Contents

Windows GDI

4/20/2022 • 2 minutes to read • Edit Online

Purpose

The Microsoft Windows graphics device interface (GDI) enables applications to use graphics and formatted text on both the video display and the printer. Windows-based applications do not access the graphics hardware directly. Instead, GDI interacts with device drivers on behalf of applications.

Where applicable

GDI can be used in all Windows-based applications.

Developer audience

This API is designed for use by C/C++ programmers. Familiarity with the Windows message-driven architecture is required.

Run-time requirements

For information on which operating systems are required to use a particular function, see the Requirements section of the documentation for the function.

In this section

- Bitmaps
- Brushes
- Clipping
- Colors
- Coordinate Spaces and Transformations
- Device Contexts
- Filled Shapes
- Fonts and Text
- Lines and Curves
- Metafiles
- Multiple Display Monitors
- Painting and Drawing
- Paths
- Pens
- Printing and Print Spooler
- Rectangles
- Regions

Related topics

DirectX

GDI+

OpenGL

Windows Image Acquisition

Security Considerations: GDI

4/20/2022 • 2 minutes to read • Edit Online

This topic provides information about security considerations related to GDI. This topic does not provide all you need to know about security issues. Instead, use it as a starting point and reference for this technology area.

GDI generally has few security concerns because it deals with display rather than input. However, here are a few issues that you should consider.

Bitmaps, metafiles, and fonts are complex structures that could become corrupted. It is good practice to try to ensure that these items are uncorrupted and from a trustworthy source.

An application can specify the security descriptor for some of the printing and spooling APIs. You should take care when setting the security descriptor.

Related topics

Microsoft Security Central

Security Developer Center

Security TechCenter

Bitmaps (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

A *bitmap* is a graphical object used to create, manipulate (scale, scroll, rotate, and paint), and store images as files on a disk. This overview describes the bitmap classes and bitmap operations.

- About Bitmaps
- Using Bitmaps
- Bitmap Reference

About Bitmaps

4/20/2022 • 3 minutes to read • Edit Online

A bitmap is one of the GDI objects that can be selected into a *device context* (DC). Device contexts are structures that define a set of graphic objects and their associated attributes, and graphic modes that affect output. The table below describes the GDI objects that can be selected into a device context.

GRAPHIC OBJECT	DESCRIPTION
Bitmaps	Creates, manipulates (scale, scroll, rotate, and paint), and stores images as files on a disk.
Brushes	Paints the interior of polygons, ellipses, and paths.
Fonts	Draws text on video displays and other output devices.
Logical Palette	A color palette created by an application and associated with a given device context.
Paths	One or more figures (or shapes) that are filled and/or outlined.
Pens	A graphics tool that an application uses to draw lines and curves.
Regions	A rectangle, polygon, or ellipse (or a combination of two or more of these shapes) that can be filled, painted, inverted, framed, and used to perform hit testing (testing for the cursor location).

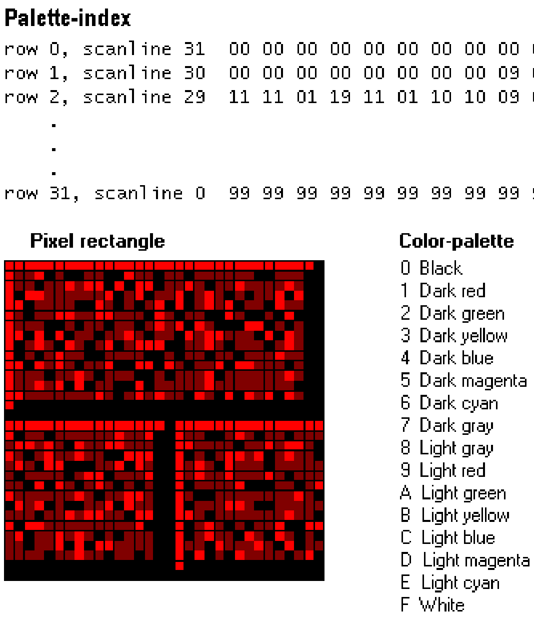
From a developer's perspective, a bitmap consists of a collection of structures that specify or contain the following elements:

- A header that describes the resolution of the device on which the rectangle of pixels was created, the dimensions of the rectangle, the size of the array of bits, and so on.
- A logical palette.
- An array of bits that defines the relationship between pixels in the bitmapped image and entries in the logical palette.

A bitmap size is related to the type of image it contains. Bitmap images can be either monochrome or color. In an image, each pixel corresponds to one or more bits in a bitmap. Monochrome images have a ratio of 1 bit per pixel (bpp). Color imaging is more complex. The number of colors that can be displayed by a bitmap is equal to two raised to the number of bits per pixel. Thus, a 256-color bitmap requires 8 bpp (2^8 = 256).

Control Panel applications are examples of applications that use bitmaps. When you select a background (or wallpaper) for your desktop, you actually select a bitmap, which the system uses to paint the desktop background. The system creates the selected background pattern by repeatedly drawing a 32-by-32 pixel pattern on the desktop.

The following illustration shows the developer's perspective of the bitmap found in the file Redbrick.bmp. It shows a palette array, a 32-by-32 pixel rectangle, and the index array that maps colors from the palette to pixels in the rectangle.



Palette-index

```
row 0, scanline 31  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
row 1, scanline 30  00 00 00 00 00 00 00 00 09 00 00 00 00 00 00 00
row 2, scanline 29  11 11 01 19 11 01 10 10 09 09 01 09 11 11 01 90
    .
    .
    .
row 31, scanline 0  99 99 99 99 99 99 99 99 99 99 99 99 99 99 99 90
```

Pixel rectangle

Color-palette

0	Black
1	Dark red
2	Dark green
3	Dark yellow
4	Dark blue
5	Dark magenta
6	Dark cyan
7	Dark gray
8	Light gray
9	Light red
A	Light green
B	Light yellow
C	Light blue
D	Light magenta
E	Light cyan
F	White

In the preceding example, the rectangle of pixels was created on a VGA display device using a palette of 16 colors. A 16-color palette requires 4-bit indexes; therefore, the array that maps palette colors to pixel colors is composed of 4-bit indexes as well. (For more information about logical color-palettes, see Colors.)

NOTE

In the above bitmap, the system maps indexes to pixels beginning with the bottom scan line of the rectangular region and ending with the top scan line. A *scan line* is a single row of adjacent pixels on a video display. For example, the first row of the array (row 0) corresponds to the bottom row of pixels, scan line 31. This is because the above bitmap is a bottom-up device-independent bitmap (DIB), a common type of bitmap. In top-down DIBs and in device-dependent bitmaps (DDB), the system maps indexes to pixels beginning with the top scan line.

The following topics describe different areas of bitmaps.

- Bitmap Classifications
- Bitmap Header Types
- JPEG and PNG Extensions for Specific Bitmap Functions and Structures
- Bitmaps, Device Contexts, and Drawing Surfaces
- Bitmap Creation
- Bitmap Rotation
- Bitmap Scaling
- Bitmaps as Brushes
- Bitmap Storage
- Bitmap Compression
- Alpha Blending
- Smooth Shading
- ICM-Enabled Bitmap Functions

Bitmap Classifications

4/20/2022 • 2 minutes to read • Edit Online

There are two classes of bitmaps:

- Device-independent bitmaps (DIB). The DIB file format was designed to ensure that bitmapped graphics created using one application can be loaded and displayed in another application, retaining the same appearance as the original.
- Device-dependent bitmaps (DDB), also known as GDI bitmaps, were the only bitmaps available in early versions of 16-bit Microsoft Windows (prior to version 3.0). However, as display technology improved and as the variety of available display devices increased, certain inherent problems surfaced which could only be solved using DIBs. For example, there was no method of storing (or retrieving) the resolution of the display type on which a bitmap was created, so a drawing application could not quickly determine whether a bitmap was suitable for the type of video display device on which the application was running.

  Despite these problems, DDBs (also known as compatible bitmaps) are still useful for better GDI performance and for other situations.

Device-Independent Bitmaps

4/20/2022 • 3 minutes to read • Edit Online

A device-independent bitmap (DIB) contains a *color table*. A color table describes how pixel values correspond to *RGB* color values, which describe colors that are produced by emitting light. Thus, a DIB can achieve the proper color scheme on any device. A DIB contains the following color and dimension information:

- The color format of the device on which the rectangular image was created.
- The resolution of the device on which the rectangular image was created.
- The palette for the device on which the image was created.
- An array of bits that maps red, green, blue (**RGB**) triplets to pixels in the rectangular image.
- A data-compression identifier that indicates the data compression scheme (if any) used to reduce the size of the array of bits.

The color and dimension information is stored in a **BITMAPINFO** structure, which consists of a **BITMAPINFOHEADER** structure followed by two or more **RGBQUAD** structures. The **BITMAPINFOHEADER** structure specifies the dimensions of the pixel rectangle, describes the device's color technology, and identifies the compression schemes used to reduce the size of the bitmap. The **RGBQUAD** structures identify the colors that appear in the pixel rectangle.

There are two varieties of DIBs:

- A bottom-up DIB, in which the origin lies at the lower-left corner.
- A top-down DIB, in which the origin lies at the upper-left corner.

If the height of a DIB, as indicated by the **Height** member of the bitmap information header structure, is a positive value, it is a bottom-up DIB; if the height is a negative value, it is a top-down DIB. Top-down DIBs cannot be compressed.

The color format is specified in terms of a count of color planes and color bits. The count of color planes is always 1; the count of color bits is 1 for monochrome bitmaps, 4 for VGA bitmaps, and 8, 16, 24, or 32 for bitmaps on other color devices. An application retrieves the number of color bits that a particular display (or printer) uses by calling the **GetDeviceCaps** function, specifying BITSPIXEL as the second argument.

The resolution of a display device is specified in pixels-per-meter. An application can retrieve the horizontal resolution for a video display, or printer, by following this three-step process.

1. Call the **GetDeviceCaps** function, specifying HORZRES as the second argument.
2. Call **GetDeviceCaps** a second time, specifying HORZSIZE as the second argument.
3. Divide the first return value by the second return value.

The application can retrieve the vertical resolution by using the same three-step process with different parameters: VERTRES in place of HORZRES, and VERTSIZE in place of HORZSIZE.

The palette is represented by an array of **RGBQUAD** structures that specify the red, green, and blue intensity components for each color in a display device's color palette. Each color index in the palette array maps to a specific pixel in the rectangular region associated with the bitmap. The size of this array, in bits, is equivalent to the width of the rectangle, in pixels, multiplied by the height of the rectangle, in pixels, multiplied by the count of color bits for the device. An application can retrieve the size of the device's palette by calling the **GetDeviceCaps** function, specifying NUMCOLORS as the second argument.

Windows supports the compression of the palette array for 8-bpp and 4-bpp bottom-up DIBs. These arrays can be compressed by using the run-length encoding (RLE) scheme. The RLE scheme uses 2-byte values, the first byte specifying the number of consecutive pixels that use a color index and the second byte specifying the index. For more information about bitmap compression, see the description of the **BITMAPINFOHEADER**, **BITMAPFILEHEADER**, **BITMAPV4HEADER**, and **BITMAPV5HEADER** structures.

An application can create a DIB from a DDB by initializing the required structures and calling the **GetDIBits** function. To determine whether a device supports this function, call the **GetDeviceCaps** function, specifying RC_DI_BITMAP as the RASTERCAPS flag.

An application that needs to copy a bitmap can use **TransparentBlt** to copy all pixels in a source bitmap to a destination bitmap except those pixels that match the transparent color.

An application can use a DIB to set pixels on the display device by calling the **SetDIBitsToDevice** or the **StretchDIBits** function. To determine whether a device supports the **SetDIBitsToDevice** function, call the **GetDeviceCaps** function, specifying RC_DIBTODEV as the RASTERCAPS flag. Specify RC_STRETCHDIB as the RASTERCAPS flag to determine if the device supports **StretchDIBits**.

An application that simply needs to display a pre-existing DIB can use the **SetDIBitsToDevice** function. For example, a spreadsheet application can open existing charts and display them in a window by using the **SetDIBitsToDevice** function. To repeatedly redraw a bitmap in a window, however, the application should use the **BitBlt** function. For example, a multimedia application that combines animated graphics with sound would benefit from calling the **BitBlt** function because it executes faster than **SetDIBitsToDevice**.

Device-Dependent Bitmaps

4/20/2022 • 2 minutes to read • Edit Online

Device-dependent bitmaps (DDBs) are described by using a single structure, the **BITMAP** structure. The members of this structure specify the width and height of a rectangular region, in pixels; the width of the array that maps entries from the device palette to pixels; and the device's color format, in terms of color planes and bits per pixel. An application can retrieve the color format of a device by calling the **GetDeviceCaps** function and specifying the appropriate constants. Note that a DDB does not contain color values; instead, the colors are in a device-dependent format. For more information, see Color in Bitmaps. Because each device can have its own set of colors, a DDB created for one device may not display well on a different device.

To use a DDB in a device context, it must have the color organization of that device context. Thus, a DDB is often called a *compatible bitmap* and it usually has better GDI performance than a DIB. For example, to create a bitmap for video memory, it is best to use a compatible bitmap with the same color format as the primary display. Once in video memory, rendering to the bitmap and displaying it to the screen are significantly faster than from a system memory surface or directly from a DIB.

In addition to enabling better GDI performance, compatible bitmaps are used to capture images (see Capturing an Image) and to create bitmaps at run time for menus see "Creating the Bitmap" in (see Using Menus).

To transfer a bitmap between devices with different color organization, use **GetDIBits** to convert the compatible bitmap to a DIB and call **SetDIBits** or **StretchDIBits** to display the DIB to the second device.

There are two types of DDBs: discardable and nondiscardable. A discardable DDB is a bitmap that the system discards if the bitmap is not selected into a DC and if system memory is low. The **CreateDiscardableBitmap** function creates discardable bitmaps. The **CreateBitmap**, **CreateCompatibleBitmap**, and **CreateBitmapIndirect** functions create nondiscardable bitmaps.

An application can create a DDB from a DIB by initializing the required structures and calling the **CreateDIBitmap** function. Specifying CBM_INIT in the call to **CreateDIBitmap** is equivalent to calling the **CreateCompatibleBitmap** function to create a DDB in the format of the device and then calling the **SetDIBits** function to translate the DIB bits to the DDB. To determine whether a device supports the **SetDIBits** function, call the **GetDeviceCaps** function, specifying RC_DI_BITMAP as the RASTERCAPS flag.

Bitmap Header Types

4/20/2022 • 4 minutes to read • Edit Online

The bitmap has four basic header types:

- BITMAPCOREHEADER
- BITMAPINFOHEADER
- BITMAPV4HEADER
- BITMAPV5HEADER

The four types of bitmap headers are differentiated by the **Size** member, which is the first **DWORD** in each of the structures.

The BITMAPV5HEADER structure is an extended BITMAPV4HEADER structure, which is an extended BITMAPINFOHEADER structure. However, the **BITMAPINFOHEADER** and BITMAPCOREHEADER have only the **Size** member in common with other bitmap header structures.

The BITMAPCOREHEADER and BITMAPV4HEADER formats have been superseded by BITMAPINFOHEADER and BITMAPV5HEADER formats, respectively. The **BITMAPCOREHEADER** and **BITMAPV4HEADER** formats are presented for completeness and backward compatibility.

The format for a DIB is the following (for more information, see Bitmap Storage):

- a BITMAPFILEHEADER structure
- either a BITMAPCOREHEADER, a BITMAPINFOHEADER, a BITMAPV4HEADER, or a BITMAPV5HEADER structure.
- an optional color table, which is either a set of RGBQUAD structures or a set of RGBTRIPLE structures.
- the bitmap data
- optional Profile data

A color table describes how pixel values correspond to RGB color values. RGB is a model for describing colors that are produced by emitting light.

Profile data refers to either the profile file name (linked profile) or the actual profile bits (embedded profile). The file format places the profile data at the end of the file. The profile data is placed just after the color table (if present). However, if the function receives a packed DIB, the profile data comes after the bitmap bits, like in the file format.

Profile data will only exist for BITMAPV5HEADER structures where **bV5CSType** is PROFILE_LINKED or PROFILE_EMBEDDED. For functions that receive packed DIBs, the profile data comes after the bitmap data.

A palettized device is any device that uses palettes to assign colors. The classic example of a palettized device is a display running in 8 bit color depth (that is, 256 colors). The display in this mode uses a small color table to assign colors to a bitmap. The colors in a bitmap are assigned to the closest color in the palette that the device is using. The palettized device does not create an optimal palette for displaying the bitmap; it simply uses whatever is in the current palette. Applications are responsible for creating a palette and selecting it into the system. In general, 16-, 24-, and 32-bits-per-pixel (bpp) bitmaps do not contain color tables (a.k.a. optimal palettes for the bitmap); the application is responsible for generating a optimal palette in this case. However, 16-, 24-, and 32-bpp bitmaps can contain such optimal color tables for displaying on palettized devices; in this case the application just needs to create a palette based off the color table present in the bitmap file.

Bitmaps that are 1, 4, or 8 bpp must have a color table with a maximum size based on the bpp. The maximum size for 1, 4, and 8 bpp bitmaps is 2 to the power of the bpp. Thus, a 1 bpp bitmap has a maximum of two colors, the 4 bpp bitmap has a maximum of 16 colors, and the 8 bpp bitmap has a maximum of 256 colors.

Bitmaps that are 16-, 24-, or 32-bpp do not require color tables, but may have them to specify colors for palettized devices. If a color table is present for 16-, 24-, or 32-bpp bitmap, the **biClrUsed** member specifies the size of the color table and the color table must have that many colors in it. If **biClrUsed** is zero, there is no color table.

The red, green, and blue bitfield masks for BI_BITFIELD bitmaps immediately follow the BITMAPINFOHEADER, BITMAPV4HEADER, and BITMAPV5HEADER structures. The **BITMAPV4HEADER** and **BITMAPV5HEADER** structures contain additional members for red, green, and blue masks as follows.

MEMBER	MEANING
RedMask	Color mask that specifies the red component of each pixel, valid only if the **Compression** member is set to BI_BITFIELDS.
GreenMask	Color mask that specifies the green component of each pixel, valid only if the **Compression** member is set to BI_BITFIELDS.
BlueMask	Color mask that specifies the blue component of each pixel, valid only if the **Compression** member is set to BI_BITFIELDS.

When the **biCompression** member of BITMAPINFOHEADER is set to BI_BITFIELDS and the function receives an argument of type **LPBITMAPINFO**, the color masks will immediately follow the header. The color table, if present, will follow the color masks. BITMAPCOREHEADER bitmaps do not support color masks.

By default, bitmap data is bottom-up in its format. Bottom-up means that the first scan line in the bitmap data is the last scan line to be displayed. For example, the 0^{th} pixel of the 0^{th} scan line of the bitmap data of a 10-pixel by 10-pixel bitmap will be the 0^{th} pixel of the 9^{th} scan line of the displayed or printed image. Run-length encoded (RLE) format bitmaps and BITMAPCOREHEADER bitmaps cannot be top-down bitmaps. The scan lines are **DWORD** aligned, except for RLE-compressed bitmaps. They must be padded for scan line widths, in bytes, that are not evenly divisible by four, except for RLE compressed bitmaps. For example, a 10- by 10-pixel 24-bpp bitmap will have two padding bytes at the end of each scan line.

JPEG and PNG Extensions for Specific Bitmap Functions and Structures

4/20/2022 • 2 minutes to read • Edit Online

On certain versions of Microsoft Windows, the **StretchDIBits** and **SetDIBitsToDevice** functions allow JPEG and PNG images to be passed as the source image to printer devices. This extension is not intended as a means to supply general JPEG and PNG decompression to applications, but rather to allow applications to send JPEG- and PNG-compressed images directly to printers having hardware support for JPEG and PNG images.

The **BITMAPINFOHEADER**, **BITMAPV4HEADER** and **BITMAPV5HEADER** structures are extended to allow specification of **biCompression** values indicating that the bitmap data is a JPEG or PNG image. These compression values are only valid for **SetDIBitsToDevice** and **StretchDIBits** when the *hdc* parameter specifies a printer device. To support metafile spooling of the printer, the application should not rely on the return value to determine whether the device supports the JPEG or PNG file. The application must issue QUERYESCSUPPORT with the corresponding escape before calling **SetDIBitsToDevice** and **StretchDIBits**. If the validation escape fails, the application must then fall back on its own JPEG or PNG support to decompress the image into a bitmap.

Bitmaps, Device Contexts, and Drawing Surfaces

4/20/2022 • 2 minutes to read • Edit Online

A *device context* (DC) is a data structure defining the graphics objects, their associated attributes, and the graphics modes affecting output on a device. To create a DC, call the **CreateDC** function; to retrieve a DC, call the **GetDC** function.

Before returning a handle that identifies that DC, the system selects a drawing surface into the DC. If the application called the **CreateDC** function to create a device context for a VGA display, the dimensions of this drawing surface are 640-by-480 pixels. If the application called the **GetDC** function, the dimensions reflect the size of the client area.

Before an application can begin drawing, it must select a bitmap with the appropriate width and height into the DC by calling the **SelectObject** function. When an application passes the handle to the DC to one of the graphics device interface (GDI) drawing functions, the requested output appears on the drawing surface selected into the DC.

For more information, see Memory Device Contexts.

Bitmap Creation

4/20/2022 • 2 minutes to read • Edit Online

To create a bitmap, use the **CreateBitmap**, **CreateBitmapIndirect**, or **CreateCompatibleBitmap** function, **CreateDIBitmap**, and **CreateDiscardableBitmap**.

These functions allow you to specify the width and height, in pixels, of the bitmap. The **CreateBitmap** and **CreateBitmapIndirect** function also allow you to specify the number of color planes and the number of bits required to identify the color. On the other hand, the **CreateCompatibleBitmap** and **CreateDiscardableBitmap** functions use a specified device context to obtain the number of color planes and the number of bits required to identify the color.

The **CreateDIBitmap** function creates a device-dependent bitmap from a device-independent bitmap. It contains a color table that describes how pixel values correspond to RGB color values. For more information, see Device-Dependent Bitmaps and Device-Independent Bitmaps.

After the bitmap has been created, you cannot change its size, number of color planes, or number of bits required to identify the color.

When you no longer need a bitmap, call the **DeleteObject** function to delete it.

Bitmap Rotation

4/20/2022 • 2 minutes to read • Edit Online

To copy a bitmap into a parallelogram; use the **PlgBlt** function, which performs a bit-block transfer from a rectangle in a source device context into a parallelogram in a destination device context. To rotate the bitmap, an application must provide the coordinates, in world units, to be used for the corners of the parallelogram. (For more information about rotation and world units, see Coordinate Spaces and Transformations.)

Bitmap Scaling

4/20/2022 • 2 minutes to read • Edit Online

The **StretchBlt** function scales a bitmap by performing a bit-block transfer from a rectangle in a source device context into a rectangle in a destination device context. However, unlike the **BitBlt** function, which duplicates the source rectangle dimensions in the destination rectangle, **StretchBlt** allows an application to specify the dimensions of both the source and the destination rectangles. When the destination bitmap is smaller than the source bitmap, the system combines rows or columns of color data (or both) in the bitmap before rendering the corresponding image on the display device. The system combines the color data according to the specified stretch mode, which the application defines by calling the **SetStretchBltMode** function. When the destination bitmap is larger than the source bitmap, the system scales or magnifies each pixel in the resultant image accordingly.

Bitmaps as Brushes

4/20/2022 • 2 minutes to read • Edit Online

A number of functions use the brush currently selected into a device context to perform bitmap operations. For example, the **PatBlt** function replicates the brush in a rectangular region within a window, and the **FloodFill** function replicates the brush inside an area in a window bounded by the specified color (unlike **PatBlt**, **FloodFill** does fill nonrectangular shapes).

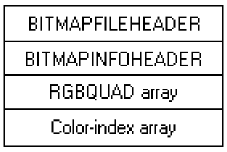
The **FloodFill** function replicates the brush within a region bounded by a specified color. However, unlike the **PatBlt** function, **FloodFill** does not combine the color data for the brush with the color data for the pixels on the display; it simply sets the color of all pixels within the enclosed region on the display to the color of the brush that is currently selected into the device context.

Bitmap Storage

4/20/2022 • 5 minutes to read • Edit Online

Bitmaps should be saved in a file that uses the established bitmap file format and assigned a name with the three-character .bmp extension. The established bitmap file format consists of a **BITMAPFILEHEADER** structure followed by a **BITMAPINFOHEADER**, **BITMAPV4HEADER**, or **BITMAPV5HEADER** structure. An array of **RGBQUAD** structures (also called a color table) follows the bitmap information header structure. The color table is followed by a second array of indexes into the color table (the actual bitmap data).

The bitmap file format is shown in the following illustration.



The members of the **BITMAPFILEHEADER** structure identify the file; specify the size of the file, in bytes; and specify the offset from the first byte in the header to the first byte of bitmap data. The members of the **BITMAPINFOHEADER**, **BITMAPV4HEADER**, or **BITMAPV5HEADER** structure specify the width and height of the bitmap, in pixels; the color format (count of color planes and color bits-per-pixel) of the display device on which the bitmap was created; whether the bitmap data was compressed before storage and the type of compression used; the number of bytes of bitmap data; the resolution of the display device on which the bitmap was created; and the number of colors represented in the data. The **RGBQUAD** structures specify the RGB intensity values for each of the colors in the device's palette.

The color-index array associates a color, in the form of an index to an **RGBQUAD** structure, with each pixel in a bitmap. Thus, the number of bits in the color-index array equals the number of pixels times the number of bits needed to index the **RGBQUAD** structures. For example, an 8x8 black-and-white bitmap has a color-index array of 8 * 8 * 1 = 64 bits, because one bit is needed to index two colors. The Redbrick.bmp, mentioned in About Bitmaps, is a 32x32 bitmap with 16 colors; its color-index array is 32 * 32 * 4 = 4096 bits because four bits index 16 colors.

To create a color-index array for a top-down bitmap, start at the top line in the bitmap. The index of the **RGBQUAD** for the color of the left-most pixel is the first *n* bits in the color-index array (where *n* is the number of bits needed to indicate all of the **RGBQUAD** structures). The color of the next pixel to the right is the next *n* bits in the array, and so forth. After you reach the right-most pixel in the line, continue with the left-most pixel in the line below. Continue until you finish with the entire bitmap. If it is a bottom-up bitmap, start at the bottom line of the bitmap instead of the top line, still going from left to right, and continue to the top line of the bitmap.

The following hexadecimal output shows the contents of the file Redbrick.bmp.

```
0000    42 4d 76 02 00 00 00 00  00 00 76 00 00 00 28 00
0010    00 00 20 00 00 00 20 00  00 00 01 00 04 00 00 00
0020    00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0030    00 00 00 00 00 00 00 00  00 00 00 00 80 00 00 80
0040    00 00 00 80 80 00 80 00  00 00 80 00 80 00 80 80
0050    00 00 80 80 80 00 c0 c0  c0 00 00 00 ff 00 00 ff
0060    00 00 00 ff ff 00 ff 00  00 00 ff 00 ff 00 ff ff
0070    00 00 ff ff ff 00 00 00  00 00 00 00 00 00 00 00
0080    00 00 00 00 00 00 00 00  00 00 00 00 00 00 09 00
0090    00 00 00 00 00 00 11 11  01 19 11 01 10 10 09 09
00a0    01 09 11 11 01 90 11 01  19 09 09 91 11 10 09 11
00b0    09 11 19 10 90 11 19 01  19 19 10 10 11 10 09 01
00c0    91 10 91 09 10 10 90 99  11 11 11 11 19 00 09 01
00d0    91 01 01 19 00 99 11 10  11 91 99 11 09 90 09 91
00e0    01 11 11 11 91 10 09 19  01 00 11 90 91 10 09 01
00f0    11 99 10 01 11 11 91 11  11 19 10 11 99 10 09 10
0100    01 11 11 11 19 10 11 09  09 10 19 10 10 10 09 01
0110    11 19 00 01 10 19 10 11  11 01 99 01 11 90 09 19
0120    11 91 11 91 01 11 19 10  99 00 01 19 09 10 09 19
0130    10 91 11 01 11 11 91 01  91 19 11 00 99 90 09 01
0140    01 99 19 01 91 10 19 91  91 09 11 99 11 10 09 91
0150    11 10 11 91 99 10 90 11  01 11 11 19 11 90 09 11
0160    00 19 10 11 01 11 99 99  99 99 99 99 99 99 09 99
0170    99 99 99 99 99 99 00 00  00 00 00 00 00 00 00 00
0180    00 00 00 00 00 00 90 00  00 00 00 00 00 00 00 00
0190    00 00 00 00 00 00 99 11  11 11 19 10 19 19 11 09
01a0    10 90 91 90 91 00 91 19  19 09 01 10 09 01 11 11
01b0    91 11 11 11 10 00 91 11  01 19 10 11 10 01 01 11
01c0    90 11 11 11 91 00 99 09  19 10 11 90 09 90 91 01
01d0    19 09 91 11 01 00 90 10  19 11 00 11 11 00 10 11
01e0    01 10 11 19 11 00 90 19  10 91 01 90 19 99 00 11
01f0    91 01 11 01 91 00 99 09  09 01 10 11 91 01 10 91
0200    99 11 10 90 91 00 91 11  00 10 11 01 10 19 19 09
0210    10 00 99 01 01 00 91 01  19 91 19 91 11 09 10 11
0220    00 91 00 10 90 00 99 01  11 10 09 10 10 19 09 01
0230    91 90 11 09 11 00 90 99  11 11 11 90 19 01 19 01
0240    91 01 01 19 09 00 91 10  11 91 99 09 09 90 11 91
0250    01 19 11 11 91 00 91 19  01 00 11 00 91 10 11 01
0260    11 11 10 01 11 00 99 99  99 99 99 99 99 99 99 99
0270    99 99 99 99 99 90
```

The following table shows the data bytes associated with the structures in a bitmap file.

STRUCTURE	CORRESPONDING BYTES
BITMAPFILEHEADER	0x00 0x0D
BITMAPINFOHEADER	0x0E 0x35
RGBQUAD array	0x36 0x75
Color-index array	0x76 0x275

Bitmap Compression

4/20/2022 • 3 minutes to read • Edit Online

Windows supports formats for compressing bitmaps that define their colors with 8 or 4 bits-per-pixel. Compression reduces the disk and memory storage required for the bitmap.

When the **Compression** member of the bitmap information header structure is BI_RLE8, a run-length encoding (RLE) format is used to compress an 8-bit bitmap. This format can be compressed in encoded or absolute modes. Both modes can occur anywhere in the same 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 character that denotes the end of a line, the end of a bitmap, or a delta, depending on the value of the second byte. The interpretation of the escape depends on the value of the second byte of the pair, which can be one of the following values.

VALUE	MEANING
0	End of line.
1	End of bitmap.
2	Delta. The 2 bytes following the escape contain unsigned values indicating the offset to the right and up of the next pixel from the current position.

- In *absolute mode*, the first byte is zero and the second byte is a value in the range 03H through FFH. The second byte represents the number of bytes that follow, each of which contains the color index of a single pixel. When the second byte is two or less, the escape has the same meaning as encoded mode. In absolute mode, each run must be zero-padded to end on a 16-bit word 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]
```

The bitmap expands 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 up
78 78
end of line
1E 1E 1E 1E 1E 1E 1E 1E 1E
end of RLE bitmap
```

When the **Compression** member is BI_RLE4, the bitmap is compressed by 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 4 bits and one in its low-order 4 bits. The first of the pixels is drawn using the color specified by the high-order 4 bits, the second is drawn using the color in the low-order 4 bits, the third is drawn using the color in the high-order 4 bits, and so on, until all the pixels specified by the first byte have been drawn.
- In absolute mode, the first byte is zero. The second byte contains the number of color indexes that follow. Subsequent bytes contain color indexes in their high- and low-order 4 bits, 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 described for BI_RLE8 also apply to BI_RLE4 compression.

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
```

The bitmap expands 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 up
7 8 7 8
end of line
1 E 1 E 1 E 1 E 1
end of RLE bitmap
```

Alpha Blending (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

Alpha blending is used to display an alpha bitmap, which is a bitmap that has transparent or semi-transparent pixels. In addition to a red, green, and blue color channel, each pixel in an alpha bitmap has a transparency component known as its *alpha channel*. The alpha channel typically contains as many bits as a color channel. For example, an 8-bit alpha channel can represent 256 levels of transparency, from 0 (the entire bitmap is transparent) to 255 (the entire bitmap is opaque).

Alpha blending mechanisms are invoked by calling **AlphaBlend**, which references the **BLENDFUNCTION** structure.

Alpha values per pixel are only supported for 32-bpp BI_RGB. This formula is defined as:

```
typedef struct {
  BYTE   Blue;
  BYTE   Green;
  BYTE   Red;
  BYTE   Alpha;
};
```

This is represented in memory as shown in the following table.

31:24	23:16	15:08	07:00
Alpha	Red	Green	Blue

Bitmaps may also be displayed with a transparency factor applied to the entire bitmap. Any bitmap format can be displayed with a global constant alpha value by setting **SourceConstantAlpha** in the **BLENDFUNCTION** structure. The global constant alpha value has 256 levels of transparency, from 0 (entire bitmap is completely transparent) to 255 (entire bitmap is completely opaque). The global constant alpha value is combined with the per-pixel alpha value.

For an example, see Alpha Blending a Bitmap.

Smooth Shading

4/20/2022 • 2 minutes to read • Edit Online

Smooth shading is a method of shading a region with a color gradient. Including color information, along with the bounds of drawing primitive, specifies the color gradient. GDI linearly interpolates the color of the inside of the primitive passed on the color endpoints. Color and vertex information is included with position information in the **TRIVERTEX** structure.

Use the **GradientFill** function to fill a triangle or rectangle structure. To fill a triangle with smooth shading, call **GradientFill** with the three triangle endpoints. To fill a rectangle with smooth shading, call **GradientFill** with the upper-left and lower-right rectangle coordinates. **GradientFill** references the **TRIVERTEX**, **GRADIENT_RECT**, and **GRADIENT_TRIANGLE** structures.

For an example, see Drawing a Shaded Triangle.

ICM-Enabled Bitmap Functions

4/20/2022 • 2 minutes to read • Edit Online

Microsoft Image Color Management (ICM) ensures that a color image, graphic object, or text object is rendered as closely as possible to its original intent on any device, despite differences in imaging technologies and color capabilities between devices. Whether you are scanning an image or other graphic on a color scanner, downloading it over the Internet, viewing or editing it onscreen, or printing it on paper, film, or other media, ICM 2.0 helps you keep colors consistent and accurate. For more information about ICM, see Windows Color System.

There are various functions in the graphics device interface (GDI) that use or operate on color data. The following bitmap functions are enabled for use with ICM:

- BitBlt
- CreateDIBitmap
- CreateDIBSection
- MaskBlt
- SetDIBColorTable
- StretchBlt
- SetDIBits
- SetDIBitsToDevice
- StretchDIBits

Using Bitmaps

4/20/2022 • 2 minutes to read • Edit Online

- Capturing an Image
- Scaling an Image
- Storing an Image
- Alpha Blending a Bitmap
- Drawing a Shaded Rectangle
- Drawing a Shaded Triangle
- Testing a Printer for JPEG or PNG Support
- Sizing a JPEG or PNG Image

Capturing an Image

4/20/2022 • 7 minutes to read • Edit Online

You can use a bitmap to capture an image, and you can store the captured image in memory, display it at a different location in your application's window, or display it in another window.

In some cases, you may want your application to capture images and store them only temporarily. For example, when you scale or zoom a picture created in a drawing application, the application must temporarily save the normal view of the image and display the zoomed view. Later, when the user selects the normal view, the application must replace the zoomed image with a copy of the normal view that it temporarily saved.

To store an image temporarily, your application must call **CreateCompatibleDC** to create a DC that is compatible with the current window DC. After you create a compatible DC, you create a bitmap with the appropriate dimensions by calling the **CreateCompatibleBitmap** function and then select it into this device context by calling the **SelectObject** function.

After the compatible device context is created and the appropriate bitmap has been selected into it, you can capture the image. The **BitBlt** function captures images. This function performs a bit block transfer that is, it copies data from a source bitmap into a destination bitmap. However, the two arguments to this function are not bitmap handles. Instead, **BitBlt** receives handles that identify two device contexts and copies the bitmap data from a bitmap selected into the source DC into a bitmap selected into the target DC. In this case, the target DC is the compatible DC, so when **BitBlt** completes the transfer, the image has been stored in memory. To redisplay the image, call **BitBlt** a second time, specifying the compatible DC as the source DC and a window (or printer) DC as the target DC.

Code example

This section contains a code example that captures an image of the entire desktop, scales it down to the current window size, and then saves it to a file (as well as displaying it in the client area).

To try out the code example, begin by creating a new project in Visual Studio based on the **Windows Desktop Application** project template. It's important to name the new project `GDI_CapturingAnImage` so that the code listing below will compile (for example, it includes `GDI_CapturingAnImage.h`, which will exist in your new project if you name it as suggested).

Open the `GDI_CapturingAnImage.cpp` source code file in your new project, and replace its contents with the listing below. Then build and run. Each time you resize the window, you'll see the captured screenshot displayed in the client area.

```
// GDI_CapturingAnImage.cpp : Defines the entry point for the application.
//

#include "framework.h"
#include "GDI_CapturingAnImage.h"

#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE hInst;                                // current instance
WCHAR szTitle[MAX_LOADSTRING];                  // The title bar text
WCHAR szWindowClass[MAX_LOADSTRING];            // the main window class name

// Forward declarations of functions included in this code module:
ATOM                MyRegisterClass(HINSTANCE hInstance);
```

```
BOOL                InitInstance(HINSTANCE, int);
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK    About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY wWinMain(_In_ HINSTANCE hInstance,
    _In_opt_ HINSTANCE hPrevInstance,
    _In_ LPWSTR    lpCmdLine,
    _In_ int       nCmdShow)
{
    UNREFERENCED_PARAMETER(hPrevInstance);
    UNREFERENCED_PARAMETER(lpCmdLine);

    // TODO: Place code here.

    // Initialize global strings
    LoadStringW(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadStringW(hInstance, IDC_GDICAPTURINGANIMAGE, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance);

    // Perform application initialization:
    if (!InitInstance(hInstance, nCmdShow))
    {
        return FALSE;
    }

    HACCEL hAccelTable = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDC_GDICAPTURINGANIMAGE));

    MSG msg;

    // Main message loop:
    while (GetMessage(&msg, nullptr, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }

    return (int)msg.wParam;
}

//
//  FUNCTION: MyRegisterClass()
//
//  PURPOSE: Registers the window class.
//
ATOM MyRegisterClass(HINSTANCE hInstance)
{
    WNDCLASSEXW wcex;

    wcex.cbSize = sizeof(WNDCLASSEX);

    wcex.style = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc = WndProc;
    wcex.cbClsExtra = 0;
    wcex.cbWndExtra = 0;
    wcex.hInstance = hInstance;
    wcex.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_GDICAPTURINGANIMAGE));
    wcex.hCursor = LoadCursor(nullptr, IDC_ARROW);
    wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wcex.lpszMenuName = MAKEINTRESOURCEW(IDC_GDICAPTURINGANIMAGE);
    wcex.lpszClassName = szWindowClass;
    wcex.hIconSm = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_SMALL));

    return RegisterClassExW(&wcex);
}

//
```

```
//   FUNCTION: InitInstance(HINSTANCE, int)
//
//   PURPOSE: Saves instance handle and creates main window
//
//   COMMENTS:
//
//        In this function, we save the instance handle in a global variable and
//        create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    hInst = hInstance; // Store instance handle in our global variable

    HWND hWnd = CreateWindowW(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, nullptr, nullptr, hInstance, nullptr);

    if (!hWnd)
    {
        return FALSE;
    }

    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    return TRUE;
}

// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF
// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO
// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
// PARTICULAR PURPOSE.
//
// Copyright (c) Microsoft Corporation. All rights reserved

//
//   FUNCTION: CaptureAnImage(HWND hWnd)
//
//   PURPOSE: Captures a screenshot into a window ,and then saves it in a .bmp file.
//
//   COMMENTS:
//
//      Note: This function attempts to create a file called captureqwsx.bmp
//

int CaptureAnImage(HWND hWnd)
{
    HDC hdcScreen;
    HDC hdcWindow;
    HDC hdcMemDC = NULL;
    HBITMAP hbmScreen = NULL;
    BITMAP bmpScreen;
    DWORD dwBytesWritten = 0;
    DWORD dwSizeofDIB = 0;
    HANDLE hFile = NULL;
    char* lpbitmap = NULL;
    HANDLE hDIB = NULL;
    DWORD dwBmpSize = 0;

    // Retrieve the handle to a display device context for the client
    // area of the window.
    hdcScreen = GetDC(NULL);
    hdcWindow = GetDC(hWnd);

    // Create a compatible DC, which is used in a BitBlt from the window DC.
    hdcMemDC = CreateCompatibleDC(hdcWindow);

    if (!hdcMemDC)
    {
        MessageBox(hWnd, L"CreateCompatibleDC has failed", L"Failed", MB_OK);
        goto done;
    }

    // Get the client area for size calculation.
    RECT rcClient;
    GetClientRect(hWnd, &rcClient);

    // This is the best stretch mode.
    SetStretchBltMode(hdcWindow, HALFTONE);

    // The source DC is the entire screen, and the destination DC is the current window (HWND).
    if (!StretchBlt(hdcWindow,
        0, 0,
        rcClient.right, rcClient.bottom,
        hdcScreen,
        0, 0,
        GetSystemMetrics(SM_CXSCREEN),
        GetSystemMetrics(SM_CYSCREEN),
        SRCCOPY))
    {
        MessageBox(hWnd, L"StretchBlt has failed", L"Failed", MB_OK);
        goto done;
    }

    // Create a compatible bitmap from the Window DC.
    hbmScreen = CreateCompatibleBitmap(hdcWindow, rcClient.right - rcClient.left, rcClient.bottom -
rcClient.top);

    if (!hbmScreen)
    {
        MessageBox(hWnd, L"CreateCompatibleBitmap Failed", L"Failed", MB_OK);
        goto done;
    }

    // Select the compatible bitmap into the compatible memory DC.
    SelectObject(hdcMemDC, hbmScreen);

    // Bit block transfer into our compatible memory DC.
    if (!BitBlt(hdcMemDC,
        0, 0,
        rcClient.right - rcClient.left, rcClient.bottom - rcClient.top,
        hdcWindow,
        0, 0,
        SRCCOPY))
    {
        MessageBox(hWnd, L"BitBlt has failed", L"Failed", MB_OK);
        goto done;
    }

    // Get the BITMAP from the HBITMAP.
    GetObject(hbmScreen, sizeof(BITMAP), &bmpScreen);

    BITMAPFILEHEADER   bmfHeader;
    BITMAPINFOHEADER   bi;

    bi.biSize = sizeof(BITMAPINFOHEADER);
    bi.biWidth = bmpScreen.bmWidth;
    bi.biHeight = bmpScreen.bmHeight;
    bi.biPlanes = 1;
    bi.biBitCount = 32;
    bi.biCompression = BI_RGB;
    bi.biSizeImage = 0;
    bi.biXPelsPerMeter = 0;
    bi.biYPelsPerMeter = 0;
    bi.biClrUsed = 0;
    bi.biClrImportant = 0;

    dwBmpSize = ((bmpScreen.bmWidth * bi.biBitCount + 31) / 32) * 4 * bmpScreen.bmHeight;
    // Starting with 32-bit Windows, GlobalAlloc and LocalAlloc are implemented as wrapper functions that
    // call HeapAlloc using a handle to the process's default heap. Therefore, GlobalAlloc and LocalAlloc
    // have greater overhead than HeapAlloc.
    hDIB = GlobalAlloc(GHND, dwBmpSize);
    lpbitmap = (char*)GlobalLock(hDIB);

    // Gets the "bits" from the bitmap, and copies them into a buffer
    // that's pointed to by lpbitmap.
    GetDIBits(hdcWindow, hbmScreen, 0,
        (UINT)bmpScreen.bmHeight,
        lpbitmap,
        (BITMAPINFO*)&bi, DIB_RGB_COLORS);

    // A file is created, this is where we will save the screen capture.
    hFile = CreateFile(L"captureqwsx.bmp",
        GENERIC_WRITE,
        0,
        NULL,
        CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL, NULL);

    // Add the size of the headers to the size of the bitmap to get the total file size.
    dwSizeofDIB = dwBmpSize + sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER);

    // Offset to where the actual bitmap bits start.
    bmfHeader.bfOffBits = (DWORD)sizeof(BITMAPFILEHEADER) + (DWORD)sizeof(BITMAPINFOHEADER);

    // Size of the file.
    bmfHeader.bfSize = dwSizeofDIB;

    // bfType must always be BM for Bitmaps.
    bmfHeader.bfType = 0x4D42; // BM.

    WriteFile(hFile, (LPSTR)&bmfHeader, sizeof(BITMAPFILEHEADER), &dwBytesWritten, NULL);
    WriteFile(hFile, (LPSTR)&bi, sizeof(BITMAPINFOHEADER), &dwBytesWritten, NULL);
    WriteFile(hFile, (LPSTR)lpbitmap, dwBmpSize, &dwBytesWritten, NULL);

    // Unlock and Free the DIB from the heap.
    GlobalUnlock(hDIB);
    GlobalFree(hDIB);

    // Close the handle for the file that was created.
    CloseHandle(hFile);

    // Clean up.
done:
    DeleteObject(hbmScreen);
    DeleteObject(hdcMemDC);
    ReleaseDC(NULL, hdcScreen);
    ReleaseDC(hWnd, hdcWindow);

    return 0;
}

//
//  FUNCTION: WndProc(HWND, UINT, WPARAM, LPARAM)
//
//  PURPOSE: Processes messages for the main window.
//
//  WM_COMMAND  - process the application menu
//  WM_PAINT    - Paint the main window
//  WM_DESTROY  - post a quit message and return
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
    case WM_COMMAND:
```

```
    {
        int wmId = LOWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
        case IDM_ABOUT:
            DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
            break;
        case IDM_EXIT:
            DestroyWindow(hWnd);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
        }
    }
    break;
    case WM_PAINT:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint(hWnd, &ps);
        CaptureAnImage(hWnd);
        EndPaint(hWnd, &ps);
    }
    break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}

// Message handler for about box.
INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    UNREFERENCED_PARAMETER(lParam);
    switch (message)
    {
    case WM_INITDIALOG:
        return (INT_PTR)TRUE;

    case WM_COMMAND:
        if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
        {
            EndDialog(hDlg, LOWORD(wParam));
            return (INT_PTR)TRUE;
        }
        break;
    }
    return (INT_PTR)FALSE;
}
```

Scaling an Image

4/20/2022 • 12 minutes to read • Edit Online

Some applications scale images; that is, they display zoomed or reduced views of an image. For example, a drawing application may provide a zoom feature that enables the user to view and edit a drawing on a pixel-by-pixel basis.

