_PIE (0x0002)	Can perform pie wedges.
CC_CHORD (0x0004)	Can perform chord arcs.
CC_ELLIPSES (0x0008)	Can perform ellipses.
CC_WIDE (0x0010)	Can perform wide lines.
CC_STYLED (0x0020)	Can perform styled lines.
CC_WIDESTYLED (0x0040)	Can perform lines that are wide and styled.
CC_INTERIORS (0x0080)	Can perform interiors.
CC_ROUNDRECT (0x0100)	Can perform round rectangles.

All other values are reserved.

dpLines
Specifies whether the device driver can perform polylines and lines. The **dpLines** member also specifies whether the interior of those figures that can be handled can be brushed in, and whether the borders of those figures that can be handled can be drawn with wide lines, styled lines, or lines that are both wide and styled. The **dpLines** member can be a combination of the following values.

Value	Meaning
LC_NONE (0x0000)	Lines not supported.
LC_POLYLINE (0x0002)	Can perform polylines.
LC_WIDE (0x0010)	Can perform wide lines.
LC_STYLED (0x0020)	Can perform styled lines.
LC_WIDESTYLED (0x0040)	Can perform wide styled lines.
LC_INTERIORS (0x0080)	Can perform interiors.

All other values are reserved. The high byte must be zero.

dpPolygonals
Specifies whether the device driver can perform polygons, rectangles, and scan lines. The **dpPolygonals** member also specifies whether the interior of those figures that can be handled can be brushed in, and whether the borders of those figures that can be handled can be drawn with wide lines, styled lines, or lines that are both wide and styled. The **dpPolygonals** member can be a combination of the following values.

Value	Meaning
PC_NONE (0x0000)	Polygons not supported.
PC_ALTPOLYGON (0x0001)	Can perform alternate-fill polygons.
PC_RECTANGLE (0x0002)	Can perform rectangles.
PC_WINDPOLYGON (0x0004)	Can perform winding-number-fill polygons.
PC_SCANLINE (0x0008)	Can perform scan lines.
PC_WIDE (0x0010)	Can perform wide borders.
PC_STYLED (0x0020)	Can perform styled borders.
PC_WIDESTYLED (0x0040)	Can perform borders that are wide and styled.
PC_INTERIORS (0x0080)	Can perform interiors.

All other values are reserved. The high byte must be zero.

dpText
Specifies the level of text support the device driver provides. The **dpText** member can be a combination of the following values.

Value	Meaning
TC_OP_CHARACTER (0x0001)	Can generate character-precision text. If this value is not given (or implied by the TC_OP_STROKE value), the driver can generate string-precision text only.
TC_OP_STROKE (0x0002)	Can generate stroke-precision text. The value implies the TC_OP_CHARACTER value.
TC_CP_STROKE (0x0004)	Can draw partially clipped characters. If this value is not given, the character must be entirely within the clipping region to be drawn.
TC_CR_90 (0x0008)	Can rotate characters in 90-degree increments. If this value is not given (or implied by the TC_CR_ANY value), the driver can not rotate text.
TC_CR_ANY (0x0010)	Can rotate characters to any angle. This value implies the TC_CR_90 value.
TC_SF_X_YINDEP (0x0020)	Can scale characters independently along the *x*- and *y*-axes. If this value is not given, the driver may be able to scale characters but not independently along the axes.

Value	Meaning
TC_SA_DOUBLE (0x0040)	Can scale characters by doubling. If this value is not given (or implied by the TC_SA_INTEGER or TC_SA_CONTIN values), the driver cannot scale text.
TC_SA_INTEGER (0x0080)	Can scale characters by integral multiples. This value implies the TC_SA_DOUBLE value.
TC_SA_CONTIN (0x0100)	Can scale characters by any multiple. This value implies the TC_SA_DOUBLE and TC_SA_INTEGER values.
TC_EA_DOUBLE (0x0200)	Can generate bold characters by doubling the weight. If this value is not given, the driver cannot modify character weights.
TC_IA_ABLE (0x0400)	Can generate italic characters by skewing.
TC_UA_ABLE (0x0800)	Can generate underlined characters.
TC_SO_ABLE (0x1000)	Can generate struck-out characters.
TC_RA_ABLE (0x2000)	Can use raster fonts to generate text.
TC_VA_ABLE (0x4000)	Can use vector fonts to generate text.
TC_RESERVED (0x8000)	Reserved; must be zero.

If a device claims to have an ability, it must have it for all fonts, whether realized by the device or provided by GDI.

dpClip

Specifies whether the device can clip output. This member can be one of the following values.

Value	Meaning
CP_NONE (0)	Device cannot clip.
CP_RECTANGLE (1)	Device can output using a single rectangle.
CP_REGION (2)	Device can output using a region (which may be several rectangles).

dpRaster

Specifies raster abilities. This member can be a combination of the following values.

Value	Meaning
RC_NONE (0x0000)	Device has no raster capabilities.
RC_BITBLT (0x0001)	Can transfer bitmaps. The driver exports the **BitBlt** function.
RC_BANDING (0x0002)	Requires banding support.
RC_SCALING (0x0004)	Requires scaling support.