Applications scale images by calling the StretchBlt function. Like the BitBlt function, **StretchBlt** copies bitmap data from a bitmap in a source device context (DC) into a bitmap in a target DC. However, unlike the **BitBlt** function, **StretchBlt** scales the image based on the specified dimensions of the source and target rectangles. If the source rectangle is larger than the target rectangle, the resultant image will appear to have shrunk; if the source rectangle is smaller than the target rectangle, the resultant image will appear to have expanded.

If the target rectangle is smaller than the source rectangle, StretchBlt removes color data from the image according to a specified stretch mode as shown in the following table.

STRETCH MODE	METHOD
BLACKONWHITE	Performs a logical AND operation on the color data for the eliminated pixels and the color data for the remaining pixels.
WHITEONBLACK	Performs a logical OR operation on the color data for the eliminated pixels and the color data for the remaining pixels.
COLORONCOLOR	Eliminates the color data of the deleted pixels completely.
HALFTONE	Approximates the original (source) color data in the destination.

You set the stretch mode by calling the SetStretchBltMode function.

The following example code is taken from an application that demonstrates all four of the stretch modes available with the StretchBlt function.

```
#include "stdafx.h"
#include "GDIBitmapScaling.h"
#include <commctrl.h>
#include <CommDlg.h>




#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE hInst;                                // current instance
TCHAR szTitle[MAX_LOADSTRING];                    // The title bar text
TCHAR szWindowClass[MAX_LOADSTRING];            // the main window class name

// Forward declarations of functions included in this code module:
ATOM                MyRegisterClass(HINSTANCE hInstance);
BOOL                InitInstance(HINSTANCE, int);
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
```

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
{
    UNREFERENCED_PARAMETER(hPrevInstance);
    UNREFERENCED_PARAMETER(lpCmdLine);

     // TODO: Place code here.
    MSG msg;
    HACCEL hAccelTable;

    // Initialize global strings
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadString(hInstance, IDC_GDIBITMAPSCALING, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance);

    // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }

    hAccelTable = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDC_GDIBITMAPSCALING));

    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }

    return (int) msg.wParam;
}



//
//  FUNCTION: MyRegisterClass()
//
//  PURPOSE: Registers the window class.
//
//  COMMENTS:
//
//    This function and its usage are only necessary if you want this code
//    to be compatible with Win32 systems prior to the 'RegisterClassEx'
//    function that was added to Windows 95. It is important to call this function
//    so that the application will get 'well formed' small icons associated
//    with it.
//
ATOM MyRegisterClass(HINSTANCE hInstance)
{
    WNDCLASSEX wcex;

    wcex.cbSize = sizeof(WNDCLASSEX);

    wcex.style            = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc    = WndProc;
    wcex.cbClsExtra        = 0;
    wcex.cbWndExtra        = 0;
    wcex.hInstance        = hInstance;
    wcex.hIcon            = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_GDIBITMAPSCALING));
    wcex.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground    = (HBRUSH)(COLOR_WINDOW+1);
    wcex.lpszMenuName    = MAKEINTRESOURCE(IDC_GDIBITMAPSCALING);
    wcex.lpszClassName    = szWindowClass;
    wcex.hIconSm        = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_SMALL));

    return RegisterClassEx(&wcex);
}

//
//   FUNCTION: InitInstance(HINSTANCE, int)
//
//   PURPOSE: Saves instance handle and creates main window
//
//   COMMENTS:
//
//        In this function, we save the instance handle in a global variable and
//        create and display the main program window.
//

#define NEW_DIB_FORMAT(lpbih) (lpbih->biSize != sizeof(BITMAPCOREHEADER))
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;

   hInst = hInstance; // Store instance handle in our global variable

   hWnd = CreateWindow(szWindowClass, szTitle, WS_SYSMENU,
      CW_USEDEFAULT, 0, 1024, 768, NULL, NULL, hInstance, NULL);

   if (!hWnd)
   {
      return FALSE;
   }

   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   return TRUE;
}

static   HCURSOR hcurSave;

WORD DIBNumColors (LPVOID lpv)
{
    INT                 bits;
    LPBITMAPINFOHEADER  lpbih = (LPBITMAPINFOHEADER)lpv;
    LPBITMAPCOREHEADER  lpbch = (LPBITMAPCOREHEADER)lpv;

    /*  With the BITMAPINFO format headers, the size of the palette
     *  is in biClrUsed, whereas in the BITMAPCORE - style headers, it
     *  is dependent on the bits per pixel ( = 2 raised to the power of
     *  bits/pixel).
     */
    if (NEW_DIB_FORMAT(lpbih)) {
      if (lpbih->biClrUsed != 0)
        return (WORD)lpbih->biClrUsed;
      bits = lpbih->biBitCount;
    }
    else
      bits = lpbch->bcBitCount;

    if (bits > 8)
      return 0; /* Since biClrUsed is 0, we dont have a an optimal palette */
    else
      return (1 << bits);
}
/* Macro to determine to round off the given value to the closest byte */
#define WIDTHBYTES(i)   ((((i)+31) >> 5) << 2)
/******************************************************************************
 *                                                                            *
 *  FUNCTION   : DIBInfo(HANDLE hbi, LPBITMAPINFOHEADER lpbih)                *
 *  FUNCTION    : DIBInfo(HANDLE hbi, LPBITMAPINFOHEADER lpbih)
 *                                                                            *
 *  PURPOSE    : Retrieves the DIB info associated with a CF_DIB              *
 *               format memory block.                                         *
 *                                                                            *
 *  RETURNS    : TRUE  - if successful.                                       *
 *               FALSE - otherwise                                            *
 *                                                                            *
 ******************************************************************************/
BOOL DIBInfo (HANDLE hbi, LPBITMAPINFOHEADER lpbih)
{
    if (hbi){
      *lpbih = *(LPBITMAPINFOHEADER)hbi;

      /* fill in the default fields */
      if (NEW_DIB_FORMAT(lpbih)) {
        if (lpbih->biSizeImage == 0L)
          lpbih->biSizeImage = WIDTHBYTES(lpbih->biWidth*lpbih->biBitCount) * lpbih->biHeight;

        if (lpbih->biClrUsed == 0L)
          lpbih->biClrUsed = DIBNumColors (lpbih);
      }

      return TRUE;
    }
    return FALSE;
}
/* flags for mmioSeek() */
#ifndef SEEK_SET
#define SEEK_SET        0               /* seek to an absolute position */
#define SEEK_CUR        1               /* seek relative to current position */
#define SEEK_END        2               /* seek relative to end of file */
#endif  /* ifndef SEEK_SET */
VOID ReadPackedFileHeader(HFILE hFile, LPBITMAPFILEHEADER lpbmfhdr, LPDWORD lpdwOffset)
{
    *lpdwOffset = _llseek(hFile, 0L, (UINT) SEEK_CUR);
    _hread(hFile, (LPSTR) &lpbmfhdr->bfType, sizeof(WORD)); /* read in bfType*/
    _hread(hFile, (LPSTR) &lpbmfhdr->bfSize, sizeof(DWORD) * 3); /* read in last 3 dwords*/
}




/* macro to determine if resource is a DIB */
#define ISDIB(bft) ((bft) == BFT_BITMAP)

/* Header signatutes for various resources */
#define BFT_ICON   0x4349   /* 'IC' */
#define BFT_BITMAP 0x4d42   /* 'BM' */
#define BFT_CURSOR 0x5450   /* 'PT' */
HANDLE ReadDIBBitmapInfo (INT hFile)
{
    DWORD              dwOffset;
    HANDLE             hbi = NULL;
    INT                size;
    INT                i;
    WORD               nNumColors;
    LPRGBQUAD          lprgbq;
    BITMAPINFOHEADER   bih;
    BITMAPCOREHEADER   bch;
    LPBITMAPINFOHEADER lpbih;
    BITMAPFILEHEADER   bf;
    DWORD              dwDWMasks= 0;
    DWORD              dwWidth = 0;
    DWORD              dwHeight = 0;
    WORD               wPlanes, wBitCount;

    if (hFile == HFILE_ERROR)
        return NULL;
    /* Read the bitmap file header */
    ReadPackedFileHeader(hFile, &bf, &dwOffset);

    /* Do we have a RC HEADER? */
    if (!ISDIB (bf.bfType)) {
      bf.bfOffBits = 0L;
        _llseek(hFile, dwOffset, (UINT)SEEK_SET); /* seek back to beginning of file */
    }

    if (sizeof(bih) != _hread(hFile, (LPSTR)&bih, (UINT)sizeof(bih)))
      return FALSE;

    nNumColors = DIBNumColors (&bih);

    /* Check the nature (BITMAPINFO or BITMAPCORE) of the info. block
     * and extract the field information accordingly. If a BITMAPCOREHEADER,
     * transfer it's field information to a BITMAPINFOHEADER-style block
     */
    switch (size = (INT)bih.biSize){
        case sizeof (BITMAPINFOHEADER):
            break;

        case sizeof (BITMAPCOREHEADER):

            bch = *(LPBITMAPCOREHEADER)&bih;

            dwWidth   = (DWORD)bch.bcWidth;
            dwHeight  = (DWORD)bch.bcHeight;
            wPlanes   = bch.bcPlanes;
            wBitCount = bch.bcBitCount;

            bih.biSize           = sizeof(BITMAPINFOHEADER);
            bih.biWidth          = dwWidth;
            bih.biHeight         = dwHeight;
            bih.biPlanes         = wPlanes;
            bih.biBitCount       = wBitCount;
            bih.biCompression    = BI_RGB;
            bih.biSizeImage      = 0;
            bih.biXPelsPerMeter  = 0;
            bih.biYPelsPerMeter  = 0;
            bih.biClrUsed        = nNumColors;
            bih.biClrImportant   = nNumColors;

            _llseek(hFile, (LONG)sizeof (BITMAPCOREHEADER) - sizeof (BITMAPINFOHEADER), (UINT)SEEK_CUR);
            break;

        default:
            /* Not a DIB! */
            return NULL;
    }

    /*  Fill in some default values if they are zero */
    if (bih.biSizeImage == 0){
        bih.biSizeImage = WIDTHBYTES((DWORD)bih.biWidth * bih.biBitCount) * bih.biHeight;
    }
    if (bih.biClrUsed == 0)
        bih.biClrUsed = DIBNumColors(&bih);

    /* Allocate for the BITMAPINFO structure and the color table. */
    if ((bih.biBitCount == 16) || (bih.biBitCount == 32))
      dwDWMasks = sizeof(DWORD) * 3;
    hbi = GlobalAlloc (GPTR, (LONG)bih.biSize + nNumColors * sizeof(RGBQUAD) + dwDWMasks);
    if (!hbi)
        return NULL;
    lpbih = (LPBITMAPINFOHEADER)hbi;
    *lpbih = bih;

    /* Get a pointer to the color table */
    lprgbq = (LPRGBQUAD)((LPSTR)lpbih + bih.biSize);
    if (nNumColors){
        if (size == sizeof(BITMAPCOREHEADER)){
            /* Convert a old color table (3 byte RGBTRIPLEs) to a new
             * color table (4 byte RGBQUADs)
             */
            _hread(hFile, (LPSTR)lprgbq, (UINT)nNumColors * sizeof(RGBTRIPLE));

            for (i = nNumColors - 1; i >= 0; i--){
                RGBQUAD rgbq;

                rgbq.rgbRed      = ((RGBTRIPLE*)lprgbq)[i].rgbtRed;
                rgbq.rgbBlue     = ((RGBTRIPLE*)lprgbq)[i].rgbtBlue;
                rgbq.rgbGreen    = ((RGBTRIPLE*)lprgbq)[i].rgbtGreen;
                rgbq.rgbReserved = (BYTE)0;

                lprgbq[i] = rgbq;
            }
        }
        else
            _hread(hFile, (LPSTR)lprgbq, (UINT)nNumColors * sizeof(RGBQUAD));
    } else
        if (dwDWMasks)
           _hread(hFile, (LPSTR)lprgbq, dwDWMasks);

    if (bf.bfOffBits != 0L){
        _llseek(hFile, dwOffset + bf.bfOffBits, (UINT)SEEK_SET);
        }

    return hbi;
}
HGLOBAL GlobalFreeDIB(HGLOBAL hDIB)
{
   LPBITMAPINFOHEADER lpbi = (LPBITMAPINFOHEADER)hDIB;

   if (!lpbi->biClrImportant)
     return GlobalFree(hDIB);

   if (GlobalFlags((HGLOBAL)lpbi->biClrImportant) == GMEM_INVALID_HANDLE) {
    SetLastError(0);
    return GlobalFree(hDIB);
  } else
    return GlobalFree((HANDLE)lpbi->biClrImportant);
}
/******************************************************************************
 *                                                                            *
 *  FUNCTION   :  ColorTableSize(LPVOID lpv)                                  *
 *                                                                            *
 *  PURPOSE    :  Calculates the palette size in bytes. If the info. block    *
 *                is of the BITMAPCOREHEADER type, the number of colors is    *
 *                multiplied by 3 to give the palette size, otherwise the     *
 *                number of colors is multiplied by 4.                        *
 *                                                                            *
 *  RETURNS    :  Color table size in number of bytes.                        *
 *                                                                            *
 *****************************************************************************/
WORD ColorTableSize (LPVOID lpv)
{
    LPBITMAPINFOHEADER lpbih = (LPBITMAPINFOHEADER)lpv;

    if (NEW_DIB_FORMAT(lpbih))
    {
      if (((LPBITMAPINFOHEADER)(lpbih))->biCompression == BI_BITFIELDS)
         /* Remember that 16/32bpp dibs can still have a color table */
         return (sizeof(DWORD) * 3) + (DIBNumColors (lpbih) * sizeof (RGBQUAD));
      else
         return (DIBNumColors (lpbih) * sizeof (RGBQUAD));
    }
    else
```

```
      return (DIBNumColors (lpbih) * sizeof (RGBTRIPLE));
}

/****************************************************************************
 *                                                                          *
 *  FUNCTION   :OpenDIB(LPSTR szFilename)                                   *
 *                                                                          *
 *  PURPOSE    :Open a DIB file and create a memory DIB -- a memory handle  *
 *              containing BITMAPINFO, palette data and the bits.           *
 *                                                                          *
 *  RETURNS    :A handle to the DIB.                                        *
 *                                                                          *
 ****************************************************************************/
HANDLE OpenDIB (LPSTR szFilename)
{
    HFILE               hFile;
    BITMAPINFOHEADER    bih;
    LPBITMAPINFOHEADER  lpbih;
    DWORD               dwLen = 0;
    DWORD               dwBits;
    HANDLE              hDIB;
    HANDLE              hMem;
    OFSTRUCT            of;

    /* Open the file and read the DIB information */
    hFile = OpenFile(szFilename, &of, (UINT)OF_READ);
    if (hFile == HFILE_ERROR)
        return NULL;

    hDIB = ReadDIBBitmapInfo(hFile);
    if (!hDIB)
        return NULL;
    DIBInfo(hDIB, &bih);

    /* Calculate the memory needed to hold the DIB */
    dwBits = bih.biSizeImage;
    dwLen  = bih.biSize + (DWORD)ColorTableSize (&bih) + dwBits;

    /* Try to increase the size of the bitmap info. buffer to hold the DIB */
    hMem = GlobalReAlloc(hDIB, dwLen, GMEM_MOVEABLE);

    if (!hMem){
        GlobalFreeDIB(hDIB);
        hDIB = NULL;
    }
    else
        hDIB = hMem;

    /* Read in the bits */
    if (hDIB){
        lpbih = (LPBITMAPINFOHEADER)hDIB;
        _hread(hFile, (LPSTR)lpbih + (WORD)lpbih->biSize + ColorTableSize(lpbih), dwBits);
    }
    _lclose(hFile);

    return hDIB;
}/* Macros to display/remove hourglass cursor for lengthy operations */
#define StartWait() hcurSave = SetCursor(LoadCursor(NULL, IDC_WAIT))
#define EndWait()   SetCursor(hcurSave)
/****************************************************************************
 *                                                                          *
 *  FUNCTION   : BitmapFromDIB(HANDLE hDIB, HPALETTE hPal)                  *
 *                                                                          *
 *  PURPOSE    : Will create a DDB (Device Dependent Bitmap) given a global *
 *               handle to a memory block in CF_DIB format                  *
 *                                                                          *
 *  RETURNS    : A handle to the DDB.                                       *
 *                                                                          *
 ****************************************************************************/
```

```
HBITMAP BitmapFromDIB (HANDLE hDIB, HPALETTE  hPal)
{
    LPBITMAPINFOHEADER  lpbih;
    HPALETTE            hPalOld;
    HDC                 hDC;
    HBITMAP             hBitmap;

    StartWait();

    if (!hDIB)
        return NULL;

    lpbih = (LPBITMAPINFOHEADER)hDIB;

    if (!lpbih)
        return NULL;

    hDC = GetDC(NULL);

    if (hPal){
        hPalOld = SelectPalette(hDC, hPal, FALSE);
        RealizePalette(hDC);
    }

    hBitmap = CreateDIBitmap(hDC,
                lpbih,
                CBM_INIT,
                (LPSTR)lpbih + lpbih->biSize + ColorTableSize(lpbih),
                (LPBITMAPINFO)lpbih,
                DIB_RGB_COLORS );

    if (hPal)
        SelectPalette(hDC, hPalOld, FALSE);

    ReleaseDC(NULL, hDC);

    EndWait();

    return hBitmap;
}
/******************************************************************************
 *                                                                            *
 *  FUNCTION   : DrawBitmap(HDC hDC, int x, int y,                            *
 *                          HBITMAP hBitmap, DWORD dwROP)                     *
 *                                                                            *
 *  PURPOSE    : Draws bitmap <hBitmap> at the specified position in DC <hDC> *
 *                                                                            *
 *  RETURNS    : Return value of BitBlt()                                     *
 *                                                                            *
 *****************************************************************************/
BOOL DrawBitmap (HDC hDC, INT x, INT y, HBITMAP hBitmap, DWORD dwROP)
{
    HDC       hDCBits;
    BITMAP    Bitmap;
    BOOL      bResult;

    if (!hDC || !hBitmap)
        return FALSE;

    hDCBits = CreateCompatibleDC(hDC);
    GetObject(hBitmap, sizeof(BITMAP), (LPSTR)&Bitmap);
    SelectObject(hDCBits, hBitmap);
    bResult = BitBlt(hDC, x, y, Bitmap.bmWidth, Bitmap.bmHeight, hDCBits, 0, 0, dwROP);
    DeleteDC(hDCBits);

    return bResult;
}

  /*FUNCTION: WndProc(HWND, UINT, WPARAM, LPARAM)
  PURPOSE:  Processes messages for the main window.

  WM_COMMAND    - process the application menu
  WM_PAINT    - Paint the main window
  WM_DESTROY    - post a quit message and return*/



#define ID_LOADBITMAP 1

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    //Handle to a GDI device context
    HDC hDC;
    //Handle to a DDB(device-dependent bitmap)
    HBITMAP hBitmap;


    HFONT hFont;
    NONCLIENTMETRICS ncm={0};
    ncm.cbSize= sizeof(NONCLIENTMETRICS);


    static HWND hwndButton;
    static HWND hwndButtonExit;
    static HANDLE hDIB = NULL;

    char szDirName[MAX_PATH];
    char szFilename[MAX_PATH]="\0";
    char szBitmapName[MAX_PATH]="\\Waterfall.bmp";
    //char szBitmapName[MAX_PATH]="\\tulips256.bmp";
    OPENFILENAMEA ofn;


    switch (message)
    {

    case WM_CREATE:

        //Creates a font from the current theme's caption font
        SystemParametersInfo(SPI_GETNONCLIENTMETRICS, NULL, &ncm, NULL);
        hFont = CreateFontIndirect(&ncm.lfCaptionFont);

        //Gets the device context for the current window
        hDC = GetDC(hWnd);

        //Gets the directory of the current project and loads Waterfall.bmp
        GetCurrentDirectoryA(MAX_PATH, szDirName);
        strcat_s(szDirName, szBitmapName);
        strcat_s(szFilename,szDirName);
        hDIB = OpenDIB(szFilename);
        hBitmap = BitmapFromDIB(hDIB, NULL);

        //Draws Waterfall.bmp as a device dependent bitmap
        DrawBitmap(hDC,0,0,hBitmap,SRCCOPY);
        InvalidateRect(hWnd, NULL, FALSE);
        ReleaseDC(hWnd,hDC);

        //Draws the "Load Bitmap" button
        hwndButton = CreateWindowW(TEXT("button"),TEXT("Load Bitmap"),
            WS_CHILD | WS_VISIBLE | WS_BORDER, 600, 200, 150,50,
            hWnd,
            (HMENU)ID_LOADBITMAP,
            ((LPCREATESTRUCT) lParam)-> hInstance, NULL);

        //Set the font of the button to the theme's caption font
        //Set the font of the button to the theme's caption font
        SendMessage(hwndButton, WM_SETFONT, (WPARAM)hFont, TRUE );


        return 0;


    case WM_COMMAND:
        wmId    = LOWORD(wParam);
        wmEvent = HIWORD(wParam);


        //if Load Bitmap button is pressed
        if (wmId == ID_LOADBITMAP && wmEvent == BN_CLICKED)
        {

            //Get the current directory name, and store in szDirName
            GetCurrentDirectoryA(MAX_PATH, szDirName);

            //Set all structure members to zero.
            ZeroMemory(&ofn, sizeof(OPENFILENAMEA));

            //Initializing the OPENFILENAMEA structure
            ofn.lStructSize = sizeof(OPENFILENAMEA);
            ofn.hwndOwner = hWnd;
            ofn.lpstrFilter ="BMP Files (*.BMP)\0 *.BMP\0\0";
            ofn.lpstrFile = szFilename;
            ofn.nMaxFile = sizeof(szFilename);
            ofn.lpstrDefExt = "BMP";
            ofn.lpstrInitialDir = szDirName;
            ofn.Flags = OFN_EXPLORER | OFN_SHOWHELP | OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;



            if (GetOpenFileNameA(&ofn)) {
               hDIB = OpenDIB((LPSTR)szFilename);
               if (!hDIB)
                  MessageBox(hWnd, TEXT("Unable to load file!"), TEXT("Oops"), MB_ICONSTOP);
            } else {
                if (strlen((const char *)szFilename) != 0)
                  MessageBox(hWnd, TEXT("Unable to load file!"), TEXT("Oops"), MB_ICONSTOP);

                return 0;
            }





            InvalidateRect(hWnd, NULL, FALSE);


        }


        break;
    case WM_PAINT:
        {
            //Initializing arrays for boxes
            POINT pRect1[5] = {{0,0},{400,0},{400,400},{0,400},{0,0}};
            POINT pRect2[5] = {{0,500}, {200, 500}, {200, 700}, {0,700},{0,500}};
            POINT pRect3[5] = {{210,500}, {410, 500}, {410, 700}, {210,700},{210,500}};
            POINT pRect4[5] = {{420,500}, {620, 500}, {620, 700}, {420,700},{420,500}};
            POINT pRect5[5] = {{630,500}, {830, 500}, {830, 700}, {630,700},{630,500}};

            //For the white background
            RECT clientRect;
            HRGN hRegion1;
            HRGN hRegion2;
            HRGN hRegion3;
            HBRUSH hBGBrush;
```

```
HBRUSH hBGBrush;

//Handle to a logical font
HFONT hFont;

//Get the caption font that is currently in use
SystemParametersInfo(SPI_GETNONCLIENTMETRICS, NULL, &ncm, NULL);
hFont = CreateFontIndirect(&ncm.lfCaptionFont);

//Begin drawing
hDC = BeginPaint(hWnd, &ps);

//Draw and fill rectangles for the background
GetClientRect(hWnd, &clientRect);
hRegion1 = CreateRectRgn(clientRect.left,clientRect.top,clientRect.right,clientRect.bottom);
hBGBrush = CreateSolidBrush(RGB(255,255,255));
FillRgn(hDC, hRegion1, hBGBrush);

//Create an HBITMAP(device dependent bitmap) to be drawn
hBitmap = BitmapFromDIB(hDIB,NULL);
//Draw the DDB
DrawBitmap(hDC,0,0,hBitmap,SRCCOPY);


if(hDIB)
{
    hRegion2 = CreateRectRgn(401,0,clientRect.right,401);
    hRegion3 = CreateRectRgn(0,401,clientRect.right,clientRect.bottom);
    FillRgn(hDC,hRegion2,hBGBrush);
    FillRgn(hDC,hRegion3,hBGBrush);
    //Set stretch mode as BLACKONWHITE and then copy from the
    //source rectangle into the smaller rectangle
    SetStretchBltMode(hDC,BLACKONWHITE);
    StretchBlt(hDC,0,500,200,200,hDC, 0,0,400,400,SRCCOPY);

    //Set stretch mode as WHITEONBLACK and then copy from the
    //source rectangle into the smaller rectangle
    SetStretchBltMode(hDC,WHITEONBLACK);
    StretchBlt(hDC,210,500,200,200, hDC, 0,0,400,400, SRCCOPY);

    //Set stretch mode as COLORONCOLOR and then copy from the
    //source rectangle into the smaller rectangle
    SetStretchBltMode(hDC,COLORONCOLOR);
    StretchBlt(hDC,420,500,200,200, hDC, 0,0,400,400, SRCCOPY);

    //Set stretch mode as HALFTONE and then copy from the
    //source rectangle into the smaller rectangle
    SetStretchBltMode(hDC,HALFTONE);
    StretchBlt(hDC,630,500,200,200, hDC, 0,0,400,400, SRCCOPY);

}
//Select the caption font created earlier
SelectObject(hDC,hFont);

//Create captions for each demonstration of color loss modes
TextOut(hDC,50,480,TEXT("BLACKONWHITE"),12);
TextOut(hDC,250,480,TEXT("WHITEONBLACK"),12);
TextOut(hDC,460,480,TEXT("COLORONCOLOR"),12);
TextOut(hDC,680,480,TEXT("HALFTONE"),8);
DeleteObject(hFont);

//Selecting the stock object pen to draw with
SelectObject(hDC, GetStockObject(DC_PEN));
//The pen is gray
SetDCPenColor(hDC, RGB(80,80,80));

//Polylines are drawn from arrays of POINTs
Polyline(hDC, pRect1, 5);
Polyline(hDC, pRect2, 5);
Polyline(hDC, pRect3, 5);
```

```
            Polyline(hDC, pRect3, 5);
            Polyline(hDC, pRect4, 5);
            Polyline(hDC, pRect5, 5);

            FillRgn(hDC,hRegion2,hBGBrush);
            EndPaint(hWnd, &ps);

            break;
        }
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Storing an Image

4/20/2022 • 3 minutes to read • Edit Online

Many applications store images permanently as files. For example, drawing applications store pictures, spreadsheet applications store charts, CAD applications store drawings, and so on.

If you are writing an application that stores a bitmap image in a file, you should use the bitmap file format described in Bitmap Storage. To store a bitmap in this format, you must use a **BITMAPINFOHEADER**, a **BITMAPV4HEADER**, or a **BITMAPV5HEADER** structure and an array of **RGBQUAD** structures, as well as an array of palette indexes.

The following example code defines a function that uses a **BITMAPINFO** structure and allocates memory for and initializes members within a **BITMAPINFOHEADER** structure. Note that the **BITMAPINFO** structure cannot be used with either a **BITMAPV4HEADER** or a **BITMAPV5HEADER** structure.

```
PBITMAPINFO CreateBitmapInfoStruct(HWND hwnd, HBITMAP hBmp)
{
    BITMAP bmp;
    PBITMAPINFO pbmi;
    WORD    cClrBits;

    // Retrieve the bitmap color format, width, and height.
    if (!GetObject(hBmp, sizeof(BITMAP), (LPSTR)&bmp))
        errhandler("GetObject", hwnd);

    // Convert the color format to a count of bits.
    cClrBits = (WORD)(bmp.bmPlanes * bmp.bmBitsPixel);
    if (cClrBits == 1)
        cClrBits = 1;
    else if (cClrBits <= 4)
        cClrBits = 4;
    else if (cClrBits <= 8)
        cClrBits = 8;
    else if (cClrBits <= 16)
        cClrBits = 16;
    else if (cClrBits <= 24)
        cClrBits = 24;
    else cClrBits = 32;

    // Allocate memory for the BITMAPINFO structure. (This structure
    // contains a BITMAPINFOHEADER structure and an array of RGBQUAD
    // data structures.)

     if (cClrBits < 24)
         pbmi = (PBITMAPINFO) LocalAlloc(LPTR,
                    sizeof(BITMAPINFOHEADER) +
                    sizeof(RGBQUAD) * (1<< cClrBits));

     // There is no RGBQUAD array for these formats: 24-bit-per-pixel or 32-bit-per-pixel

     else
         pbmi = (PBITMAPINFO) LocalAlloc(LPTR,
                    sizeof(BITMAPINFOHEADER));

    // Initialize the fields in the BITMAPINFO structure.

    pbmi->bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
    pbmi->bmiHeader.biWidth = bmp.bmWidth;
    pbmi->bmiHeader.biHeight = bmp.bmHeight;
    pbmi->bmiHeader.biPlanes = bmp.bmPlanes;
    pbmi->bmiHeader.biBitCount = bmp.bmBitsPixel;
    if (cClrBits < 24)
        pbmi->bmiHeader.biClrUsed = (1<<cClrBits);

    // If the bitmap is not compressed, set the BI_RGB flag.
    pbmi->bmiHeader.biCompression = BI_RGB;

    // Compute the number of bytes in the array of color
    // indices and store the result in biSizeImage.
    // The width must be DWORD aligned unless the bitmap is RLE
    // compressed.
    pbmi->bmiHeader.biSizeImage = ((pbmi->bmiHeader.biWidth * cClrBits +31) & ~31) /8
                                  * pbmi->bmiHeader.biHeight;
    // Set biClrImportant to 0, indicating that all of the
    // device colors are important.
     pbmi->bmiHeader.biClrImportant = 0;
     return pbmi;
 }
```

The following example code defines a function that initializes the remaining structures, retrieves the array of palette indices, opens the file, copies the data, and closes the file.

```
void CreateBMPFile(HWND hwnd, LPTSTR pszFile, PBITMAPINFO pbi,
                  HBITMAP hBMP, HDC hDC)
 {
     HANDLE hf;                 // file handle
    BITMAPFILEHEADER hdr;       // bitmap file-header
    PBITMAPINFOHEADER pbih;     // bitmap info-header
    LPBYTE lpBits;              // memory pointer
    DWORD dwTotal;              // total count of bytes
    DWORD cb;                   // incremental count of bytes
    BYTE *hp;                   // byte pointer
    DWORD dwTmp;

    pbih = (PBITMAPINFOHEADER) pbi;
    lpBits = (LPBYTE) GlobalAlloc(GMEM_FIXED, pbih->biSizeImage);

    if (!lpBits)
         errhandler("GlobalAlloc", hwnd);

    // Retrieve the color table (RGBQUAD array) and the bits
    // (array of palette indices) from the DIB.
    if (!GetDIBits(hDC, hBMP, 0, (WORD) pbih->biHeight, lpBits, pbi,
        DIB_RGB_COLORS))
    {
        errhandler("GetDIBits", hwnd);
    }

    // Create the .BMP file.
    hf = CreateFile(pszFile,
                   GENERIC_READ | GENERIC_WRITE,
                   (DWORD) 0,
                    NULL,
                   CREATE_ALWAYS,
                   FILE_ATTRIBUTE_NORMAL,
                   (HANDLE) NULL);
    if (hf == INVALID_HANDLE_VALUE)
        errhandler("CreateFile", hwnd);
    hdr.bfType = 0x4d42;        // 0x42 = "B" 0x4d = "M"
    // Compute the size of the entire file.
    hdr.bfSize = (DWORD) (sizeof(BITMAPFILEHEADER) +
                 pbih->biSize + pbih->biClrUsed
                 * sizeof(RGBQUAD) + pbih->biSizeImage);
    hdr.bfReserved1 = 0;
    hdr.bfReserved2 = 0;

    // Compute the offset to the array of color indices.
    hdr.bfOffBits = (DWORD) sizeof(BITMAPFILEHEADER) +
                    pbih->biSize + pbih->biClrUsed
                    * sizeof (RGBQUAD);

    // Copy the BITMAPFILEHEADER into the .BMP file.
    if (!WriteFile(hf, (LPVOID) &hdr, sizeof(BITMAPFILEHEADER),
        (LPDWORD) &dwTmp,  NULL))
    {
       errhandler("WriteFile", hwnd);
    }

    // Copy the BITMAPINFOHEADER and RGBQUAD array into the file.
    if (!WriteFile(hf, (LPVOID) pbih, sizeof(BITMAPINFOHEADER)
                  + pbih->biClrUsed * sizeof (RGBQUAD),
                  (LPDWORD) &dwTmp, ( NULL)))
        errhandler("WriteFile", hwnd);

    // Copy the array of color indices into the .BMP file.
    dwTotal = cb = pbih->biSizeImage;
    hp = lpBits;
    if (!WriteFile(hf, (LPSTR) hp, (int) cb, (LPDWORD) &dwTmp,NULL))
           errhandler("WriteFile", hwnd);
    // Close the .BMP file.
     if (!CloseHandle(hf))
           errhandler("CloseHandle", hwnd);

    // Free memory.
    GlobalFree((HGLOBAL)lpBits);
}
```

Alpha Blending a Bitmap

4/20/2022 • 3 minutes to read • Edit Online

The following code sample divides a window into three horizontal areas. Then it draws an alpha-blended bitmap in each of the window areas as follows:

- In the top area, constant alpha = 50% but there is no source alpha.
- In the middle area, constant alpha = 100% (disabled) and source alpha is 0 (transparent) in the middle of the bitmap and 0xff (opaque) elsewhere.
- In the bottom area, constant alpha = 75% and source alpha changes.

```
void DrawAlphaBlend (HWND hWnd, HDC hdcwnd)
{
    HDC hdc;               // handle of the DC we will create
    BLENDFUNCTION bf;      // structure for alpha blending
    HBITMAP hbitmap;       // bitmap handle
    BITMAPINFO bmi;        // bitmap header
    VOID *pvBits;          // pointer to DIB section
    ULONG   ulWindowWidth, ulWindowHeight;      // window width/height
    ULONG   ulBitmapWidth, ulBitmapHeight;      // bitmap width/height
    RECT    rt;            // used for getting window dimensions
    UINT32   x,y;          // stepping variables
    UCHAR ubAlpha;         // used for doing transparent gradient
    UCHAR ubRed;
    UCHAR ubGreen;
    UCHAR ubBlue;
    float fAlphaFactor;    // used to do premultiply

    // get window dimensions
    GetClientRect(hWnd, &rt);

    // calculate window width/height
    ulWindowWidth = rt.right - rt.left;
    ulWindowHeight = rt.bottom - rt.top;

    // make sure we have at least some window size
    if ((!ulWindowWidth) || (!ulWindowHeight))
        return;

    // divide the window into 3 horizontal areas
    ulWindowHeight = ulWindowHeight / 3;

    // create a DC for our bitmap -- the source DC for AlphaBlend
    hdc = CreateCompatibleDC(hdcwnd);

    // zero the memory for the bitmap info
    ZeroMemory(&bmi, sizeof(BITMAPINFO));

    // setup bitmap info
    // set the bitmap width and height to 60% of the width and height of each of the three horizontal areas.
Later on, the blending will occur in the center of each of the three areas.
    bmi.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
    bmi.bmiHeader.biWidth = ulBitmapWidth = ulWindowWidth - (ulWindowWidth/5)*2;
    bmi.bmiHeader.biHeight = ulBitmapHeight = ulWindowHeight - (ulWindowHeight/5)*2;
    bmi.bmiHeader.biPlanes = 1;
    bmi.bmiHeader.biBitCount = 32;         // four 8-bit components
    bmi.bmiHeader.biCompression = BI_RGB;
    bmi.bmiHeader.biSizeImage = ulBitmapWidth * ulBitmapHeight * 4;

    // create our DIB section and select the bitmap into the dc
    hbitmap = CreateDIBSection(hdc, &bmi, DIB_RGB_COLORS, &pvBits, NULL, 0x0);
    hbitmap = CreateDIBSection(hdc, &bmi, DIB_RGB_COLORS, &pvBits, NULL, 0x0);
    SelectObject(hdc, hbitmap);

    // in top window area, constant alpha = 50%, but no source alpha
    // the color format for each pixel is 0xaarrggbb
    // set all pixels to blue and set source alpha to zero
    for (y = 0; y < ulBitmapHeight; y++)
        for (x = 0; x < ulBitmapWidth; x++)
            ((UINT32 *)pvBits)[x + y * ulBitmapWidth] = 0x000000ff;

    bf.BlendOp = AC_SRC_OVER;
    bf.BlendFlags = 0;
    bf.SourceConstantAlpha = 0x7f;  // half of 0xff = 50% transparency
    bf.AlphaFormat = 0;             // ignore source alpha channel

    if (!AlphaBlend(hdcwnd, ulWindowWidth/5, ulWindowHeight/5,
                    ulBitmapWidth, ulBitmapHeight,
                    hdc, 0, 0, ulBitmapWidth, ulBitmapHeight, bf))
        return;                     // alpha blend failed

    // in middle window area, constant alpha = 100% (disabled), source
    // alpha is 0 in middle of bitmap and opaque in rest of bitmap
    for (y = 0; y < ulBitmapHeight; y++)
        for (x = 0; x < ulBitmapWidth; x++)
            if ((x > (int)(ulBitmapWidth/5)) && (x < (ulBitmapWidth-ulBitmapWidth/5)) &&
                (y > (int)(ulBitmapHeight/5)) && (y < (ulBitmapHeight-ulBitmapHeight/5)))
                //in middle of bitmap: source alpha = 0 (transparent).
                // This means multiply each color component by 0x00.
                // Thus, after AlphaBlend, we have a, 0x00 * r,
                // 0x00 * g,and 0x00 * b (which is 0x00000000)
                // for now, set all pixels to red
                ((UINT32 *)pvBits)[x + y * ulBitmapWidth] = 0x00ff0000;
            else
                // in the rest of bitmap, source alpha = 0xff (opaque)
                // and set all pixels to blue
                ((UINT32 *)pvBits)[x + y * ulBitmapWidth] = 0xff0000ff;
            endif;

    bf.BlendOp = AC_SRC_OVER;
    bf.BlendFlags = 0;
    bf.AlphaFormat = AC_SRC_ALPHA;  // use source alpha
    bf.SourceConstantAlpha = 0xff;  // opaque (disable constant alpha)

    if (!AlphaBlend(hdcwnd, ulWindowWidth/5, ulWindowHeight/5+ulWindowHeight, ulBitmapWidth, ulBitmapHeight,
hdc, 0, 0, ulBitmapWidth, ulBitmapHeight, bf))
        return;

    // bottom window area, use constant alpha = 75% and a changing
    // source alpha. Create a gradient effect using source alpha, and
    // then fade it even more with constant alpha
    ubRed = 0x00;
    ubGreen = 0x00;
    ubBlue = 0xff;

    for (y = 0; y < ulBitmapHeight; y++)
        for (x = 0; x < ulBitmapWidth; x++)
        {
            // for a simple gradient, base the alpha value on the x
            // value of the pixel
            ubAlpha = (UCHAR)((float)x / (float)ulBitmapWidth * 255);
            //calculate the factor by which we multiply each component
            fAlphaFactor = (float)ubAlpha / (float)0xff;
            // multiply each pixel by fAlphaFactor, so each component
            // is less than or equal to the alpha value.
            ((UINT32 *)pvBits)[x + y * ulBitmapWidth]
                = (ubAlpha << 24) |                       //0xaa000000
                 ((UCHAR)(ubRed * fAlphaFactor) << 16) |  //0x00rr0000
                 ((UCHAR)(ubGreen * fAlphaFactor) << 8) | //0x0000gg00
                 ((UCHAR)(ubBlue   * fAlphaFactor));      //0x000000bb
        }

    bf.BlendOp = AC_SRC_OVER;
    bf.BlendFlags = 0;
    bf.AlphaFormat = AC_SRC_ALPHA;   // use source alpha
    bf.SourceConstantAlpha = 0xbf;   // use constant alpha, with
                                     // 75% opaqueness

    AlphaBlend(hdcwnd, ulWindowWidth/5,
               ulWindowHeight/5+2*ulWindowHeight, ulBitmapWidth,
               ulBitmapHeight, hdc, 0, 0, ulBitmapWidth,
               ulBitmapHeight, bf);

    // do cleanup
    DeleteObject(hbitmap);
    DeleteDC(hdc);

}
```

Drawing a Shaded Rectangle

4/20/2022 • 2 minutes to read • Edit Online

To draw a shaded rectangle, define a TRIVERTEX array with two elements and a single GRADIENT_RECT structure. The following code example shows how to draw a shaded rectangle using the GradientFill function with the GRADIENT_FILL_RECT mode defined.

```
// Create an array of TRIVERTEX structures that describe
// positional and color values for each vertex. For a rectangle,
// only two vertices need to be defined: upper-left and lower-right.
TRIVERTEX vertex[2] ;
vertex[0].x     = 0;
vertex[0].y     = 0;
vertex[0].Red   = 0x0000;
vertex[0].Green = 0x8000;
vertex[0].Blue  = 0x8000;
vertex[0].Alpha = 0x0000;

vertex[1].x     = 300;
vertex[1].y     = 80;
vertex[1].Red   = 0x0000;
vertex[1].Green = 0xd000;
vertex[1].Blue  = 0xd000;
vertex[1].Alpha = 0x0000;

// Create a GRADIENT_RECT structure that
// references the TRIVERTEX vertices.
GRADIENT_RECT gRect;
gRect.UpperLeft  = 0;
gRect.LowerRight = 1;

// Draw a shaded rectangle.
GradientFill(hdc, vertex, 2, &gRect, 1, GRADIENT_FILL_RECT_H);
```

The following image shows the drawing output of the preceding code example.

Related topics

Bitmaps Overview

Bitmap Functions

Drawing a Shaded Triangle

EMRGRADIENTFILL

GRADIENT_RECT

GradientFill

TRIVERTEX

Drawing a Shaded Triangle

4/20/2022 • 2 minutes to read • Edit Online

To draw a shaded triangle, define a TRIVERTEX structure with three elements and a single GRADIENT_TRIANGLE structure. The following code example shows how to draw a shaded triangle using the GradientFill function with the GRADIENT_FILL_TRIANGLE mode defined.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
    case WM_COMMAND:
        wmId    = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
        case IDM_ABOUT:
            DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
            break;
        case IDM_EXIT:
            DestroyWindow(hWnd);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
        }
        break;
    case WM_PAINT:
        {hdc = BeginPaint(hWnd, &ps);
// Create an array of TRIVERTEX structures that describe
// positional and color values for each vertex.
TRIVERTEX vertex[3];
vertex[0].x     = 150;
vertex[0].y     = 0;
vertex[0].Red   = 0xff00;
vertex[0].Green = 0x8000;
vertex[0].Blue  = 0x0000;
vertex[0].Alpha = 0x0000;

vertex[1].x     = 0;
vertex[1].y     = 150;
vertex[1].Red   = 0x9000;
vertex[1].Green = 0x0000;
vertex[1].Blue  = 0x9000;
vertex[1].Alpha = 0x0000;

vertex[2].x     = 300;
vertex[2].y     = 150;
vertex[2].Red   = 0x9000;
vertex[2].Green = 0x0000;
vertex[2].Blue  = 0x9000;
vertex[2].Alpha = 0x0000;

// Create a GRADIENT_TRIANGLE structure that
// references the TRIVERTEX vertices.
GRADIENT_TRIANGLE gTriangle;
gTriangle.Vertex1 = 0;
gTriangle.Vertex2 = 1;
gTriangle.Vertex2 = 1;
gTriangle.Vertex3 = 2;

// Draw a shaded triangle.
GradientFill(hdc, vertex, 3, &gTriangle, 1, GRADIENT_FILL_TRIANGLE);
        EndPaint(hWnd, &ps);
        break;}
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

The following image shows the drawing output of the preceding code example.

Related topics

Bitmaps Overview

Bitmap Functions

Drawing a Shaded Rectangle

EMRGRADIENTFILL

GRADIENT_TRIANGLE

GradientFill

TRIVERTEX

Testing a Printer for JPEG or PNG Support

4/20/2022 • 2 minutes to read • Edit Online

The **SetDIBitsToDevice** function uses color data from a DIB to set the pixels in the specified rectangle on the device that is associated with the destination device context.

SetDIBitsToDevice is extended to allow a JPEG or PNG image to be passed as the source image.

For example:

```
//
// pvJpgImage points to a buffer containing the JPEG image
// nJpgImageSize is the size of the buffer
// ulJpgWidth is the width of the JPEG image
// ulJpgHeight is the height of the JPEG image
//

//
// Check if CHECKJPEGFORMAT is supported (device has JPEG support)
// and use it to verify that device can handle the JPEG image.
//

ul = CHECKJPEGFORMAT;

if (
    // Check if CHECKJPEGFORMAT exists:

    (ExtEscape(hdc, QUERYESCSUPPORT,
               sizeof(ul), &ul, 0, 0) > 0) &&

    // Check if CHECKJPEGFORMAT executed without error:

    (ExtEscape(hdc, CHECKJPEGFORMAT,
               pvJpgImage, nJpgImageSize, sizeof(ul), &ul) > 0) &&

    // Check status code returned by CHECKJPEGFORMAT:

    (ul == 1)
   )
{
    //
    // Initialize the BITMAPINFO.
    //

    memset(&bmi, 0, sizeof(bmi));
    bmi.bmiHeader.biSize        = sizeof(BITMAPINFOHEADER);
    bmi.bmiHeader.biWidth       = ulJpgWidth;
    bmi.bmiHeader.biHeight      = -ulJpgHeight; // top-down image
    bmi.bmiHeader.biPlanes      = 1;
    bmi.bmiHeader.biBitCount    = 0;
    bmi.bmiHeader.biCompression = BI_JPEG;
    bmi.bmiHeader.biSizeImage   = nJpgImageSize;

    //
    // Do the SetDIBitsToDevice.
    //

    iRet = SetDIBitsToDevice(hdc,
                             ulDstX, ulDstY,
                             ulDstWidth, ulDstHeight,
                             0, 0,
                             0, ulJpgHeight,
                             pvJpgImage,
                             &bmi,
                             DIB_RGB_COLORS);

    if (iRet == GDI_ERROR)
        return FALSE;
}
else
{
    //
    // Decompress image into a DIB and call SetDIBitsToDevice
    // with the DIB instead.
    //
}
```

Sizing a JPEG or PNG Image

4/20/2022 • 2 minutes to read • Edit Online

The **StretchDIBits** function copies the color data for a rectangle of pixels in a DIB to the specified destination rectangle. If the destination rectangle is larger than the source rectangle, this function stretches the rows and columns of color data to fit the destination rectangle. If the destination rectangle is smaller than the source rectangle, **StretchDIBits** compresses the rows and columns by using the specified raster operation.

StretchDIBits is extended to allow a JPEG or PNG image to be passed as the source image.

For example:

```
// pvJpgImage points to a buffer containing the JPEG image
// nJpgImageSize is the size of the buffer
// ulJpgWidth is the width of the JPEG image
// ulJpgHeight is the height of the JPEG image
//

//
// Check if CHECKJPEGFORMAT is supported (device has JPEG support)
// and use it to verify that device can handle the JPEG image.
//

ul = CHECKJPEGFORMAT;

if (
    // Check if CHECKJPEGFORMAT exists:

    (ExtEscape(hdc, QUERYESCSUPPORT,
               sizeof(ul), &ul, 0, 0) > 0) &&

    // Check if CHECKJPEGFORMAT executed without error:

    (ExtEscape(hdc, CHECKJPEGFORMAT,
               nJpgImageSize, pvJpgImage, sizeof(ul), &ul) > 0) &&

    // Check status code returned by CHECKJPEGFORMAT:

    (ul == 1)
   )
{
    //
    // Initialize the BITMAPINFO.
    //

    memset(&bmi, 0, sizeof(bmi));
    bmi.bmiHeader.biSize        = sizeof(BITMAPINFOHEADER);
    bmi.bmiHeader.biWidth       = ulJpgWidth;
    bmi.bmiHeader.biHeight      = -ulJpgHeight; // top-down image
    bmi.bmiHeader.biPlanes      = 1;
    bmi.bmiHeader.biBitCount    = 0;
    bmi.bmiHeader.biCompression = BI_JPEG;
    bmi.bmiHeader.biSizeImage   = nJpgImageSize;

    //
    // Do the StretchDIBits.
    //

    iRet = StretchDIBits(hdc,
                         // destination rectangle
                         ulDstX, ulDstY, ulDstWidth, ulDstHeight,
                         // source rectangle
                         0, 0, ulJpgWidth, ulJpgHeight,
                         pvJpgImage,
                         &bmi,
                         DIB_RGB_COLORS,
                         SRCCOPY);

    if (iRet == GDI_ERROR)
        return FALSE;
}
else
{
    //
    // Decompress image into a DIB and call StretchDIBits
    // with the DIB instead.
    //
}
```

Bitmap Reference

4/20/2022 • 2 minutes to read • Edit Online

The following elements are used with bitmaps:

- Bitmap Functions
- Bitmap Structures
- Bitmap Macros

Bitmap Functions (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used with bitmaps.

FUNCTION	DESCRIPTION
AlphaBlend	Displays a bitmap with transparent or semitransparent pixels.
BitBlt	Performs a bit-block transfer.
CreateBitmap	Creates a bitmap.
CreateBitmapIndirect	Creates a bitmap.
CreateCompatibleBitmap	Creates a bitmap compatible with a device.
CreateDIBitmap	Creates a device-dependent bitmap (DDB) from a DIB.
CreateDIBSection	Creates a DIB that applications can write to directly.
ExtFloodFill	Fills an area of the display surface with the current brush.
GetBitmapDimensionEx	Gets the dimensions of a bitmap.
GetDIBColorTable	Retrieves RGB color values from a DIB section bitmap.
GetDIBits	Copies a bitmap into a buffer.
GetPixel	Gets the RGB color value of the pixel at a given coordinate.
GetStretchBltMode	Gets the current stretching mode.
GradientFill	Fills rectangle and triangle structures.
LoadBitmap	Loads a bitmap from a module's executable file.
MaskBlt	Combines the color data in the source and destination bitmaps.
PlgBlt	Performs a bit-block transfer.
SetBitmapDimensionEx	Sets the preferred dimensions to a bitmap.
SetDIBColorTable	Sets RGB values in a DIB.
SetDIBits	Sets the pixels in a bitmap using color data from a DIB.

FUNCTION	DESCRIPTION
SetDIBitsToDevice	Sets the pixels in a rectangle using color data from a DIB.
SetPixel	Sets the color for a pixel.
SetPixelV	Sets a pixel to the best approximation of a color.
SetStretchBltMode	Sets the bitmap stretching mode.
StretchBlt	Copies a bitmap and stretches or compresses it.
StretchDIBits	Copies the color data in a DIB.
TransparentBlt	Performs a bit-block transfer of color data.

Obsolete Functions

The following functions are provided only for compatibility with 16-bit versions of Microsoft Windows:

- CreateDiscardableBitmap
- FloodFill
- GetBitmapBits
- SetBitmapBits

Bitmap Structures

4/20/2022 • 2 minutes to read • Edit Online

The following structures are used with bitmaps:

- BITMAP
- BITMAPCOREHEADER
- BITMAPCOREINFO
- BITMAPFILEHEADER
- BITMAPINFO
- BITMAPINFOHEADER
- BITMAPV4HEADER
- BITMAPV5HEADER
- BLENDFUNCTION
- COLORADJUSTMENT
- DIBSECTION
- GRADIENT_RECT
- GRADIENT_TRIANGLE
- RGBQUAD
- RGBTRIPLE
- SIZE
- TRIVERTEX

Bitmap Macros

4/20/2022 • 2 minutes to read • Edit Online

The following macro is used with bitmaps.

MAKEROP4

Brushes (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

A *brush* is a graphics tool that applications use to paint the interior of polygons, ellipses, and paths. Drawing applications use brushes to paint shapes; word processing applications use brushes to paint rules; computer-aided design (CAD) applications use brushes to paint the interiors of cross-section views; and spreadsheet applications use brushes to paint the sections of pie charts and the bars in bar graphs.

- About Brushes
- Using Brushes
- Brush Reference

About Brushes

4/20/2022 • 2 minutes to read • Edit Online

There are two types of brushes: logical and physical. A *logical brush* is a description of the ideal bitmap that an application uses to paint shapes. A *physical brush* is the actual bitmap that a device driver creates based on an application's logical-brush definition. For more information about bitmaps, see Bitmaps.

When an application calls one of the functions that creates a brush, it retrieves a handle that identifies a logical brush. When the application passes this handle to the **SelectObject** function, the device driver for the corresponding display or printer creates the physical brush.

The following topics describe brushes:

- Brush Origin
- Logical Brush Types
- Pattern Block Transfer
- ICM-Enabled Brush Functions

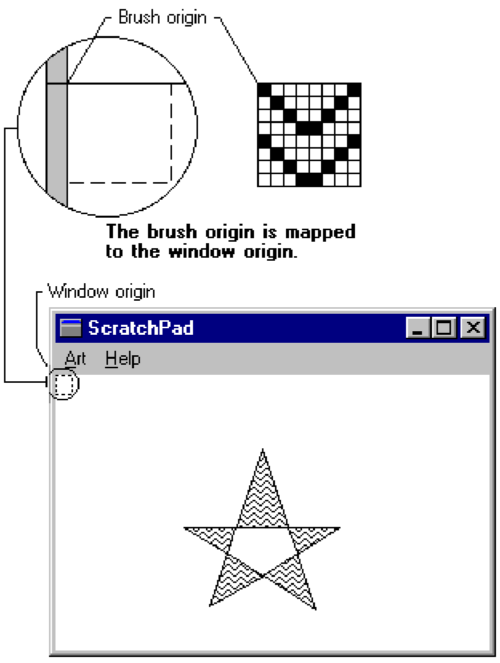
Brush Origin

4/20/2022 • 2 minutes to read • Edit Online

When an application calls a drawing function to paint a shape, the system positions a brush at the start of the paint operation and maps a pixel in the brush bitmap to the client area at the *window origin*, which is the upper-left corner of the window. The coordinates of the pixel that the system maps are called the *brush origin*. The default brush origin is located in the upper-left corner of the brush bitmap, at the coordinates (0,0). The system then copies the brush across the client area, forming a pattern that is as tall as the bitmap. The copy operation continues, row by row, until the entire client area is filled. However, the brush pattern is visible only within the boundaries of the specified shape.

There are instances when the default brush origin should not be used. For example, it may be necessary for an application to use the same brush to paint the backgrounds of its parent and child windows and blend a child window's background with that of the parent window. To do this, the application should reset the brush origin by calling the SetBrushOrgEx function and shifting the origin the required number of pixels. (An application can retrieve the current brush origin by calling the GetBrushOrgEx function.)

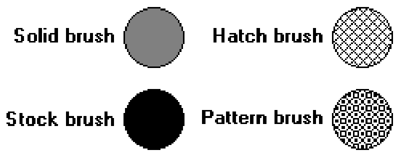
The following illustration shows a five-pointed star filled by using an application-defined brush. The illustration shows a zoomed image of the brush, as well as the location to which it was mapped at the beginning of the paint operation.



Logical Brush Types

4/20/2022 • 2 minutes to read • Edit Online

There are four types of logical brushes: solid, stock, hatch, and pattern. These brushes are shown in the following illustration.



The stock and hatch types each have several predefined brushes.

The **CreateBrushIndirect** function creates a logical brush with a specified style, color, and pattern.

Solid Brush

4/20/2022 • 2 minutes to read • Edit Online

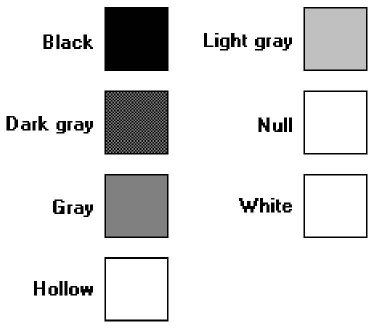
A *solid brush* is a logical brush that contains 64 pixels of the same color. An application can create a solid logical brush by calling the **CreateSolidBrush** function, specifying the color of the brush required. After creating the solid brush, the application can select it into its device context and use it to paint filled shapes.

Stock Brush

4/20/2022 • 2 minutes to read • Edit Online

There are seven predefined logical stock brushes maintained by the graphics device interface (GDI). There are also 21 predefined logical stock brushes maintained by the window management interface (USER).

The following illustration shows rectangles painted by using the seven predefined stock brushes.



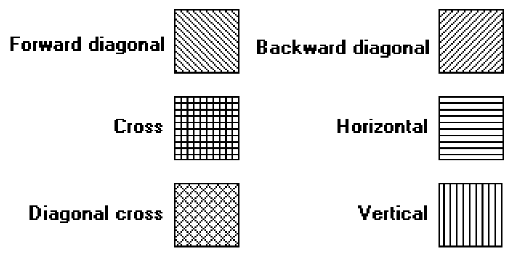
An application can retrieve a handle identifying one of the seven stock brushes by calling the GetStockObject function, specifying the brush type.

The 21 stock brushes maintained by the window management interface correspond to the colors of window elements such as menus, scroll bars, and buttons. An application can obtain a handle identifying one of these brushes by calling the GetSysColorBrush function and specifying a system-color value. An application can retrieve the color corresponding to a particular window element by calling the GetSysColor function. An application can set the color corresponding to a window element by calling the SetSysColors function.

Hatch Brush

4/20/2022 • 2 minutes to read • Edit Online

There are six predefined logical hatch brushes maintained by GDI. The following rectangles were painted by using the six predefined hatch brushes.

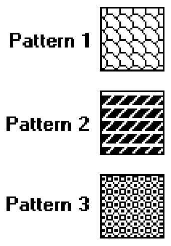


An application can create a hatch brush by calling the **CreateHatchBrush** function, specifying one of the six hatch styles.

Pattern Brush

4/20/2022 • 2 minutes to read • Edit Online

A pattern (or custom) brush is created from an application-defined bitmap or device-independent bitmap (DIB). The following rectangles were painted by using different pattern brushes.



To create a logical pattern brush, an application must first create a bitmap. After creating the bitmap, the application can create the logical pattern brush by calling the **CreatePatternBrush** or **CreateDIBPatternBrushPt** function, supplying a handle that identifies the bitmap (or DIB). The brushes that appear in the preceding illustration were created from monochrome bitmaps. For a description of bitmaps, DIBs, and the functions that create them, see Bitmaps.

Pattern Block Transfer

4/20/2022 • 2 minutes to read • Edit Online

The name of the **PatBlt** function (an abbreviation for pattern block transfer) implies that this function simply replicates the brush (or pattern) until it fills a specified rectangle. However, the function is actually much more powerful. Before replicating the brush, it combines the color data for the pattern with the color data for the existing pixels on the video display by using a raster operation (ROP). An ROP is a bitwise operation that is applied to the bits of color data for the replicated brush and the bits of color data for the target rectangle on the display device. There are 256 ROPs; however, the **PatBlt** function recognizes only those that require a pattern and a destination (not those that require a source). The following table identifies the most common ROPs.

ROP	DESCRIPTION
PATCOPY	Copies the pattern to the destination bitmap.
PATINVERT	Combines the destination bitmap with the pattern by using the Boolean XOR operator.
DSTINVERT	Inverts the destination bitmap.
BLACKNESS	Turns all output to binary zeros.
WHITENESS	Turns all output to binary ones.

For more information, see Raster Operation Codes.

ICM-Enabled Brush Functions

4/20/2022 • 2 minutes to read • Edit Online

Microsoft Image Color Management (ICM) ensures that a color image, graphic, or text object is rendered as close as possible to its original intent on any device, despite differences in imaging technologies and color capabilities between devices. Whether you are scanning an image or other graphic on a color scanner, downloading it over the Internet, viewing or editing it on the screen, or outputting it to paper, film, or other media, ICM 2.0 helps you keep its colors consistent and accurate. For more information about ICM, see Windows Color System.

The following brush functions are enabled for use with ICM:

- CreateBrushIndirect
- CreateDIBPatternBrush
- CreateDIBPatternBrushPt
- CreateHatchBrush
- CreatePatternBrush
- CreateSolidBrush

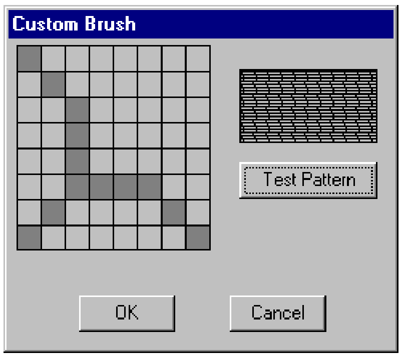
Using Brushes

4/20/2022 • 6 minutes to read • Edit Online

You can use a brush to paint the interior of virtually any shape by using a graphics device interface (GDI) function. This includes the interiors of rectangles, ellipses, polygons, and paths. Depending on the requirements of your application, you can use a solid brush of a specified color, a stock brush, a hatch brush, or a pattern brush.

This section contains code samples that demonstrate the creation of a custom brush dialog box. The dialog box contains a grid that represents the bitmap the system uses as a brush. A user can use this grid to create a pattern-brush bitmap and then view the custom pattern by clicking the **Test Pattern** button.

The following illustration shows a pattern created by using the **Custom Brush** dialog box.



To display a dialog box, you must first create a dialog box template. The following dialog box template defines the **Custom Brush** dialog box.

```
CustBrush DIALOG 6, 18, 160, 118
STYLE WS_DLGFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION
CAPTION "Custom Brush"
FONT 8, "MS Sans Serif"
BEGIN
    CONTROL         "", IDD_GRID, "Static", SS_BLACKFRAME |
                    WS_CHILD, 3, 2, 83, 79
    CONTROL         "", IDD_RECT, "Static", SS_BLACKFRAME |
                    WS_CHILD, 96, 11, 57, 28
    PUSHBUTTON      "Test Pattern", IDD_PAINTRECT, 96, 47, 57, 14
    PUSHBUTTON      "OK", IDD_OK, 29, 98, 40, 14
    PUSHBUTTON      "Cancel", IDD_CANCEL, 92, 98, 40, 14
END
```

The **Custom Brush** dialog box contains five controls: a bitmap-grid window, a pattern-viewing window, and three push buttons, labeled **Test Pattern**, **OK**, and **Cancel**. The **Test Pattern** push button enables the user to view the pattern. The dialog box template specifies the overall dimensions of the dialog box window, assigns a value to each control, specifies the location of each control, and so forth. For more information, see Dialog Boxes.

The control values in the dialog box template are constants that have been defined as follows in the application's header file.

```
#define IDD_GRID      120
#define IDD_RECT      121
#define IDD_PAINTRECT 122
#define IDD_OK        123
#define IDD_CANCEL    124
```

After you create a dialog box template and include it in the application's resource-definition file, you must write a dialog procedure. This procedure processes messages that the system sends to the dialog box. The following excerpt from an application's source code shows the dialog box procedure for the **Custom Brush** dialog box and the two application-defined functions it calls.

```
BOOL CALLBACK BrushDlgProc( HWND hdlg, UINT message, LONG wParam,
                            LONG lParam)
{
    static HWND hwndGrid;       // grid-window control
    static HWND hwndBrush;      // pattern-brush control
    static RECT rctGrid;        // grid-window rectangle
    static RECT rctBrush;       // pattern-brush rectangle
    static UINT bBrushBits[8];  // bitmap bits
    static RECT rect[64];       // grid-cell array
    static HBITMAP hbm;         // bitmap handle
    HBRUSH hbrush;              // current brush
    HBRUSH hbrushOld;           // default brush
    HRGN hrgnCell;              // test-region handle
    HDC hdc;                    // device context (DC) handle
    int x, y, deltaX, deltaY;   // drawing coordinates
    POINTS ptlHit;              // mouse coordinates
    int i;                      // count variable

    switch (message)
        {
        case WM_INITDIALOG:

            // Retrieve a window handle for the grid-window and
            // pattern-brush controls.

            hwndGrid = GetDlgItem(hdlg, IDD_GRID);
            hwndBrush = GetDlgItem(hdlg, IDD_RECT);

            // Initialize the array of bits that defines the
            // custom brush pattern with the value 1 to produce a
            // solid white brush.

            for (i=0; i<8; i++)
                bBrushBits[i] = 0xFF;

            // Retrieve dimensions for the grid-window and pattern-
            // brush controls.

            GetClientRect(hwndGrid, &rctGrid);
            GetClientRect(hwndBrush, &rctBrush);

            // Determine the width and height of a single cell.

            deltaX = (rctGrid.right - rctGrid.left)/8;
            deltaY = (rctGrid.bottom - rctGrid.top)/8;

            // Initialize the array of cell rectangles.

            for (y=rctGrid.top, i=0; y < rctGrid.bottom; y += deltaY)
            {
                for(x=rctGrid.left; x < (rctGrid.right - 8) && i < 64;
                        x += deltaX, i++)
                {
                    rect[i].left = x; rect[i].top = y;
                    rect[i].right = x + deltaX;
                    rect[i].bottom = y + deltaY;
                }
            }
            return FALSE;


        case WM_PAINT:

            // Draw the grid.

            hdc = GetDC(hwndGrid);

            for (i=rctGrid.left; i<rctGrid.right;
                 i+=(rctGrid.right - rctGrid.left)/8)
            {
                 MoveToEx(hdc, i, rctGrid.top, NULL);
                 LineTo(hdc, i, rctGrid.bottom);
            }
            for (i=rctGrid.top; i<rctGrid.bottom;
                 i+=(rctGrid.bottom - rctGrid.top)/8)
            {
                 MoveToEx(hdc, rctGrid.left, i, NULL);
                 LineTo(hdc, rctGrid.right, i);
            }
            ReleaseDC(hwndGrid, hdc);
            return FALSE;


        case WM_LBUTTONDOWN:

            // Store the mouse coordinates in a POINT structure.

            ptlHit = MAKEPOINTS((POINTS FAR *)lParam);

            // Create a rectangular region with dimensions and
            // coordinates that correspond to those of the grid
            // window.

            hrgnCell = CreateRectRgn(rctGrid.left, rctGrid.top,
                        rctGrid.right, rctGrid.bottom);

            // Retrieve a window DC for the grid window.

            hdc = GetDC(hwndGrid);

            // Select the region into the DC.

            SelectObject(hdc, hrgnCell);

            // Test for a button click in the grid-window rectangle.

            if (PtInRegion(hrgnCell, ptlHit.x, ptlHit.y))
            {
                // A button click occurred in the grid-window
                // rectangle; isolate the cell in which it occurred.

                for(i=0; i<64; i++)
                {
                    DeleteObject(hrgnCell);

                    hrgnCell = CreateRectRgn(rect[i].left,
                               rect[i].top,
                               rect[i].right, rect[i].bottom);

                    if (PtInRegion(hrgnCell, ptlHit.x, ptlHit.y))
                    {
                        InvertRgn(hdc, hrgnCell);
                        // Set the appropriate brush bits.

                         if (i % 8 == 0)
                            bBrushBits[i/8] = bBrushBits[i/8] ^ 0x80;
                         else if (i % 8 == 1)
                            bBrushBits[i/8] = bBrushBits[i/8] ^ 0x40;
                         else if (i % 8 == 2)
                            bBrushBits[i/8] = bBrushBits[i/8] ^ 0x20;
                         else if (i % 8 == 3)
                            bBrushBits[i/8] = bBrushBits[i/8] ^ 0x10;
                         else if (i % 8 == 4)
                            bBrushBits[i/8] = bBrushBits[i/8] ^ 0x08;
                         else if (i % 8 == 5)
                            bBrushBits[i/8] = bBrushBits[i/8] ^ 0x04;
                         else if (i % 8 == 6)
                            bBrushBits[i/8] = bBrushBits[i/8] ^ 0x02;
                         else if (i % 8 == 7)
                            bBrushBits[i/8] = bBrushBits[i/8] ^ 0x01;

                      // Exit the "for" loop after the bit is set.

                         break;
                    } // end if

                } // end for

            } // end if

            // Release the DC for the control.

            ReleaseDC(hwndGrid, hdc);
            return TRUE;


        case WM_COMMAND:
            switch (wParam)
            {
               case IDD_PAINTRECT:

                    hdc = GetDC(hwndBrush);

                    // Create a monochrome bitmap.

                    hbm = CreateBitmap(8, 8, 1, 1,
                          (LPBYTE)bBrushBits);

                    // Select the custom brush into the DC.

                    hbrush = CreatePatternBrush(hbm);

                    hbrushOld = SelectObject(hdc, hbrush);

                    // Use the custom brush to fill the rectangle.

                    Rectangle(hdc, rctBrush.left, rctBrush.top,
                              rctBrush.right, rctBrush.bottom);

                    // Clean up memory.
                    SelectObject(hdc, hbrushOld);
                    DeleteObject(hbrush);
                    DeleteObject(hbm);

                    ReleaseDC(hwndBrush, hdc);
                    return TRUE;

                case IDD_OK:

                case IDD_CANCEL:
                    EndDialog(hdlg, TRUE);
```

```
                    return TRUE;

                } // end switch
                break;
            default:
                return FALSE;
            }
    }


    int GetStrLngth(LPTSTR cArray)
    {
        int i = 0;

        while (cArray[i++] != 0);
        return i-1;

    }

    DWORD RetrieveWidth(LPTSTR cArray, int iLength)
    {
        int i, iTmp;
        double dVal, dCount;

        dVal = 0.0;
        dCount = (double)(iLength-1);
        for (i=0; i<iLength; i++)
        {
            iTmp = cArray[i] - 0x30;
            dVal = dVal + (((double)iTmp) * pow(10.0, dCount--));
        }

        return (DWORD)dVal;
    }
```

The dialog box procedure for the **Custom Brush** dialog box processes four messages, as described in the following table.

MESSAGE	ACTION
WM_INITDIALOG	Retrieves a window handle and dimensions for the grid-window and pattern-brush controls, computes the dimensions of a single cell in the grid-window control, and initializes an array of grid-cell coordinates.
WM_PAINT	Draws the grid pattern in the grid-window control.
WM_LBUTTONDOWN	Determines whether the cursor is within the grid-window control when the user presses the left mouse button. If so, the dialog box procedure inverts the appropriate grid cell and records the state of that cell in an array of bits that is used to create the bitmap for the custom brush.
WM_COMMAND	Processes input for the three push button controls. If the user clicks the **Test Pattern** button, the dialog box procedure paints the Test Pattern control with the new custom brush pattern. If the user clicks the **OK** or **Cancel** button, the dialog box procedure performs actions accordingly.

For more information about messages and message processing, see Messages and Message Queues.

After you write the dialog box procedure, include the function definition for the procedure in the application's header file and then call the dialog box procedure at the appropriate point in the application.

The following excerpt from the application's header file shows the function definition for the dialog box procedure and the two functions it calls.

```
BOOL CALLBACK BrushDlgProc(HWND, UINT, WPARAM, LPARAM);
int GetStrLngth(LPTSTR);
DWORD RetrieveWidth(LPTSTR, int);
```

Finally, the following code shows how the dialog box procedure is called from the application's source-code file.

```
    DialogBox((HANDLE)GetModuleHandle(NULL),
        (LPTSTR)"CustBrush",
        hWnd,
        (DLGPROC) BrushDlgProc);
```

This call is usually made in response to the user choosing an option from the application's menu.

Brush Reference

4/20/2022 • 2 minutes to read • Edit Online

The following elements are used with brushes:

- Brush Functions
- Brush Structures

Brush Functions (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used with brushes.

FUNCTION	DESCRIPTION
CreateBrushIndirect	Creates a brush with a specified style, color, and pattern
CreateDIBPatternBrushPt	Creates a brush with the pattern from a DIB
CreateHatchBrush	Creates a brush with a hatch pattern and color
CreatePatternBrush	Creates a brush with a bitmap pattern
CreateSolidBrush	Creates a brush with a solid color
GetBrushOrgEx	Gets the brush origin for a device context
GetSysColorBrush	Gets a handle to a brush that corresponds to a color index
PatBlt	Paints a rectangle
SetBrushOrgEx	Sets the brush origin for a device context
SetDCBrushColor	Sets the current device context brush color.

Obsolete Functions

The following functions are provided only for compatibility with 16-bit versions of Windows.

CreateDIBPatternBrush

Brush Structures

4/20/2022 • 2 minutes to read • Edit Online

The following structures are used with brushes:

- LOGBRUSH
- LOGBRUSH32

Clipping (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

Clipping is the process of limiting output to a region or path within the client area of an application window. The following sections discuss clipping.

- About Clipping
- Using Clipping
- Clipping Reference

About Clipping

4/20/2022 • 2 minutes to read • Edit Online

Clipping is used by applications in a variety of ways. Word processing and spreadsheet applications clip keyboard input to keep it from appearing in the margins of a page or spreadsheet. Computer-aided design (CAD) and drawing applications clip graphics output to keep it from overwriting the edges of a drawing or picture.

A *clipping region* is a region with edges that are either straight lines or curves. A *clip path* is a region with edges that are straight lines, Bézier curves, or combinations of both. For more information about regions, see Regions. For more information about paths, see Paths.

This overview covers the following topics:

- Clipping Regions
- Clip Paths

Clipping Regions

4/20/2022 • 2 minutes to read • Edit Online

A clipping region is one of the graphic objects that an application can select into a device context (DC). It is typically rectangular. Some device contexts provide a predefined or default clipping region while others do not. For example, if you obtain a device context handle from the **BeginPaint** function, the DC contains a predefined rectangular clipping region that corresponds to the invalid rectangle that requires repainting. However, when you obtain a device context handle by calling the **GetDC** function with a **NULL** *hWnd* parameter, or by calling the **CreateDC** function, the DC does not contain a default clipping region. For more information about device contexts returned by the **BeginPaint** function, see Painting and Drawing . For more information about device contexts returned by the **CreateDC** and **GetDC** functions, see Device Contexts.

Applications can perform a variety of operations on clipping regions. Some of these operations require a handle identifying the region and some do not. For example, an application can perform the following operations directly on a device context's clipping region.

- Determine whether graphics output appears within the region's borders by passing coordinates of the corresponding line, arc, bitmap, text, or filled shape to the **PtVisible** function.
- Determine whether part of the client area intersects a region by calling the **RectVisible** function.
- Move the existing region by a specified offset by calling the **OffsetClipRgn** function.
- Exclude a rectangular part of the client area from the current clipping region by calling the **ExcludeClipRect** function.
- Combine a rectangular part of the client area with the current clipping region by calling the **IntersectClipRect** function.

After obtaining a handle identifying the clipping region, an application can perform any operation that is common with regions, such as:

- Combining a copy of the current clipping region with a second region by calling the **CombineRgn** function.
- Compare a copy of the current clipping region to a second region by calling the **EqualRgn** function.
- Determine whether a point lies within the interior of a copy of the current clipping region by calling the **PtInRegion** function.

Clip Paths

4/20/2022 • 2 minutes to read • Edit Online

Like a clipping region, a clip path is another graphics object that an application can select into a device context. Unlike a clipping region, a clip path is always created by an application and it is used for clipping to one or more irregular shapes. For example, an application can use the lines and curves that form the outlines of characters in a string of text to define a clip path.

To create a clip path, it's first necessary to create a path that describes the required irregular shape. Paths are created by calling the appropriate graphics device interface (GDI) drawing functions after calling the **BeginPath** function and before calling the **EndPath** function. This collection of functions is called a path bracket. For more information about paths and path brackets, see Paths.

After the path is created, it can be converted to a clip path by calling the **SelectClipPath** function, identifying a device context, and specifying a usage mode. The usage mode determines how the system combines the new clip path with the device context's original clipping region. The following table describes the usage modes.

MODE	DESCRIPTION
RGN_AND	The clip path includes the intersection (overlapping areas) of the device context's clipping region and the current path.
RGN_COPY	The clip path is the current path.
RGN_DIFF	The clip path includes the device context's clipping region with any intersecting parts of the current path excluded.
RGN_OR	The clip path includes the union (combined areas) of the device context's clipping region and the current path.
RGN_XOR	The clip path includes the union of the device context's clipping region and the current path but excludes the intersection.

Using Clipping

4/20/2022 • 2 minutes to read • Edit Online

This section contains example code that shows how to generate a clip path consisting of a character string. The example creates a logical font and uses it to draw a string within a clip path, then fills the path by drawing horizontal and vertical lines.

```
// DoClipPat - Draws a clip path using the specified string
// Return value - TRUE if successful; FALSE otherwise
// lplf - address of a LOGFONT structure that defines the font to
//        use to draw the clip path
// lpsz - address of a string to use for the clip path

BOOL DoClipPath(LPLOGFONT lplf, LPSTR lpsz)
{
    LOGFONT lf;           // logical font structure
    HFONT hfont;          // new logical font handle
    HFONT hfontOld;       // original logical font handle
    HDC hdc;              // display DC handle
    int nXStart, nYStart; // drawing coordinates
    RECT rc;              // rectangle structure for painting window
    SIZE sz;              // size structure that receives text extents
    int nStrLen;          // length of the string
    int i;                // loop counter
        HRESULT hr;
        size_t * pcch;

    // Retrieve a cached DC for the window.

    hdc = GetDC(hwnd);

    // Erase the current window contents.

    GetClientRect(hwnd, &rc);
    FillRect(hdc, &rc, GetStockObject(WHITE_BRUSH));

    // Use the specified font to create a logical font and select it
    // into the DC.

    hfont = CreateFontIndirect(lplf);
    if (hfont == NULL)
        return FALSE;
    hfontOld = SelectObject(hdc, hfont);

    // Create a clip path.

        hr = StringCchLength(lpsz, STRSAFE_MAX_CCH, pcch);
        if (FAILED(hr))
        {
        // TODO: write error handler
        }
        nStrLen = *pcch
    BeginPath(hdc);
        TextOut(hdc, nXStart, nYStart, lpsz, nStrLen);
    EndPath(hdc);
    SelectClipPath(hdc, RGN_DIFF);

    // Retrieve the dimensions of the rectangle surrounding
    // the text.

    GetTextExtentPoint32(hdc, lpsz, nStrLen, &sz);
    // Draw horizontal lines through the clip path.

    for (i = nYStart + 1; i < (nYStart + sz.cy); i += 3)
    {
       MoveToEx(hdc, nXStart, i, (LPPOINT) NULL);
       LineTo(hdc, (nXStart + sz.cx), i);
    }

    // Draw vertical lines through the clip path.

    for (i = nXStart + 1; i < (nXStart + sz.cx); i += 3)
    {
       MoveToEx(hdc, i, nYStart, (LPPOINT) NULL);
       LineTo(hdc, i, (nYStart + sz.cy));
    }

    // Select the original font into the DC and release the DC.

    SelectObject(hdc, hfontOld);
    DeleteObject(hfont);
    ReleaseDC(hwnd, hdc);

    return TRUE;
}
```

For an example that demonstrates how an application creates a rectangular clipping region, see Regions.

Clipping Reference

4/20/2022 • 2 minutes to read • Edit Online

The following elements are used with clipping.

- Clipping Functions

Clipping Functions

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used with clipping.

FUNCTION	DESCRIPTION
ExcludeClipRect	Creates a new clipping region that consists of the existing clipping region minus the specified rectangle.
ExtSelectClipRgn	Combines the specified region with the current clipping region using the specified mode.
GetClipBox	Retrieves the dimensions of the tightest bounding rectangle that can be drawn around the current visible area on the device.
GetClipRgn	Retrieves a handle identifying the current application-defined clipping region for the specified device context.
GetMetaRgn	Retrieves the current metaregion for the specified device context.
GetRandomRgn	Copies the system clipping region of a specified device context to a specific region.
IntersectClipRect	Creates a new clipping region from the intersection of the current clipping region and the specified rectangle.
OffsetClipRgn	Moves the clipping region of a device context by the specified offsets.
PtVisible	Determines whether the specified point is within the clipping region of a device context.
RectVisible	Determines whether any part of the specified rectangle lies within the clipping region of a device context.
SelectClipPath	Selects the current path as a clipping region for a device context, combining the new region with any existing clipping region by using the specified mode.
SelectClipRgn	Selects a region as the current clipping region for the specified device context.
SetMetaRgn	Intersects the current clipping region for the specified device context with the current metaregion and saves the combined region as the new metaregion for the specified device context.

Colors

4/20/2022 • 2 minutes to read • Edit Online

Color is an important element in the pictures and images generated by applications. This overview describes how applications can manage and use colors with pens, brushes, text, or bitmaps.