Value	Meaning
RC_BITMAP64 (0x0008)	Supports bitmaps that are larger than 64K bytes.
RC_GDI20_OUTPUT (0x0010)	Supports the Windows 2.*x* functions. The driver exports the **ExtTextOut**, **GetCharWidth**, and **FastBorder** functions.
RC_GDI20_STATE (0x0020)	Supports state blocks in device contexts.
RC_SAVEBITMAP (0x0040)	Saves bitmaps locally in "shadow" memory. Driver exports the **SaveScreenBitmap** function.
RC_DI_BITMAP (0x0080)	Can get and set device-independent bitmaps (DIBs). The Driver exports the **DeviceBitmapBits** function.
RC_PALETTE (0x0100)	Can do color-palette management.
RC_DIBTODEV (0x0200)	Can transfer device-independent bitmaps directly to device. The driver exports the **SetDIBitsToDevice** function.
RC_BIGFONT (0x0400)	Supports Windows 3.*x* fonts. If this value is not given, GDI ensures that the driver receives Windows 2.*x* fonts only.
RC_STRETCHBLT (0x0800)	Can stretch and compress bitmaps while transferring the bitmap. The driver exports the **StretchBlt** function.
RC_FLOODFILL (0x1000)	Can perform flood filling. The driver exports the **FloodFill** function.
RC_STRETCHDIB (0x2000)	Can stretch and compress device-independent bitmaps while transferring the bitmap. The driver exports the **StretchDIBits** function.
RC_OP_DX_OUTPUT (0x4000)	Can fill opaque rectangle and set character widths on calls to the **ExtTextOut** function.
RC_DEVBITS (0x8000)	Supports device bitmaps. Driver exports the **BitmapBits** and **SelectBitmap** function.

dpAspectX
Specifies the relative width of a device pixel. This value, in the range 1 to 1000, helps specify the device's aspect ratio.

dpAspectY
Specifies the relative height of a device pixel. This value, in the range 1 to 1000, helps specify the device's aspect ratio.

dpAspectXY
Specifies the relative diagonal width of a device pixel. This value, in the range 1 to 1000, helps specify the device's aspect ratio. It must be equal to the square root of the sum of the squares of the **dpAspectX** and **dpAspectY** members.

dpStyleLen
Specifies the minimum length of a dot generated by a styled pen. The length is relative to the width of a device pixel and should be given in the same units as the **dpAspectX** member. For example, if **dpAspectX** is 5 and the minimum length required is 3 pixels, the **dpStyleLen** member should be 15.

dpMLoWin
Specifies the width and height of the metric (low resolution) window. Width is **dpHorzSize***10; height is **dpVertSize***10.

dpMLoVpt
Specifies the horizontal and vertical resolutions of the metric (low resolution) viewport. Horizontal is **dpHorzRes**; vertical is **–dpVertRes**.

dpMHiWin
Specifies the width and height of the metric (high resolution) window. Width is **dpHorzSize***100; height is **dpVertSize***100.

dpMHiVpt
Specifies the horizontal and vertical resolutions of the metric (high resolution) viewport. Horizontal is **dpHorzRes**; vertical is **–dpVertRes**.

dpELoWin
Specifies the width and height of the English (low resolution) window. Width is **dpHorzSize***1000; height is **dpVertSize***1000.

dpELoVpt
Specifies the horizontal and vertical resolutions of the English (low resolution) viewport. Horizontal is **dpHorzRes***254; vertical is **–dpVertRes***254.

dpEHiWin
Specifies the width and height of the English (high resolution) window. Width is **dpHorzSize***10,000; height is **dpVertSize***10,000.

dpEHiVpt
Specifies the horizontal and vertical resolutions of the English (high resolution) viewport. Horizontal is **dpHorzRes***254; vertical is **–dpVertRes***254.

dpTwpWin
Specifies the width and height of the twip window. There are 20 twips per 1 printer's point and 72 printer's points per inch. Width is **dpHorzSize***14400; height is **dpVertSize***14400.

dpTwpVpt
Specifies the horizontal and vertical resolutions of the twip viewport. Horizontal is **dpHorzRes***254; vertical is **–dpVertRes***254.

dpLogPixelsX
Specifies the number of pixels per logical inch along a horizontal line on the display surface. This is used to match fonts.

dpLogPixelsY
Specifies the number of pixels per logical inch along a vertical line on the display surface. This is used to match fonts.

dpDCManage
Specifies whether the device driver can manage multiple device contexts (DC). This member can be one of the following values.

Value	Meaning
0x0000	Driver allows multiple DCs. It creates a new **PDEVICE** for each DC that specifies a new device and filename pair, but uses the same **PDEVICE** for any subsequent DCs that specify the same device and filename pair.
DC_SPDEVICE (0x0001)	Driver allows multiple DCs but it creates a new **PDEVICE** for each DC regardless of whether the device and filename pairs are the same.
DC_1PDEVICE (0x0002)	Driver allows multiple DCs but only if all DCs have unique device and filename pairs. The driver creates a **PDEVICE** for each DC. The driver returns an error on any attempt to create a second DC with an existing device and filename pair.
DC_IGNOREDFNP (0x0004)	Driver allows multiple DCs but only creates one **PDEVICE**. All DCs share the same **PDEVICE** regardless of the device and filename pairs.
0x0006	Driver allows only one DC. The driver returns an error on any attempt to create a second DC.

The values 0x0003, 0x0005, and 0x0007 are not valid and must not be used.

dpCaps1
Specifies additional raster abilities. The member can be one of the following values.