- About Colors
- Using Color
- Color Reference

About Colors

4/20/2022 • 2 minutes to read • Edit Online

Color can be used to communicate ideas, show relationships between items, and improve the appeal and quality of output. Windows enables applications to discover the color capabilities of given devices and to choose from the available colors those that best suit their needs.

This overview provides information on the following topics:

- Color Basics
- Color Palettes

Although not described in this overview, image color matching is an important feature of color management that helps ensure that color images look the same whether displayed on screen or printed on paper. For more information, see Windows Color System.

Color Basics

4/20/2022 • 2 minutes to read • Edit Online

The color capabilities of devices, such as displays and printers, can range from monochrome to thousands of colors. Because an application may need to generate output for devices throughout this range, it should be prepared to handle varying color capabilities.

An application can discover the number of colors available for a given device by using the **GetDeviceCaps** function to retrieve the NUMCOLORS value. This value specifies the count of colors available for use by the application. Usually, this count corresponds to a physical property of the output device, such as the number of inks in the printer or the number of distinct color signals the display adapter can transmit to the monitor.

Although the NUMCOLORS value specifies the count of colors, it does not identify what the available colors are. An application can discover what colors are available by enumerating all pens having the PS_SOLID type. Because the device driver that supports a given device usually has a full range of solid pens and because the system requires that solid pens have only colors that the device can generate, enumerating these pens is often equivalent to enumerating the colors. An application can enumerate the pens by using the **EnumObjects** function. For a code example, see Enumerating Colors.

For more information, see the following topics:

- Color Values
- Color Approximations and Dithering
- Color in Bitmaps
- Color Mixing

Color Values

4/20/2022 • 2 minutes to read • Edit Online

Color is defined as a combination of three primary colors red, green, and blue. the system identifies a color by giving it a color value (sometimes called an RGB triplet), which consists of three 8-bit values specifying the intensities of its color components. Black has the minimum intensity for red, green, and blue, so the color value for black is (0, 0, 0). White has the maximum intensity for red, green, and blue, so its color value is (255, 255, 255).

NOTE

If image color matching is enabled, the definition of color and the meaning of a color value depends on the type of color space that is currently set for the device context.

The system and applications use parameters and variables having the COLORREF type to pass and store color values. For example, the **EnumObjects** function identifies the color of each pen by setting the **lopnColor** member in a **LOGPEN** structure to a color value. Applications can extract the individual values of the red, green, and blue components from a color value by using the **GetRValue**, **GetGValue**, and **GetBValue** macros, respectively. Applications can create a color value from individual component values by using the **RGB** macro. When creating or examining a logical palette, an application uses the **RGBQUAD** structure to define color values and to examine individual component values.

Color Approximations and Dithering

4/20/2022 • 2 minutes to read • Edit Online

Although an application can use color without regard to the color capabilities of the device, the resulting output may not be as informative and pleasing as output for which color is carefully chosen. Few, if any, devices guarantee an exact match for every possible color value; therefore, if an application requests a color that the device cannot generate, the system approximates that color by using a color that the device can generate. For example, if an application attempts to create a red pen for a black and white printer, it will receive a black pen instead the system uses black as the approximation for red.

An application can discover whether the system will approximate a given color by using the **GetNearestColor** function. The function takes a color value and returns the color value of the closest matching color the device can generate. The method the system uses to determine this approximation depends on the device driver and its color capabilities. In most cases, the approximated color's overall intensity is closest to that of the requested color.

When an application creates a pen or sets the color for text, the system always approximates a color if no exact match exists. When an application creates a solid brush, the system may attempt to simulate the requested color by dithering. *Dithering* simulates a color by alternating two or more colors in a pattern. For example, different shades of pink can be simulated by alternating different combinations of red and white. Depending on the colors and the pattern, dithering can produce reasonable simulations. It is most useful for monochrome devices, because it expands the number of available "colors" well beyond simple black and white.

The method used to create dithered colors depends on the device driver. Most device drivers use a standard dithering algorithm, which generates a pattern based on the intensity values of the requested red, green, and blue colors. In general, any requested color that cannot be generated by the device is subject to simulation, but an application is not notified when the system simulates a color. Furthermore, an application cannot modify or change the dithering algorithm of the device driver. An application, however, can bypass the algorithm by creating and using pattern brushes. In this way, the application creates its own dithered colors by combining solid colors in the bitmap that it uses to create the brush.

Color in Bitmaps

4/20/2022 • 2 minutes to read • Edit Online

The system handles colors in bitmaps differently than colors in pens, brushes, and text. Compatible bitmaps, created by using the **CreateBitmap** or **CreateCompatibleBitmap** function, are device specific and retain color information in a device-dependent format. No color values are used, and the colors are not subject to approximations and dithering.

Device-independent bitmaps (DIBs) retain color information either as color values or color palette indexes. If color values are used, the colors are subject to approximation, but not dithering. Color palette indexes can only be used with devices that support color palettes. Although the system does not approximate or dither colors identified by indexes, the resulting color may be different than that intended, because the indexes yield valid results only in the context of the color palette that was current at the time the bitmap was created. If the palette changes, so do the colors in the bitmap. For more information on palette indexes, see Default Palette and **PALETTEINDEX**.

In addition to referencing the logical palette, an application can also reference a value in a DIB color table. To select a color in a DIB color table, call **DIBINDEX**. Note that this is only possible for a device context that has a DIB selected into it.

Color Mixing

4/20/2022 • 2 minutes to read • Edit Online

Color mixing lets an application create new colors by combining the pen or brush color with colors in the existing image. The application can choose either to draw the pen or brush color as is (effectively drawing over any existing image) or to mix the color with the colors already present.

The foreground mix mode, sometimes called the binary raster operation, determines how these colors are mixed. An application can merge colors, preserving all components of both colors; mask colors, removing or moderating components that are not common; or exclusively mask colors, removing or moderating components that are common. There are several variations on these basic mixing operations.

Color mixing is subject to color approximation. If the result of color mixing is a color that the device cannot generate, the system approximates the result, using a color it can generate. If an application mixes dithered colors, the individual colors used to create the dithered color are mixed, and the results are subject to color approximation.

An application sets the foreground mix mode by using the **SetROP2** function and retrieves the current mode by using the **GetROP2** function.

Although there is a background mix mode, that mode does not control the mixing of colors. Instead, it specifies whether a background color is used when drawing styled lines, hatched brushes, and text.

Color Palettes (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

A color palette is an array that contains color values identifying the colors that can currently be displayed or drawn on the output device. Color palettes are used by devices that are capable of generating many colors but that can only display or draw a subset of these at any given time. For such devices, the system maintains a *system palette* to track and manage the current colors of the device. Applications do not have direct access to the system palette. Instead, the system associates a default palette with each device context. Applications can use the colors in the default palette or define their own colors by creating *logical palettes* and associating them with individual device contexts.

An application can determine whether a device supports color palettes by checking for the RC_PALETTE bit in the RASTERCAPS value returned by the **GetDeviceCaps** function.

Default Palette

4/20/2022 • 2 minutes to read • Edit Online

The *default palette* is an array of color values identifying the colors that can be used with a device context by default. the system associates the default palette with a context whenever an application creates a context for a device that supports color palettes. The default palette ensures that colors are available for use by an application without any further action.

The default palette typically has 20 entries (colors), but the exact number of entries may vary from device to device. This number is equal to the NUMCOLORS value returned by the **GetDeviceCaps** function. An application can retrieve the color values for colors in the default palette by enumerating solid pens, the same technique used to discover the colors available on nonpalette devices. The colors in the default palette depend on the device. Display devices, for example, often use the 16 standard colors of the VGA display and 4 other colors defined by Windows. Print devices may use other default colors.

When using the default palette, applications use color values to specify pen and text colors. If the requested color is not in the palette, The system approximates the color by using the closest color in the palette. If an application requests a solid brush color that is not in the palette, the system simulates the color by dithering with colors that are in the palette.

To avoid approximations and dithering, applications can also specify pen, brush, and text colors by using color palette indexes rather than color values. A color palette index is an integer value that identifies a specific palette entry. Applications can use color palette indexes in place of color values but must use the **PALETTEINDEX** macro to create the indexes.

Color palette indexes are only useful for devices that support color palettes. To avoid this device dependence, applications that use the same code to draw to both palette and nonpalette devices should use palette-relative color values to specify pen, brush, and text colors. These values are identical to color values except when creating solid brushes. (On palette devices, a solid brush color specified by a palette-relative color value is subject to color approximation instead of dithering.) Applications must use the **PALETTERGB** macro to create palette-relative color values.

The system does not allow an application to change the entries in the default palette. To use colors other than those in the default palette, an application must create its own logical palette and select the palette into the device context.

Logical Palette

4/20/2022 • 2 minutes to read • Edit Online

A *logical palette* is a color palette that an application creates and associates with a given device context. Logical palettes let applications define and use colors that meet their specific needs. Applications can create any number of logical palettes, using them for separate device contexts or switching between them for a single device context. The maximum number of palettes that an application can create depends on the resources of the system.

An application creates a logical palette by using the **CreatePalette** function. The application fills a LOGPALETTE structure, which specifies the number of entries and the color values for each entry, and then the application passes the structure to **CreatePalette**. The function returns a palette handle that the application uses in all subsequent operations to identify the palette. To use colors in the logical palette, the application selects the palette into a device context by using the **SelectPalette** function and then realizes the palette by using the **RealizePalette** function. The colors in the palette are available as soon as the logical palette is realized.

An application should limit the size of its logical palettes to just enough entries to represent the colors needed. Applications cannot create logical palettes larger than the maximum palette size, a device-dependent value. Applications can obtain the maximum size by using the **GetDeviceCaps** function to retrieve the SIZEPALETTE value.

Although an application can specify any color value for a given entry in a logical palette, not all colors can be generated by the given device. The system does not provide a way to discover which colors are supported, but the application can discover the total number of these colors by retrieving the color resolution of the device. The color resolution, specified in color bits per pixel, is equal to the COLORRES value returned by the **GetDeviceCaps** function. A device that has a color resolution of 18 has 262,144 possible colors. If an application requests a color that is not supported, the system chooses an appropriate approximation.

Once a logical palette is created, an application can change colors in the palette by using the **SetPaletteEntries** function. If the logical palette has been selected and realized, changing the palette does not immediately affect the colors being displayed. The application must use the **UnrealizeObject** and **RealizePalette** functions to update the colors. In some cases, the application may need to deselect, unrealize, select, and realize the logical palette to ensure that the colors are updated exactly as requested. If an application selects a logical palette into more than one device context, changes to the logical palette affect all device contexts for which it is selected.

An application can change the number of entries in a logical palette by using the **ResizePalette** function. If the application reduces the size, the remaining entries are unchanged. If the application extends the size, the system sets the color for each new entry to black (0, 0, 0) and the flag to zero.

An application can retrieve the color and flag values for entries in a given logical palette by using the **GetPaletteEntries** function. An application can retrieve the index for the entry in a given logical palette that most closely matches a specified color value by using the **GetNearestPaletteIndex** function.

When an application no longer needs a logical palette, it can delete it by using the **DeleteObject** function. The application must make sure the logical palette is no longer selected into a device context before deleting the palette.

Palette Animation

4/20/2022 • 2 minutes to read • Edit Online

Palette animation is a technique to simulate motion by rapidly changing the colors of selected entries in a color palette. An application can carry out palette animation by creating a logical palette that contains "reserved" entries and then using the **AnimatePalette** function to change colors in those reserved entries.

An application creates a reserved entry in a logical palette by setting the **peFlags** member of the **PALETTEENTRY** structure to the PC_RESERVED flag. Once this logical palette is selected and realized, the application can call the **AnimatePalette** function to change one or more reserved entries. If the given palette is associated with the active window, the system updates the colors on the screen immediately.

System Palette

4/20/2022 • 3 minutes to read • Edit Online

The system maintains a *system palette* for each device that uses palettes. The system palette contains the color values for all colors that can currently be displayed or drawn by the device. Other than viewing the contents of the system palette, applications cannot access the system palette directly. Instead, the system has complete control of the system palette and permits access only through the use of logical palettes.

An application can view the contents of the system palette by using the **GetSystemPaletteEntries** function. This function retrieves the contents of one or more entries, up to the total number of entries in the system palette. The total is always equal to the number returned for the SIZEPALETTE value by the **GetDeviceCaps** function and is the same as the maximum size for any given logical palette.

Although applications cannot change colors in the system palette directly, they may cause changes when realizing logical palettes. To realize a palette, the system examines each requested color and attempts to find an entry in the system palette that contains an exact match. If the system finds a matching color, it maps the logical palette index to the corresponding system palette index. If the system does not find an exact match, it copies the requested color to an unused system palette entry before mapping the indexes. If all system palette entries are in use, the system maps the logical palette index to the system palette entry whose color most closely matches the requested color. After this mapping is set, applications cannot override it. For example, applications cannot use system palette indexes to specify colors; only logical palette indexes are permitted.

Applications can modify the way indexes are mapped by setting the **peFlags** member of the **PALETTEENTRY** structure to selected values when creating the logical palette. For example, the PC_NOCOLLAPSE flag directs the system to immediately copy the requested color to an unused system palette entry regardless of whether a system palette entry already contains that color. Also, the PC_EXPLICIT flag directs the system to map the logical palette index to an explicitly given system palette index. (The application gives the system palette index in the low-order word of the **PALETTEENTRY** structure.)

Palettes can be realized as either a background palette or a foreground palette by specifying **TRUE** or **FALSE** respectively for the *bForceBackground* parameter in the **SelectPalette** function. There can be only one foreground palette in the system at a time. If the window is the currently active window or a descendent of the currently active window, it can realize a foreground palette. Otherwise the palette is realized as a background palette regardless of the value of the *bForceBackground* parameter. The critical property of a foreground palette is that, when realized, it can overwrite all entries (except for the static entries) in the system palette. The system accomplishes this by marking all of the entries that are not static in the system palette as unused before the realization of a foreground palette, thereby eliminating all of the used entries. No preprocessing occurs on the system palette for a background palette realization. The foreground palette sets all of the possible nonstatic colors. Background palettes can set only what remains open and are prioritized in a first-come, first-serve manner. Typically, applications use background palettes for child windows which realize their own individual palettes. This helps minimize the number of changes that occur to the system palette.

An unused system palette entry is any entry that is not reserved and does not contain a static color. Reserved entries are explicitly marked with the PC_RESERVED value. These entries are created when an application realizes a logical palette for palette animation. Static color entries are created by the system and correspond to the colors in the default palette. The **GetDeviceCaps** function can be used to retrieve the NUMRESERVED value, which specifies the number of system palette entries reserved for static colors.

Because the system palette has a limited number of entries, selecting and realizing a logical palette for a given device may affect the colors associated with other logical palettes for the same device. These color changes are especially dramatic when they occur on the display. An application can make sure that reasonable colors are used for its currently selected logical palette by resetting the palette before each use. An application resets the palette by calling the **UnrealizeObject** and **RealizePalette** functions. Using these functions causes the system to remap the colors in the logical palette to reasonable colors in the system palette.

System Palette and Static Colors

4/20/2022 • 2 minutes to read • Edit Online

Ordinarily, the system palette entries that the system reserves for static colors cannot be changed. An application can override this default behavior by using the **SetSystemPaletteUse** function to reduce the number of static color entries and, thereby, increase the number of unused system palette entries. However, because changing the static colors can have an immediate and dramatic effect on all windows on the display, an application should not call **SetSystemPaletteUse**, unless it has a maximized window and the input focus.

When an application calls **SetSystemPaletteUse** with the SYSPAL_NOSTATIC value, the system frees all but two of the reserved entries, allowing those entries to receive new color values when the application subsequently realizes its logical palette. The remaining two static color entries remain reserved and are set to white and black. An application can restore the reserved entries by calling **SetSystemPaletteUse** with the SYSPAL_STATIC value. It can discover the current system palette usage by using the **GetSystemPaletteUse** function.

Furthermore, after setting the system palette usage to SYSPAL_NOSTATIC, the application must immediately realize its logical palette, call the **GetSysColor** function to save the current system color settings, call the **SetSysColors** function to set the system colors to reasonable values using black and white, and finally send the **WM_SYSCOLORCHANGE** message to other top-level windows to allow them to be redrawn with the new system colors. When setting system colors using black and white, the application should make sure adjacent or overlapping items, such as window frames and borders, are set to black and white, respectively.

Before the application loses the input focus, closes its window, or terminates, it must immediately call **SetSystemPaletteUse** with the SYSPAL_STATIC value, realize its logical palette, restore the system colors to their previous values, and send the **WM_SYSCOLORCHANGE** message. The system sends a **WM_PAINT** message to any window that is affected by a system color change. Applications that have brushes using the existing system colors should delete those brushes and re-create them using the new system colors.

Palette Messages

4/20/2022 • 2 minutes to read • Edit Online

Changes to the system palette for the display device can have dramatic and sometimes undesirable effects on the colors used in windows on the desktop. To minimize the impact of these changes, the system provides a set of messages that help applications manage their logical palettes while ensuring that colors in the active window are as close as possible to the colors intended.

The system sends a WM_QUERYNEWPALETTE message to a top-level or overlapped window just before activating the window. This message gives an application the opportunity to select and realize its logical palette so that it receives the best possible mapping of colors for its logical palette. When the application receives the message, it should use the SelectPalette, UnrealizeObject, and RealizePalette functions to select and realize the logical palette. Doing so directs the system to update colors in the system palette so that its colors match as many colors in the logical palette as possible.

When an application causes changes to the system palette as a result of realizing its logical palette, the system sends a WM_PALETTECHANGED message to all top-level and overlapped windows. This message gives applications the opportunity to update the colors in the client areas of their windows, replacing colors that have changed with colors that more closely match the intended colors. An application that receives the **WM_PALETTECHANGED** message should use UnrealizeObject and RealizePalette to reset the logical palettes associated with all inactive windows and then update the colors in the client area for each inactive window by using the UpdateColors function. This technique does not guarantee the greatest number of exact color matches; however, it does ensure that colors in the logical palette are mapped to reasonable colors in the system palette.

NOTE

To avoid creating an infinite loop, an application should never realize the palette for the window whose handle matches the handle passed in the *wParam* parameter of the WM_PALETTECHANGED message.

The UpdateColors function typically updates a client area of an inactive window faster than redrawing the area. However, because **UpdateColors** performs color translation based on the color of each pixel before the system palette changed, each call to this function results in the loss of some color accuracy. This means **UpdateColors** cannot be used to update colors when the window becomes active. In such cases, the application should redraw the client area.

The system may send the WM_QUERYNEWPALETTE message when changes to the logical palette are made. Also, the system may send the WM_PALETTEISCHANGING message to all top-level and overlapped windows when the system palette is about to change.

Halftone Palette and Color Adjustment

4/20/2022 • 2 minutes to read • Edit Online

Halftone palettes are intended to be used whenever the stretching mode of a device context is set to HALFTONE. An application creates a halftone palette by using the **CreateHalftonePalette** function. The application must select and realize this palette into the device context before calling the **StretchBlt** or **StretchDIBits** function.

The system automatically adjusts the input color of source bitmaps whenever applications call the **StretchBlt** and **StretchDIBits** functions and the stretching mode of a device context is set to HALFTONE. These color adjustments affect certain attributes of the image, such as contrast and brightness. An application can set the color adjustment values by using the **SetColorAdjustment** function. The application can retrieve the color adjustment values for the specified device context by using the **GetColorAdjustment** function.

Using Color

4/20/2022 • 2 minutes to read • Edit Online

Enumerating Colors

4/20/2022 • 2 minutes to read • Edit Online

You can determine how many colors a device supports and what those colors are by retrieving the count of colors for the device and enumerating the colors of the solid pens. To retrieve the number of colors, use the **GetDeviceCaps** function with the NUMCOLORS value. To enumerate solid pens, use the **EnumObjects** function and a corresponding callback function that receives information about each pen.

```
// GetTheColors - returns the count and color values of solid colors
// Returns a pointer to the array containing colors
// hdc - handle to device context

COLORREF *GetTheColors(HDC hdc)
{
    int cColors;
    COLORREF *aColors;

    // Get the number of colors.
    cColors = GetDeviceCaps(hdc, NUMCOLORS);

    // Allocate space for the array.
    aColors = (COLORREF *)LocalAlloc(LPTR, sizeof(COLORREF) *
        (cColors+1));

    // Save the count of colors in first element.
    aColors[0] = (LONG)cColors;

    // Enumerate all pens and save solid colors in the array.
    EnumObjects(hdc, OBJ_PEN, (GOBJENUMPROC)MyEnumProc, (LONG)aColors);

    // Refresh the count of colors.
    aColors[0] = (LONG)cColors;

    return aColors;
}

int MyEnumProc(LPVOID lp, LPBYTE lpb)
{
    LPLOGPEN lopn;
    COLORREF *aColors;
    int iColor;

    lopn = (LPLOGPEN)lp;
    aColors = (COLORREF *)lpb;

    if (lopn->lopnStyle==PS_SOLID)
    {
        // Check for too many colors.
        if ((iColor = (int)aColors[0]) <= 0)
             return 0;
        aColors[iColor] = lopn->lopnColor;
        aColors[0]--;
    }
    return 1;
}
```

Creating Colored Pens and Brushes

4/20/2022 • 2 minutes to read • Edit Online

Although you can specify any color for a pen when creating it, the system uses only colors that are available on the device. This means the system uses the closest matching color when it realizes the pen for drawing. When creating brushes, the system generates a dithered color if you specify a color that the device does not support. In either case, you can use the **RGB** macro to specify a color when creating a pen or brush.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;
    RECT clientRect;
    RECT textRect;
    HRGN bgRgn;
    HBRUSH hBrush;
    HPEN hPen;


    switch (message)
    {

    case WM_PAINT:
        {
        hdc = BeginPaint(hWnd, &ps);

        // Fill the client area with a brush
        GetClientRect(hWnd, &clientRect);
        bgRgn = CreateRectRgnIndirect(&clientRect);
        hBrush = CreateSolidBrush(RGB(200,200,200));
        FillRgn(hdc, bgRgn, hBrush);


        hPen = CreatePen(PS_DOT,1,RGB(0,255,0));
        SelectObject(hdc, hPen);
        SetBkColor(hdc, RGB(0,0,0));
        Rectangle(hdc, 10,10,200,200);

        // Text caption
        SetBkColor(hdc, RGB(255,255,255));
        SetRect(&textRect, 10, 210, 200,200);
        DrawText(hdc,TEXT("PS_DOT"),-1,&textRect, DT_CENTER | DT_NOCLIP);


        hPen = CreatePen(PS_DASHDOTDOT,1,RGB(0,255,255));
        SelectObject(hdc, hPen);
        SetBkColor(hdc, RGB(255,0,0));
        SelectObject(hdc,CreateSolidBrush(RGB(0,0,0)));
        Rectangle(hdc, 210,10,400,200);

        // Text caption
        SetBkColor(hdc, RGB(255,200,200));
        SetRect(&textRect, 210, 210, 400,200);
        DrawText(hdc,TEXT("PS_DASHDOTDOT"),-1,&textRect, DT_CENTER | DT_NOCLIP);


        hPen = CreatePen(PS_DASHDOT,1,RGB(255,0,0));
        SelectObject(hdc, hPen);
        SetBkColor(hdc, RGB(255,255,0));
        SelectObject(hdc,CreateSolidBrush(RGB(100,200,255)));
        Rectangle(hdc, 410,10,600,200);

        // Text caption
        SetBkColor(hdc, RGB(200,255,200));
        SetRect(&textRect, 410, 210, 600,200);
        DrawText(hdc,TEXT("PS_DASHDOT"),-1,&textRect, DT_CENTER | DT_NOCLIP);


        // When fnPenStyle is PS_SOLID, nWidth may be more than 1.
        // Also, if you set the width of any pen to be greater than 1,
        // then it will draw a solid line, even if you try to select another style.
        hPen = CreatePen(PS_SOLID,5,RGB(255,0,0));
        SelectObject(hdc, hPen);
        // Setting the background color doesn't matter
        // when the style is PS_SOLID
        SetBkColor(hdc, RGB(255,255,255));
        SelectObject(hdc,CreateSolidBrush(RGB(200,100,50)));
        Rectangle(hdc, 10,300,200,500);

        // Text caption
        SetBkColor(hdc, RGB(200,200,255));
        SetRect(&textRect, 10, 510, 200,500);
        DrawText(hdc,TEXT("PS_SOLID"),-1,&textRect, DT_CENTER | DT_NOCLIP);

        hPen = CreatePen(PS_DASH,1,RGB(0,255,0));
        SelectObject(hdc, hPen);
        SetBkColor(hdc, RGB(0,0,0));
        SelectObject(hdc,CreateSolidBrush(RGB(200,200,255)));
        Rectangle(hdc, 210,300,400,500);

        // Text caption
        SetBkColor(hdc, RGB(255,255,200));
        SetRect(&textRect, 210, 510, 400,200);
        DrawText(hdc,TEXT("PS_DASH"),-1,&textRect, DT_CENTER | DT_NOCLIP);

        hPen = CreatePen(PS_NULL,1,RGB(0,255,0));
        SelectObject(hdc, hPen);
        // Setting the background color doesn't matter
        // when the style is PS_NULL
        SetBkColor(hdc, RGB(0,0,0));
        SelectObject(hdc,CreateSolidBrush(RGB(255,255,255)));
        Rectangle(hdc, 410,300,600,500);

        // Text caption
        SetBkColor(hdc, RGB(200,255,255));
        SetRect(&textRect, 410, 510, 600,500);
        DrawText(hdc,TEXT("PS_NULL"),-1,&textRect, DT_CENTER | DT_NOCLIP);


        // Clean up
        DeleteObject(bgRgn);
        DeleteObject(hBrush);
        DeleteObject(hPen);

        GetStockObject(WHITE_BRUSH);
        GetStockObject(DC_PEN);

        EndPaint(hWnd, &ps);
        break;
        }


    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
```

```
  }
```

Color Reference (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

The following elements are associated with color.

- Color Functions
- Color Structures
- Color Macros
- Color Messages

Color Functions

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used with color.

FUNCTION	DESCRIPTION
AnimatePalette	Replaces entries in the specified logical palette.
CreateHalftonePalette	Creates a halftone palette for the specified device context (DC).
CreatePalette	Creates a logical palette.
GetColorAdjustment	Retrieves the color adjustment values for the specified DC.
GetNearestColor	Retrieves a color value identifying a color from the system palette that will be displayed when the specified color value is used.
GetNearestPaletteIndex	Retrieves the index for the entry in the specified logical palette most closely matching a specified color value.
GetPaletteEntries	Retrieves a specified range of palette entries from the given logical palette.
GetSystemPaletteEntries	Retrieves a range of palette entries from the system palette that is associated with the specified DC.
GetSystemPaletteUse	Retrieves the current state of the system (physical) palette for the specified DC.
RealizePalette	Maps palette entries from the current logical palette to the system palette.
ResizePalette	Increases or decreases the size of a logical palette based on the specified value.
SelectPalette	Selects the specified logical palette into a device context.
SetColorAdjustment	Sets the color adjustment values for a DC using the specified values.
SetPaletteEntries	Sets RGB (red, green, blue) color values and flags in a range of entries in a logical palette.
SetSystemPaletteUse	Allows an application to specify whether the system palette contains 2 or 20 static colors.
UnrealizeObject	Resets the origin of a brush or resets a logical palette.

FUNCTION	DESCRIPTION
UpdateColors	Updates the client area of the specified device context by remapping the current colors in the client area to the currently realized logical palette.

Color Structures

4/20/2022 • 2 minutes to read • Edit Online

The following structures are used with color:

- COLORREF
- LOGPALETTE
- PALETTEENTRY

COLORREF

4/20/2022 • 2 minutes to read • Edit Online

The COLORREF value is used to specify an RGB color.

```
typedef DWORD COLORREF;
typedef DWORD* LPCOLORREF;
```

Remarks

When specifying an explicit RGB color, the **COLORREF** value has the following hexadecimal form:

`0x00bbggrr`

The low-order byte contains a value for the relative intensity of red; the second byte contains a value for green; and the third byte contains a value for blue. The high-order byte must be zero. The maximum value for a single byte is 0xFF.

To create a **COLORREF** color value, use the RGB macro. To extract the individual values for the red, green, and blue components of a color value, use the **GetRValue**, GetGValue, and GetBValue macros, respectively.

Example

```
// Color constants.
const COLORREF rgbRed   =  0x000000FF;
const COLORREF rgbGreen =  0x0000FF00;
const COLORREF rgbBlue  =  0x00FF0000;
const COLORREF rgbBlack =  0x00000000;
const COLORREF rgbWhite =  0x00FFFFFF;
```

Example from Windows Classic Samples on GitHub.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Windef.h (include Windows.h)

See also

Colors Overview

Color Structures

GetBValue

GetGValue

GetRValue

RGB

Color Macros

4/20/2022 • 2 minutes to read • Edit Online

The following macros are used with color:

- DIBINDEX
- GetBValue
- GetGValue
- GetRValue
- PALETTEINDEX
- PALETTERGB
- RGB

Color Messages

4/20/2022 • 2 minutes to read • Edit Online

The following messages are used with color:

- WM_PALETTECHANGED
- WM_PALETTEISCHANGING
- WM_QUERYNEWPALETTE
- WM_SYSCOLORCHANGE

WM_PALETTECHANGED message

4/20/2022 • 2 minutes to read • Edit Online

The **WM_PALETTECHANGED** message is sent to all top-level and overlapped windows after the window with the keyboard focus has realized its logical palette, thereby changing the system palette. This message enables a window that uses a color palette but does not have the keyboard focus to realize its logical palette and update its client area.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,
  UINT  uMsg,
  WPARAM wParam,
  LPARAM lParam
);
```

Parameters

wParam

A handle to the window that caused the system palette to change.

lParam

This parameter is not used.

Remarks

This message must be sent to all top-level and overlapped windows, including the one that changed the system palette. If any child windows use a color palette, this message must be passed on to them as well.

To avoid creating an infinite loop, a window that receives this message must not realize its palette, unless it determines that *wParam* does not contain its own window handle.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

See also

Colors Overview

Color Messages

WM_PALETTEISCHANGING

WM_QUERYNEWPALETTE

WM_PALETTEISCHANGING message

4/20/2022 • 2 minutes to read • Edit Online

The **WM_PALETTEISCHANGING** message informs applications that an application is going to realize its logical palette.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,
  UINT  uMsg,
  WPARAM wParam,
  LPARAM lParam
);
```

Parameters

wParam

A handle to the window that is going to realize its logical palette.

lParam

This parameter is not used.

Return value

If an application processes this message, it should return zero.

Remarks

The application changing its palette does not wait for acknowledgment of this message before changing the palette and sending the **WM_PALETTECHANGED** message. As a result, the palette may already be changed by the time an application receives this message.

If the application either ignores or fails to process this message and a second application realizes its palette while the first is using palette indexes, there is a strong possibility that the user will see unexpected colors during subsequent drawing operations.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

See also

Colors Overview

Color Messages

WM_PALETTECHANGED

WM_QUERYNEWPALETTE

WM_QUERYNEWPALETTE message

4/20/2022 • 2 minutes to read • Edit Online

The **WM_QUERYNEWPALETTE** message informs a window that it is about to receive the keyboard focus, giving the window the opportunity to realize its logical palette when it receives the focus.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,
  UINT  uMsg,
  WPARAM wParam,
  LPARAM lParam
);
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

If the window realizes its logical palette, it must return **TRUE**; otherwise, it must return **FALSE**.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

See also

Colors Overview

Color Messages

WM_PALETTECHANGED

WM_PALETTEISCHANGING

WM_SYSCOLORCHANGE message

4/20/2022 • 2 minutes to read • Edit Online

The **WM_SYSCOLORCHANGE** message is sent to all top-level windows when a change is made to a system color setting.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,
  UINT  uMsg,
  WPARAM wParam,
  LPARAM lParam
);
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Remarks

The system sends a **WM_PAINT** message to any window that is affected by a system color change.

Applications that have brushes using the existing system colors should delete those brushes and re-create them using the new system colors.

Top level windows that use common controls must forward the **WM_SYSCOLORCHANGE** message to the controls; otherwise, the controls will not be notified of the color change. This ensures that the colors used by your common controls are consistent with those used by other user interface objects. For example, a toolbar control uses the "3D Objects" color to draw its buttons. If the user changes the 3D Objects color but the **WM_SYSCOLORCHANGE** message is not forwarded to the toolbar, the toolbar buttons will remain in their original color while the color of other buttons in the system changes.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

See also

Colors Overview

Color Messages

WM_PAINT

Coordinate Spaces and Transformations

4/20/2022 • 2 minutes to read • Edit Online

Applications use coordinate spaces and transformations to scale, rotate, translate, shear, and reflect graphics output. A *coordinate space* is a planar space that locates two-dimensional objects by using two reference axes that are perpendicular to each other. There are four coordinate spaces: world, page, device, and physical device (client area, desktop, or page of printer paper).

A *transformation* is an algorithm that alters ("transforms") the size, orientation, and shape of objects. Transformations also transfer a graphics object from one coordinate space to another. Ultimately, the object appears on the physical device, which is usually a screen or printer.

This chapter is divided into three topics:

- About Coordinate Spaces and Transformations
- Using Coordinate Spaces and Transformations
- Coordinate Space and Transformation Reference

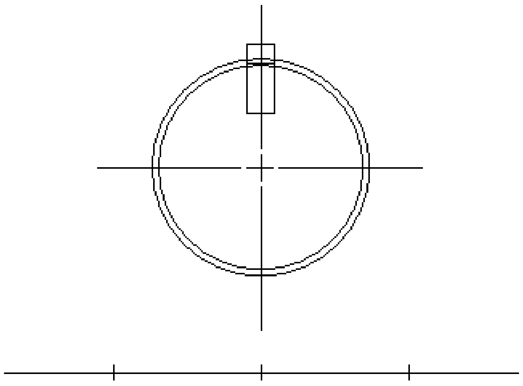
About Coordinate Spaces and Transformations

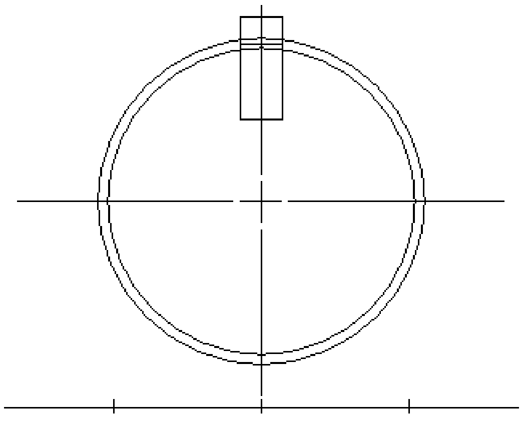
4/20/2022 • 2 minutes to read • Edit Online

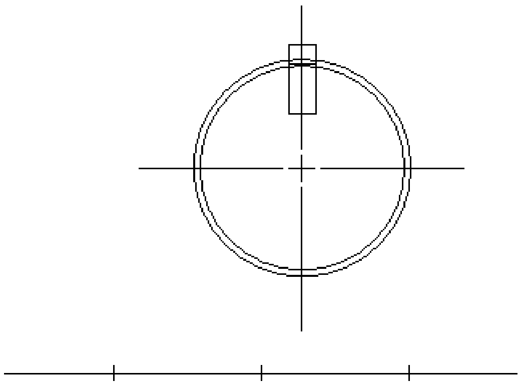
Coordinate spaces and transformations are used by the following types of applications:

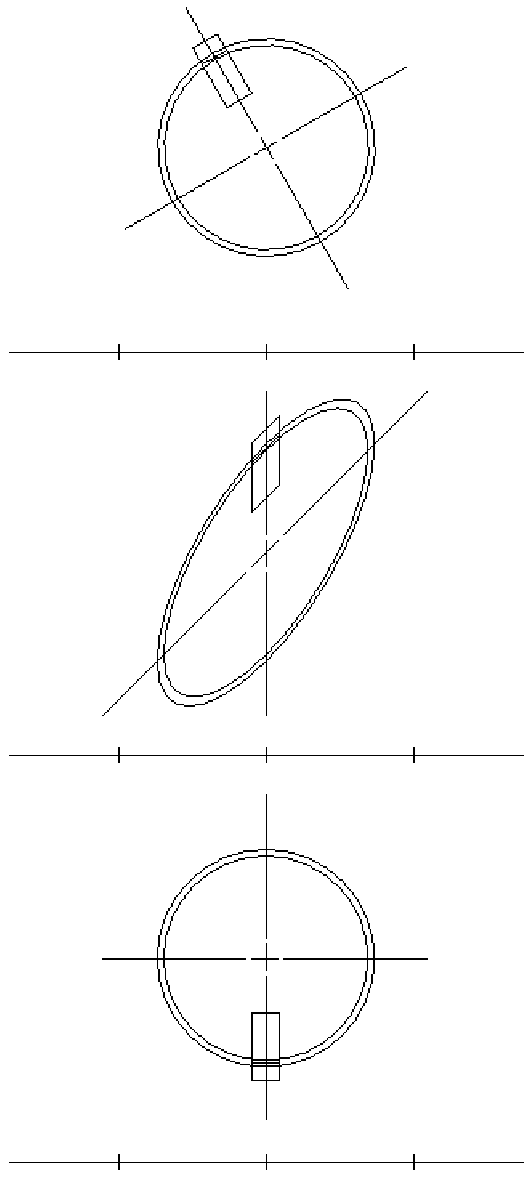
- Desktop publishing applications (to "zoom" parts of a page or to display adjacent pages in a window).
- Computer-aided design (CAD) applications (to rotate objects, scale drawings, or create perspective views).
- Spreadsheet applications (to move and size graphs).

The following illustrations show successive views of an object created in a drawing application. The first illustration shows the object as it appears in the original drawing; the succeeding five illustrations show the effects of applying various transformations.









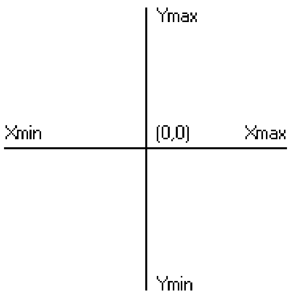
The following subtopics describe various aspects of coordinate spaces and transformations.

- Transformation of Coordinate Spaces
- World-Space to Page-Space Transformations
- Page-Space to Device-Space Transformations
- Device-Space to Physical-Device Transformation
- Default Transformations

Transformation of Coordinate Spaces

4/20/2022 • 2 minutes to read • Edit Online

A *coordinate space* is a planar space based on the Cartesian coordinate system. This system provides a means of specifying the location of each point on a plane. It requires two axes that are perpendicular and equal in length. The following illustration shows a coordinate space.



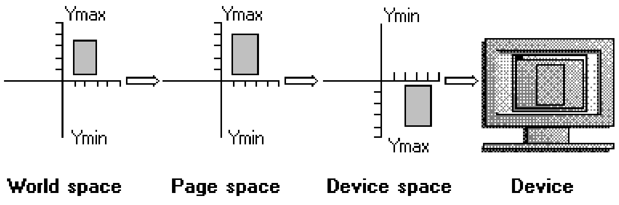
The system supports four coordinate spaces, as described in the following table.

COORDINATE SPACE	DESCRIPTION
world	Used optionally as the starting coordinate space for graphics transformations. It allows scaling, translation, rotation, shearing, and reflection. World space measures 2^32 units high by 2^32 units wide.
page	Used either as the next space after world space or as the starting space for graphics transformations. It sets the mapping mode. Page space also measures 2^32 units high by 2^32 units wide.
device	Used as the next space after page space. It only allows translation, which ensures the origin of the device space maps to the proper location in physical device space. Device space measures 2^27 units high by 2^27 units wide.
physical device	The final (output) space for graphics transformations. It usually refers to the client area of the application window; however, it can also include the entire desktop, a complete window (including the frame, title bar, and menu bar), or a page of printer or plotter paper, depending on the function that obtained the handle to the device context. Physical device dimensions vary according to the dimensions set by the display, printer, or plotter technology.

Page space works with device space to provide applications with device-independent units, such as millimeters and inches. This overview refers to both world space and page space as logical space.

To depict output on a physical device, the system copies (or maps) a rectangular region from one coordinate space into the next using a transformation until the output appears in its entirety on the physical device. Mapping begins in the application's world space if the application has called the SetWorldTransform function; otherwise, mapping occurs in page space. As the system copies each point within the rectangular region from one space into another, it applies an algorithm called a transformation. A *transformation* alters (or transforms) the size, orientation, and shape of objects that are copied from one coordinate space into another. Although a transformation affects an object as a whole, it is applied to each point, or to each line, in the object.

The following illustration shows a typical transformation performed by using the SetWorldTransform function.



World-Space to Page-Space Transformations

4/20/2022 • 2 minutes to read • Edit Online

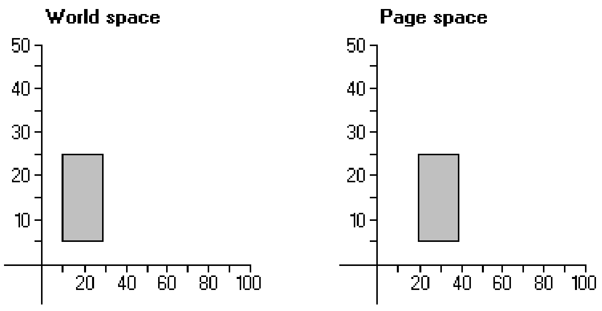
World-space to page-space transformations support translation and scaling. In addition, they support rotation, shear, and reflection capabilities. The following sections describe these transformations, illustrate their effects, and provide the algorithms used to achieve them.

Translation

4/20/2022 • 2 minutes to read • Edit Online

Some applications translate (or shift) objects drawn in the client area. by calling the **SetWorldTransform** function to set the appropriate world-space to page-space transformation. The SetWorldTransform function receives a pointer to an **XFORM** structure containing the appropriate values. The eDx and eDy members of XFORM specify the horizontal and vertical translation components, respectively.

When *translation* occurs, each point in an object is shifted vertically, horizontally, or both, by a specified amount. The following illustration shows a 20- by 20-unit rectangle that was translated to the right by 10 units when copied from world-coordinate space to page-coordinate space.



In the preceding illustration, the x-coordinate of each point in the rectangle is 10 units greater than the original x-coordinate.

Horizontal translation can be represented by the following algorithm.

```
x' = x + Dx
```

Where x' is the new x-coordinate, x is the original x-coordinate, and Dx is the horizontal distance moved.

Vertical translation can be represented by the following algorithm.

```
y' = y + Dy
```

Where y' is the new y-coordinate, y is the original y-coordinate, and Dy is the vertical distance moved.

The horizontal and vertical translation transformations can be combined into a single operation by using a 3-by-3 matrix.