Value	Meaning
C1_TRANSPARENT (0x0001)	Performs **BitBlt** and **StretchBlt** functions with a transparent background.
TC_TT_ABLE (0x0002)	Informs GDI that the driver is capable of producing TrueType as raster fonts. The driver must call the **dmExtTextOut** function to draw the raster font into the bitmap. The value is similar to TC_RA_ABLE.

All other values are reserved.

dpSpotSizeX
Specifies the horizontal spot size for TrueType fonts on this device.

dpSpotSizeY
Specifies the vertical spot size for TrueType fonts on this device.

dpPalColors
Specifies the total number of simultaneous colors available in Windows 3.*x* for palette-capable devices. Nonpalette-capable devices ignore this value.

dpPalReserved
Specifies the even number of reserved system colors available in Windows 3.*x* for palette-capable devices. Nonpalette-capable devices ignore this value.

dpPalResolution
Specifies the palette resolution, which equals the number of bits going into video DACS. Nonpalette-capable devices ignore this value.

Comments

The **dpText** member requires that for each precision level that the precision levels below it are also set. For example, the TC_SA_INTEGER value requires that the the TC_SA_DOUBLE value be set, and the TC_SA_CONTIN value requires that all three be set. Since it is required that the lowest precision level of each ability be supported, no value is provided in **dpText** for the lowest level of each ability.

The **dpAspectX**, **dpAspectY**, and **dpAspectXY** members specify the relative width, height, and diagonal width of a device pixel and correspond directly to the device's aspect ratio. For devices whose pixels do not have integral diagonal widths, the member values can be multiplied by a convenient factor to preserve information. For example, pixels on a device with a 1 to 1 aspect ratio have a diagonal width of 1.414. For good results, the aspect members should be set to 100, 100, and 141, respectively. For numerical stability, the member values should be kept under 1000.

The window/viewport pair members are the numerator and denominator of the scale fraction used to correct for the device aspect ratio, and to set to a fixed unit of measurement, either metric or English. These numbers should be integers in the range of –32768 to 32767. When calculating these constants, out-of-range values can be divided by some number to bring them back into range as long as the corresponding window or viewport constant is divided by the same number.

The **dpRaster** member is also used to indicate a scaling device. If the RC_SCALING value is set, the device does graphics scaling. Certain devices perform graphics at one resolution and text at another. Some applications require that character cells be an integral number of pixels. If a device reported that its graphics resolution was 75 dpi but its text resolution was 300 dpi, then its character cells would not be an integral number of pixels (since they were digitized at 300 dpi). To get around this problem, GDI uses scaling devices. The device driver registers itself as a 300 dpi device and all the graphics at 300 dpi are scaled to 75 dpi. Any device that scales must have the RC_SCALING value set. Scaling always reduces the resolution; it never increases it. GDI calls the **Control** function with GETSCALINGFACTOR escape before output to a device. The scaling factor is a shift count that is a power of two. Therefore, a scaling factor of 2 means reduce by 4, and a scaling factor of 1 means reduce by 2.

The number of reserved colors on the palette is always 20, with 16 corresponding to the VGA colors and 4 special colors. Half of the reserved palette colors are placed at the beginning and half at the end of the palette.

See Also **Enable**, **Control**, GETSCALINGFACTOR, **PDEVICE**

JUST_VALUE_STRUCT

```
typedef struct tagJUST_VALUE_STRUCT {
    short nCharExtra;
    WORD  nCharCount;
    short nBreakExtra;
    WORD  nBreakCount;
} JUST_VALUE_STRUCT;
```

The **JUST_VALUE_STRUCT** structures contains the values to use when justifying text.

Members

nCharExtra
Specifies in font units the total extra space that must be distributed over **nCharCount** characters.

nCharCount
Specifies the number of characters over which the **nCharExtra** member is distributed.

nBreakExtra
Specifies in font units the total extra space that is distributed over **nBreakCount** break characters.

nBreakCount
Specifies the number of break characters over which **nBreakExtra** units are distributed.

See Also SETALLJUSTVALUES

KERNPAIR

```
typedef struct tagKERNPAIR {
    union {
        BYTE each [2];
        WORD both;
    } kpPair;
    short  kpKernAmount;
} KERNPAIR;
```

The **KERNPAIR** structure contains the amount of kerning to apply to a given pair of characters.

Members

kpPair
Specifies the character pair. This 2-byte member contains the character codes in the order in which the pair is to be printed.

kpKernAmount
Specifies the amount of kerning to apply to the character pair.

See Also

GETPAIRKERNTABLE

KERNTRACK

```
typedef struct tagKERNTRACK {
    short ktDegree;
    short ktMinSize;
    short ktMinAmount;
    short ktMaxSize;
    short ktMaxAmount;
} KERNTRACK;
```

The **KERNTRACK** structure contains information used to adapt kerning values for scaled fonts.

Members

ktDegree
Specifies the degree.

ktMinSize
Specifies the minimum size.

ktMinAmount
Specifies the minimum amount.

ktMaxSize
Specifies the maximum size.

ktMaxAmount
Specifies the maximum amount.

See Also GETTRACKKERNTABLE

LBRUSH

```
typedef struct tagLBRUSH {
    short   lbStyle;
    long    lbColor;
    short   lbHatch;
    long    lbBkColor;
} LBRUSH;
```

The **LBRUSH** structure contains style and color information for a brush.