```
                      |1   0   0|
|x' y' 1| = |x y 1| * |0   1   0|
                      |Dx  Dy  1|
```

(The rules of matrix multiplication state that the number of rows in one matrix must equal the number of columns in the other. The integer 1 in the matrix |x y 1| is a placeholder that was added to meet this requirement.)

The 3-by-3 matrix that produced the illustrated translation transformation contains the following values.

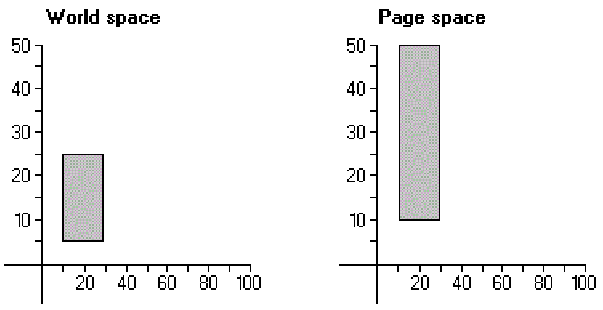
```
|1  0  0|
|0  1  0|
|10 0  1|
```

Scaling

4/20/2022 • 2 minutes to read • Edit Online

Most CAD and drawing applications provide features that scale output created by the user. Applications that include scaling (or zoom) capabilities call the SetWorldTransform function to set the appropriate world-space to page-space transformation. This function receives a pointer to an XFORM structure containing the appropriate values. The eM11 and eM22 members of XFORM specify the horizontal and vertical scaling components, respectively.

When *scaling* occurs, the vertical and horizontal lines (or vectors), that constitute an object, are stretched or compressed with respect to the x- or y-axis. The following illustration shows a 20-by-20-unit rectangle scaled vertically to twice its original height when copied from world-coordinate space to page-coordinate space.



In the preceding illustration, the vertical lines that define the original rectangle's side measure 20 units, while the vertical lines that define the scaled rectangle's sides measure 40 units.

Vertical scaling can be represented by the following algorithm.

```
y' = y * Dy
```

Where y' is the new length, y is the original length, and Dy is the vertical scaling factor.

Horizontal scaling can be represented by the following algorithm.

```
x' = x * Dx
```

Where x' is the new length, x is the original length, and Dx is the horizontal scaling factor.

The vertical and horizontal scaling transformations can be combined into a single operation by using a 2-by-2 matrix.

```
|x' y'|  =  |Dx   0|  *  |x y|
            |0   Dy|
```

The 2-by-2 matrix that produced the scaling transformation contains the following values.

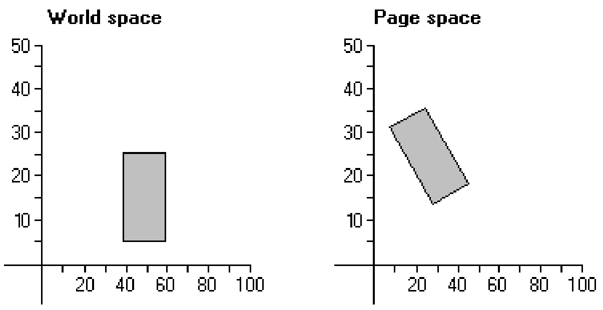
```
|1   0|
|0   2|
```

Rotation

4/20/2022 • 2 minutes to read • Edit Online

Many CAD applications provide features that rotate objects drawn in the client area. Applications that include rotation capabilities use the SetWorldTransform function to set the appropriate world-space to page-space transformation. This function receives a pointer to an XFORM structure containing the appropriate values. The eM11, eM12, eM21, and eM22 members of XFORM specify respectively, the cosine, sine, negative sine, and cosine of the angle of rotation.

When *rotation* occurs, the points that constitute an object are rotated with respect to the coordinate-space origin. The following illustration shows a 20-by-20-unit rectangle rotated 30 degrees when copied from world-coordinate space to page-coordinate space.



In the preceding illustration, each point in the rectangle was rotated 30 degrees with respect to the coordinate-space origin.

The following algorithm computes the new x-coordinate (x ') for a point (x,y) that is rotated by angle A with respect to the coordinate-space origin.

```
x' = (x * cos A) - (y * sin A)
```

The following algorithm computes the y-coordinate (y ') for a point (x,y) that is rotated by the angle A with respect to the origin.

```
y' = (x * sin A) + (y * cos A)
```

The two rotation transformations can be combined in a 2-by-2 matrix as follows.

```
|x' y'| == |x y| * | cos A   sin A|
                   |-sin A   cos A|
```

The 2-by-2 matrix that produced the rotation contains the following values.

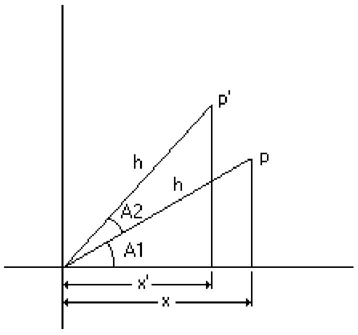
```
| .8660    .5000|
|-.5000    .8660|
```

Rotation Algorithm Derivation

Rotation algorithms are based on trigonometry's addition theorem stating that the trigonometric function of a sum of two angles (A1 and A2) can be expressed in terms of the trigonometric functions of the two angles.

```
sin(A1 + A2) = (sin A1 * cos A2) + (cos A1 * sin A2)
cos(A1 + A2) = (cos A1 * cos A2) - (sin A1 * sin A2)
```

The following illustration shows a point p rotated counterclockwise to a new position p'. In addition, it shows two triangles formed by a line drawn from the coordinate-space origin to each point and a line drawn from each point through the x-axis.



Using trigonometry, the x-coordinate of point p can be obtained by multiplying the length of the hypotenuse h by the cosine of A1.

```
x = h * cos A1
```

The y-coordinate of point p can be obtained by multiplying the length of the hypotenuse h by the sine of A1.

```
y = h * sin A1
```

Likewise, the x-coordinate of point p' can be obtained by multiplying the length of the hypotenuse h by the cosine of (A1 +A2).

```
x' = h * cos (A1 + A2)
```

Finally, the y-coordinate of point p' can be obtained by multiplying the length of the hypotenuse h by the sine of (A1 +A2).

```
y' = h * sin (A1 + A2)
```

Using the addition theorem, the preceding algorithms become the following:

```
x' = (h * cos A1 * cos A2) - (h * sin A1 * sin A2)
y' = (h * cos A1 * sin A2) + (h * sin A1 * cos A2)
```

The rotation algorithms for a given point rotated by angle A2 can be obtained by substituting x for each occurrence of (h * cos A1) and by substituting y for each occurrence of (h * sin A1).

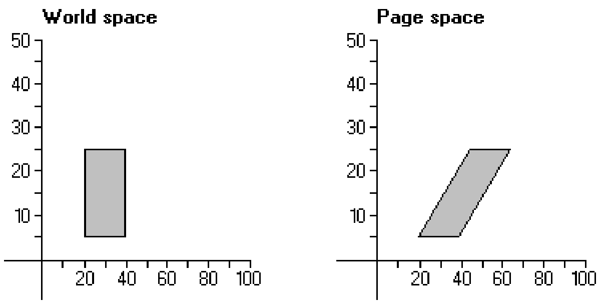
```
x' = (x * cos A2) - (y * sin A2)
y' = (x * sin A2) + (y * cos A2)
```

Shear

4/20/2022 • 2 minutes to read • Edit Online

Some applications provide features that shear objects drawn in the client area. Applications that use shear capabilities use the SetWorldTransform function to set appropriate values in the world-space to page-space transformation. This function receives a pointer to an XFORM structure containing the appropriate values. The eM12 and eM21 members of XFORM specify the horizontal and vertical proportionality constants, respectively.

There are two components of the *shear transformation*. The first alters the vertical lines in an object; the second alters the horizontal lines. The following illustration shows a 20-by-20-unit rectangle sheared horizontally when copied from world space to page space.



A horizontal shear can be represented by the following algorithm:

```
x' = x + (Sx * y)
```

where x is the original x-coordinate, Sx is the proportionality constant, and x' is the result of the shear transformation.

A vertical shear can be represented by the following algorithm:

```
y' = y + (Sy * x)
```

where y is the original y-coordinate, Sy is the proportionality constant, and y' is the result of the shear transformation.

The horizontal-shear and vertical-shear transformations can be combined into a single operation using a 2-by-2 matrix.

```
|x' y'| == |x y| * |  1   Sx|
                   | Sy    1|
```

The 2-by-2 matrix that produced the shear contains the following values:

```
|1    1|
|0    1|
```

Reflection

4/20/2022 • 2 minutes to read • Edit Online

Some applications provide features that reflect (or mirror) objects drawn in the client area. Applications that contain reflection capabilities use the **SetWorldTransform** function to set the appropriate values in the world-space to page-space transformation. This function receives a pointer to an **XFORM** structure containing the appropriate values. The eM11 and eM22 members of XFORM specify the horizontal and vertical reflection components, respectively.

The *reflection transformation* creates a mirror image of an object with respect to either the x- or y-axis. In short, reflection is just negative scaling. To produce a horizontal reflection, x-coordinates are multiplied by 1. To produce a vertical reflection, y-coordinates are multiplied by 1.

Horizontal reflection can be represented by the following algorithm:

```
x' = -x
```

where x is the x-coordinate and x' is the result of the reflection.

The 2-by-2 matrix that produced horizontal reflection contains the following values:

```
|-1    0|
|0     1|
```

Vertical reflection can be represented by the following algorithm:

```
y' = -y
```

where y is the y-coordinate and y' is the result of the reflection.

The 2-by-2 matrix that produced vertical reflection contains the following values:

```
|1    0|
|0   -1|
```

The horizontal-reflection and vertical-reflection operations can be combined into a single operation by using the following 2-by-2 matrix:

```
|-1    0|
|0    -1|
```

Combined World-to-Page Space Transformations

4/20/2022 • 2 minutes to read • Edit Online

The five world-to-page transformations can be combined into a single 3-by-3 matrix. The **CombineTransform** function can be used to combine two world-space to page-space transformations. The combined transformations can be used to alter output associated with a particular device context (DC) by calling the **SetWorldTransform** function and supplying the elements for this matrix. When an application calls **SetWorldTransform**, it stores the elements of the 3-by-3 matrix in an **XFORM** structure. The members of this structure correspond to the first two columns of a 3-by-3 matrix; the last column of the matrix is not required because its values are constant.

The elements of the current world transformation matrix can be revived by calling the **GetWorldTransform** function and supplying a pointer to an **XFORM** structure.

Page-Space to Device-Space Transformations

4/20/2022 • 2 minutes to read • Edit Online

The page-space to device-space transformation determines the mapping mode for all graphics output associated with a particular DC. A *mapping mode* is a scaling transformation that specifies the size of the units used for drawing operations. The mapping mode may also perform translation. In some cases, the mapping mode alters the orientation of the x-axis and y-axis in device space. The following topics describe the mapping modes.

- Mapping Modes and Translations
- Predefined Mapping Modes
- Application-Defined Mapping Modes

Mapping Modes and Translations

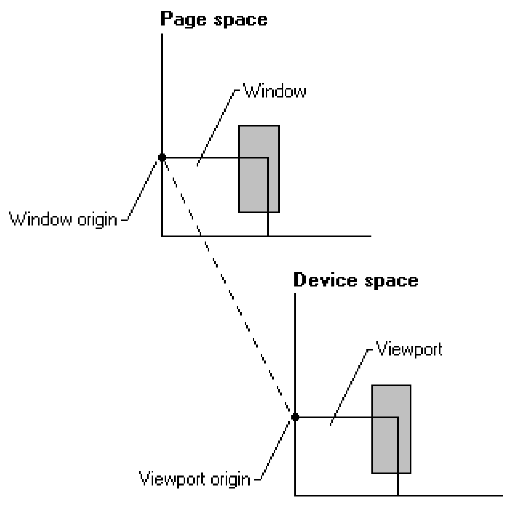
4/20/2022 • 4 minutes to read • Edit Online

The mapping modes are described in the following table.

MAPPING MODE	DESCRIPTION
MM_ANISOTROPIC	Each unit in page space is mapped to an application-specified unit in device space. The axis may or may not be equally scaled (for example, a circle drawn in world space may appear to be an ellipse when depicted on a given device). The orientation of the axis is also specified by the application.
MM_HIENGLISH	Each unit in page space is mapped to 0.001 inch in device space. The value of x increases from left to right. The value of y increases from bottom to top.
MM_HIMETRIC	Each unit in page space is mapped to 0.01 millimeter in device space. The value of x increases from left to right. The value of y increases from bottom to top.
MM_ISOTROPIC	Each unit in page space is mapped to an application-defined unit in device space. The axes are always equally scaled. The orientation of the axes may be specified by the application.
MM_LOENGLISH	Each unit in page space is mapped to 0.01 inch in device space. The value of x increases from left to right. The value of y increases from bottom to top.
MM_LOMETRIC	Each unit in page space is mapped to 0.1 millimeter in device space. The value of x increases from left to right. The value of y increases from bottom to top.
MM_TEXT	Each unit in page space is mapped to one pixel; that is, no scaling is performed at all. When no translation is in effect (this is the default), page space in the MM_TEXT mapping mode is equivalent to physical device space. The value of x increases from left to right. The value of y increases from top to bottom.
MM_TWIPS	Each unit in page space is mapped to one twentieth of a printer's point (1/1440 inch). The value of x increases from left to right. The value of y increases from bottom to top.

To set a mapping mode, call the **SetMapMode** function. Retrieve the current mapping mode for a DC by calling the **GetMapMode** function.

The page-space to device-space transformations consist of values calculated from the points given by the window and viewport. In this context, the window refers to the logical coordinate system of the page space, while the viewport refers to the device coordinate system of the device space. The window and viewport each consist of an origin, a horizontal ("x") extent, and a vertical ("y") extent). The window parameters are in logical coordinates; the viewport in device coordinates (pixels). The system combines the origins and extents from both the window and viewport to create the transformation. This means that the window and viewport each specify half of the factors needed to define the transformation used to map points in page space to device space. Thus, the system maps the window origin to the viewport origin and the window extents to the viewport extents, as shown in the following illustration.



The window and viewport extents establish a ratio or scaling factor used in the page-space to device-space transformations. For the six predefined mapping modes (MM_HIENGLISH, MM_LOENGLISH, MM_HIMETRIC, MM_LOMETRIC, MM_TEXT, and MM_TWIPS), the extents are set by the system when SetMapMode is called. They cannot be changed. The other two mapping modes (MM_ISOTROPIC and MM_ANISOTROPIC) require that the extents are specified. This is done by calling **SetMapMode** to set the appropriate mode and then calling the SetWindowExtEx and SetViewportExtEx functions to specify the extents. In the MM_ISOTROPIC mapping mode, it is important to call **SetWindowExtEx** before calling **SetViewportExtEx**.

The window and viewport origins establish the translation used in the page-space to device-space transformations. Set the window and viewport origins by using the SetWindowOrgEx and SetViewportOrgEx functions. The origins are independent of the extents, and an application can set them regardless of the current mapping mode. Changing a mapping mode does not affect the currently set origins (although it can affect the extents). Origins are specified in absolute units that the current mapping mode does not affect. To alter the origins, use the OffsetWindowOrgEx and OffsetViewportOrgEx functions.

The following formula shows the math involved in converting a point from page space to device space.

```
Dx = ((Lx - WOx) * VEx / WEx) + VOx
```

The following variables are involved.

```
Dx     x value in device units
Lx     x value in logical units (also known as page space units)
WOx     window x origin
VOx     viewport x origin
WEx     window x-extent
VEx     viewport x-extent
```

The same equation with y replacing x transforms the y componentof a point.

The formula first offsets the point from its coordinate origin. This value, no longer biased by the origin, is then scaled into the destination coordinate system by the ratio of the extents. Finally, the scaled value is offset by the destination origin to its final mapping.

The LPtoDP and DPtoLP functions may be used to convert from logical points to device points and from device points to logical points, respectively.

Predefined Mapping Modes

4/20/2022 • 2 minutes to read • Edit Online

Of the six predefined mapping modes, one is device dependent (MM_TEXT) and the remaining five (MM_HIENGLISH, MM_LOENGLISH, MM_HIMETRIC, MM_LOMETRIC, and MM_TWIPS) are device independent.

The default mapping mode is MM_TEXT. One logical unit equals one pixel. Positive x is to the right, and positive y is down. This mode maps directly to the device's coordinate system. The logical-to-physical mapping involves only an offset in x and y that is defined by the application-controlled window and viewport origins. The viewport and window extents are all set to 1, creating a one-to-one mapping.

Applications that display geometric shapes (circles, squares, polygons, and so on) make use of one of the device-independent mapping modes. For example, if you are writing an application to provide charting capabilities for a spreadsheet program and want to guarantee that the diameter of each pie chart is 2 inches, use the MM_LOENGLISH mapping mode and call the appropriate functions to draw and fill the chart. Specifying MM_LOENGLISH, guarantees that the diameter of the chart is consistent on any display or printer. If MM_TEXT is used instead of MM_LOENGLISH, a chart that appears circular on a VGA display would appear elliptical on an EGA display and would appear very small on a 300 dpi (dots per inch) laser printer.

Application-Defined Mapping Modes

4/20/2022 • 2 minutes to read • Edit Online

The two application-defined mapping modes (MM_ISOTROPIC and MM_ANISOTROPIC) are provided for application-specific mapping modes. The MM_ISOTROPIC mode guarantees that logical units in the x-direction and in the y-direction are equal, while the MM_ANISOTROPIC mode allows the units to differ. A CAD or drawing application can benefit from the MM_ISOTROPIC mapping mode but may need to specify logical units that correspond to the increments on an engineer's scale (1/64 inch). These units would be difficult to obtain with the predefined mapping modes (MM_HIENGLISH or MM_HIMETRIC); however, they can easily be obtained by selecting the MM_ISOTROPIC (or MM_ANISOTROPIC) mode. The following example shows how to set logical units to 1/64 inch:

```
SetMapMode(hDC, MM_ISOTROPIC);
SetWindowExtEx(hDC, 64, 64, NULL);
SetViewportExtEx(hDC, GetDeviceCaps(hDC, LOGPIXELSX),
                      GetDeviceCaps(hDC, LOGPIXELSY), NULL);
```

Device-Space to Physical-Device Transformation

4/20/2022 • 2 minutes to read • Edit Online

The device-space to physical-device transformation is unique in several respects. For example, it is limited to translation and is controlled by the system. The sole purpose of this transformation is to ensure that the origin of device space is mapped to the proper point on the physical device. There are no functions to set this transformation, nor are there any functions to retrieve related data.

Default Transformations

4/20/2022 • 2 minutes to read • Edit Online

Whenever an application creates a DC and immediately begins calling GDI drawing or output functions, it takes advantage of the default page-space to device-space, and device-space to client-area transformations. A world-to-page space transformation cannot happen until the application first calls the **SetGraphicsMode** function to set the mode to GM_ADVANCED and then calls the **SetWorldTransform** function.

Use of MM_TEXT (the default page-space to device-space transformation) results in a one-to-one mapping; that is, a given point in page space maps to the same point in device space. As previously mentioned, this transformation is not specified by a matrix. Instead, it is obtained by dividing the width of the viewport by the width of the window and the height of the viewport by the height of the window. In the default case, the viewport dimensions are 1-pixel by 1-pixel and the window dimensions are 1-page unit by 1-page unit.

The device-space to physical-device (client area, desktop, or printer paper) transformation always results in a one-to-one mapping; that is, one unit in device space is always equivalent to one unit in the client area, on the desktop, or on a page. The sole purpose of this transformation is translation; it ensures that output appears correctly in an application's window no matter where that window is moved on the desktop.

The one unique aspect of MM_TEXT is the orientation of the y-axis in page space. In MM_TEXT, the positive y-axis extends downward and the negative y-axis extends upward.

Using Coordinate Spaces and Transformations

4/20/2022 • 3 minutes to read • Edit Online

This section contains an example that demonstrates the following tasks:

- Drawing graphics with predefined units.
- Centering graphics in the application's client area.
- Scaling graphics output to half its original size.
- Translating graphics output 3/4 of an inch to the right.
- Rotating graphics 30 degrees.
- Shearing graphics output along the x-axis.
- Reflecting graphics output about an imaginary horizontal axis drawn through its midpoint.

The following example was used to create the illustrations that appear earlier in this overview.

```
void TransformAndDraw(int iTransform, HWND hWnd)
{
    HDC hDC;
    XFORM xForm;
    RECT rect;

    // Retrieve a DC handle for the application's window.

    hDC = GetDC(hWnd);

    // Set the mapping mode to LOENGLISH. This moves the
    // client area origin from the upper left corner of the
    // window to the lower left corner (this also reorients
    // the y-axis so that drawing operations occur in a true
    // Cartesian space). It guarantees portability so that
    // the object drawn retains its dimensions on any display.

    SetGraphicsMode(hDC, GM_ADVANCED);
    SetMapMode(hDC, MM_LOENGLISH);

    // Set the appropriate world transformation (based on the
    // user's menu selection).

    switch (iTransform)
    {
        case SCALE:        // Scale to 1/2 of the original size.
            xForm.eM11 = (FLOAT) 0.5;
            xForm.eM12 = (FLOAT) 0.0;
            xForm.eM21 = (FLOAT) 0.0;
            xForm.eM22 = (FLOAT) 0.5;
            xForm.eDx  = (FLOAT) 0.0;
            xForm.eDy  = (FLOAT) 0.0;
            SetWorldTransform(hDC, &xForm);
            break;

        case TRANSLATE:   // Translate right by 3/4 inch.
            xForm.eM11 = (FLOAT) 1.0;
            xForm.eM12 = (FLOAT) 0.0;
            xForm.eM21 = (FLOAT) 0.0;
            xForm.eM22 = (FLOAT) 1.0;
            xForm.eDx  = (FLOAT) 75.0;
            xForm.eDy  = (FLOAT) 0.0;
            SetWorldTransform(hDC, &xForm);
            break;
        case ROTATE:      // Rotate 30 degrees counterclockwise.
            xForm.eM11 = (FLOAT) 0.8660;
            xForm.eM12 = (FLOAT) 0.5000;
            xForm.eM21 = (FLOAT) -0.5000;
            xForm.eM22 = (FLOAT) 0.8660;
            xForm.eDx  = (FLOAT) 0.0;
            xForm.eDy  = (FLOAT) 0.0;
            SetWorldTransform(hDC, &xForm);
            break;

        case SHEAR:       // Shear along the x-axis with a
                          // proportionality constant of 1.0.
            xForm.eM11 = (FLOAT) 1.0;
            xForm.eM12 = (FLOAT) 1.0;
            xForm.eM21 = (FLOAT) 0.0;
            xForm.eM22 = (FLOAT) 1.0;
            xForm.eDx  = (FLOAT) 0.0;
            xForm.eDy  = (FLOAT) 0.0;
            SetWorldTransform(hDC, &xForm);
            break;

        case REFLECT:     // Reflect about a horizontal axis.
            xForm.eM11 = (FLOAT) 1.0;
            xForm.eM12 = (FLOAT) 0.0;
            xForm.eM21 = (FLOAT) 0.0;
            xForm.eM22 = (FLOAT) -1.0;
            xForm.eDx  = (FLOAT) 0.0;
            xForm.eDy  = (FLOAT) 0.0;
            SetWorldTransform(hDC, &xForm);
            break;

        case NORMAL:      // Set the unity transformation.
            xForm.eM11 = (FLOAT) 1.0;
            xForm.eM12 = (FLOAT) 0.0;
            xForm.eM21 = (FLOAT) 0.0;
            xForm.eM22 = (FLOAT) 1.0;
            xForm.eDx  = (FLOAT) 0.0;
            xForm.eDy  = (FLOAT) 0.0;
            SetWorldTransform(hDC, &xForm);
            break;

    }

    // Find the midpoint of the client area.

    GetClientRect(hWnd, (LPRECT) &rect);
    DPtoLP(hDC, (LPPOINT) &rect, 2);

    // Select a hollow brush.

    SelectObject(hDC, GetStockObject(HOLLOW_BRUSH));

    // Draw the exterior circle.

    Ellipse(hDC, (rect.right / 2 - 100), (rect.bottom / 2 + 100),
        (rect.right / 2 + 100), (rect.bottom / 2 - 100));

    // Draw the interior circle.

    Ellipse(hDC, (rect.right / 2 -94), (rect.bottom / 2 + 94),
        (rect.right / 2 + 94), (rect.bottom / 2 - 94));

    // Draw the key.

    Rectangle(hDC, (rect.right / 2 - 13), (rect.bottom / 2 + 113),
        (rect.right / 2 + 13), (rect.bottom / 2 + 50));
    Rectangle(hDC, (rect.right / 2 - 13), (rect.bottom / 2 + 96),
        (rect.right / 2 + 13), (rect.bottom / 2 + 50));
    // Draw the horizontal lines.

    MoveToEx(hDC, (rect.right/2 - 150), (rect.bottom / 2 + 0), NULL);
    LineTo(hDC, (rect.right / 2 - 16), (rect.bottom / 2 + 0));

    MoveToEx(hDC, (rect.right / 2 - 13), (rect.bottom / 2 + 0), NULL);
    LineTo(hDC, (rect.right / 2 + 13), (rect.bottom / 2 + 0));

    MoveToEx(hDC, (rect.right / 2 + 16), (rect.bottom / 2 + 0), NULL);
    LineTo(hDC, (rect.right / 2 + 150), (rect.bottom / 2 + 0));

    // Draw the vertical lines.

    MoveToEx(hDC, (rect.right/2 + 0), (rect.bottom / 2 - 150), NULL);
    LineTo(hDC, (rect.right / 2 + 0), (rect.bottom / 2 - 16));

    MoveToEx(hDC, (rect.right / 2 + 0), (rect.bottom / 2 - 13), NULL);
    LineTo(hDC, (rect.right / 2 + 0), (rect.bottom / 2 + 13));

    MoveToEx(hDC, (rect.right / 2 + 0), (rect.bottom / 2 + 16), NULL);
    LineTo(hDC, (rect.right / 2 + 0), (rect.bottom / 2 + 150));

    ReleaseDC(hWnd, hDC);
}
```

Coordinate Space and Transformation Reference

4/20/2022 • 2 minutes to read • Edit Online

The following elements are used with coordinate spaces and transformations.

- Coordinate Space and Transformation Functions
- Coordinate Space and Transformation Structures

Coordinate Space and Transformation Functions

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used with coordinate spaces and transformations.

FUNCTION	DESCRIPTION
ClientToScreen	Converts the client-area coordinates of a specified point to screen coordinates.
CombineTransform	Concatenates two world-space to page-space transformations.
DPtoLP	Converts device coordinates into logical coordinates.
GetCurrentPositionEx	Retrieves the current position in logical coordinates.
GetDisplayAutoRotationPreferences	Gets the orientation preferences of the display.
GetGraphicsMode	Retrieves the current graphics mode for the specified device context.
GetMapMode	Retrieves the current mapping mode.
GetViewportExtEx	Retrieves the x-extent and y-extent of the current viewport for the specified device context.
GetViewportOrgEx	Retrieves the x-coordinates and y-coordinates of the viewport origin for the specified device context.
GetWindowExtEx	Retrieves the x-extent and y-extent of the window for the specified device context.
GetWindowOrgEx	Retrieves the x-coordinates and y-coordinates of the window origin for the specified device context.
GetWorldTransform	Retrieves the current world-space to page-space transformation.
LPtoDP	Converts logical coordinates into device coordinates.
MapWindowPoints	Converts (maps) a set of points from a coordinate space relative to one window to a coordinate space relative to another window.
ModifyWorldTransform	Changes the world transformation for a device context using the specified mode.
OffsetViewportOrgEx	Modifies the viewport origin for a device context using the specified horizontal and vertical offsets.

FUNCTION	DESCRIPTION
OffsetWindowOrgEx	Modifies the window origin for a device context using the specified horizontal and vertical offsets.
ScaleViewportExtEx	Modifies the viewport for a device context using the ratios formed by the specified multiplicands and divisors.
ScaleWindowExtEx	Modifies the window for a device context using the ratios formed by the specified multiplicands and divisors.
ScreenToClient	Converts the screen coordinates of a specified point on the screen to client coordinates.
SetDisplayAutoRotationPreferences	Sets the orientation preferences of the display.
SetGraphicsMode	Sets the graphics mode for the specified device context.
SetMapMode	Sets the mapping mode of the specified device context.
SetViewportExtEx	Sets the horizontal and vertical extents of the viewport for a device context by using the specified values.
SetViewportOrgEx	Specifies which device point maps to the window origin (0,0).
SetWindowExtEx	Sets the horizontal and vertical extents of the window for a device context by using the specified values.
SetWindowOrgEx	Specifies which window point maps to the viewport origin (0,0).
SetWorldTransform	Sets a two-dimensional linear transformation between world space and page space for the specified device context.

Coordinate Space and Transformation Structures

4/20/2022 • 2 minutes to read • Edit Online

The following structure is used with coordinate spaces and transformations.

XFORM

Device Contexts

4/20/2022 • 2 minutes to read • Edit Online

A *device context* is a structure that defines a set of graphic objects and their associated attributes, as well as the graphic modes that affect output. The *graphic objects* include a pen for line drawing, a brush for painting and filling, a bitmap for copying or scrolling parts of the screen, a palette for defining the set of available colors, a region for clipping and other operations, and a path for painting and drawing operations. The remainder of this section is divided into the following three areas.

- About Device Contexts
- Using the Device Context Functions
- Device Context Reference

About Device Contexts

4/20/2022 • 2 minutes to read • Edit Online

Device independence is one of the chief features of Microsoft Windows. Applications can draw and print output on a variety of devices. The software that supports this device independence is contained in two dynamic-link libraries. The first, Gdi.dll, is referred to as the graphics device interface (GDI); the second is referred to as a device driver. The name of the second depends on the device where the application draws output. For example, if the application draws output in the client area of its window on a VGA display, this library is Vga.dll; if the application prints output on an Epson FX-80 printer, this library is Epson9.dll.

An application must inform GDI to load a particular device driver and, once the driver is loaded, to prepare the device for drawing operations (such as selecting a line color and width, a brush pattern and color, a font typeface, a clipping region, and so on). These tasks are accomplished by creating and maintaining a device context (DC). A DC is a structure that defines a set of graphic objects and their associated attributes, and the graphic modes that affect output. The graphic objects include a pen for line drawing, a brush for painting and filling, a bitmap for copying or scrolling parts of the screen, a palette for defining the set of available colors, a region for clipping and other operations, and a path for painting and drawing operations. Unlike most of the structures, an application never has direct access to the DC; instead, it operates on the structure indirectly by calling various functions.

This overview provides information on the following topics:

- Graphic Objects
- Graphic Modes
- Device Context Types
- Device Context Operations
- ICM-Enabled Device Context Functions

An important concept is the layout of a DC or a window, which describes the order in which GDI objects and text are revealed (either left-to-right or right-to-left). For more information, see "Window Layout and Mirroring" in `Window Features` and the `GetLayout` and `SetLayout` functions.

Graphic Objects

4/20/2022 • 2 minutes to read • Edit Online

The pen, brush, bitmap, palette, region, and path associated with a DC are referred to as its graphic objects. The following attributes are associated with each of these objects.

GRAPHIC OBJECT	ASSOCIATED ATTRIBUTES
Bitmap	Size, in bytes; dimensions, in pixels; color-format; compression scheme; and so on.
Brush	Style, color, pattern, and origin.
Palette	Colors and size (or number of colors).
Font	Typeface name, width, height, weight, character set, and so on.
Path	Shape.
Pen	Style, width, and color.
Region	Location and dimensions.

When an application creates a DC, the system automatically stores a set of default objects in it (there is no default bitmap or path). An application can examine the attributes of the default objects by calling the GetCurrentObject and GetObject functions. The application can change these defaults by creating a new object and selecting it into the DC. An object is selected into a DC by calling the SelectObject function.

An application can set the current brush color to a specified color value with SetDCBrushColor.

The GetDCBrushColor function returns the DC brush color. The SetDCPenColor function sets the pen color to a specified color value. The GetDCPenColor function returns the DC pen color.

Graphic Modes

4/20/2022 • 2 minutes to read • Edit Online

Windows supports five graphic modes that allow an application to specify how colors are mixed, where output appears, how the output is scaled, and so on. These modes, which are stored in a DC, are described in the following table.

GRAPHICS MODE	DESCRIPTION
Background	Defines how background colors are mixed with existing window or screen colors for bitmap and text operations.
Drawing	Defines how foreground colors are mixed with existing window or screen colors for pen, brush, bitmap, and text operations.
Mapping	Defines how graphics output is mapped from logical (or world) space onto the window, screen, or printer paper.
Polygon-fill	Defines how the brush pattern is used to fill the interior of complex regions.
Stretching	Defines how bitmap colors are mixed with existing window or screen colors when the bitmap is compressed (or scaled down).

As it does with graphic objects, the system initializes a DC with default graphic modes. An application can retrieve and examine these default modes by calling the following functions.

GRAPHICS MODE	FUNCTION
Background	GetBkMode
Drawing	GetROP2
Mapping	GetMapMode
Polygon-fill	GetPolyFillMode
Stretching	GetStretchBltMode

An application can change the default modes by calling one of the following functions.

GRAPHICS MODE	FUNCTION
Background	SetBkMode

GRAPHICS MODE	FUNCTION
Drawing	SetROP2
Mapping	SetMapMode
Polygon-fill	SetPolyFillMode
Stretching	SetStretchBltMode

Device Context Types

4/20/2022 • 2 minutes to read • Edit Online

There are four types of DCs: display, printer, memory (or compatible), and information. Each type serves a specific purpose, as described in the following table.

DEVICE CONTEXT	DESCRIPTION
Display	Supports drawing operations on a video display.
Printer	Supports drawing operations on a printer or plotter.
Memory	Supports drawing operations on a bitmap.
Information	Supports the retrieval of device data.

Display Device Contexts

4/20/2022 • 2 minutes to read • Edit Online

An application obtains a display DC by calling the **BeginPaint**, **GetDC**, or **GetDCEx** function and identifying the window in which the corresponding output will appear. Typically, an application obtains a display DC only when it must draw in the client area. However, one may obtain a window device context by calling the **GetWindowDC** function. When the application is finished drawing, it must release the DC by calling the **EndPaint** or **ReleaseDC** function.

There are five types of DCs for video displays:

- Class
- Common
- Private
- Window
- Parent

Class Device Contexts

Class device contexts are supported strictly for compatibility with 16-bit versions of Windows. When writing your application, avoid using the class device context; use a private device context instead.

Common Device Contexts

Common device contexts are display DCs maintained in a special cache by the system. Common device contexts are used in applications that perform infrequent drawing operations. Before the system returns the DC handle, it initializes the common device context with default objects, attributes, and modes. Any drawing operations performed by the application use these defaults unless one of the GDI functions is called to select a new object, change the attributes of an existing object, or select a new mode.

Because only a limited number of common device contexts exist, an application should release them after it has finished drawing. When the application releases a common device context, any changes to the default data are lost.

Private Device Contexts

Private device contexts are display DCs that, unlike common device contexts, retain any changes to the default data even after an application releases them. Private device contexts are used in applications that perform numerous drawing operations such as computer-aided design (CAD) applications, desktop-publishing applications, drawing and painting applications, and so on. Private device contexts are not part of the system cache and therefore need not be released after use. The system automatically removes a private device context after the last window of that class has been destroyed.

An application creates a private device context by first specifying the CS_OWNDC window-class style when it initializes the **style** member of the **WNDCLASS** structure and calls the **RegisterClass** function. (For more information about window classes, see Window Classes.)

After creating a window with the CS_OWNDC style, an application can call the **GetDC**, **GetDCEx**, or **BeginPaint** function once to obtain a handle identifying a private device context. The application can continue using this handle (and the associated DC) until it deletes the window created with this class. Any changes to graphic objects and their attributes, or graphic modes are retained by the system until the window is deleted.

Window Device Contexts

A *window device context* enables an application to draw anywhere in a window, including the nonclient area. Window device contexts are typically used by applications that process the **WM_NCPAINT** and **WM_NCACTIVATE** messages for windows with custom nonclient areas. Using a window device context is not recommended for any other purpose. For more information; see **GetWindowDC**.

Parent Device Contexts

A *parent device context* enables an application to minimize the time necessary to set up the clipping region for a window. An application typically uses parent device contexts to speed up drawing for control windows without requiring a private or class device context. For example, the system uses parent device contexts for push button and edit controls. Parent device contexts are intended for use with child windows only, never with top-level or pop-up windows. For more information; see Parent Display Device Contexts.

Printer Device Contexts (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

The printer DC can be used when printing on a dot-matrix printer, ink-jet printer, laser printer, or plotter. An application creates a printer DC by calling the **CreateDC** function and supplying the appropriate arguments (the name of the printer driver, the name of the printer, the file or device name for the physical output medium, and other initialization data). When an application has finished printing, it deletes the printer DC by calling the **DeleteDC** function. An application must delete (rather than release) a printer DC; the **ReleaseDC** function fails when an application attempts to use it to release a printer DC.

For more information, see Printer Output.

Memory Device Contexts

4/20/2022 • 2 minutes to read • Edit Online

To enable applications to place output in memory rather than sending it to an actual device, use a special device context for bitmap operations called a *memory device context*. A memory DC enables the system to treat a portion of memory as a virtual device. It is an array of bits in memory that an application can use temporarily to store the color data for bitmaps created on a normal drawing surface. Because the bitmap is compatible with the device, a memory DC is also sometimes referred to as a *compatible device context*.

The memory DC stores bitmap images for a particular device. An application can create a memory DC by calling the **CreateCompatibleDC** function.

The original bitmap in a memory DC is simply a placeholder. Its dimensions are one pixel by one pixel. Before an application can begin drawing, it must select a bitmap with the appropriate width and height into the DC by calling the **SelectObject** function. To create a bitmap of the appropriate dimensions, use the **CreateBitmap**, **CreateBitmapIndirect**, or **CreateCompatibleBitmap** function. After the bitmap is selected into the memory DC, the system replaces the single-bit array with an array large enough to store color information for the specified rectangle of pixels.

When an application passes the handle returned by **CreateCompatibleDC** to one of the drawing functions, the requested output does not appear on a device's drawing surface. Instead, the system stores the color information for the resultant line, curve, text, or region in the array of bits. The application can copy the image stored in memory back onto a drawing surface by calling the **BitBlt** function, identifying the memory DC as the source device context and a window or screen DC as the target device context.

When displaying a DIB or a DDB created from a DIB on a palette device, you can improve the speed at which the image is drawn by arranging the logical palette to match the layout of the system palette. To do this, call **GetDeviceCaps** with the NUMRESERVED value to get the number of reserved colors in the system. Then call **GetSystemPaletteEntries** and fill in the first and last NUMRESERVED/2 entries of the logical palette with the corresponding system colors. For example, if NUMRESERVED is 20, you would fill in the first and last 10 entries of the logical palette with the system colors. Then fill in the remaining 256-NUMRESERVED colors of the logical palette (in our example, the remaining 236 colors) with colors from the DIB and set the PC_NOCOLLAPSE flag on each of these colors.

For more information about color and palettes, see Colors. For more information about bitmaps and bitmap operations, see Bitmaps.

Information Device Contexts

4/20/2022 • 2 minutes to read • Edit Online

The information DC is used to retrieve default device data. For example, an application can call the **CreateIC** function to create an information DC for a particular model of printer and then call the **GetCurrentObject** and **GetObject** functions to retrieve the default pen or brush attributes. Because the system can retrieve device information without creating the structures normally associated with the other types of device contexts, an information DC involves far less overhead and is created significantly faster than any of the other types. After an application finishes retrieving data by using an information DC, it must call the **DeleteDC** function.

Device Context Operations

4/20/2022 • 2 minutes to read • Edit Online

An application can perform the following operations on a device context:

- Enumerate existing graphic objects.
- Select new graphic objects.
- Delete existing graphic objects.
- Save the current graphic objects, their attributes, and the graphic modes.
- Restore previously saved graphic objects, their attributes, and the graphic modes.

In addition, an application can use a device context to:

- Determine how graphics output is translated.
- Cancel lengthy drawing operations (begun by a thread in a multithreaded application).
- Reset a printer to a particular state.

This subsection provides information on the following topics.

- Operations on Graphic Objects
- Cancellation of Drawing Operations
- Retrieving Device Data
- Saving, Restoring, and Resetting a Device Context

Operations on Graphic Objects

4/20/2022 • 2 minutes to read • Edit Online

After an application creates a display or printer DC, it can begin drawing on the associated device or, in the case of the memory DC, it can begin drawing on the bitmap stored in memory. However, before drawing begins, and sometimes while drawing is in progress, it is often necessary to replace the default objects with new objects.

An application can examine a default object's attributes by calling the GetCurrentObject and GetObject functions. The **GetCurrentObject** function returns a handle identifying the current pen, brush, palette, bitmap, or font, and the **GetObject** function initializes a structure containing that object's attributes.

Some printers provide resident pens, brushes, and fonts that can be used to improve drawing speed in an application. Two functions can be used to enumerate these objects: EnumObjects and EnumFontFamilies. If the application must enumerate resident pens or brushes, it can call the **EnumObjects** function to examine the corresponding attributes. If the application must enumerate resident fonts, it can call the **EnumFontFamilies** function (which can also enumerate GDI fonts).

Once an application determines that a default object needs replacing, it creates a new object by calling one of the following creation functions.

GRAPHIC OBJECT	FUNCTION
Bitmap	CreateBitmap, CreateBitmapIndirect, CreateCompatibleBitmap, CreateDiscardableBitmap, CreateDIBitmap
Brush	CreateBrushIndirect, CreateDIBPatternBrush, CreateDIBPatternBrushPt, CreateHatchBrush, CreatePatternBrush, CreateSolidBrush
Color Palette	CreatePalette
Font	CreateFont, CreateFontIndirect
Pen	CreatePen, CreatePenIndirect, ExtCreatePen
Region	CreateEllipticRgn, CreateEllipticRgnIndirect, CreatePolygonRgn, CreatePolyPolygonRgn, CreateRectRgn, CreateRectRgnIndirect, CreateRoundRectRgn

Each of these functions returns a handle identifying a new object. After an application retrieves a handle, it must call the SelectObject function to replace the default object. However, the application should save the handle identifying the default object and use this handle to replace the new object when it is no longer needed. When the application finishes drawing with the new object, it must restore the default object by calling the **SelectObject** function and then delete the new object by calling the DeleteObject function. Failing to delete objects causes serious performance problems.

Cancellation of Drawing Operations

4/20/2022 • 2 minutes to read • Edit Online

When complex drawing applications perform lengthy graphics operations, they consume valuable system resources. By taking advantage of the system's multitasking features, an application can use threads and the **CancelDC** function to manage these operations. For example, if the graphics operation performed by thread A is consuming needed resources, thread B can call the CancelDC function to halt that operation.

Retrieving Device Data

4/20/2022 • 2 minutes to read • Edit Online

Applications can use the following functions to retrieve device data using a device context: **GetDeviceCaps** and **DeviceCapabilities**.

GetDeviceCaps retrieves general device data for the following devices:

- Raster displays
- Dot-matrix printers
- Ink-jet printers
- Laser printers
- Vector plotters
- Raster cameras

The data includes the supported capabilities of the device, including device resolution (for video displays), color format (for video displays and color printers), number of graphic objects, raster capabilities, curve drawing, line drawing, polygon drawing, and text drawing. An application retrieves this data by supplying a handle identifying the appropriate device context, as well as an index specifying the type of data the function is to retrieve.

The **DeviceCapabilities** function retrieves data specific to printers, including the number of available paper bins, the duplex capabilities of the printer, the resolutions supported by the printer, the maximum and minimum supported paper size, and so on. An application retrieves this data by supplying strings specifying a printer device and port, as well as an index specifying the type of data that the function is to retrieve.

Saving, Restoring, and Resetting a Device Context

4/20/2022 • 2 minutes to read • Edit Online

The following functions enable an application to save, restore, and reset a device context: **SaveDC**, **RestoreDC**, and **ResetDC**. The SaveDC function records on a special GDI stack the current DC's graphic objects and their attributes, and graphic modes. A drawing application can call this function before a user begins drawing and save the application's original state providing a clean slate for the user. To return to this original state, the application calls the RestoreDC function.

ResetDC is provided to reset printer DC data. An application calls this function to reset the paper orientation, paper size, output scaling factor, number of copies to be printed, paper source (or bin), duplex mode, and so on. Typically, an application calls this function after a user has changed one of the printer options and the system has issued a **WM_DEVMODECHANGE** message.

ICM-Enabled Device Context Functions

4/20/2022 • 2 minutes to read • Edit Online

Microsoft Image Color Management (ICM) ensures that a color image, graphic, or text object is rendered as closely as possible to its original intent on any device, despite differences in imaging technologies and color capabilities between devices. (For more information, see Windows Color System.)

There are various functions in the graphics device interface (GDI) that use or operate on color data. The following device context functions are enabled for use with ICM:

- CreateCompatibleDC
- CreateDC
- GetDCBrushColor
- GetDCPenColor
- ResetDC
- SelectObject
- SetDCBrushColor
- SetDCPenColor

Using the Device Context Functions

4/20/2022 • 2 minutes to read • Edit Online

Obtaining a Private Display Device Context

4/20/2022 • 2 minutes to read • Edit Online

An application performing numerous drawing operations in the client area of its window must use a private display DC. To create this type of DC, the application must specify the **CS_OWNDC** constant for the style member of the **WNDCLASS** structure when registering the window class. After registering the window class, the application obtains a handle identifying a private display DC by calling the **GetDC** function.

The following example shows how to create a private display DC.

```
#include <windows.h>    // required for all Windows-based applications
#include <stdio.h>      // C run-time header for i/o
#include <string.h>     // C run-time header for strtok_s
#include "dc.h"         // specific to this program

// Function prototypes.

BOOL InitApplication(HINSTANCE);
long FAR PASCAL MainWndProc(HWND, UINT, UINT, LONG);

// Global variables

HINSTANCE hinst;       // handle to current instance
HDC hdc;               // display device context handle

BOOL InitApplication(HINSTANCE hinstance)
{
    WNDCLASS  wc;

    // Fill in the window class structure with parameters
    // describing the main window.

    wc.style = CS_OWNDC;         // Private-DC constant
    wc.lpfnWndProc = (WNDPROC) MainWndProc;

    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hinstance;

    wc.hIcon = LoadIcon((HINSTANCE) NULL,
        MAKEINTRESOURCE(IDI_APPLICATION));

    wc.hCursor = LoadCursor((HINSTANCE) NULL,
        MAKEINTRESOURCE(IDC_ARROW));

    wc.hbrBackground = GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName =  "GenericMenu";
    wc.lpszClassName = "GenericWClass";

    // Register the window class and return the resulting code.

    return RegisterClass(&wc);
}

LRESULT APIENTRY MainWndProc(
        HWND hwnd,          // window handle
        UINT message,       // type of message
        WPARAM wParam,      // additional information
        LPARAM lParam)      // additional information
{
    PAINTSTRUCT ps;              // paint structure

    // Retrieve a handle identifying the private DC.

    hdc = GetDC(hwnd);

    switch (message)
    {
        case WM_PAINT:
              hdc = BeginPaint(hwnd, &ps);

        // Draw and paint using private DC.

                 EndPaint(hwnd, &ps);

        case WM_DESTROY:
                     PostQuitMessage(0);
                     break;
        default:
                return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Retrieving the Capabilities of a Printer

4/20/2022 • 2 minutes to read • Edit Online

Not every output device supports the entire set of graphics functions. For example, because of hardware limitations, most vector plotters do not support bit-block transfers. An application can determine whether a device supports a particular graphics function by calling the **GetDeviceCaps** function, specifying the appropriate index, and examining the return value.

The following example shows how an application tests a printer to determine whether it supports bit-block transfers.

```
// Examine the raster capabilities of the device
// identified by hdcPrint to verify that it supports
// the BitBlt function.

if ((GetDeviceCaps(hdcPrint, RASTERCAPS)
       & RC_BITBLT) == 0)
{
   DeleteDC(hdcPrint);
   break;
}

else
{
    // Print the bitmap using the printer DC.
}
```

Retrieve object attributes, select new objects

4/20/2022 • 2 minutes to read • Edit Online

An application can retrieve the attributes for a pen, brush, palette, font, or bitmap by calling the GetCurrentObject and GetObject functions. The **GetCurrentObject** function returns a handle identifying the object currently selected into the DC; the **GetObject** function returns a structure that describes the attributes of the object.

The following example shows how an application can retrieve the current brush attributes and use the retrieved data to determine whether it is necessary to select a new brush.

```
HDC hdc;                     // display DC handle
HBRUSH hbrushNew, hbrushOld; // brush handles
HBRUSH hbrush;               // brush handle
LOGBRUSH lb;                 // logical-brush structure

// Retrieve a handle identifying the current brush.

hbrush = GetCurrentObject(hdc, OBJ_BRUSH);

// Retrieve a LOGBRUSH structure that contains the
// current brush attributes.

GetObject(hbrush, sizeof(LOGBRUSH), &lb);

// If the current brush is not a solid-black brush,
// replace it with the solid-black stock brush.

if ((lb.lbStyle != BS_SOLID)
       || (lb.lbColor != 0x000000))
{
    hbrushNew = GetStockObject(BLACK_BRUSH);
    hbrushOld = SelectObject(hdc, hbrushNew);
}

// Perform painting operations with the white brush.


// After completing the last painting operation with the new
// brush, the application should select the original brush back
// into the device context and delete the new brush.
// In this example, hbrushNew contains a handle to a stock object.
// It is not necessary (but it is not harmful) to call
// DeleteObject on a stock object. If hbrushNew contained a handle
// to a brush created by a function such as CreateBrushIndirect,
// it would be necessary to call DeleteObject.

SelectObject(hdc, hbrushOld);
DeleteObject(hbrushNew);
```

NOTE

The application saved the original brush handle when calling the SelectObject function the first time. This handle is saved so that the original brush can be selected back into the DC after the last painting operation has been completed with the new brush. After the original brush is selected back into the DC, the new brush is deleted, freeing memory in the GDI heap.

Setting and Retrieving the Device Context Brush Color Value

4/20/2022 • 2 minutes to read • Edit Online

The following example shows how an application can retrieve the current DC brush color by using the **SetDCBrushColor** and the **GetDCBrushColor** functions.

```
SelectObject(hdc,GetStockObject(DC_BRUSH));
SetDCBrushColor(hdc,RGB(00,0xff,00));
PatBlt(hdc,0,0,200,200,PATCOPY);
SetDCBrushColor(hdc,RGB(00,00,0xff));
PatBlt(hdc,0,0,200,200,PATCOPY);
```

Setting the Pen or Brush Color

4/20/2022 • 2 minutes to read • Edit Online

The following example shows how an application can change the DC pen color by using the GetStockObject function or SetDCPenColor and the SetDCBrushColor functions.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
    case WM_COMMAND:
        wmId    = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Parse the menu selections:
        switch (wmId)
        {
        case IDM_ABOUT:
            DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
            break;
        case IDM_EXIT:
            DestroyWindow(hWnd);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
        }
        break;
    case WM_PAINT:

        {
            hdc = BeginPaint(hWnd, &ps);
        //    Initializing original object
            HGDIOBJ original = NULL;

        //    Saving the original object
            original = SelectObject(hdc,GetStockObject(DC_PEN));

        //    Rectangle function is defined as...
        //    BOOL Rectangle(hdc, xLeft, yTop, yRight, yBottom);

        //    Drawing a rectangle with just a black pen
        //    The black pen object is selected and sent to the current device context
        //  The default brush is WHITE_BRUSH
            SelectObject(hdc, GetStockObject(BLACK_PEN));
            Rectangle(hdc,0,0,200,200);

        //    Select DC_PEN so you can change the color of the pen with
        //    COLORREF SetDCPenColor(HDC hdc, COLORREF color)
            SelectObject(hdc, GetStockObject(DC_PEN));

        //    Select DC_BRUSH so you can change the brush color from the
        //    default WHITE_BRUSH to any other color
            SelectObject(hdc, GetStockObject(DC_BRUSH));

        //    Set the DC Brush to Red
        //    The RGB macro is declared in "Windowsx.h"
            SetDCBrushColor(hdc, RGB(255,0,0));

        //    Set the Pen to Blue
```

```
            SetDCPenColor(hdc, RGB(0,0,255));

        //    Drawing a rectangle with the current Device Context
            Rectangle(hdc,100,300,200,400);

        //    Changing the color of the brush to Green
            SetDCBrushColor(hdc, RGB(0,255,0));
            Rectangle(hdc,300,150,500,300);

        //    Restoring the original object
            SelectObject(hdc,original);

        // It is not necessary to call DeleteObject to delete stock objects.
        }

        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Getting Information on a Display Monitor

4/20/2022 • 2 minutes to read • Edit Online

The following code sample shows how to use **EnumDisplayDevices** to get information on a display monitor.

```
BOOL GetDisplayMonitorInfo(int nDeviceIndex, LPSTR lpszMonitorInfo)
{
    FARPROC EnumDisplayDevices;
    HINSTANCE  hInstUser32;
    DISPLAY_DEVICE DispDev;
    char szSaveDeviceName[33];  // 32 + 1 for the null-terminator
    BOOL bRet = TRUE;
        HRESULT hr;

    hInstUser32 = LoadLibrary("c:\\windows\User32.DLL");
    if (!hInstUser32) return FALSE;

    // Get the address of the EnumDisplayDevices function
    EnumDisplayDevices = (FARPROC)GetProcAddress(hInstUser32,"EnumDisplayDevicesA");
    if (!EnumDisplayDevices) {
        FreeLibrary(hInstUser32);
        return FALSE;
    }

    ZeroMemory(&DispDev, sizeof(DispDev));
    DispDev.cb = sizeof(DispDev);

    // After the first call to EnumDisplayDevices,
    // DispDev.DeviceString is the adapter name
    if (EnumDisplayDevices(NULL, nDeviceIndex, &DispDev, 0))
        {
                hr = StringCchCopy(szSaveDeviceName, 33, DispDev.DeviceName);
                if (FAILED(hr))
                {
                // TODO: write error handler
                }

        // After second call, DispDev.DeviceString is the
        // monitor name for that device
        EnumDisplayDevices(szSaveDeviceName, 0, &DispDev, 0);

                // In the following, lpszMonitorInfo must be 128 + 1 for
                // the null-terminator.
                hr = StringCchCopy(lpszMonitorInfo, 129, DispDev.DeviceString);
                if (FAILED(hr))
                {
                // TODO: write error handler
                }

    } else    {
        bRet = FALSE;
    }

    FreeLibrary(hInstUser32);

    return bRet;
}
```

Device Context Reference

4/20/2022 • 2 minutes to read • Edit Online

The following elements are associated with device contexts:

- Device Context Functions
- Device Context Structures
- Device Context Messages

Device Context Functions

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used with device contexts.

FUNCTION	DESCRIPTION
CancelDC	Cancels any pending operation on the specified device context.
ChangeDisplaySettings	Changes the settings of the default display device to the specified graphics mode.
ChangeDisplaySettingsEx	Changes the settings of the specified display device to the specified graphics mode.
CreateCompatibleDC	Creates a memory device context compatible with the specified device.
CreateDC	Creates a device context for a device using the specified name.
CreateIC	Creates an information context for the specified device.
DeleteDC	Deletes the specified device context.
DeleteObject	Deletes a logical pen, brush, font, bitmap, region, or palette, freeing all system resources associated with the object.
DeviceCapabilities	Retrieves the capabilities of a printer device driver.
DrawEscape	Provides drawing capabilities of the specified video display that are not directly available through the graphics device interface.
EnumDisplayDevices	Retrieves information about the display devices in a system.
EnumDisplaySettings	Retrieves information about one of the graphics modes for a display device.
EnumDisplaySettingsEx	Retrieves information about one of the graphics modes for a display device.
EnumObjects	Enumerates the pens or brushes available for the specified device context.
EnumObjectsProc	An application-defined callback function used with the EnumObjects function.
GetCurrentObject	Retrieves a handle to an object of the specified type that has been selected into the specified device context.

FUNCTION	DESCRIPTION
GetDC	Retrieves a handle to a display device context for the client area of a specified window or for the entire screen.
GetDCBrushColor	Retrieves the current brush color for the specified device context.
GetDCEx	Retrieves a handle to a display device context for the client area of a specified window or for the entire screen.
GetDCOrgEx	Retrieves the final translation origin for a specified device context.
GetDCPenColor	Retrieves the current pen color for the specified device context.
GetDeviceCaps	Retrieves device-specific information for the specified device.
GetLayout	Retrieves the layout of a device context.
GetObject	Retrieves information for the specified graphics object.
GetObjectType	Retrieves the type of the specified object.
GetStockObject	Retrieves a handle to one of the stock pens, brushes, fonts, or palettes.
ReleaseDC	Releases a device context, freeing it for use by other applications.
ResetDC	Updates the specified printer or plotter device context using the specified information.
RestoreDC	Restores a device context to the specified state.
SaveDC	Saves the current state of the specified device context by copying data describing selected objects and graphic modes to a context stack.
SelectObject	Selects an object into the specified device context.
SetDCBrushColor	Sets the current device context brush color to the specified color value.
SetDCPenColor	Sets the current device context pen color to the specified color value.
SetLayout	Sets the layout for a device context.
WindowFromDC	Returns a handle to the window associated with a device context.

Device Context Structures

4/20/2022 • 2 minutes to read • Edit Online

The following structures are used with device contexts.

DISPLAY_DEVICE

VIDEOPARAMETERS

Device Context Messages

4/20/2022 • 2 minutes to read • Edit Online

The following message is used with device contexts.

WM_DEVMODECHANGE

WM_DEVMODECHANGE message

4/20/2022 • 2 minutes to read • Edit Online

The **WM_DEVMODECHANGE** message is sent to all top-level windows whenever the user changes device-mode settings.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,
  UINT  uMsg,
  WPARAM wParam,
  LPARAM lParam
);
```

Parameters

hwnd

A handle to a window.

uMsg

WM_DEVMODECHANGE

wParam

This parameter is not used.

lParam

A pointer to a string that specifies the device name.

Return value

An application should return zero if it processes this message.

Remarks

This message cannot be sent directly to a window. To send the **WM_DEVMODECHANGE** message to all top-level windows, use the **SendMessageTimeout** function with the *hWnd* parameter set to HWND_BROADCAST.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

See also

Device Contexts Overview

Device Context Messages

Filled Shapes

4/20/2022 • 2 minutes to read • Edit Online

Filled shapes are geometric forms that are outlined by using the current pen and filled by using the current brush. There are five filled shapes:

- Ellipse
- Chord
- Pie
- Polygon
- Rectangle

This overview describes the filled shapes.

- About Filled Shapes
- Using Filled Shapes
- Filled Shape Reference

About Filled Shapes

4/20/2022 • 2 minutes to read • Edit Online

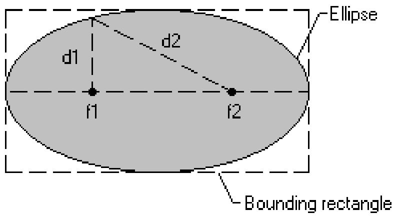
Applications use filled shapes for a variety of tasks. Spreadsheet applications, for example, use filled shapes to construct charts and graphs, and drawing and painting applications use filled shapes to allow the user to draw figures and illustrations.

- Ellipse
- Chord
- Pie
- Polygon
- Rectangle

About Ellipses

4/20/2022 • 2 minutes to read • Edit Online

An ellipse is a closed curve defined by two fixed points (f1 and f2) such that the sum of the distances (d1 +d2) from any point on the curve to the two fixed points is constant. The following illustration shows an ellipse drawn by using the **Ellipse** function.

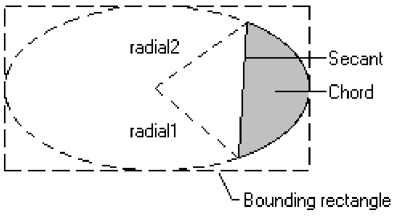


When calling **Ellipse**, an application supplies the coordinates of the upper-left and lower-right corners of the ellipse's bounding rectangle. A *bounding rectangle* is the smallest rectangle completely surrounding the ellipse. When the system draws the ellipse, it excludes the right and lower sides if no world transformations are set. Therefore, for any rectangle measuring x units wide by y units high, the associated ellipse measures x1 units wide by y1 units high. If the application sets a world transformation by calling the **SetWorldTransform** or **ModifyWorldTransform** function, the system includes the right and lower sides.

About Chords

4/20/2022 • 2 minutes to read • Edit Online

A *chord* is a region bounded by the intersection of an ellipse and a line segment called a *secant*. The following illustration shows a chord drawn by using the **Chord** function.



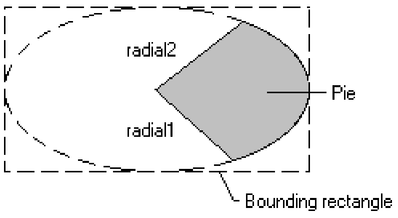
When calling **Chord**, an application supplies the coordinates of the upper-left and lower-right corners of the ellipse's bounding rectangle, as well as the coordinates of two points defining two radials. A radial is a line drawn from the center of an ellipse's bounding rectangle to a point on the ellipse.

When the system draws the curved part of the chord, it does so by using the current arc direction for the specified device context. The default arc direction is counterclockwise. You can have your application reset the arc direction by calling the **SetArcDirection** function.

About Pies

4/20/2022 • 2 minutes to read • Edit Online

A *pie* is a region bounded by the intersection of an ellipse curve and two radials. The following illustration shows a pie drawn by using the **Pie** function.



When calling **Pie**, an application supplies the coordinates of the upper-left and lower-right corners of the ellipse's bounding rectangle, as well as the coordinates of two points defining two radials.

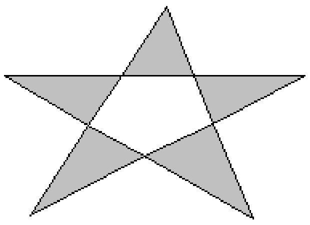
When the system draws the curved part of the pie, it uses the current arc direction for the given device context. The default arc direction is counterclockwise. An application can reset the arc direction by calling the **SetArcDirection** function.

About Polygons

4/20/2022 • 2 minutes to read • Edit Online

A *polygon* is a filled shape with straight sides. The sides of a polygon are drawn by using the current pen. When the system fills a polygon, it uses the current brush and the current polygon fill mode. The two fill modes, alternate (the default) and winding, determine whether regions within a complex polygon are filled or left unpainted. An application can select either mode by calling the **SetPolyFillMode** function. For more information about polygon fill modes, see Regions.

The following illustration shows a polygon drawn by using **Polygon**.



In addition to drawing a single polygon by using **Polygon**, an application can draw multiple polygons by using the **PolyPolygon** function.

Drawing Rectangles

4/20/2022 • 2 minutes to read • Edit Online

A rectangle is a four-sided polygon whose opposing sides are parallel and equal in length. Although an application can draw a rectangle by calling the **Polygon** function, supplying the coordinates of each corner, the **Rectangle** function provides a simpler method. This function requires only the coordinates for the upper-left and the lower-right corners. When an application calls the **Rectangle** function, the system draws the rectangle, excluding the right and lower sides if no world transformation is set for the given device context.

If a world transformation has been set by using the **SetWorldTransform** or **ModifyWorldTransform** function, the system includes the right and lower edges.

In addition to drawing a traditional rectangle, you can draw rectangles with rounded corners. The **RoundRect** function requires that the application supply the coordinates of the lower-left and upper-right corners, as well as the width and height of the ellipse used to round each corner.

Applications can use the following functions to manipulate rectangles.

FUNCTION	DESCRIPTION
FillRect	Repaints the interior of a rectangle.
FrameRect	Redraws the sides of a rectangle.
InvertRect	Inverts the colors that appear within the interior of a rectangle.

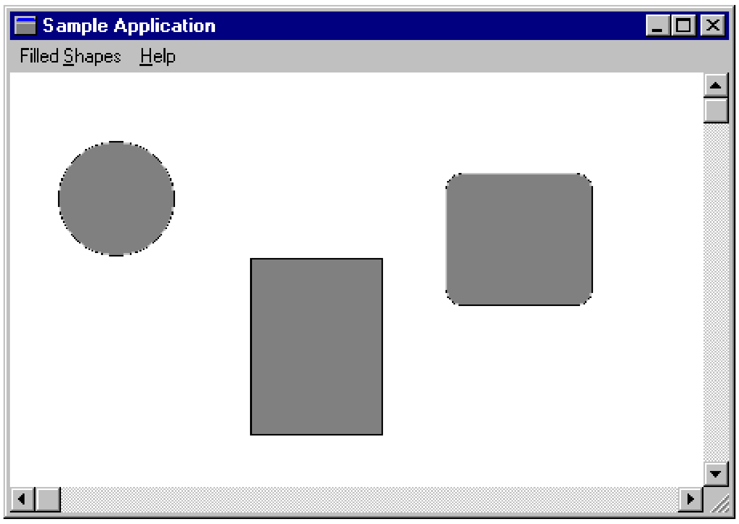
Using Filled Shapes

4/20/2022 • 4 minutes to read • Edit Online

This section illustrates how to use filled shape functions. The example uses the main window procedure from an application that enables the user to draw ellipses, rectangles, and rectangles with rounded corners.

The user draws a filled shape by selecting a particular shape from the menu, positioning the cursor at the upper-left corner of the shape (or the shape's bounding rectangle in the case of an ellipse), and then dragging the mouse until the desired dimensions are obtained.

The following illustration shows three filled shapes drawn using the sample code in this section.



To enable the user to draw filled shapes, include the following popup menu in your application.

```
    POPUP "Filled &Shapes"
    {
       MENUITEM "&Ellipse",   IDM_ELLIPSE
       MENUITEM "&Rectangle", IDM_RECTANGLE
       MENUITEM "R&oundRect", IDM_ROUNDRECT
    }
```

The menu item values in the menu template are constants that you must define as follows in your application's header file.

```
#define IDM_ELLIPSE    1100
#define IDM_RECTANGLE  1200
#define IDM_ROUNDRECT  1300
```

Finally, include the following window procedure in your application.

```
BOOL CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam,
                          LPARAM lParam)
{
    HDC hdc;          // handle to device context (DC)
    PAINTSTRUCT ps;   // paint data for Begin/EndPaint
    POINT ptClientUL; // client area upper left corner
    POINT ptClientLR; // client area lower right corner
    static HDC hdcCompat; // handle to DC for bitmap
    static POINT pt;      // x- and y-coordinates of cursor
    static RECT rcTarget; // rect to receive filled shape
    static RECT rcClient; // client area rectangle
    static BOOL fSizeEllipse; // TRUE if ellipse is sized
    static BOOL fDrawEllipse;   // TRUE if ellipse is drawn
    static BOOL fDrawRectangle; // TRUE if rectangle is drawn
    static BOOL fSizeRectangle; // TRUE if rectangle is sized
    static BOOL fSizeRoundRect; // TRUE if rounded rect is sized
    static BOOL fDrawRoundRect; // TRUE if rounded rect is drawn
    static int nEllipseWidth;   // width for round corners
    static int nEllipseHeight;  // height for round corners

    switch (uMsg)
    {
        case WM_COMMAND:
            switch (wParam)
            {

                // Set the appropriate flag to indicate which
                // filled shape the user is drawing.

                case IDM_ELLIPSE:
                    fSizeEllipse = TRUE;
                break;

                case IDM_RECTANGLE:
                    fSizeRectangle = TRUE;
                break;

                case IDM_ROUNDRECT:
                    fSizeRoundRect = TRUE;
                break;

                default:
                    return DefWindowProc(hwnd, uMsg, wParam,
                         lParam);
            }
            break;


        case WM_CREATE:
            nEllipseWidth = 20;
            nEllipseHeight = 20;

            return 0;

        case WM_PAINT:


            BeginPaint(hwnd, &ps);

            // Because the default brush is white, select
            // a different brush into the device context
            // to demonstrate the painting of filled shapes.

            SelectObject(ps.hdc, GetStockObject(GRAY_BRUSH));

            // If one of the filled shape "draw" flags is TRUE,
            // draw the corresponding shape.

            if (fDrawEllipse)
            {
                Ellipse(ps.hdc, rcTarget.left, rcTarget.top,
                    rcTarget.right, rcTarget.bottom);
                fDrawEllipse = FALSE;
                rcTarget.left = rcTarget.right = 0;
                rcTarget.top = rcTarget.bottom = 0;
            }

            if (fDrawRectangle)
            {
                Rectangle(ps.hdc, rcTarget.left, rcTarget.top,
                    rcTarget.right, rcTarget.bottom);
                fDrawRectangle = FALSE;
                rcTarget.left = rcTarget.right = 0;
                rcTarget.top = rcTarget.bottom = 0;
            }

            if (fDrawRoundRect)
            {
                RoundRect(ps.hdc, rcTarget.left, rcTarget.top,
                    rcTarget.right, rcTarget.bottom,
                    nEllipseWidth, nEllipseHeight);
                fDrawRectangle = FALSE;
                rcTarget.left = rcTarget.right = 0;
                rcTarget.top = rcTarget.bottom = 0;
            }

            EndPaint(hwnd, &ps);
            break;

        case WM_SIZE:

            // Convert the client coordinates of the client area
            // rectangle to screen coordinates and save them in a
            // rectangle. The rectangle is passed to the ClipCursor
            // function during WM_LBUTTONDOWN processing.

            GetClientRect(hwnd, &rcClient);
            ptClientUL.x = rcClient.left;
            ptClientUL.y = rcClient.top;
            ptClientLR.x = rcClient.right;
            ptClientLR.y = rcClient.bottom;
            ClientToScreen(hwnd, &ptClientUL);
            ClientToScreen(hwnd, &ptClientLR);
            SetRect(&rcClient, ptClientUL.x, ptClientUL.y,
                ptClientLR.x, ptClientLR.y);
            return 0;

        case WM_LBUTTONDOWN:

            // Restrict the cursor to the client area.
            // This ensures that the window receives a matching
            // WM_LBUTTONUP message.

            ClipCursor(&rcClient);

            // Save the coordinates of the cursor.

            pt.x = (LONG) LOWORD(lParam);
            pt.y = (LONG) HIWORD(lParam);

            // If the user chooses one of the filled shapes,
            // set the appropriate flag to indicate that the
            // shape is being sized.

             if (fDrawEllipse)
                fSizeEllipse = TRUE;

            return 0;

        case WM_MOUSEMOVE:

            // If one of the "size" flags is set, draw
            // the target rectangle as the user drags
            // the mouse.

            if ((wParam && MK_LBUTTON)
                    && (fSizeEllipse || fSizeRectangle
                    || fSizeRoundRect))
            {

                // Set the mixing mode so that the pen color is the
                // inverse of the background color. The previous
                // rectangle can then be erased by drawing
                // another rectangle on top of it.

                hdc = GetDC(hwnd);
                SetROP2(hdc, R2_NOTXORPEN);

                // If a previous target rectangle exists, erase
                // it by drawing another rectangle on top.

                if (!IsRectEmpty(&rcTarget))
                {
                    Rectangle(hdc, rcTarget.left, rcTarget.top,
                        rcTarget.right, rcTarget.bottom);
                }

                // Save the coordinates of the target rectangle.
                // Avoid invalid rectangles by ensuring that the
                // value of the left coordinate is greater than
                // that of the right, and that the value of the
                // bottom coordinate is greater than that of
                // the top.

                if ((pt.x < (LONG) LOWORD(lParam)) &&
                        (pt.y > (LONG) HIWORD(lParam)))
                {
                    SetRect(&rcTarget, pt.x, HIWORD(lParam),
                        LOWORD(lParam), pt.y);
                }

                else if ((pt.x > (LONG) LOWORD(lParam)) &&
                        (pt.y > (LONG) HIWORD(lParam)))
                {
                    SetRect(&rcTarget, LOWORD(lParam),
                        HIWORD(lParam), pt.x, pt.y);
                }

                else if ((pt.x > (LONG) LOWORD(lParam)) &&
                        (pt.y < (LONG) HIWORD(lParam)))
                {
                    SetRect(&rcTarget, LOWORD(lParam), pt.y,
                        pt.x, HIWORD(lParam));
                }
                else
                {
                    SetRect(&rcTarget, pt.x, pt.y, LOWORD(lParam),
                        HIWORD(lParam));
                }

                // Draw the new target rectangle.

                Rectangle(hdc, rcTarget.left, rcTarget.top,
                    rcTarget.right, rcTarget.bottom);
                ReleaseDC(hwnd, hdc);
            }
            return 0;

        case WM_LBUTTONUP:

            // If one of the "size" flags is TRUE, reset it to FALSE,
            // and then set the corresponding "draw" flag. Invalidate
            // the appropriate rectangle and issue a WM_PAINT message.

            if (fSizeEllipse)
            {
            {
                fSizeEllipse = FALSE;
                fDrawEllipse = TRUE;
            }

            if (fSizeRectangle)
            {
                fSizeRectangle = FALSE;
                fDrawRectangle = TRUE;
            }

            if (fSizeRoundRect)
            {
                fSizeRoundRect = FALSE;
                fDrawRoundRect = TRUE;
            }

            if (fDrawEllipse || fDrawRectangle || fDrawRoundRect)
            {
                InvalidateRect(hwnd, &rcTarget, TRUE);
                UpdateWindow(hwnd);
            }

            // Release the cursor.

            ClipCursor((LPRECT) NULL);
            return 0;

        case WM_DESTROY:

            // Destroy the background brush, compatible bitmap,
            // and bitmap.

            DeleteDC(hdcCompat);
            PostQuitMessage(0);
            break;