Members

lbStyle
Specifies brush style. GDI requires that drivers support at least the following brush styles.

Value	Meaning
BS_SOLID (0)	Brush is a single solid or dithered color.
BS_HOLLOW (1)	Brush has no color or pattern.
BS_HATCHED (2)	Brush has a predefined pattern.
BS_PATTERN (3)	Brush has the pattern as specified by a given bitmap.

lbColor
Specifies a foreground color for brushes having BS_SOLID and BS_HATCHED values. For BS_HOLLOW and BS_PATTERN values, this member is zero. Brush colors are specified as physical colors. For palette-capable devices, this member specifies a physical-color index if the high-order byte is 0xFF.

lbHatch
Specifies a brush pattern for brushes having BS_HATCHED style or a global memory handle for a bitmap for brushes having BS_PATTERN style. For BS_SOLID and BS_HOLLOW values, this member is zero.

For BS_HATCHED brushes, this member specifies a brush pattern and can be one of the following values.

Value	Description
HS_HORIZONTAL	Horizontal hatch
HS_VERTICAL	Vertical hatch
HS_FDIAGONAL	45-degree upward hatch (left to right)
HS_BDIAGONAL	45-degree downward hatch (left to right)
HS_CROSS	Horizontal and vertical cross hatch
HS_DIAGCROSS	45-degree cross hatch

For BS_PATTERN style, this member is the global memory handle to a **PBITMAP** structure specifying the pattern.

lbBkColor
Specifies the background color for brushes having BS_HATCHED style. Brush colors are specified as physical colors. For palette-capable devices, this member specifies a physical color index if the high-order byte is 0xFF.

See Also **RealizeObject**

LFONT

```
typedef struct {
    short lfHeight;
    short lfWidth;
    short lfEscapement;
    short lfOrientation;
    short lfWeight;
    BYTE  lfItalic;
    BYTE  lfUnderline;
    BYTE  lfStrikeOut;
    BYTE  lfCharSet;
    BYTE  lfOutPrecision;
    BYTE  lfClipPrecision;
    BYTE  lfQuality;
    BYTE  lfPitchAndFamily;
    BYTE  lfFaceName [32];
} LFONT;
```

The **LFONT** structure contains the logical attributes for a font.

Members

lfHeight
Specifies the height of the font in device units. If this member is greater than zero, the driver should choose a font whose cell height matches the given height. If this member is zero, the driver should choose a font having a reasonable default size. If this member is less than zero, the driver should choose a font whose character height (that is, cell height less internal leading) matches the absolute value of this member. In all cases, the driver should choose the largest font that does not exceed the requested height and, if there is no such font, choose the next smallest font available.

lfWidth
Specifies the average width of characters in the font in device units. If this member is zero, the driver should choose an available font whose digitization aspect ratio (the **dfVertRes** member of the **FONTINFO** structure) most closely matches the aspect ratio of the device (the **dpAspectY** member in the **GDIINFO** structure). When comparing fonts, the driver should compare the absolute values of the differences between the digitization aspect ratio and the device aspect ratio.

lfEscapement
Specifies the angle, counterclockwise from the *x*-axis in tenths of a degree, of the vector passing through the origin of all the characters in the string.

lfOrientation
Specifies the angle, counterclockwise from the *x*-axis in tenths of a degree, of the baseline of the character.

lfWeight
Specifies the weight of the font ranging from 1 to 1000, with 400 being the value for the standard font. If this member is zero, the driver should choose a reasonable weight.

lfItalic
Specifies whether the font is to be italic. If the low bit is set, the font is to be italic. All other bits must be zero.

lfUnderline
Specifies whether the font is to be underlined. If the low bit is set, the font is to be underlined. All other bits must be zero.

lfStrikeOut
Specifies whether the font is to be struck out. If the low bit is set, the font is to be struck out. All other bits must be zero.

lfCharSet
Specifies the character set to be used. It can be one of the following values.

Value	Meaning
ANSI_CHARSET (0x00)	Indicates the ANSI character set.
SYMBOL_CHARSET (0x02)	Indicates the Symbol character set.
OEM_CHARSET (0xFF)	Indicates an OEM-specific character set. The characters and corresponding character codes depend on the computer.

lfOutPrecision

Specifies the required output precision for text. This member can have one of the following values.

Value	Meaning
OUT_DEFAULT_PRECIS (0x00)	Chooses a reasonable font.
OUT_STRING_PRECIS (0x01)	Chooses the font whose size (height and width) most closely matches the requested size. The driver may disregard the requested orientation and escapement, but other attributes must match.
OUT_CHARACTER_PRECIS (0x02)	Chooses the font whose size (height and width) most closely matches the requested size. The driver may disregard the requested orientation, but other attributes must match.
OUT_STROKE_PRECIS (0x03)	Chooses a font whose attributes exactly match the requested attributes.

lfClipPrecision

Specifies the required clipping precision for text. This member can be one of the following values.

Value	Meaning
CLIP_DEFAULT_PRECIS (0x00)	Chooses a reasonable font.
CLIP_CHARACTER_PRECIS (0x01)	Chooses a font that allows clipping of individual characters. The driver must be able to clip a character if any portion of it lies outside the clipping rectangle.
CLIP_STROKE_PRECIS (0x02)	Chooses a font that allows clipping of portions of a character. The driver must be able to clip any portion of a character that lies outside the clipping rectangle.

lfQuality

Specifies the required quality for text. This member can have one of the following values.