        default:
            return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
    return (LRESULT) NULL;
}
```

Filled Shape Reference

4/20/2022 • 2 minutes to read • Edit Online

The following elements are used with filled shapes:

- Filled Shape Functions

Filled Shape Functions

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used with filled shapes.

FUNCTION	DESCRIPTION
Chord	Draws an area bounded by an ellipse and a line segment.
Ellipse	Draws an ellipse.
FillRect	Fills a rectangle using a brush.
FrameRect	Draws a border around a rectangle using a brush.
InvertRect	Inverts the color values of the pixels in a rectangle.
Pie	Draws a pie-shaped wedge bounded by an ellipse and two radials.
Polygon	Draws a polygon.
PolyPolygon	draws a series of closed polygons.
Rectangle	Draws a rectangle.
RoundRect	Draws a rectangle with rounded corners.

Fonts and Text (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

Fonts are used to draw text on video displays and other output devices. The font and text functions enable you to install, select, and query different fonts.

About Fonts

4/20/2022 • 2 minutes to read • Edit Online

To a person trained in the mechanics of manuscript composition or familiar with standard typography, some of the typographic terms used in this overview may be unusual. Most of the differences between standard typography and text on Microsoft Windows reflect changes in technology. The original typographic terms were based on hot-metal composition, whereas the terms used here, which appear as member names for the font and text output structures, reflect a new technology based on laser-printer output and composition performed on a personal computer using desktop publishing software.

The following sections discuss fonts:

- Font Elements
- Font Families
- Raster, Vector, TrueType, and OpenType Fonts
- Character Sets Used by Fonts
- Font Installation and Deletion
- Fonts from Multiple Resource Files
- Font Creation and Selection
- Embedded Fonts

Font Elements

4/20/2022 • 2 minutes to read • Edit Online

A *font* is a collection of characters and symbols that share a common design. The three major elements of this design are referred to as typeface, style, and size.

Typeface

The term typeface refers to specific characteristics of characters and symbols in the font, such as the width of the thick and thin strokes that compose the characters and the presence or absence of serifs. A serif is the short cross line at the ends of an unconnected stroke. A font or typeface without serifs is usually called a sans-serif font.

Style

The term style refers to the weight and slant of a font. Font weights can range from thin to black. The following is a list of possible weights for fonts (from lightest to heaviest):

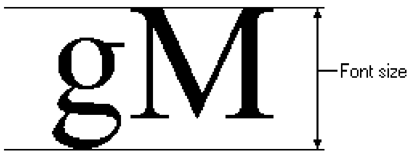
Thin Extralight Light Normal Medium Semibold Bold Extrabold Heavy

Three terms categorize the slant of a font: roman, oblique, and italic.

The characters in a roman font are upright. The characters in an oblique font are artificially slanted. The slant is achieved by performing a shear transformation on the characters from a roman font. The characters in an italic font are truly slanted and appear as they were designed. For more information on shearing, see Coordinate Spaces and Transformations.

Size

The *font size* is an imprecise value. It can generally be determined by measuring the distance from the bottom of a lowercase g to the top of an adjacent uppercase M, as shown in the following illustration.



A font's size is specified in units called points. A point is .013837 of an inch. Following the point system devised by Pierre Simon Fournier, it is common practice to approximate a point as 1/72 inch.

Font Families

4/20/2022 • 2 minutes to read • Edit Online

A *font family* is a set of fonts having common stroke width and serif characteristics. There are five font families. A sixth family allows an application to use the default font. The following table describes the font-families.

FONT FAMILY	DESCRIPTION
Decorative	Specifies a novelty font. An example is Old English.
Dontcare	Specifies a generic family name. This name is used when information about a font does not exist or does not matter. The default font is used.
Modern	Specifies a monospace font with or without serifs. Monospace fonts are usually modern; examples include Pica, Elite, and Courier New.
Roman	Specifies a proportional font with serifs. An example is Times New Roman.
Script	Specifies a font that is designed to look like handwriting; examples include Script and Cursive.
Swiss	Specifies a proportional font without serifs. An example is Arial.

These font families correspond to constants found in the Wingdi.h file: FF_DECORATIVE, FF_DONTCARE, FF_MODERN, FF_ROMAN, FF_SCRIPT, and FF_SWISS. An application uses these constants when it creates a font, selects a font, or retrieves information about a font.

Fonts within a family are distinguished by size (10 point, 24 point, and so on) and style (regular, italic, and so on).

Raster, Vector, TrueType, and OpenType Fonts

4/20/2022 • 2 minutes to read • Edit Online

Applications can use four different kinds of font technologies to display and print text:

- Raster
- Vector
- TrueType
- Microsoft OpenType

The differences between these fonts reflect the way that the *glyph* for each character or symbol is stored in the respective font-resource file:

- In raster fonts, a glyph is a bitmap that the system uses to draw a single character or symbol in the font.
- In vector fonts, a glyph is a collection of line endpoints that define the line segments that the system uses to draw a character or symbol in the font.
- In TrueType and OpenType fonts, a glyph is a collection of line and curve commands as well as a collection of hints.

The system uses the line and curve commands to define the outline of the bitmap for a character or symbol in the TrueType or Microsoft OpenType font. The system uses the hints to adjust the length of the lines and shapes of the curves used to draw the character or symbol. These hints and the respective adjustments are based on the amount of scaling used to reduce or increase the size of the bitmap. An OpenType font is equivalent to a TrueType font except that an OpenType font allows PostScript glyph definitions in addition to TrueType glyph definitions.

Because the bitmaps for each glyph in a raster font are designed for a specific resolution of device, raster fonts are generally considered to be device dependent. Vector fonts, on the other hand, are not device dependent, because each glyph is stored as a collection of scalable lines. However, vector fonts are generally drawn more slowly than raster or TrueType and OpenType fonts. TrueType and OpenType fonts provide both relatively fast drawing speed and true device independence. By using the hints associated with a glyph, a developer can scale the characters from a TrueType or OpenType font up or down and still maintain their original shape.

As previously mentioned, the glyphs for a font are stored in a font-resource file. A font-resource file is actually a DLL that contains only data, there is no code. For raster and vector fonts, this data is divided into two parts: a header describing the font's metrics and the glyph data. A font-resource file for a raster or vector font is identified by the .fon file name extension. For TrueType and OpenType fonts, there are two files for each font: the first file contains a relatively short header and the second contains the actual font data. The first file is identified by an .fot extension and the second is identified by a .ttf extension.

Character Sets Used by Fonts

4/20/2022 • 2 minutes to read • Edit Online

All fonts use a character set. A character set contains punctuation marks, numerals, uppercase and lowercase letters, and all other printable characters. Each element of a character set is identified by a number.

Most character sets in use are supersets of the U.S. ASCII character set, which defines characters for the 96 numeric values from 32 through 127. There are five major groups of character sets:

- Windows
- Unicode
- OEM (original equipment manufacturer)
- Symbol
- Vendor-specific

Windows Character Set

The Windows character set is the most commonly used character set. It is essentially equivalent to the ANSI character set. The blank character is the first character in the Windows character set. It has a hexadecimal value of 0x20 (decimal 32). The last character in the Windows character set has a hexadecimal value of 0xFF (decimal 255).

Many fonts specify a default character. Whenever a request is made for a character that is not in the font, the system provides this default character. Many fonts using the Windows character set specify the period (.) as the default character. TrueType and OpenType fonts typically use an open box as the default character.

Fonts use a break character called a quad to separate words and justify text. Most fonts using the Windows character set specify that the blank character will serve as the break character.

Unicode Character Set

The Windows character set uses 8 bits to represent each character; therefore, the maximum number of characters that can be expressed using 8 bits is 256 (2^8). This is usually sufficient for Western languages, including the diacritical marks used in French, German, Spanish, and other languages. However, Eastern languages employ thousands of separate characters, which cannot be encoded by using a single-byte coding scheme. With the proliferation of computer commerce, double-byte coding schemes were developed so that characters could be represented in 8-bit, 16-bit, 24-bit, or 32-bit sequences. This requires complicated passing algorithms; even so, using different code sets could yield entirely different results on two different computers.

To address the problem of multiple coding schemes, the Unicode standard for data representation was developed. A 16-bit character coding scheme, Unicode can represent 65,536 (2^16) characters, which is enough to include all languages in computer commerce today, as well as punctuation marks, mathematical symbols, and room for expansion. Unicode establishes a unique code for every character to ensure that character translation is always accurate.

OEM Character Set

The OEM character set is typically used in full-screen MS-DOS sessions for screen display. Characters 32 through 127 are usually the same in the OEM, U.S. ASCII, and Windows character sets. The other characters in the OEM character set (0 through 31 and 128 through 255) correspond to the characters that can be displayed in a full-screen MS-DOS session. These characters are generally different from the Windows characters.

Symbol Character Set

The Symbol character set contains special characters typically used to represent mathematical and scientific formulas.

Vendor-Specific Character Sets

Many printers and other output devices provide fonts based on character sets that differ from the Windows and OEM setsfor example, the Extended Binary Coded Decimal Interchange Code (EBCDIC) character set. To use one of these character sets, the printer driver translates from the Windows character set to the vendor-specific character set.

Font Installation and Deletion

4/20/2022 • 2 minutes to read • Edit Online

An application can use a font to draw text only if that font is either resident on a specified device or installed in the system font table. The font table is an internal array that identifies all nondevice fonts that are available to an application. An application can retrieve the names of fonts currently installed on a device or stored in the internal font table by calling the EnumFontFamilies or ChooseFont functions.

To temporarily install a font, call AddFontResource or AddFontResourceEx. These functions load a font that is stored in a font-resource file. However, this is a temporary installation because after a reboot the font will not be present.

To install a font that will remain after the system is rebooted, use one of the following methods:

- Go to the Control Panel, click the **Fonts** icon, and select **Install New Fonts** from the **File** menu. The font is available to an application even before the reboot. However, in a terminal server situation the font is available for the current session but is not available for other sessions until after a reboot.
- Copy the font into the %windir%\fonts folder. Then, either go to the Control Panel and click the **Fonts** icon, or call AddFontResource or AddFontResourceEx. The font is available to an application even before the reboot. However, in a terminal server situation the font is available for the current session but is not available for other sessions until after a reboot. If you only copy the font into the %windir%\fonts folder, the font will be available only after the system is rebooted.

When an application finishes using an installed font, it must remove that font by calling the RemoveFontResource function.

A font installed from a location other than the %windir%\fonts folder cannot be modified when loaded in any active session, including session 0. Any attempt to change, replace, or delete will, therefore, be blocked. If modification to a font is necessary:

- *Temporary fonts* get loaded only in the current session. Before attempting any font modifications, call RemoveFontResource to force the current session to unload the font.
- *Permanent fonts* remain installed after reboot and are loaded by all created sessions. Call RemoveFontResource to force the current session to unload the font. Then, in the font registry key (**HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Fonts**) find and remove the registry value associated with the font. Finally, reboot the machine to ensure the font isn't loaded in any session. After reboot, proceed with your font modification/deletion.

Whenever an application calls the functions that add and delete font resources, it should also call the SendMessage function and send a WM_FONTCHANGE message to all top-level windows in the system. This message notifies other applications that the internal font table has been altered by an application that added or removed a font.

Fonts from Multiple Resource Files

4/20/2022 • 2 minutes to read • Edit Online

Typically, a font is contained in a single font resource file. However, the information for some fonts is spread among several files. For example, Type 1 multiple master fonts require two files:

- .pfm for the font metrics
- .pfb for the font bits

To add a font from multiple files to the system, use the AddFontResource or AddFontResourceEx functions. The *lpszFilename* parameter in these functions must point to a string that contains the file names separated by the vertical bar or pipe (|). For example, to specify abcxxxxx.pfm and abcxxxxx.pfb for a Type 1 font, use the string "abcxxxxx.pfm | abcxxxxx.pfb."

AddFontResourceEx differs from AddFontResource in that the application calling **AddFontResourceEx** can specify the font as private to itself or non-enumerable.

To add a font from a memory image, use AddFontMemResourceEx. This allows an application to use a font that is embedded in a document or a webpage.

To remove a font that came from multiple resource files, call RemoveFontResource or RemoveFontResourceEx, depending on the function used to add the font. You must specify the same flags that were used to add the font. To remove a font that was added from a memory image, use RemoveFontMemResourceEx.

Using a font that comes from multiple font-resource files is identical to using a font from a single resource file.

Font Creation and Selection

4/20/2022 • 4 minutes to read • Edit Online

The **Font** common dialog box simplifies the process of creating and selecting fonts. By initializing the **CHOOSEFONT** structure and calling the **CHOOSEFONT** function, an application can support the same font-selection interface that previously required many lines of custom code. (For more information about the **Font** common dialog box, see Common Dialog Box Library.)

Selection by the User

Most font creation and selection operations involve the user. For example, word processing applications let the user select unique fonts for headings, footnotes, and body text. After the user selects a font by using the **Font** dialog box and presses the **OK** button, the **CHOOSEFONT** function initializes the members of a **LOGFONT** structure with the attributes of the requested font. To use this font for text-output operations, an application must first create a logical font and then select that font into its device context. A *logical font* is an application-supplied description of an ideal font. A developer can create a logical font by calling the **CreateFont** or the **CreateFontIndirect** functions. In this case, the application would call **CreateFontIndirect** and supply a pointer to the **LOGFONT** structure initialized by **CHOOSEFONT**. In general, it is more efficient to call **CreateFontIndirect** because **CreateFont** requires several parameters while **CreateFontIndirect** requires only one pointer to **LOGFONT**.

Before an application can actually begin drawing text with a logical font, it must find the closest match from the fonts stored internally on the device and the fonts whose resources have been loaded into the operating system. The fonts stored on the device or in the operating system are called *physical fonts*. The process of finding the physical font that most closely matches a specified logical font is called font mapping. This process occurs when an application calls the **SelectObject** function and supplies a handle identifying a logical font. Font mapping is performed by using an internal algorithm that compares the attributes of the requested logical font against the attributes of available physical fonts. When the font mapper algorithm completes its search and determines the closest possible match, the **SelectObject** function returns and the application can begin drawing text with the new font.

The **SetMapperFlags** function specifies whether the font mapper algorithm searches only for physical fonts with aspect ratios that match the physical device. The aspect ratio for a device is the ratio formed by the width and the height of a pixel on that device.

The system font (also known as the shell or default font) is the font used for text in the title bars, menus, and dialog boxes.

Special Font Selection Considerations

Although most font selection operations involve the user, there are some instances where this is not true. For example, a developer may want to use a unique font in an application to draw text in a control window. To select an appropriate font, the application must be able to determine what fonts are available, create a logical font that describes one of these available fonts, and then select that font into the appropriate device context.

An application can enumerate the available fonts by using the **EnumFonts** or **EnumFontFamilies** functions. **EnumFontFamilies** is recommended because it enumerates all the styles associated with a family name. This can be useful for fonts with many or unusual styles and for fonts that cross international borders.

Once an application has enumerated the available fonts and located an appropriate match, it should use the values returned by the font enumeration function to initialize the members of a **LOGFONT** structure. Then it can call the CreateFontIndirect function, passing to it a pointer to the initialized LOGFONT structure. If the CreateFontIndirect function is successful, the application can then select the logical font by calling the SelectObject function. When initializing the members of the **LOGFONT** structure, be sure to specify a specific character set in the **lfCharSet** member. This member is important in the font mapping process and the results will be inconsistent if this member is not initialized correctly. If you specify a typeface name in the **lfFaceName** member of the **LOGFONT** structure, make sure that the **lfCharSet** value matches the character set of the typeface specified in **lfFaceName**. For example, if you want to select a font such as MS Mincho, **lfCharSet** must be set to the predefined value SHIFTJIS_CHARSET.

The fonts for many East Asian languages have two typeface names: an English name and a localized name. CreateFont, CreateFontIndirect, and CreateFontIndirectEx take the localized typeface name for a system locale that matches the language, but they take the English typeface name for all other system locales. The best method is to try one name and, on failure, try the other. Note that EnumFonts, EnumFontFamilies, and EnumFontFamiliesEx return the English typeface name if the system locale does not match the language of the font. Starting with Windows 2000, this is no longer a problem because the font mapper for CreateFont, CreateFontIndirect, and CreateFontIndirectEx recognizes either typeface name, regardless of locale.

Embedded Fonts

4/20/2022 • 2 minutes to read • Edit Online

Embedding a font is the technique of bundling a document and the fonts it contains into a file for transmission to another computer. Embedding a font guarantees that a font specified in a transmitted file will be present on the computer receiving the file. Not all fonts can be moved from computer to computer, however, since most fonts are licensed to only one computer at a time. Only TrueType and OpenType fonts can be embedded.

Applications should embed a font in a document only when requested by a user. An application cannot be distributed along with documents that contain embedded fonts, nor can an application itself contain an embedded font. Whenever an application distributes a font, in any format, the proprietary rights of the owner of the font must be acknowledged.

It may be a violation of a font vendor's proprietary rights or user license agreement to embed any fonts where embedding is not permitted or to fail to observe the following guidelines on embedding fonts. A font's license may give only read/write permission for a font to be installed and used on the destination computer. Or the license may give read-only permission. Read-only permission allows a document to be viewed and printed (but not modified) by the destination computer; documents with read-only embedded fonts are themselves read-only. Read-only embedded fonts may not be unbundled from the document and installed on the destination computer.

An application can determine the license status by calling the **GetOutlineTextMetrics** function and examining the **otmfsType** member of the **OUTLINETEXTMETRIC** structure. If bit 1 of **otmfsType** is set, embedding is not permitted for the font. If bit 1 is clear, the font can be embedded. If bit 2 is set, the embedding is read-only.

To embed a TrueType font, an application can use the **GetFontData** function to read the font file. Setting the *dwTable* and *dwOffset* parameters of **GetFontData** to 0L and the *cbData* parameter to 1L ensures that the application reads the entire font file from the beginning.

Several functions are available to embed OpenType fonts depending on the character width and where the font data resides. To embed an OpenType Unicode font that resides in a device context, an application can use **TTEmbedFont**. To embed an OpenType UCS-4 font that resides in a device context, an application can use **TTEmbedFontEx**. To embed an OpenType Unicode font that resides in a font file, an application can use **TTEmbedFontFromFile**. For additional information on OpenType font embedding, see the Font Embedding Reference.

After an application retrieves the font data, it can store the data with the document by using any applicable format. Most applications build a font directory in the document, listing the embedded fonts and whether the embedding is read/write or read-only. An application can use the **otmpStyleName** and **otmFamilyName** members of the **OUTLINETEXTMETRIC** structure to identify the font.

If the read-only bit is set for the embedded font, applications must encrypt the font data before storing it with the document. The encryption method need not be complicated; for example, using the XOR operator to combine the font data with an application-defined constant is adequate and fast.

About Text Output

4/20/2022 • 2 minutes to read • Edit Online

Text output is the most common type of graphic output found within the client area; it is used by applications in different ways. Word processing and desktop publishing applications create documents with formatted text; spreadsheet applications use text, numbers, and symbols to specify formulas, label columns, and list values; database applications create records and display queries with text, and CAD applications use text to label objects and display dimensions.

There are functions to format and draw text in an application's client area and on a page of printer paper. These functions can be divided into two categories: those that format the text (or prepare it for output) and those that actually draw the text. The formatting functions align text, set the intercharacter spacing, set the text and text-background colors, and justify text. The drawing functions draw individual characters (or symbols) or entire strings of text.

When working in Microsoft Windows, hard line breaks are specified with the carriage-return/line feed pair (\r\n).

For more information, see the following topics:

- Formatting Text
- Drawing Text
- Complex Scripts
- ClearType Antialiasing

Formatting Text (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

The formatting functions can be divided into three categories: those that retrieve or set the text-formatting attributes for a device context, those that retrieve character widths, and those that retrieve string widths and heights.

Text-Formatting Attributes

4/20/2022 • 3 minutes to read • Edit Online

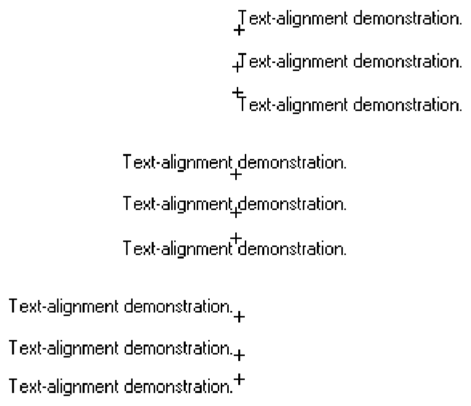
An application can use six functions to set the text-formatting attributes for a device context: SetBkColor, SetBkMode, SetTextAlign, SetTextCharacterExtra, SetTextColor, and SetTextJustification. These functions affect the text alignment, the intercharacter spacing, the text justification, and text and background colors. In addition, six other functions can be used to retrieve the current text formatting attributes for any device context: GetBkColor, GetBkMode, GetTextAlign, GetTextCharacterExtra, GetTextColor, and GetTextExtentPoint32.

Text Alignment

Applications can use the SetTextAlign function to specify how the system should position the characters in a string of text when they call one of the drawing functions. This function can be used to position headings, page numbers, callouts, and so on. The system positions a string of text by aligning a reference point on an imaginary rectangle that surrounds the string, with the current cursor position or with a point passed as an argument to one of the text drawing functions. The **SetTextAlign** function lets the application specify the location of this reference point. The following is a list of the possible reference point locations.

LOCATION	DESCRIPTION
left/bottom	The reference point is located at the lower-left corner of the rectangle.
left/base line	The reference point is located at the intersection of the character-cell base line and the left edge of the rectangle.
left/top	The reference point is located at the top-left corner of the rectangle.
center/bottom	The reference point is located at the center of the bottom of the rectangle.
center/base line	The reference point is located at the intersection of the character-cell base line and the center of the rectangle.
center/top	The reference point is located at the center of the top of the rectangle.
right/bottom	The reference point is located at the lower-right corner of the rectangle.
right/base line	The reference point is located at the intersection of the character-cell base line and the right edge of the rectangle.
right/top	The reference point is located at the top-right corner of the rectangle.

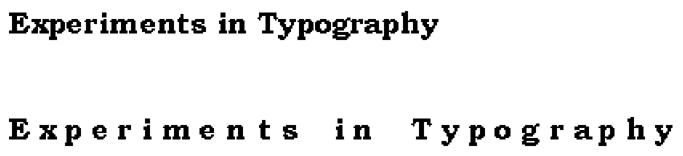
The following illustration shows a string of text drawn by calling the TextOut function. Before drawing the text, the SetTextAlign function was called to relocate the reference point at each one of the nine possible locations.


Text-alignment demonstration.
Text-alignment demonstration.
Text-alignment demonstration.
Text-alignment demonstration.
Text-alignment demonstration.
Text-alignment demonstration.
Text-alignment demonstration.
Text-alignment demonstration.
Text-alignment demonstration.

The default text alignment for a device context is the upper-left corner of the imaginary rectangle that surrounds the text. An application can retrieve the current text-alignment setting for any device context by calling the GetTextAlign function.

Intercharacter Spacing

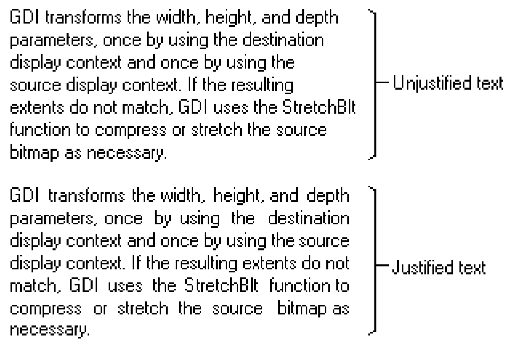
Applications can use the SetTextCharacterExtra function to alter the intercharacter spacing for all text output operations in a specified device context. The following illustration shows a string of text drawn twice by calling the TextOut function. Before drawing the text the second time, the **SetTextCharacterExtra** function was called to increment the intercharacter spacing.


Experiments in Typography
E x p e r i m e n t s   i n   T y p o g r a p h y

The default intercharacter spacing value for any device context is zero. An application can retrieve the current intercharacter spacing value for a device context by calling the GetTextCharacterExtra function.

Text Justification

Applications can use the GetTextExtentPoint32 and SetTextJustification functions to justify a line of text. Text justification is a common operation in any desktop publishing and in most word processing applications. The **GetTextExtentPoint32** function computes the width and height of a string of text. After the width is computed, the application can call the **SetTextJustification** function to distribute extra spacing between each of the words in a line of text. The following illustration shows a paragraph of text printed twice: in the first paragraph, the text was not justified; in the second paragraph, the text was justified by calling the **GetTextExtentPoint32** and **SetTextJustification** functions.


GDI transforms the width, height, and depth parameters, once by using the destination display context and once by using the source display context. If the resulting extents do not match, GDI uses the StretchBlt function to compress or stretch the source bitmap as necessary.
Unjustified text
GDI transforms the width, height, and depth parameters, once by using the destination display context and once by using the source display context. If the resulting extents do not match, GDI uses the StretchBlt function to compress or stretch the source bitmap as necessary.
Justified text

Text and Background Color

Applications can use the SetTextColor function to set the color of text drawn in the client-area of their windows, as well as the color of text drawn on a color printer. An application can use the SetBkColor function to set the color that appears behind each character and the SetBkMode function to specify how the system should blend the selected background color with the current color or colors on the video display.

The default text color for a display device context is black; the default background color is white; and the default background mode is OPAQUE. An application can retrieve the current text color for a device context by calling the GetTextColor function. An application can retrieve the current background color for a device context by calling the GetBkColor function and the current background mode by calling the GetBkMode function.

Character Widths

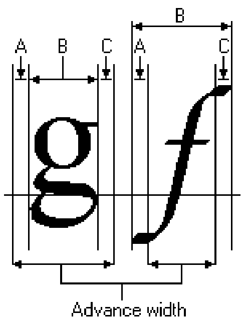
4/20/2022 • 2 minutes to read • Edit Online

Applications need to retrieve character-width data when they perform such tasks as fitting strings of text to page or column widths. There are four functions that an application can use to retrieve character-width data. Two of these functions retrieve the character-advance width and two of these functions retrieve actual character-width data.

An application can use the GetCharWidth32 and GetCharWidthFloat functions to retrieve the advance width for individual characters or symbols in a string of text. The advance width is the distance that the cursor on a video display or the print-head on a printer must advance before printing the next character in a string of text. The **GetCharWidth32** function returns the advance width as an integer value. If greater precision is required, an application can use the **GetCharWidthFloat** function to retrieve fractional advance-width values.

An application can retrieve actual character-width data by using the GetCharABCWidths and **GetCharABCWidthsFloat** functions. The **GetCharABCWidthsFloat** function works with all fonts. The **GetCharABCWidths** function only works with TrueType and OpenType fonts. For more information about TrueType and OpenType fonts, see Raster, Vector, TrueType, and OpenType Fonts.

The following illustration shows the three components of a character width:



The A spacing is the width to add to the current position before placing the character. The B spacing is the width of the character itself. The C spacing is the white space to the right of the character. The total advance width is determined by calculating the sum of A+B+C. The character cell is an imaginary rectangle that surrounds each character or symbol in a font. Because characters can overhang or underhang the character cell, either or both of the A and C increments can be a negative number.

String Widths and Heights

4/20/2022 • 2 minutes to read • Edit Online

In addition to retrieving character-width data for individual characters, applications also need to compute the width and height of entire strings. Two functions retrieve string-width and height measurements: GetTextExtentPoint32, and GetTabbedTextExtent. If the string does not contain tab characters, an application can use the **GetTextExtentPoint32** function to retrieve the width and height of a specified string. If the string contains tab characters, an application should call the GetTabbedTextExtent function.

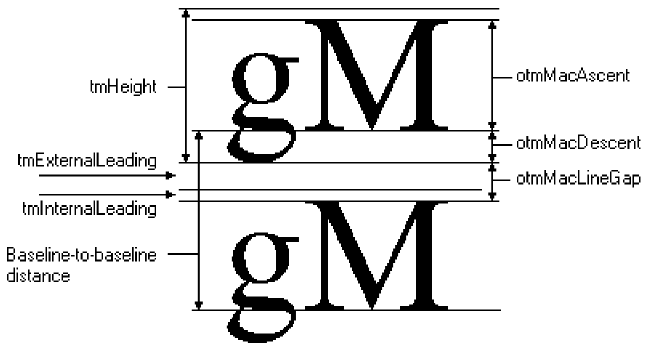
Applications can use the GetTextExtentExPoint function for word-wrapping operations. This function returns the number of characters from a specified string that fit within a specified space.

Font Ascenders and Descenders

Some applications determine the line spacing between text lines of different sizes by using a font's maximum ascender and descender. An application can retrieve these values by calling the GetTextMetrics function and then checking the **tmAscent** and **tmDescent** members of the TEXTMETRIC.

The maximum ascent and descent are different from the typographic ascent and descent. In TrueType and OpenType fonts, the typographic ascent and descent are typically the top of the f glyph and bottom of the g glyph. An application can retrieve the typographic ascender and descender for a TrueType or OpenType font by calling the GetOutlineTextMetrics function and checking the values in the **otmMacAscent** and **otmMacDescent** members of the OUTLINETEXTMETRIC structure.

The following figure shows the difference between the vertical text metric values returned in the NEWTEXTMETRIC and OUTLINETEXTMETRIC structures. (The names beginning with otm are members of the **OUTLINETEXTMETRIC** structure.)



Font Dimensions

An application can retrieve the physical dimensions of a TrueType or OpenType font by calling the GetOutlineTextMetrics function. An application can retrieve the physical dimensions of any other font by calling the GetTextMetrics function. To determine the dimensions of an output device, an application can call the GetDeviceCaps function. **GetDeviceCaps** returns both physical and logical dimensions.

A logical inch is a measure the system uses to present legible fonts on the screen and is approximately 30 to 40 percent larger than a physical inch. The use of logical inches precludes an exact match between the output of the screen and printer. Developers should be aware that the text on a screen is not simply a scaled version of the text that will appear on the page, particularly if graphics are incorporated into the text.

Drawing Text (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

After an application selects the appropriate font, sets the required text-formatting options, and computes the necessary character width and height values for a string of text, it can begin drawing characters and symbols by calling any of the text-output functions:

- DrawText
- DrawTextEx
- ExtTextOut
- PolyTextOut
- TabbedTextOut
- TextOut

When an application calls one of these functions, the operating system passes the call to the graphics engine, which in turn passes the call to the appropriate device driver. At the device driver level, all of these calls are supported by one or more calls to the driver's own ExtTextOut or TextOut function. An application will achieve the fastest execution by calling **ExtTextOut**, which is quickly converted into an **ExtTextOut** call for the device. However, there are instances when an application should call one of the other three functions; for example, to draw multiple lines of text within the borders of a specified rectangular region, it is more efficient to call **DrawText**. To create a multicolumn table with justified columns of text, it is more efficient to call **TabbedTextOut**.

Complex Scripts

4/20/2022 • 2 minutes to read • Edit Online

While the functions discussed in the preceding work well for many languages, they may not deal with the needs of complex scripts. *Complex scripts* are languages whose printed form is not rendered in a simple way. For example, a complex script may allow bidirectional rendering, contextual shaping of glyphs, or combining characters. Due to these special requirements, the control of text output must be very flexible.

Functions that display text TextOut, ExtTextOut, TabbedTextOut, DrawText, and GetTextExtentExPoint have been extended to support complex scripts. In general, this support is transparent to the application. However, applications should save characters in a buffer and display a whole line of text at one time, so that the complex script shaping modules can use context to reorder and generate glyphs correctly. In addition, because the width of a glyph can vary by context, applications should use GetTextExtentExPoint to determine line length rather than using cached character widths.

In addition, complex script-aware applications should consider adding support for right-to-left reading order and right alignment to their applications. You can toggle the reading order or alignment between left and right with the following code:

```
// Save lAlign (this example uses static variables)
static LONG lAlign = TA_LEFT;

// When user toggles alignment (assuming TA_CENTER is not supported).

lAlign = TA_RIGHT;

// When the user toggles reading order.

lAlign = TA_RTLREADING;

// Before calling ExtTextOut, for example, when processing WM_PAINT

SetTextAlign (hDc, lAlign);
```

To toggle both attributes at once, execute the following statement and then call SetTextAlign and ExtTextOut, as shown previously:

```
lAlign = TA_RIGHT^TA_RTLREADING;  //pre-inline !
```

You can also process complex scripts with Uniscribe. Uniscribe is a set of functions that allow a fine degree of control for complex scripts. For more information, see Uniscribe and Processing Complex Scripts.

ClearType Antialiasing

4/20/2022 • 2 minutes to read • Edit Online

Microsoft ClearType antialiasing is a smoothing method that improves font display resolution over traditional antialiasing. It dramatically improves readability on color LCD monitors with a digital interface, such as those in laptops and high-quality flat desktop displays. Readability on CRT screens is also somewhat improved.

However, ClearType is dependent on the orientation and ordering of the LCD stripes. Currently, ClearType is implemented only for LCDs with vertical stripes that are ordered RGB. In particular, this affects tablet PCs, where the display can be oriented in any direction, and those screens that can be turned from landscape to portrait.

ClearType antialiasing is allowed:

- For 16-, 24-, and 32-bit color (disabled for 256 colors or less)
- For screen DC and memory DC (not for printer DC)
- For TrueType fonts and OpenType fonts with TrueType outlines

ClearType antialiasing is disabled:

- Under terminal server client
- For bitmap fonts, vector fonts, device fonts, Type 1 fonts, or Postscript OpenType fonts without TrueType outlines
- If the font has tuned embedded bitmaps, only for those font sizes that contain the embedded bitmaps

To activate ClearType antialiasing, call **SystemParametersInfo** once to turn on font smoothing and then a second time to set the smoothing type to FE_FONTSMOOTHINGCLEARTYPE, as shown in the following code sample:

```
SystemParametersInfo(SPI_SETFONTSMOOTHING,
                     TRUE,
                     0,
                     SPIF_UPDATEINIFILE | SPIF_SENDCHANGE);
SystemParametersInfo(SPI_SETFONTSMOOTHINGTYPE,
                     0,
                     (PVOID)FE_FONTSMOOTHINGCLEARTYPE,
                     SPIF_UPDATEINIFILE | SPIF_SENDCHANGE);
```

You can adjust the appearance of text by changing the contrast value used in the ClearType algorithm. The default is 1,400, but it can be any value from 1,000 to 2,200. Depending on the display device and the user's sensitivity to colors, a higher or lower contrast value may improve readability. To change the contrast, call **SystemParametersInfo** with SPI_SETFONTSMOOTHINGCONTRAST. The following code sets the contrast value to 1,600.

```
SystemParametersInfo(SPI_SETFONTSMOOTHINGCONTRAST,
                     0,
                     (PVOID)1600,
                     SPIF_UPDATEINIFILE | SPIF_SENDCHANGE);
```

You should consider the following details for application compatibility:

- Text rendering with ClearType is slightly slower than with standard antialiasing.
- Applications should not use XOR to display selected text. Applications should set the background color and

redisplay the selected text.

- Applications should not paint the same text on top of itself in transparent mode. If this occurs, the edge pixels that are antialiased will color merge with themselves instead of with the background color. This results in darkened and colorful edges.
- Applications should not paint text by painting the characters individually when in opaque mode because the edge of a character may be clipped by the following character. This occurs because a character that is smoothed with ClearType may have a negative A or C width where the regular character has a positive A or C width. Only the B width of the character is guaranteed to be the same. Likewise, applications should be careful if smoothed text is next to unsmoothed text.
- If an application renders text and then manipulates the bitmap, font smoothing should be turned off by setting the **lfQuality** member of the LOGFONT structure to NONANTIALIASED_QUALITY. For example, a game may add a bitmap shadow effect, or text rendered into a bitmap may be scaled to produce a thumbview.
- If the user is running in portrait mode (that is, monitor striping is horizontal) ClearType antialiasing should be disabled.

The *fdwQuality* parameter in CreateFont and the **lfQuality** member of LOGFONT accept the CLEARTYPE_QUALITY flag. Rasterization of fonts created with this flag will use the ClearType rasterizer. This flag has no effect on previous versions of the operating system.

Using the Font and Text-Output Functions

4/20/2022 • 2 minutes to read • Edit Online

This section describes how you can use the font and text-output functions to draw normal text, draw text from different fonts on the same line, rotate lines of text, display the font-selection common dialog-box, enumerate fonts, and so on:

- Using a stock font to draw text
- Creating a logical font
- Enumerating the installed fonts
- Checking the text capabilities of a device
- Setting the text alignment
- Drawing text from different fonts on the same line
- Rotating lines of text
- Retrieving character outlines
- Using portable TrueType metrics
- Using PANOSE numbers
- Specifying length of text-output string

Using a Stock Font to Draw Text

4/20/2022 • 2 minutes to read • Edit Online

The system provides six stock fonts. A stock font is a logical font that an application can obtain by calling the GetStockObject function and specifying the requested font. The following list contains the values that you can specify to obtain a stock font.

VALUE	MEANING
ANSI_FIXED_FONT	Specifies a monospace font based on the Windows character set. A Courier font is typically used.
ANSI_VAR_FONT	Specifies a proportional font based on the Windows character set. MS Sans Serif is typically used.
DEVICE_DEFAULT_FONT	Specifies the preferred font for the specified device. This is typically the System font for display devices; however, for some dot-matrix printers this is a font that is resident on the device. (Printing with this font is usually faster than printing with a downloaded, bitmap font).
OEM_FIXED_FONT	Specifies a monospace font based on an OEM character set. For IBM computers and compatibles, the OEM font is based on the IBM PC character set.
SYSTEM_FONT	Specifies the System font. This is a proportional font based on the Windows character set, and is used by the operating system to display window titles, menu names, and text in dialog boxes. The System font is always available. Other fonts are available only if they have been installed.
SYSTEM_FIXED_FONT	Specifies a monospace font compatible with the System font in early versions of Windows.

For more information on fonts, see About Fonts.

The following example retrieves a handle to the variable stock font, selects it into a device context, and then writes a string using that font:

```
HFONT hFont, hOldFont;

// Retrieve a handle to the variable stock font.
hFont = (HFONT)GetStockObject(ANSI_VAR_FONT);

// Select the variable stock font into the specified device context.
if (hOldFont = (HFONT)SelectObject(hdc, hFont))
{
    // Display the text string.
    TextOut(hdc, 10, 50, L"Sample ANSI_VAR_FONT text", 25);

    // Restore the original font.
    SelectObject(hdc, hOldFont);
}
```

If other stock fonts are not available, GetStockObject returns a handle to the System font (SYSTEM_FONT). You should use stock fonts only if the mapping mode for your application's device context is MM_TEXT.

Creating a Logical Font

4/20/2022 • 2 minutes to read • Edit Online

You can use the **Font** common dialog box to display available fonts. The **ChooseFont** dialog box is displayed after an application initializes the members of a CHOOSEFONT structure and calls the CHOOSEFONT function. After the user selects one of the available fonts and presses the **OK** button, the **ChooseFont** function initializes a LOGFONT structure with the relevant data. Your application can then call the CreateFontIndirect function and create a logical font based on the user's request. The following example demonstrates how this is done.

```
HFONT FAR PASCAL MyCreateFont( void )
{
    CHOOSEFONT cf;
    LOGFONT lf;
    HFONT hfont;

    // Initialize members of the CHOOSEFONT structure.

    cf.lStructSize = sizeof(CHOOSEFONT);
    cf.hwndOwner = (HWND)NULL;
    cf.hDC = (HDC)NULL;
    cf.lpLogFont = &lf;
    cf.iPointSize = 0;
    cf.Flags = CF_SCREENFONTS;
    cf.rgbColors = RGB(0,0,0);
    cf.lCustData = 0L;
    cf.lpfnHook = (LPCFHOOKPROC)NULL;
    cf.lpTemplateName = (LPSTR)NULL;
    cf.hInstance = (HINSTANCE) NULL;
    cf.lpszStyle = (LPSTR)NULL;
    cf.nFontType = SCREEN_FONTTYPE;
    cf.nSizeMin = 0;
    cf.nSizeMax = 0;

    // Display the CHOOSEFONT common-dialog box.

    ChooseFont(&cf);

    // Create a logical font based on the user's
    // selection and return a handle identifying
    // that font.

    hfont = CreateFontIndirect(cf.lpLogFont);
    return (hfont);
}
```

Enumerating the Installed Fonts

4/20/2022 • 3 minutes to read • Edit Online

In some instances, an application must be able to enumerate the available fonts and select the one most appropriate for a particular operation. An application can enumerate the available fonts by calling the EnumFonts or EnumFontFamilies function. These functions send information about the available fonts to a callback function that the application supplies. The callback function receives information in LOGFONT and NEWTEXTMETRIC structures. (The **NEWTEXTMETRIC** structure contains information about a TrueType font. When the callback function receives information about a non-TrueType font, the information is contained in a TEXTMETRIC structure.) By using this information, an application can limit the user's choices to only those fonts that are available.

The **EnumFontFamilies** function is similar to the **EnumFonts** function but includes some extra functionality. **EnumFontFamilies** allows an application to take advantage of styles available with TrueType fonts. New and upgraded applications should use **EnumFontFamilies** instead of **EnumFonts**.

TrueType fonts are organized around a typeface name (for example, Courier New) and style names (for example, italic, bold, and extra-bold). The **EnumFontFamilies** function enumerates all the styles associated with a specified family name, not simply the bold and italic attributes. For example, when the system includes a TrueType font called Courier New Extra-Bold, **EnumFontFamilies** lists it with the other Courier New fonts. The capabilities of **EnumFontFamilies** are helpful for fonts with many or unusual styles and for fonts that cross international borders.

If an application does not supply a typeface name, the EnumFonts and EnumFontFamilies functions supply information about one font in each available family. To enumerate all the fonts in a device context, the application can specify **NULL** for the typeface name, compile a list of the available typefaces, and then enumerate each font in each typeface.

The following example uses the **EnumFontFamilies** function to retrieve the number of available raster, vector, and TrueType font families.

```
    UINT uAlignPrev;
    int aFontCount[] = { 0, 0, 0 };
    char szCount[8];
        HRESULT hr;
        size_t * pcch;

    EnumFontFamilies(hdc, (LPCTSTR) NULL,
        (FONTENUMPROC) EnumFamCallBack, (LPARAM) aFontCount);

    uAlignPrev = SetTextAlign(hdc, TA_UPDATECP);

    MoveToEx(hdc, 10, 50, (LPPOINT)NULL);
    TextOut(hdc, 0, 0, "Number of raster fonts: ", 24);
    itoa(aFontCount[0], szCount, 10);

        hr = StringCchLength(szCount, 9, pcch);
        if (FAILED(hr))
        {
        // TODO: write error handler
        }
    TextOut(hdc, 0, 0, szCount, *pcch);

    MoveToEx(hdc, 10, 75, (LPPOINT)NULL);
    TextOut(hdc, 0, 0, "Number of vector fonts: ", 24);
    itoa(aFontCount[1], szCount, 10);
        hr = StringCchLength(szCount, 9, pcch);
        if (FAILED(hr))
        {
        // TODO: write error handler
        }
    TextOut(hdc, 0, 0, szCount, *pcch);

    MoveToEx(hdc, 10, 100, (LPPOINT)NULL);
    TextOut(hdc, 0, 0, "Number of TrueType fonts: ", 26);
    itoa(aFontCount[2], szCount, 10);
        hr = StringCchLength(szCount, 9, pcch);
        if (FAILED(hr))
        {
        // TODO: write error handler
        }
    TextOut(hdc, 0, 0, szCount, *pcch);

    SetTextAlign(hdc, uAlignPrev);

BOOL CALLBACK EnumFamCallBack(LPLOGFONT lplf, LPNEWTEXTMETRIC lpntm, DWORD FontType, LPVOID aFontCount)
{
    int far * aiFontCount = (int far *) aFontCount;

    // Record the number of raster, TrueType, and vector
    // fonts in the font-count array.

    if (FontType & RASTER_FONTTYPE)
        aiFontCount[0]++;
    else if (FontType & TRUETYPE_FONTTYPE)
        aiFontCount[2]++;
    else
        aiFontCount[1]++;

    if (aiFontCount[0] || aiFontCount[1] || aiFontCount[2])
        return TRUE;
    else
        return FALSE;

    UNREFERENCED_PARAMETER( lplf );
    UNREFERENCED_PARAMETER( lpntm );
}
```

This example uses two masks, RASTER_FONTTYPE and TRUETYPE_FONTTYPE, to determine the type of font being enumerated. If the RASTER_FONTTYPE bit is set, the font is a raster font. If the TRUETYPE_FONTTYPE bit is set, the font is a TrueType font. If neither bit is set, the font is a vector font. A third mask, DEVICE_FONTTYPE, is set when a device (for example, a laser printer) supports downloading TrueType fonts; it is zero if the device is a display adapter, dot-matrix printer, or other raster device. An application can also use the DEVICE_FONTTYPE mask to distinguish GDI-supplied raster fonts from device-supplied fonts. The system can simulate bold, italic, underline, and strikeout attributes for GDI-supplied raster fonts, but not for device-supplied fonts.

An application can also check bits 1 and 2 in the **tmPitchAndFamily** member of the NEWTEXTMETRIC structure to identify a TrueType font. If bit 1 is 0 and bit 2 is 1, the font is a TrueType font.

Vector fonts are categorized as OEM_CHARSET instead of ANSI_CHARSET. Some applications identify vector fonts by using this information, checking the **tmCharSet** member of the **NEWTEXTMETRIC** structure. This categorization usually prevents the font mapper from choosing vector fonts unless they are specifically requested. (Most applications no longer use vector fonts because their strokes are single lines and they take longer to draw than TrueType fonts, which offer many of the same scaling and rotation features that required vector fonts.)

Checking the Text Capabilities of a Device

4/20/2022 • 2 minutes to read • Edit Online

You can use the EnumFonts and EnumFontFamilies functions to enumerate the fonts that are available in a printer-compatible memory device context. You can also use the GetDeviceCaps function to retrieve information about the text capabilities of a device. By calling the **GetDeviceCaps** function with the NUMFONTS index, you can determine the minimum number of fonts supported by a printer. (An individual printer may support more fonts than specified in the return value from **GetDeviceCaps** with the NUMFONTS index.) By using the TEXTCAPS index, you can identify many of the text capabilities of the specified device.

Setting the Text Alignment

4/20/2022 • 2 minutes to read • Edit Online

You can query and set the text alignment for a device context by using the GetTextAlign and SetTextAlign functions. The text-alignment settings determine how text is positioned relative to a specified location. Text can be aligned to the right or left of the position or centered over it; it can also be aligned above or below the point.

The following example shows a method for determining which horizontal alignment flag is set:

```
switch ((TA_LEFT | TA_RIGHT | TA_CENTER) & GetTextAlign(hdc))
{
    case TA_LEFT:
       .
       .
       .
    case TA_RIGHT:
       .
       .
       .
    case TA_CENTER:
       .
       .
       .
}
```

You can also use the SetTextAlign function to update the current position when a text-output function is called. For instance, the following example uses the SetTextAlign function to update the current position when the TextOut function is called. In this example, the *cArial* parameter is an integer that specifies the number of Arial fonts.

```
UINT uAlignPrev;
char szCount[8];
HRESULT hr;
size_t * pcch;

uAlignPrev = SetTextAlign(hdc, TA_UPDATECP);
MoveToEx(hdc, 10, 50, (LPPOINT) NULL);
TextOut(hdc, 0, 0, "Number of Arial fonts: ", 23);
itoa(cArial, szCount, 10);

hr = StringCchLength(szCount, 9, pcch);
if (FAILED(hr))
{
// TODO: write error handler
}

TextOut(hdc, 0, 0, (LPSTR) szCount, *pcch);
SetTextAlign(hdc, uAlignPrev);
```

NOTE

You should not use SetTextAlign with TA_UPDATECP when you are using ScriptStringOut, because selected text is not rendered correctly. If you must use this flag, you can unset and reset it as necessary to avoid the problem.

Drawing Text from Different Fonts on the Same Line

4/20/2022 • 5 minutes to read • Edit Online

Different type styles within a font family can have different widths. For example, bold and italic styles of a family are always wider than the roman style for a specified point size. When you display or print several type styles on a single line, you must keep track of the width of the line to avoid having characters displayed or printed on top of one another.

You can use two functions to retrieve the width (or extent) of text in the current font. The GetTabbedTextExtent function computes the width and height of a character string. If the string contains one or more tab characters, the width of the string is based upon a specified array of tab-stop positions. The GetTextExtentPoint32 function computes the width and height of a line of text.

When necessary, the system synthesizes a font by changing the character bitmaps. To synthesize a character in a bold font, the system draws the character twice: at the starting point, and again one pixel to the right of the starting point. To synthesize a character in an italic font, the system draws two rows of pixels at the bottom of the character cell, moves the starting point one pixel to the right, draws the next two rows, and continues until the character has been drawn. By shifting pixels, each character appears to be sheared to the right. The amount of shear is a function of the height of the character.

One way to write a line of text that contains multiple fonts is to use the **GetTextExtentPoint32** function after each call to TextOut and add the length to a current position. The following example writes the line "This is a sample string." using bold characters for "This is a", switches to italic characters for "sample", then returns to bold characters for "string." After printing all the strings, it restores the system default characters.

```
int XIncrement;
int YStart;
TEXTMETRIC tm;
HFONT hfntDefault, hfntItalic, hfntBold;
SIZE sz;
LPSTR lpszString1 = "This is a ";
LPSTR lpszString2 = "sample ";
LPSTR lpszString3 = "string.";
HRESULT hr;
size_t * pcch;

// Create a bold and an italic logical font.

hfntItalic = MyCreateFont();
hfntBold = MyCreateFont();


// Select the bold font and draw the first string
// beginning at the specified point (XIncrement, YStart).

XIncrement = 10;
YStart = 50;
hfntDefault = SelectObject(hdc, hfntBold);
hr = StringCchLength(lpszString1, 11, pcch);
        if (FAILED(hr))
        {
        // TODO: write error handler
        }
TextOut(hdc, XIncrement, YStart, lpszString1, *pcch);

// Compute the length of the first string and add
// this value to the x-increment that is used for the
// text-output operation.
// text-output operation.

hr = StringCchLength(lpszString1, 11, pcch);
        if (FAILED(hr))
        {
        // TODO: write error handler
        }
GetTextExtentPoint32(hdc, lpszString1, *pcch, &sz);
XIncrement += sz.cx;

// Retrieve the overhang value from the TEXTMETRIC
// structure and subtract it from the x-increment.
// (This is only necessary for non-TrueType raster
// fonts.)

GetTextMetrics(hdc, &tm);
XIncrement -= tm.tmOverhang;

// Select an italic font and draw the second string
// beginning at the point (XIncrement, YStart).

hfntBold = SelectObject(hdc, hfntItalic);
GetTextMetrics(hdc, &tm);
XIncrement -= tm.tmOverhang;
hr = StringCchLength(lpszString2, 8, pcch);
        if (FAILED(hr))
        {
        // TODO: write error handler
        }
TextOut(hdc, XIncrement, YStart, lpszString2, *pcch);

// Compute the length of the second string and add
// this value to the x-increment that is used for the
// text-output operation.

hr = StringCchLength(lpszString2, 8, pcch);
        if (FAILED(hr))
        {
        // TODO: write error handler
        }
GetTextExtentPoint32(hdc, lpszString2, *pcch, &sz);
XIncrement += sz.cx;

// Reselect the bold font and draw the third string
// beginning at the point (XIncrement, YStart).

SelectObject(hdc, hfntBold);
hr = StringCchLength(lpszString3, 8, pcch);
        if (FAILED(hr))
        {
        // TODO: write error handler
        }
TextOut(hdc, XIncrement - tm.tmOverhang, YStart, lpszString3,
            *pcch);

// Reselect the original font.

SelectObject(hdc, hfntDefault);

// Delete the bold and italic fonts.

DeleteObject(hfntItalic);
DeleteObject(hfntBold);
```

In this example, the GetTextExtentPoint32 function initializes the members of a SIZE structure with the length and height of the specified string. The GetTextMetrics function retrieves the overhang for the current font. Because the overhang is zero if the font is a TrueType font, the overhang value does not change the string placement. For raster fonts, however, it is important to use the overhang value.

The overhang is subtracted from the bold string once, to bring subsequent characters closer to the end of the string if the font is a raster font. Because overhang affects both the beginning and end of the italic string in a raster font, the glyphs start at the right of the specified location and end at the left of the endpoint of the last character cell. (The **GetTextExtentPoint32** function retrieves the extent of the character cells, not the extent of the glyphs.) To account for the overhang in the raster italic string, the example subtracts the overhang before placing the string and subtracts it again before placing subsequent characters.

The SetTextJustification function adds extra space to the break characters in a line of text. You can use the GetTextExtentPoint function to determine the extent of a string, then subtract that extent from the total amount of space the line should occupy, and use the **SetTextJustification** function to distribute the extra space among the break characters in the string. The SetTextCharacterExtra function adds extra space to every character cell in the selected font, including the break character. (You can use the GetTextCharacterExtra function to determine the current amount of extra space being added to the character cells; the default setting is zero.)

You can place characters with greater precision by using the GetCharWidth32 or GetCharABCWidths function to retrieve the widths of individual characters in a font. The **GetCharABCWidths** function is more accurate than the **GetCharWidth32** function, but it can only be used with TrueType fonts.

ABC spacing also allows an application to perform very accurate text alignment. For example, when the application right aligns a raster roman font without using ABC spacing, the advance width is calculated as the character width. This means the white space to the right of the glyph in the bitmap is aligned, not the glyph itself. By using ABC widths, applications have more flexibility in the placement and removal of white space when aligning text, because they have information that allows them to finely control intercharacter spacing.

Rotating Lines of Text

4/20/2022 • 2 minutes to read • Edit Online

You can rotate TrueType fonts at any angle. This is useful for labeling charts and other illustrations. The following example rotates a string in 10-degree increments around the center of the client area by changing the value of the **lfEscapement** and **lfOrientation** members of the LOGFONT structure used to create the font.