Value	Meaning
DEFAULT_QUALITY (0x00)	Chooses a reasonable font.
DRAFT_QUALITY (0x01)	Chooses a font that generates the most efficient, speediest output. The driver can sacrifice appearance if a speedier font has lower quality. GDI synthesizes bold, italic, underline, and strikethrough characters if needed.
PROOF_QUALITY (0x02)	Chooses a font that generates the highest-quality output. The driver should sacrifice speedy output if a slower font has higher quality. The driver should sacrifice output precision if a font that does not exactly match the requested attributes (such as size) is a higher quality. GDI synthesizes bold, italic, underline, and strikethrough characters if needed.

lfPitchAndFamily

Specifies the font pitch and font family. This member is a combination of one pitch and one family value. The pitch value can be any one of the following values.

Value	Meaning
DEFAULT_PITCH (0x00)	Chooses a reasonable font.
FIXED_PITCH (0x01)	Chooses a fixed-pitch font.
VARIABLE_PITCH (0x02)	Chooses a variable-pitch font.

The font family, which describes in a general way the look of a font, can be any one of the following values.

Value	Meaning
FF_DONTCARE (0x00)	Chooses a reasonable font.
FF_ROMAN (0x10)	Chooses a variable-pitch font with serifs.
FF_SWISS (0x20)	Chooses a variable-pitch fonts without serifs.
FF_MODERN (0x30)	Chooses a fixed-pitch font.
FF_SCRIPT (0x40)	Chooses a cursive or script font.
FF_DECORATIVE (0x50)	Chooses a novelty font.

lfFaceName
Specifies a null-terminated string specifying the name of the font. The driver should choose a font having the given name. If the string is empty (the first byte is zero), the driver should choose a reasonable font. The string, including the null terminator, does not exceed 32 bytes.

See Also **EngineRealizeFont, RealizeObject**

LPEN

```
typedef struct tagLPEN {
    long  lopnStyle;
    POINT lopnWidth;
    long  lopnColor;
} LPEN;
```

The **LPEN** structure specifies a logical pen. Pens are used to draw lines and borders.

Members **lopnStyle**
Specifies the pen style. GDI requires that drivers support at least the following pen styles.

Value	Meaning
LS_SOLID (0)	Draws solid lines.
LS_DASHED (1)	Draws dashed lines.
LS_DOTTED (2)	Draws dotted lines.
LS_DOTDASHED (3)	Draws lines with alternating dots and dashes.
LS_DASHDOTDOT (4)	Draws lines with a repeating pattern of a dash followed by two dots.
LS_NOLINE (5)	Nothing is drawn.
LS_INSIDEFRAME (6)	Creates a pen in which a line is drawn inside the frame of ellipses and rectangles. If the width of the pen is greater than 1 and the pen style is PS_INSIDEFRAME, the line is drawn inside the frame of all primitives except polygons and polylines; the pen is drawn with a logical (dithered) color if the pen color does not match an available RGB value. The PS_INSIDEFRAME style is identical to PS_SOLID if the pen width is less than or equal to 1.

lopnWidth
Specifies a **POINT** structure whose **x** member contains the pen width in device units. A zero-width pen is drawn with the system's smallest width. Negative-width pens have no width and are NULL pens. The **y** member is ignored.

lopnColor
Specifies the color for the pen. Pen colors are specified as physical colors. For palette-capable devices, the value is a color index if the high byte is 0x0FFH.

See Also **RealizeObject**

ORIENT

```
typedef struct tagORIENT {
    short Orientation;
    short Reserved[4];
} ORIENT;
```

The **ORIENT** structure contains information about the paper orientation, such as whether it is portrait or landscape.

Members

Orientation
Specifies the paper orientation. If it is 1, the orientation is portrait. If it is 2, the orientation is landscape.

Reserved
Not used; must be zero.

See Also GETSETPRINTORIENT

PATH_INFO

```
typedef struct tagPATH_INFO {
    short     RenderMode;
    BYTE      FillMode;
    BYTE      BkMode;
    LPEN      Pen;
    LBRUSH    Brush;
    DWORD     BkColor;
} PATH_INFO;
```

The **PATH_INFO** structure contains information about a path.

Members

RenderMode
Specifies how to draw the path. This member can have one of the following values.

Value	Meaning
NO_DISPLAY (0)	Path not drawn.
OPEN (1)	Drawn as an open polygon.
CLOSED (2)	Drawn as a closed polygon.

FillMode
Specifies how to fill the path. This member can have one of the following values.

Value	Meaning
ALTERNATE (1)	Filled using the alternate-fill method.
WINDING (2)	Filled using the winding-fill method.

BkMode
Specifies how to use existing colors when filling the path. This member is equivalent to the **BkMode** member of the **DRAWMODE** structure. Drivers that encounter a **BkMode** of zero should assume TRANSPARENT value and ignore **BkColor**.

Pen
Specifies which pen to use to draw the path. If **RenderMode** is set to the NO_DISPLAY value, the pen is ignored.

Brush
Specifies which brush to use to draw the path. If **RenderMode** is set to the NO_DISPLAY or OPEN values, the pen is ignored.

BkColor
Specifies which background color to use when filling the path. This member is equivalent to the **BkColor** member of the **DRAWMODE** structure.

See Also

END_PATH

PATTERNRECT

```
typedef struct tagPATTERNRECT {
    POINT prPosition;
    POINT prSize;
    WORD prStyle;
    WORD prPattern;
} PATTERNRECT;
```

The **PATTERNRECT** structure contains information about a pattern, gray scale, or black rectangle to be created by a Hewlett-Packard PCL driver.

Members

prPosition
Specifies a **POINT** structure identifying the upper-left corner of the rectangle.

prSize
Specifies a **POINT** structure identifying the lower-right corner of the rectangle.

prStyle
Specifies the type of pattern. It can be one of the following values.

Value	Meaning
0	Black rule
1	White rule
2	Gray scale
3	Device defined

prPattern
Specifies the percent of gray for a gray scale pattern or specifies one of six patterns for device-defined patterns. This member is ignored if the value is zero (black rule).

See Also

DRAWPATTERNRECT

PBITMAP

```
typedef struct tagPBITMAP {
    short bmType;
    short bmWidth;
    short bmHeight;
    short bmWidthBytes;
    BYTE  bmPlanes;
    BYTE  bmBitsPixel;
    long  bmBits;
```

```
    long  bmWidthPlanes;
    long  bmlpPDevice;
    short bmSegmentIndex;
    short bmScanSegment;
    short bmFillBytes;
    short reserved1;
    short reserved2;
} PBITMAP;
```

A **PBITMAP** structure specifies the dimensions, attributes, and bits of a physical bitmap.

Members

bmType
Specifies a physical bitmap. The member must be set to zero.

bmWidth
Specifies the width of the bitmap in pixels.

bmHeight
Specifies the height of the bitmap in raster lines.

bmWidthBytes
Specifies the number of bytes in each raster line of this bitmap. The number of bytes must be even; all raster lines must be aligned on 16-bit boundaries.

bmPlanes
Specifies the number of color planes.

bmBitsPixel
Specifies the number of color bits for each pixel.

bmBits
Points to an array of bits specifying the pixels of the bitmap. The array must be aligned on a 16-bit boundary.

bmWidthPlanes
Specifies the size in bytes of each color plane. It is equal to the product of **bmWidthBytes*bmHeight**.

bmlpPDevice
Points to the **PDEVICE** structure specifying the device for which this bitmap is compatible.

bmSegmentIndex
Specifies the segment or selector offset for segments in the bitmap array. If the bitmap is less than 64K bytes, this member is zero.

bmScanSegment
Specifies the number of raster lines contained in each segment of the bitmap array. If the bitmap is less than 64K bytes, this member is zero.

bmFillBytes
Specifies the number of extra bytes in each segment. Graphics-device interface (GDI) allocates storage for the bitmap array in 16-byte multiples.

reserved1
Reserved; do not use.

reserved2
Reserved; do not use.

Comments

If the bitmap bits exceed 64K bytes, GDI allocates a two or more segments to store the bitmap. In such cases, the **bmScanSegment** member specifies the number of raster lines stored in each segment, with **bmFillBytes** specifying any additional bytes needed to round the segment size out to a multiple of 16; no segment contains more than 64K bytes. The total number of segments is equal to the quotient of **bmHeight** divided by **bmScanSegment** rounded up by one if the remainder is not zero. The selector (or segment address) for each segment is a multiple of **bmSegmentIndex**.

GDI stores the bits in the bitmap array as raster lines, with the raster line representing the top of the bitmap stored first. If the bitmap has more than one plane, GDI stores the first raster lines for all planes at the beginning of the array, stores the second raster lines next, and so on. The following shows the layout for a 4-plane bitmap:

```
Plane 0, first raster line
Plane 1, first raster line
Plane 2, first raster line
Plane 3, first raster line
Plane 0, second raster line
        .
        .
        .
Plane 0, last raster line
Plane 1, last raster line
Plane 2, last raster line
Plane 3, last raster line
```

If the bitmap array exceeds 64K bytes, GDI splits the raster lines across several segments but retains the storage order, giving the first raster lines in the first segment and the last raster lines in the last segment. When GDI splits the raster lines, it ensures that matching raster lines from the various planes are always in the same segment. If necessary, GDI leaves a number of empty bytes (as specified by **bmFillBytes**) at the end of the segment to round out the segment size to a multiple of 16.

PDEVICE

```
typedef struct tagPDEVICE {
    short pdType;
} PDEVICE;
```

The **PDEVICE** structure contains information that a graphics driver uses to identify a device and the current state of the device. The size and content of the structure depends entirely on the driver. For example, the structure may include the current pen, the current position, the communication port of a particular device, and other state information. However, the first member in every **PDEVICE** structure must be **pdType**.

Members

pdType
Specifies the device type. If this member is nonzero, the structure identifies a device and all remaining members are driver specific. If this member is zero, the structure identifies a memory bitmap and all remaining members must be identical to a **PBITMAP** structure.

Comments

GDI allocates space for the **PDEVICE** structure when it calls the **Enable** function to initialize a device driver. The size of this structure must be specified in the **dpDEVICEsize** member of the **GDIINFO** structure.

See Also

Enable

POINT

```
typedef struct tagPOINT {
    short   x;
    short   y;
} POINT;
```

The **POINT** structure contains the *x*- and *y*-coordinates of a point.

Members

x
Specifies the *x*-coordinate of a point.

y
Specifies the *y*-coordinate of a point.

See Also

GETPHYSPAGESIZE, GETPRINTINGOFFSET

RECT

```
typedef struct tagRECT {
    short left;
    short top;
    short right;
    short bottom;
} RECT;
```

The **RECT** structure contains the coordinates of the top-left and bottom-right corners of a rectangle.

Members

left
Specifies the *x*-coordinate of the top-left corner of the rectangle.

top
Specifies the *y*-coordinate of the top-left corner of the rectangle.

right
Specifies the *x*-coordinate of the bottom-right corner of the rectangle.

bottom
Specifies the *y*-coordinate of the bottom-right corner of the rectangle.

See Also

Output(OS_RECTANGLE)

RGBQUAD

```
typedef struct tagRGBQUAD {
    BYTE rgbBlue;
    BYTE rgbGreen;
    BYTE rgbRed;
    BYTE rgbReserved;
} RGBQUAD;
```

The **RGBQUAD** structure specifies a logical color or a 16-bit color index. A logical color specifies the color desired by an application. A color index indirectly specifies a logical color by identifying a color in an array (or table) of colors.

Members

rgbBlue
Specifies the intensity of blue. It must be a value in the range 0 (no blue) to 255 (brightest blue).

rgbGreen
Specifies the intensity of green. It must be a value in the range 0 (no green) to 255 (brightest green).

rgbRed
Specifies the intensity of red. It must be a value in the range 0 (no red) to 255 (brightest red).

rgbReserved
Specifies whether the **RGBQUAD** structure specifies an RGB color or a 16-bit color index. If this member is zero, this is an RGB color. If this member is 0xFF, then the low 16 bits (**rgbBlue** and **rgbGreen** members) is a color index, not an RGB color.

Comments

When the colors are at minimum intensity (0,0,0), the result is black; when at maximum intensity (255,255,255), the result is white; and when at half intensity (127,127,127), the result is gray.

Primary colors can be combined to form new colors. For example, solid red (0,0,255) and blue (255,0,0) can form purple (255,0,255). If a device cannot display all the possible RGB color combinations, the device driver must map given RGB color values to colors the device can display. For example, in a black-and-white display with only one bit per pixel, the driver uses a cutoff intensity at which all the RGB values above the intensity are white and all below are black. One method used to compute the cutoff intensity is to add the individual color intensities according to the following formula:

```
((5*rgbGreen+3*rgbRed+rgbBlue)+4)>>3
```

If the result is greater than 128, then all the RGB values above that intensity will be white, and those below it will be black.

TEXTMETRIC

```
typedef struct tagTEXTMETRIC {
    short   tmHeight;
    short   tmAscent;
    short   tmDescent;
    short   tmInternalLeading;
    short   tmExternalLeading;
    short   tmAveCharWidth;
    short   tmMaxCharWidth;
    short   tmWeight;
    BYTE    tmItalic;
    BYTE    tmUnderlined;
    BYTE    tmStruckOut;
```

```
    BYTE    tmFirstChar;
    BYTE    tmLastChar;
    BYTE    tmDefaultChar;
    BYTE    tmBreakChar;
    BYTE    tmPitchAndFamily;
    BYTE    tmCharSet;
    short   tmOverhang;
    short   tmDigitizedAspectX;
    short   tmDigitizedAspectY;
} TEXTMETRIC;
```

The **TEXTMETRIC** structure contains a list of the basic metrics of a physical font.

Members

tmHeight
Specifies the height of the character cell. This member is equal to the sum of the **tmAscent** and **tmDescent** members.

tmAscent
Specifies the ascent of the character cell, that is, height of the cell measured from the baseline.

tmDescent
Specifies the descent of the character cell, that is, the height of the cell measured from the baseline to the bottom of the cell.

tmInternalLeading
Specifies the amount of internal leading. It is equal to the difference between the cell height (as expressed by the **tmHeight** member) and the maximum height of any character in the font (excluding the height of accent marks).

tmExternalLeading
Specifies the recommended amount of leading for the font.

tmAveCharWidth
Specifies the average width of characters in the font (loosely defined as the width of the letter "X").

tmMaxCharWidth
Specifies the maximum width of any character in the font.

tmWeight
Specifies the weight of the font.

tmItalic
If nonzero, specifies an italic font.

tmUnderlined
If nonzero, specifies an underlined font.

tmStruckOut
If nonzero, specifies a font that has been struck through.

tmFirstChar
Specifies the value of the first character defined in the font.

tmLastChar
Specifies the value of the last character defined in the font.

tmDefaultChar
Specifies the value of the character that is to be substituted for characters that are not in the font.

tmBreakChar
Specifies the value of the character that is to be used to define word breaks for text justification.

tmPitchAndFamily
Specifies the font pitch and font family. This member is a combination of one pitch and one family value. The pitch value can be any one of the following values.

Value	Meaning
DEFAULT_PITCH (0x00)	Chooses a reasonable font.
FIXED_PITCH (0x01)	Chooses a fixed-pitch font.
VARIABLE_PITCH (0x02)	Chooses a variable-pitch font.

The font family, which describes in a general way the look of a font, can be any one of the following values.

Value	Meaning
FF_DONTCARE (0x00)	Chooses a reasonable font.
FF_ROMAN (0x10)	Chooses a variable-pitch font with serifs.
FF_SWISS (0x20)	Chooses a variable-pitch fonts without serifs.
FF_MODERN (0x30)	Chooses a fixed-pitch font.
FF_SCRIPT (0x40)	Chooses a cursive or script font.
FF_DECORATIVE (0x50)	Chooses a novelty font.

tmCharSet
Specifies the character set of the font. It can be one of the following values.

Value	Meaning
ANSI_CHARSET (0x00)	Indicates the ANSI character set.
SYMBOL_CHARSET (0x02)	Indicates the Symbol character set.
OEM_CHARSET (0xFF)	Indicates an OEM-specific character set. The characters and corresponding character codes depend on the computer.

tmOverhang
Specifies the amount of additional, synthesized width of a character or character string. This member may be is nonzero if the driver synthesizes character attributes, such as bold or italic, by modifying an existing font.

tmDigitizedAspectX
Specifies the horizontal aspect ratio for which this font was designed. This member is equal to the **dfHorizRes** member of the **FONTINFO** structure.

tmDigitizedAspectY
Specifies the vertical aspect ratio for which this font was designed. This member is equal to the **dfVertRes** member of the **FONTINFO** structure.

Comments

GDI makes a string bold by expanding the intracharacter spacing and overstriking with an offset; the overhang is the distance by which the overstrike is offset. GDI italicizes a font by skewing the string, and the overhang is the amount the top of the font is skewed past the bottom of the font.

See Also

EnumDFonts

TEXTXFORM

```
typedef struct tagTEXTXFORM {
    short   txfHeight;
    short   txfWidth;
    short   txfEscapement;
    short   txfOrientation;
    short   txfWeight;
    char    txfItalic;
    char    txfUnderline;
    char    txfStrikeOut;
    char    txfOutPrecision;
    char    txfClipPrecision;
    short   txfAccelerator;
    short   txfOverhang;
} TEXTXFORM;
```

The **TEXTXFORM** structure contains information describing the actual appearance of text as displayed by the device. The **StrBlt** and **ExtTextOut** functions check the the **TEXTXFORM** structure to determine what additional actions are required to generate the desired text from the specified physical font.

Members

txfHeight
Specifies the height of characters (ascent + descent) in device units.

txfWidth
Specifies the width in device units of the bounding box of the letter "X."

txfEscapement
Specifies the angle in tenths of a degree counterclockwise from the *x*-axis of the vector passing through the origin of all the characters in the string.

txfOrientation
Specifies the angle in tenths of a degree counterclockwise from the *x*-axis of the baseline of the character.

txfWeight
Specifies the weight of the font ranging from 1 to 1000, with 400 being the value for the standard font.

txfItalic
Specifies whether the font is to be italic. If the low bit is set, the font is to be italic. All the other bits must be zero.

txfUnderline
Specifies whether the font is to be underlined. If the low bit is set, the font is to be underlined. All the other bits must be zero.

txfStrikeOut
Specifies whether to strike out the font. If the low bit is set, the font is to be struck out. All the other bits must be zero.

txfOutPrecision
Specifies the required output precision for text. This member can have one of the following values.

Value	Meaning
OUT_DEFAULT_PRECIS (0x00)	Chooses a reasonable font.
OUT_STRING_PRECIS (0x01)	Chooses the font whose size (height and width) most closely matches the requested size. The driver may disregard the requested orientation and escapement, but other attributes must match.
OUT_CHARACTER_PRECIS (0x02)	Chooses the font whose size (height and width) most closely matches the requested size. The driver may disregard the requested orientation, but other attributes must match.
OUT_STROKE_PRECIS (0x03)	Chooses a font whose attributes exactly match the requested attributes.

txfClipPrecision
Specifies the required clipping precision for text. This member can have one of the following values.

Value	Meaning
CLIP_DEFAULT_PRECIS (0x00)	Chooses a reasonable font.
CLIP_CHARACTER_PRECIS (0x01)	Chooses a font that allows clipping of individual characters. The driver must be able to clip a character if any portion of it lies outside the clipping rectangle.
CLIP_STROKE_PRECIS (0x02)	Chooses a font that allows clipping of portions of a character. The driver must be able to clip any portion of a character that lies outside the clipping rectangle.

txfAccelerator
Specifies the requested text modifications using the same format as the the **dpText** member in the **GDIINFO** structure. Each bit in this member is set if the corresponding ability is required to modify the physical font into the requested font.

txfOverhang
Specifies same information as the **tmOverhang** member in the **TEXTMETRIC** structure. This member is set by the device for device-realized fonts and is in device units. GDI uses additional overhang if it makes the font bold.

Comments

Although most of the members in the **TEXTXFORM** structure correspond to the members in the **LFONT** structure, these members may not always exactly match. For example, if the logical font specified a 19-unit font at string precision and the closest available was a 9-unit font on a device capable of doubling, then the **txfHeight** member in the structure is 18.

A driver should check the **dpText** member in its **GDIINFO** structure to determine whether the driver can carry out the requested text modifications. In particular, the driver should check the bitwise difference between the **txfAccelerator** member and the **dpText** member to determine what abilities it should simulate. If the driver can not carry out the modifications, GDI is responsible for simulating the required modifications.

See Also

EngineRealizeFont, ExtTextOut, StrBlt