```
#include "strsafe.h"
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {

    case WM_PAINT:
        {
        hdc = BeginPaint(hWnd, &ps);
        RECT rc;
        int angle;
        HGDIOBJ hfnt, hfntPrev;
        WCHAR lpszRotate[22] = TEXT("String to be rotated.");
        HRESULT hr;
        size_t pcch = 22;

// Allocate memory for a LOGFONT structure.

PLOGFONT plf = (PLOGFONT) LocalAlloc(LPTR, sizeof(LOGFONT));


// Specify a font typeface name and weight.

hr = StringCchCopy(plf->lfFaceName, 6, TEXT("Arial"));
if (FAILED(hr))
{
// TODO: write error handler
}

plf->lfWeight = FW_NORMAL;

// Retrieve the client-rectangle dimensions.

GetClientRect(hWnd, &rc);

// Set the background mode to transparent for the
// text-output operation.

SetBkMode(hdc, TRANSPARENT);

// Draw the string 36 times, rotating 10 degrees
// counter-clockwise each time.

for (angle = 0; angle < 3600; angle += 100)
{
    plf->lfEscapement = angle;
    hfnt = CreateFontIndirect(plf);
    hfntPrev = SelectObject(hdc, hfnt);

    //
    // The StringCchLength call is fitted to the lpszRotate string
    // The StringCchLength call is fitted to the lpszRotate string
    //
    hr = StringCchLength(lpszRotate, 22, &pcch);
    if (FAILED(hr))
    {
    // TODO: write error handler
    }
    TextOut(hdc, rc.right / 2, rc.bottom / 2,
        lpszRotate, pcch);
    SelectObject(hdc, hfntPrev);
    DeleteObject(hfnt);
}

// Reset the background mode to its default.

SetBkMode(hdc, OPAQUE);

// Free the memory allocated for the LOGFONT structure.

LocalFree((LOCALHANDLE) plf);
        EndPaint(hWnd, &ps);
        break;
        }
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}
```

Retrieving Character Outlines

4/20/2022 • 2 minutes to read • Edit Online

You can use the GetGlyphOutline function to retrieve the outline of a glyph from a TrueType font. The glyph outline returned by the **GetGlyphOutline** function is for a grid-fitted glyph. (A grid-fitted glyph has been modified so that its bitmap image conforms as closely as possible to the original design of the glyph.) If your application requires an unmodified glyph outline, request the glyph outline for a character in a font whose size is equal to the font's em units. (To create a font with this size, set the **lfHeight** member of the LOGFONT structure to the negative of the value of the **ntmSizeEM** member of the NEWTEXTMETRIC structure.)

GetGlyphOutline returns the outline as a bitmap or as a series of polylines and splines. When an application retrieves a glyph outline as a series of polylines and splines, the information is returned in a TTPOLYGONHEADER structure followed by as many TTPOLYCURVE structures as required to describe the glyph. All points are returned as POINTFX structures and represent absolute positions, not relative moves. The starting point specified by the **pfxStart** member of the **TTPOLYGONHEADER** structure is the point where the outline for a contour begins. The **TTPOLYCURVE** structures that follow can be either polyline records or spline records.

To render a TrueType character outline, you must use both the polyline and the spline records. The system can render both polylines and splines easily. Each polyline and spline record contains as many sequential points as possible, to minimize the number of records returned.

The starting point specified in the **TTPOLYGONHEADER** structure is always on the outline of the glyph. The specified point serves as both the starting and ending points for the contour.

This section provides information on the following topics.

- Polyline Records
- Spline Records

Polyline Records

4/20/2022 • 2 minutes to read • Edit Online

Polyline records are a series of points; lines drawn between the points describe the outline of the character. A polyline record begins with the last point in the previous record (or, for the first record in the contour, the starting point). Each point in the record is on the glyph outline and can be connected simply by using straight lines.

Spline Records

4/20/2022 • 2 minutes to read • Edit Online

Spline records represent the quadratic curves (that is, quadratic b-splines) used by TrueType. A spline record begins with the last point in the previous record (or for the first record in the contour, with the starting point). For the first spline record, the starting point and the last point in the record are on the glyph outline. For all other spline records, only the last point is on the glyph outline. All other points in the spline records are off the glyph outline and must be rendered as the control points of b-splines.

The last spline or polyline record in a contour always ends with the contour's starting point. This arrangement ensures that every contour is closed.

Because b-splines require three points (one point off the glyph outline between two points that are on the outline), you must perform some calculations when a spline record contains more than one off-curve point.

For example, if a spline record contains three points (A, B, and C) and it is not the first record, points A and B are off the glyph outline. To interpret point A, use the current position (which is always on the glyph outline) and the point on the glyph outline between points A and B. To find the midpoint (M) between A and B, you can perform the following calculation.

M = A + (B A)/2

The midpoint between consecutive off-outline points in a spline record is a point on the glyph outline, according to the definition of the spline format used in TrueType fonts.

If the current position is designated by P, the two quadratic splines defined by this spline record are (P, A, M) and (M, B, C).

Using Portable TrueType Metrics

4/20/2022 • 2 minutes to read • Edit Online

Applications that use the TrueType text metrics can achieve a high degree of printer and document portability; they can use TrueType metrics even if they must maintain compatibility with early 16-bit versions of Windows.

Design widths overcome most of the problems of device-dependent text introduced by physical devices. Design widths are a kind of logical width. Independent of any rasterization problems or scaling transformations, each glyph has a logical width and height. Composed to a logical page, each character in a string has a place that is independent of the physical device widths. Although a logical width implies that widths can be scaled linearly at all point sizes, this is not necessarily true for either nonportable or most TrueType fonts. At smaller point sizes, some glyphs are made wider relative to their height for better readability.

The characters in TrueType core fonts are designed against a 2048 by 2048 grid. The design width is the width of a character in these grid units. (TrueType supports any integer grid size up to 16,384 by 16,384; grid sizes that are integer powers of 2 scale faster than other grid sizes.)

The font outline is designed in notional units. The em square is the notional grid against which the font outline is fitted. (You can use the **otmEMSquare** member of OUTLINETEXTMETRIC and the **ntmSizeEM** member of NEWTEXTMETRIC to retrieve the size of the em square in notional units.) When a font is created that has a point size (in device units) equal to the size of its em square, the ABC widths for this font are the desired design widths. For example, assume the size of an em square is 1000 and the ABC widths of a character in the font are 150, 400, and 150. A character in this font that is 10 device units high would have ABC widths of 1.5, 4, and 1.5, respectively. Since the MM_TEXT mapping mode is most commonly used with fonts (and MM_TEXT is equivalent to device units), this is a simple calculation.

Because of the high resolution of TrueType design widths, applications that use them must take into account the large numeric values that can be created. For more information, see the following topics:

- Device vs. Design Units
- Metrics for Portable Documents

Device vs. Design Units

4/20/2022 • 2 minutes to read • Edit Online

An application can retrieve font metrics for a physical font only after the font has been selected into a device context. When a font is selected into a device context, it is scaled for the device. The font metrics specific to the device are known as device units.

Portable metrics in fonts are known as design units. To apply to a specified device, design units must be converted to device units. Use the following formula to convert design units to device units.

DeviceUnits = (*DesignUnits*/ *unitsPerEm*) * (*PointSize*/72) * *DeviceResolution*

The variables in this formula have the following meanings.

VARIABLE	DESCRIPTION
DeviceUnits	Specifies the *DesignUnits* font metric converted to device units. This value is in the same units as the value specified for *DeviceResolution*.
DesignUnits	Specifies the font metric to be converted to device units. This value can be any font metric, including the width of a character or the ascender value for an entire font.
unitsPerEm	Specifies the em square size for the font.
PointSize	Specifies size of the font, in points. (One point equals 1/72 of an inch.)
DeviceResolution	Specifies number of device units (pixels) per inch. Typical values might be 300 for a laser printer or 96 for a VGA screen.

This formula should not be used to convert device units back to design units. Device units are always rounded to the nearest pixel. The propagated round-off error can become very large, especially when an application is working with screen sizes.

To request design units, create a logical font whose height is specified as *unitsPerEm*. Applications can retrieve the value for *unitsPerEm* by calling the **EnumFontFamilies** function and checking the **ntmSizeEM** member of the **NEWTEXTMETRIC** structure.

Metrics for Portable Documents

4/20/2022 • 2 minutes to read • Edit Online

The following table specifies the most important font metrics for applications that require portable documents and the functions that allow an application to retrieve them.

FUNCTION	METRIC	USE
EnumFontFamilies	**ntmSizeEM**	Retrieval of design metrics; conversion to device metrics.
GetCharABCWidths	**ABCWidths**	Accurate placement of characters at the start and end of margins, picture boundaries, and other text breaks.
GetCharWidth32	**AdvanceWidths**	Placement of characters on a line.
GetOutlineTextMetrics	**otmfsType**	Font-embedding bits.
	otmsCharSlopeRise	Y-component for slope of cursor for italic fonts.
	otmsCharSlopeRun	X-component for slope of cursor for italic fonts.
	otmAscent	Line spacing.
	otmDescent	Line spacing.
	otmLineGap	Line spacing.
	otmpFamilyName	Font identification.
	otmpStyleName	Font identification.
	otmpFullName	Font identification (typically, family and style name).

The **otmsCharSlopeRise**, **otmsCharSlopeRun**, **otmAscent**, **otmDescent**, and **otmLineGap** members of the OUTLINETEXTMETRIC structure are scaled or transformed to correspond to the current device mode and physical height (as specified in the **tmHeight** member of the NEWTEXTMETRIC structure).

Font identification is important in those instances when an application must select the same font, for example, when a document is reopened or moved to a different operating system. The font mapper always selects the correct font when an application requests a font by full name. The family and style names provide input to the standard font dialog box, which ensures that the selection bars are properly placed.

The **otmsCharSlopeRise** and **otmsCharSlopeRun** values are used to produce a close approximation of the main italic angle of the font. For typical roman fonts, **otmsCharSlopeRise** is 1 and **otmsCharSlopeRun** is 0. For italic fonts, the values attempt to approximate the sine and cosine of the main italic angle of the font (in counterclockwise degrees past vertical); note that the italic angle for upright fonts is 0. Because these values are not expressed in design units, they should not be converted into device units.

The character placement and line spacing metrics enable an application to compute device-independent line breaks that are portable across screens, printers, typesetters, and even platforms.

To perform device-independent page layout

1. Normalize all design metrics to a common ultra-high resolution (UHR) value (for example, 65,536 DPI); this prevents round-off errors.
2. Compute line breaks based on UHR metrics and physical page width; this yields a starting point and an ending point of a line within the text stream.
3. Compute the device page width in device units (for example, pixels).
4. Fit each line of text into the device page width, using the line breaks computed in step 2.
5. Compute page breaks by using UHR metrics and the physical page length; this yields the number of lines per page.
6. Compute the line heights in device units.
7. Fit the lines of text onto the page, using the lines per page from step 5 and the line heights from step 6.

If all applications adopt these techniques, developers can virtually guarantee that documents moved from one application to another will retain their original appearance and format.

Using PANOSE Numbers

4/20/2022 • 2 minutes to read • Edit Online

TrueType font files include PANOSE numbers, which applications can use to choose a font that closely matches their specifications. The PANOSE system classifies faces by 10 different attributes. For more information about these attributes, see **PANOSE**. A **PANOSE** structure is part of the **OUTLINETEXTMETRIC** structure (whose values are filled in by calling the **GetOutlineTextMetrics** function).

The PANOSE attributes are rated individually on a scale. The resulting values are concatenated to produce a number. Given this number for a font and a mathematical metric to measure distances in the PANOSE space, an application can determine the nearest neighbors.

Specifying length of text-output string

4/20/2022 • 2 minutes to read • Edit Online

Several of the font and text-output functions have a parameter that specifies the length of the text-output string. A typical example is the *cchText* parameter of **DrawTextEx**.

Each of these functions has both an "ANSI" version and a Unicode version (for example, **DrawTextExA** and **DrawTextExW**, respectively). For the "ANSI" version of each function, the length is specified as a BYTE count and for the Unicode function it is specified as a WORD count.

It is traditional to think of this as a "character count". That is generally accurate for many languages, including English, but it is not accurate in general. In "ANSI" strings, characters in SBCS code pages take one byte each, but most characters in DBCS code pages take two bytes. Similarly, most currently defined Unicode characters reside in the Basic Multilingual Plane (BMP) and their UTF-16 representations fit in one WORD, but supplementary characters are represented in Unicode by ''surrogates'', which require two WORDs.

Each of these functions takes a length count. For the "ANSI" version of each function, the length is specified as a BYTE count length of a string not including the **NULL** terminator. For the Unicode function, the length count is the byte count divided by sizeof(WCHAR), which is 2, not including the **NULL** terminator. The character count is the count of the number of characters, which might not equal the length count of the string. In some instances, characters take more than one BYTE for ANSI (for example, DBCS character) and more than one WORD for Unicode (for example, surrogate characters). Further, the number of glyphs might not equal the number of characters because multiple characters might be composited to make one glyph. Length count is the amount of data. Character count is the number of units that are processed as one entity. Glyphs are what gets rendered. For example, in Unicode, you can have a string with length of 3, which is 2 characters and which results in 1 glyph being rendered. However, typically, most Unicode strings length, character count, and number of rendered glyphs are equal.

You can use _tcslen() to get the string length. For ANSI, _tcslen() returns the number of bytes. For Unicode, _tcslen() returns the number of WCHARs (that is, WORDs).

Special processing characters like tabs and soft hyphens that aren't always drawn can affect the drawn output. They get included in the string length and character counts, but might not be directly represented by a rendered glyph.

Some of these functions allow the caller to specify the length as -1 to indicate that the string is null-terminated; in that case the function will compute the character count automatically. Not all of the functions offer this capability. That is specified on a function-by-function basis; see the individual function documentation.

Font and Text Reference

4/20/2022 • 2 minutes to read • Edit Online

The following elements are used with fonts and text:

- Font and Text Functions
- Font and Text Structures
- Font and Text Macros
- Font and Text Messages
- Font-Package Function Error Messages

Font and Text Functions (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used with fonts and text.

FUNCTION	DESCRIPTION
AddFontMemResourceEx	Adds an embedded font to the system font table.
AddFontResource	Adds a font resource to the system font table.
AddFontResourceEx	Adds a private or non-enumerable font to the system font table.
CreateFont	Creates a logical font.
CreateFontIndirect	Creates a logical font from a structure.
CreateFontIndirectEx	Creates a logical font from a structure.
DrawText	Draws formatted text in a rectangle.
DrawTextEx	Draws formatted text in rectangle.
EnumFontFamExProc	An application definedcallback function used with EnumFontFamiliesEx to process fonts.
EnumFontFamiliesEx	Enumerates all fonts in the system with certain characteristics.
ExtTextOut	Draws a character string.
GetAspectRatioFilterEx	Gets the setting for the aspect-ratio filter.
GetCharABCWidths	Gets the widths of consecutive characters from the TrueType font.
GetCharABCWidthsFloat	Gets the widths of consecutive characters from the current font.
GetCharABCWidthsI	Gets the widths of consecutive glyph indices or from an array of glyph indices from the TrueType font.
GetCharacterPlacement	Gets information about a character string.
GetCharWidth32	Gets the widths of consecutive characters from the current font.
GetCharWidthFloat	Gets the fractional widths of consecutive characters from the current font.

FUNCTION	DESCRIPTION
GetCharWidthI	Gets the widths of consecutive glyph indices or an array of glyph indices from the current font.
GetFontData	Gets metric data for a TrueType font.
GetFontLanguageInfo	Returns information about the selected font for a display context.
GetFontUnicodeRanges	Tells which Unicode characters are supported by a font.
GetGlyphIndices	Translates a string into an array of glyph indices.
GetGlyphOutline	Gets the outline or bitmap for a character in the TrueType font.
GetKerningPairs	Gets the character-kerning pairs for a font.
GetOutlineTextMetrics	Gets text metrics for TrueType fonts.
GetRasterizerCaps	Tells whether TrueType fonts are installed.
GetTabbedTextExtent	Computes the width and height of a character string, including tabs.
GetTextAlign	Gets the text-alignment setting for a device context.
GetTextCharacterExtra	Gets the current intercharacter spacing for a device context.
GetTextColor	Gets the text color for a device context.
GetTextExtentExPoint	Gets the number of characters in a string that will fit within a space.
GetTextExtentExPointI	Gets the number of glyph indices that will fit within a space.
GetTextExtentPoint32	Computes the width and height of a string of text.
GetTextExtentPointI	Computes the width and height of an array of glyph indices.
GetTextFace	Gets the name of the font that is selected into a device context.
GetTextMetrics	Fills a buffer with the metrics for a font.
PolyTextOut	Draws several strings using the font and text colors in a device context.
RemoveFontMemResourceEx	Removes a font whose source was embedded in a document from the system font table.
RemoveFontResource	Removes the fonts in a file from the system font table.

FUNCTION	DESCRIPTION
RemoveFontResourceEx	Removes a private or non-enumerable font from the system font table.
SetMapperFlags	Alters the algorithm used to map logical fonts to physical fonts.
SetTextAlign	Sets the text-alignment flags for a device context.
SetTextCharacterExtra	Sets the intercharacter spacing.
SetTextColor	Sets the text color for a device context.
SetTextJustification	Specifies the amount of space the system should add to the break characters in a string.
TabbedTextOut	Writes a character string at a location, expanding tabs to specified values.
TextOut	Writes a character string at a location.

Obsolete Functions

These functions are provided only for compatibility with 16-bit versions of Windows.

- CreateScalableFontResource
- EnumFontFamilies
- EnumFontFamProc
- EnumFonts
- EnumFontsProc
- GetCharWidth
- GetTextExtentPoint

EnableEUDC function

4/20/2022 • 2 minutes to read • Edit Online

This function enables or disables support for end-user-defined characters (EUDC).

Syntax

```
BOOL EnableEUDC(
  _In_ HDC BOOL fEnableEUDC
);
```

Parameters

fEnableEUDC [in]

Boolean that is set to **TRUE** to enable EUDC, and to **FALSE** to disable EUDC.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

If EUDC is disabled, trying to display EUDC characters will result in missing or bad glyphs.

During multi-session, this function affects the current session only.

It is recommended that you use this function with Windows XP SP2 or later.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Library	Gdi32.lib
DLL	Gdi32.dll

Font and Text Structures

4/20/2022 • 2 minutes to read • Edit Online

The following structures are used with fonts and text.

ABC
ABCFLOAT
AXESLIST
AXISINFO
DESIGNVECTOR
DRAWTEXTPARAMS
ENUMLOGFONT
ENUMLOGFONTEX
ENUMLOGFONTEXDV
ENUMTEXTMETRIC
EXTLOGFONT
FIXED
GCP_RESULTS
GLYPHMETRICS
GLYPHSET
KERNINGPAIR
LOGFONT
MAT2
NEWTEXTMETRIC
NEWTEXTMETRICEX
OUTLINETEXTMETRIC
PANOSE
POINTFX
POLYTEXT
RASTERIZER_STATUS
SIZE
TEXTMETRIC
TTPOLYCURVE
TTPOLYGONHEADER
WCRANGE

Font and Text Macros

4/20/2022 • 2 minutes to read • Edit Online

The following macros are used with fonts and text.

MACRO	DESCRIPTION
DeleteFont	Deletes a specified font.
SelectFont	Selects a font into the specified device context (DC).

Font and Text Messages

4/20/2022 • 2 minutes to read • Edit Online

The following message is used with fonts.

WM_FONTCHANGE

WM_FONTCHANGE message

4/20/2022 • 2 minutes to read • Edit Online

An application sends the **WM_FONTCHANGE** message to all top-level windows in the system after changing the pool of font resources.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
  (HWND)  hWnd,
  WM_FONTCHANGE,
  (WPARAM)  wParam,
  (LPARAM)  lParam
);
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Remarks

An application that adds or removes fonts from the system (for example, by using the **AddFontResource** or **RemoveFontResource** function) should send this message to all top-level windows.

To send the **WM_FONTCHANGE** message to all top-level windows, an application can call the **SendMessage** function with the *hwnd* parameter set to HWND_BROADCAST.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

See also

Fonts and Text Overview

Font and Text Messages

AddFontResource

RemoveFontResource

Font Embedding Reference

4/20/2022 • 2 minutes to read • Edit Online

The Font Embedding Services Library provides a mechanism to bundle TrueType and Microsoft OpenType fonts into a document or file. Typically, a document containing embedded fonts needs those fonts for rendering the document on another computer. Embedding a font guarantees that a font specified in a file will be present on the computer receiving the file. Some fonts, however, cannot be moved to other computers due to copyright issues limiting distribution.

The following elements are used with Font Embedding Services.

- Font Embedding Functions
- Font Embedding Structures
- Font Embedding Error Messages

Font Embedding Functions

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used with embedded Microsoft OpenType fonts.

FUNCTION	DESCRIPTION
CFP_ALLOCPROC	Application-provided memory allocation function for CreateFontPackage and MergeFontPackage.
CFP_FREEPROC	Application-provided memory deallocation function for CreateFontPackage and MergeFontPackage.
CFP_REALLOCPROC	Application-provided memory reallocation function for CreateFontPackage and MergeFontPackage.
CreateFontPackage	Creates a more compact version of a specified TrueType font, in order to pass it to a printer. The resulting font may be subsetted, compressed, or both.
MergeFontPackage	Merges subset fonts created by CreateFontPackage.
READEMBEDPROC	Client-provided callback function to read stream contents from a buffer.
TTCharToUnicode	Converts an array of 8-bit character code values to 16-bit Unicode values.
TTDeleteEmbeddedFont	Releases memory used by an embedded font.
TTEmbedFont	Creates a font structure containing a subsetted wide character (16-bit) font, using a device context as the font-embedding information source.
TTEmbedFontEx	Creates a font structure containing the subsetted UCS-4 character (32-bit) font, using a device context as the font-embedding information source.
TTEmbedFontFromFileA	Creates a font structure containing a subsetted wide-character (16-bit) font, using a file as the font-embedding information source.
TTEnableEmbeddingForFacename	Adds or removes facenames from the typeface exclusion list.
TTGetEmbeddedFontInfo	Retrieves information about an embedded font.
TTGetEmbeddingType	Returns embedding privileges of a font.
TTGetNewFontName	Creates a new name for an installed embedded font.

FUNCTION	DESCRIPTION
TTIsEmbeddingEnabled	Determines if the typeface exclusion list contains a specified font.
TTIsEmbeddingEnabledForFacename	Determines whether embedding is enabled for a specified font.
TTLoadEmbeddedFont	Reads the embedded font from the document stream and installs it. Also allows a client to further restrict embedding privileges of the font.
TTRunValidationTests	Validates part or all glyph data of a wide-character (16-bit) font, in the size range specified.
TTRunValidationTestsEx	UCS-4 version of TTRunValidationTests.
WRITEEMBEDPROC	Client-provided callback function to write stream contents to a buffer.

Font Embedding Services Structures

4/20/2022 • 2 minutes to read • Edit Online

The following structures are used with embedded Microsoft OpenType fonts.

TTEMBEDINFO
TTLOADINFO
TTVALIDATIONTESTPARAMS
TTVALIDATIONTESTPARAMSEX

Font-Embedding Function Error Messages

4/20/2022 • 3 minutes to read • Edit Online

The following LONG values are returned by the font-embedding functions when errors are encountered. When functions are successful, the value E_NONE is returned.

RETURN VALUE	DESCRIPTION
E_NONE	No error.
E_ADDFONTFAILED	An error occurred when the load functions tried to add the new font using AddFontResource.
E_CHARCODECOUNTINVALID	The count of subsetted characters specified in TTEmbedFont is invalid.
E_CHARCODESETINVALID	The character set specified in TTEmbedFont is invalid.
E_COULDNTCREATETEMPFILE	The load functions could not create a temporary file needed in to install a new font or resource file.
E_DEVICETRUETYPEFONT	The specified TrueType® font is not a system font. The font may exist as a device font in a printer.
E_ERRORACCESSINGEXCLUDELIST	An error occurred while attempting to access the Typeface Exclusion List.
E_ERRORACCESSINGFACENAME	A non-DC-related error was encountered while trying to allocate an OUTLINETEXTMETRIC structure.
E_ERRORACCESSINGFONTDATA	An error was encountered while attempting to use GetFontData.
E_ERRORCOMPRESSINGFONTDATA	An error occurred while TTEmbedFont attempted to compress the font data.
E_ERRORCONVERTINGCHARS	An error prevented the conversion of a string of single-byte characters to Unicode characters. This can occur in TTCharToUnicode if either *pucCharCodes* or *pusShortCodes* are non-null values, or if the conversion fails while using MultiByteToWideChar.
E_ERRORCREATINGFONTFILE	An error occurred while attempting to create the font file.
E_ERRORDECOMPRESSINGFONTDATA	An error occurred while trying to decompress data in a font file.
E_ERROREXPANDINGFONTDATA	An error occurred while the load functions attempted to expand embedded, compressed font data.
E_ERRORGETTINGDC	An error occurred while trying to allocate a DC, halting processing.

RETURN VALUE	DESCRIPTION
E_ERRORREADINGFONTDATA	An error occurred while attempting to read font data.
E_ERRORUNICODECONVERSION	An error occurred while allocating memory to convert a name string to Unicode.
E_ERRORUSINGTEMPFILE	An error occurred while the load functions were using a temporary file to install a new font file or resource file.
E_EXCEPTION	An exception was thrown by an unknown cause.
E_FACENAMEINVALID	A null *szFaceName* parameter was passed to the function.
E_FLAGSINVALID	The *ulFlags* parameter in the current function is invalid.
E_FONTALREADYEXISTS	The embedded font has the same name and checksum as a font already installed on the system.
E_FONTDATAINVALID	Font data read from disk is not a valid embedded-font structure.
E_FONTFILECREATEFAILED	The load functions could not create the font file (.ttf)
E_FONTFILENOTFOUND	The font file of the specified file name does not exist.
E_FONTINSTALLFAILED	An attempt to install the embedded font in the system failed.
E_FONTNAMEALREADYEXISTS	The embedded font has the same name but a different checksum as a font already installed.
E_FONTNOTEMBEDDABLE	The specified font cannot be embedded due to restrictions from the font manufacturer. Embedding this font in a document violates copyright laws.
E_FONTREFERENCEINVALID	A null *phFontReference* was passed to the function.
E_HDCINVALID	The device context specified for the **TTEmbedFont** function is invalid.
E_NAMECHANGEFAILED	**TTLoadEmbeddedFont** was unable to change the name of the font being loaded.
E_NOFREEMEMORY	An internal operation failed while attempting to allocate memory.
E_NOOS2	An OS/2 table was not found in the font.
E_NOTATRUETYPEFONT	The specified font is not a TrueType font.
E_PBENABLEDINVALID	A null *pbEnabled* parameter was passed to the function.
E_PERMISSIONSINVALID	A null *pulPermissions* parameter was passed to the function.

RETURN VALUE	DESCRIPTION
E_PRIVSINVALID	The *ulPrivs* parameter specified in the load functions is invalid.
E_PRIVSTATUSINVALID	A null *pulPrivStatus* parameter was passed to the function.
E_READFROMSTREAMFAILED	An error occurred while attempting to read the embedded font structure from the stream.
E_RESOURCEFILECREATEFAILED	The load functions could not create the font resource file (.fot).
E_SAVETOSTREAMFAILED	An error occurred while attempting to save the embedded-font structure to a stream.
E_STATUSINVALID	A null *pulStatus* parameter was passed to the function.
E_STREAMINVALID	The stream specified in TTEmbedFont or the load functions is invalid.
E_SUBSETTINGFAILED	TTEmbedFont failed while attempting to create a subset of a font.
E_T2NOFREEMEMORY	An error occurred while attempting to free memory. The memory in question failed during the free operation.
E_WINDOWSAPI	An internal error occurred when one of the functions called a Windows API, such as GetTextMetrics or GetOutlineTextMetrics.
E_API_NOTIMPL	This API function is not implemented in the version of Windows on which it is running.

Font Package Function Error Messages

4/20/2022 • 4 minutes to read • Edit Online

The following LONG values are returned by the font package functions (CreateFontPackage and MergeFontPackage) when errors are encountered. When functions are successful, the value NO_ERROR is returned.

RETURN VALUE	VALUE	DESCRIPTION
NO_ERROR	0	No error occurred.
ERR_FORMAT	1006	An input data format error occurred.
ERR_GENERIC	1000	An error occurred in generic code.
ERR_MEM	1005	An error occurred during memory allocation.
ERR_NO_GLYPHS	1009	No glyphs were found.
ERR_INVALID_BASE	1085	The font contained an invalid baseline data (BASE) table. Currently this value is not used.
ERR_INVALID_CMAP	1030	The font contained an invalid character-to-glyph mapping (cmap) table.
ERR_INVALID_DELTA_FORMAT	1013	An invalid delta format was detected while trying to subset a format 1 or 2 font.
ERR_INVALID_EBLC	1086	The font contained an invalid embedded bitmap location data (EBLC) table.
ERR_INVALID_GLYF	1061	The font contained an invalid glyph data (glyf) table.
ERR_INVALID_GDEF	1083	The font contained an invalid glyph definition data (GDEF) table. Currently this value is not used.
ERR_INVALID_GPOS	1082	The font contained an invalid glyph positioning data (GPOS) table. Currently this value is not used.
ERR_INVALID_GSUB	1081	The font contained an invalid glyph substitution data (GSUB) table.
ERR_INVALID_HDMX	1089	The font contained an invalid horizontal device metrics (hdmx) table.

RETURN VALUE	VALUE	DESCRIPTION
ERR_INVALID_HEAD	1062	The font contained an invalid font header (head) table.
ERR_INVALID_HHEA	1063	The font contained an invalid horizontal header (hhea) table.
ERR_INVALID_HHEA_OR_VHEA	1072	The font contained an invalid horizontal header (hhea) table or an invalid vertical metrics header (vhea) table.
ERR_INVALID_HMTX	1064	The font contained an invalid horizontal metrics (hmtx) table.
ERR_INVALID_HMTX_OR_VMTX	1073	The font contained an invalid horizontal metrics (hmtx) table or an invalid vertical metrics (vmtx) table.
ERR_INVALID_JSTF	1084	The font contained an invalid justification data (JSTF) table.
ERR_INVALID_LTSH	1087	The font contained an invalid linear threshold data (LTSH) table.
ERR_INVALID_TTO	1080	The font was an invalid TrueType Open font.
ERR_INVALID_VDMX	1088	The font contained an invalid vertical device metrics (VDMX) table.
ERR_INVALID_LOCA	1065	The font contained an invalid index to location (loca) table.
ERR_INVALID_MAXP	1066	The font contained an invalid maximum profile (maxp) table.
ERR_INVALID_MERGE_CHECKSUMS	1011	An attempt to merge checksums for two fonts from a different mother font was unsuccessful.
ERR_INVALID_MERGE_FORMATS	1010	An attempt to merge fonts with the wrong dttf formats was unsuccessful.
ERR_INVALID_MERGE_NUMGLYPHS	1012	An attempt to merge the number of glyphs for two fonts from a different mother font was unsuccessful.
ERR_INVALID_NAME	1067	The font package name or a font name was invalid.
ERR_INVALID_POST	1068	The font contained an invalid PostScript information (post) table.

RETURN VALUE	VALUE	DESCRIPTION
ERR_INVALID_OS2	1069	The font contained an invalid OS/2 and Windows-specific metrics (OS/2) table.
ERR_INVALID_VHEA	1070	The font contained an invalid vertical metrics header (vhea) table.
ERR_INVALID_VMTX	1071	The font contained an invalid vertical metrics (vmtx) table.
ERR_INVALID_TTC_INDEX	1015	An invalid zero-based (TTC) index into the font file was passed.
ERR_MISSING_CMAP	1030	The font did not contain a cmap table.
ERR_MISSING_EBDT	1044	The font did not contain an EBDT table.
ERR_MISSING_GLYF	1031	The font did not contain a glyf table.
ERR_MISSING_HEAD	1032	The font did not contain a head table.
ERR_MISSING_HHEA	1033	The font did not contain an hhea table.
ERR_MISSING_HMTX	1034	The font did not contain an hmtx table.
ERR_MISSING_LOCA	1035	The font did not contain a loca table.
ERR_MISSING_MAXP	1036	The font did not contain a maxp table.
ERR_MISSING_NAME	1037	The font did not contain a naming (name) table.
ERR_MISSING_POST	1038	The font did not contain a post table.
ERR_MISSING_OS2	1039	The font did not contain an OS/2 table.
ERR_MISSING_VHEA	1040	The font did not contain a vhea table.
ERR_MISSING_VMTX	1041	The font did not contain a vmtx table.
ERR_MISSING_HHEA_OR_VHEA	1042	The font did not contain an hhea table or a vhea table.
ERR_MISSING_HMTX_OR_VMTX	1043	The font did not contain an hmtx table or a vmtx table.
ERR_NOT_TTC	1014	The provided value was not an index for a TTC file.
ERR_PARAMETER0	1100	Calling function parameter 0 was invalid.

RETURN VALUE	VALUE	DESCRIPTION
ERR_PARAMETER1	1101	Calling function parameter 1 was invalid.
ERR_PARAMETER2	1102	Calling function parameter 2 was invalid.
ERR_PARAMETER3	1103	Calling function parameter 3 was invalid.
ERR_PARAMETER4	1104	Calling function parameter 4 was invalid.
ERR_PARAMETER5	1105	Calling function parameter 5 was invalid.
ERR_PARAMETER6	1106	Calling function parameter 6 was invalid.
ERR_PARAMETER7	1107	Calling function parameter 7 was invalid.
ERR_PARAMETER8	1108	Calling function parameter 8 was invalid.
ERR_PARAMETER9	1109	Calling function parameter 9 was invalid.
ERR_PARAMETER10	1110	Calling function parameter 10 was invalid.
ERR_PARAMETER11	1111	Calling function parameter 11 was invalid.
ERR_PARAMETER12	1112	Calling function parameter 12 was invalid.
ERR_PARAMETER13	1113	Calling function parameter 13 was invalid.
ERR_PARAMETER14	1114	Calling function parameter 14 was invalid.
ERR_PARAMETER15	1115	Calling function parameter 15 was invalid.
ERR_PARAMETER16	1116	Calling function parameter 16 was invalid.
ERR_READCONTROL	1003	The read control structure did not match data.
ERR_READOUTOFBOUNDS	1001	A read from memory was not allowed, possibly because data was out of bounds or corrupt.

RETURN VALUE	VALUE	DESCRIPTION
ERR_VERSION	1008	Major dttf.version value of the input data was greater than the version the function can read.
ERR_WOULD_GROW	1007	The requested action caused data to grow and the application must use original data.
ERR_WRITECONTROL	1004	The write control structure did not match data.
ERR_WRITEOUTOFBOUNDS	1002	A write to memory was not allowed, possibly because data was out of bounds.

Lines and Curves

4/20/2022 • 2 minutes to read • Edit Online

Lines and curves are used to draw graphics output on raster devices. As discussed in this overview, a *line* is a set of highlighted pixels on a raster display (or a set of dots on a printed page) identified by two points: a starting point and an ending point. A *regular curve* is a set of highlighted pixels on a raster display (or dots on a printed page) that defines the perimeter (or part of the perimeter) of a conic section. An *irregular curve* is a set of pixels that defines a curve that does not fit the perimeter of a conic section.

- About Lines and Curves
- Using Lines and Curves
- Line and Curve Reference

About Lines and Curves

4/20/2022 • 2 minutes to read • Edit Online

Many types of applications use lines and curves to draw graphics output on raster devices. Computer-aided design (CAD) and drawing applications use lines and curves to outline objects, specify the centers of objects, the dimensions of objects, and so on. Spreadsheet applications use lines and curves to draw grids, charts, and graphs. Word processing applications use lines to create rules and borders on a page of text.

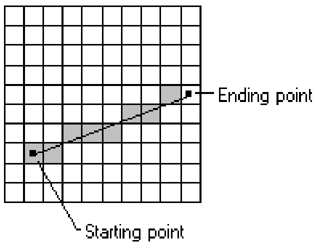
- Lines
- Curves
- Combined Lines and Curves
- Line and Curve Attributes

Lines

4/20/2022 • 2 minutes to read • Edit Online

A line is a set of highlighted pixels on a raster display (or a set of dots on a printed page) identified by two points: a starting point and an ending point. The pixel located at the starting point is always included in the line, and the pixel located at the ending point is always excluded. (This kind of line is sometimes called inclusive-exclusive.)

When an application calls one of the line-drawing functions, graphics device interface (GDI), or in some cases a device driver, determines which pixels should be highlighted. GDI is a dynamic-link library (DLL) that processes graphics function calls from an application and passes those calls to a device driver. A device driver is a DLL that receives input from GDI, converts the input to device commands, and passes those commands to the appropriate device. GDI uses a digital differential analyzer (DDA) to determine the set of pixels that define a line. A DDA determines the set of pixels by examining each point on the line and identifying those pixels on the display surface (or dots on a printed page) that correspond to the points. The following illustration shows a line, its starting point, its ending point, and the pixels highlighted by using a simple DDA.



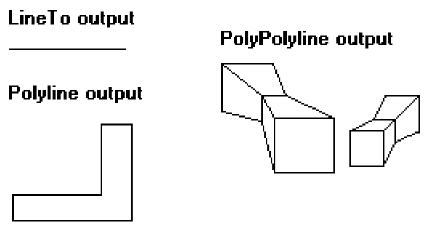
The simplest and most common DDA is the Bresenham, or incremental, DDA. A modified version of this algorithm draws lines in Windows. The incremental DDA is noted for its simplicity, but it is also noted for its inaccuracy. Because it rounds off to the nearest integer value, it sometimes fails to represent the original line requested by the application. The DDA used by GDI does not round off to the nearest integer. As a result, this new DDA produces output that is sometimes much closer in appearance to the original line requested by the application.

NOTE

If an application requires line output that cannot be achieved with the new DDA, it can draw its own lines by calling the **LineDDA** function and supplying a private DDA (**LineDDAProc**). However, the **LineDDA** function draws lines much slower than the line-drawing functions. Do not use this function within an application if speed is a primary concern.

An application can use the new DDA to draw single lines and multiple, connected line segments. An application can draw a single line by calling the **LineTo** function. This function draws a line from the current position up to, but not including, a specified ending point. An application can draw a series of connected line segments by calling the **Polyline** function, supplying an array of points that specify the ending point of each line segment. An application can draw multiple, disjointed series of connected line segments by calling the **PolyPolyline** function, supplying the required ending points.

The following illustration shows line output created by calling the **LineTo**, **Polyline**, and **PolyPolyline** functions.


LineTo output
Polyline output
PolyPolyline output

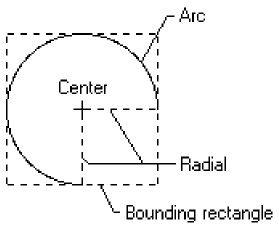
Curves

4/20/2022 • 2 minutes to read • Edit Online

A regular curve is a set of highlighted pixels on a raster display (or dots on a printed page) that define the perimeter (or part of the perimeter) of a conic section. An irregular curve is a set of pixels that define a curve that does not fit the perimeter of a conic section. The ending point is excluded from a curve just as it is excluded from a line.

When an application calls one of the curve-drawing functions, GDI breaks the curve into a number of extremely small, discrete line segments. After determining the endpoints (starting point and ending point) for each of these line segments, GDI determines which pixels (or dots) define each line by applying its DDA.

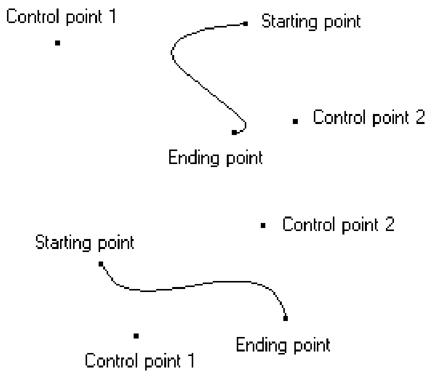
An application can draw an ellipse or part of an ellipse by calling the **Arc** function. This function draws the curve within the perimeter of an invisible rectangle called a bounding rectangle. The size of the ellipse is specified by two invisible radials extending from the center of the rectangle to the sides of the rectangle. The following illustration shows an arc (part of an ellipse) drawn by using the **Arc** function.



When calling the **Arc** function, an application specifies the coordinates of the bounding rectangle and radials. The preceding illustration shows the rectangle and radials with dashed lines while the actual arc was drawn using a solid line.

When drawing the arc of another object, the application can call the **SetArcDirection** and **GetArcDirection** functions to control the direction (clockwise or counterclockwise) in which the object is drawn. The default direction for drawing arcs and other objects is counterclockwise.

In addition to drawing ellipses or parts of ellipses, applications can draw irregular curves called Bézier curves. A *Bézier curve* is an irregular curve whose curvature is defined by four control points (p1, p2, p3, and p4). The control points p1 and p4 define the starting and ending points of the curve, and the control points p2 and p3 define the shape of the curve by marking points where the curve reverses orientation, as shown in the following diagram.



An application can draw irregular curves by calling the **PolyBezier** function, supplying the appropriate control points.

Combined Lines and Curves

4/20/2022 • 2 minutes to read • Edit Online

In addition to drawing lines or curves, applications can draw combinations of line and curve output by calling a single function. For example, an application can draw the outline of a pie chart by calling the **AngleArc** function.

The **AngleArc** function draws an arc along a circle's perimeter and draws a line connecting the starting point of the arc to the circle's center. In addition to using the **AngleArc** function, an application can also combine line and irregular curve output by using the **PolyDraw** function.

Line and Curve Attributes

4/20/2022 • 2 minutes to read • Edit Online

A device context (DC) contains attributes that affect line and curve output. The *line and curve attributes* include the current position, brush style, brush color, pen style, pen color, transformation, and so on.

The default current position for any DC is located at the point (0,0) in logical (or world) space. You can set these coordinates to a new position by calling the **MoveToEx** function and passing a new set of coordinates.

> **NOTE**
> There are two sets of line- and curve-drawing functions. The first set retains the current position in a DC, and the second set alters the position. You can identify the functions that alter the current position by examining the function name. If the function name ends with the preposition "To", the function sets the current position to the ending point of the last line drawn (**LineTo**, **ArcTo**, **PolylineTo**, or **PolyBezierTo**). If the function name does not end with this preposition, it leaves the current position intact (**Arc**, **Polyline**, or **PolyBezier**).

The default brush is a solid white brush. An application can create a new brush by calling the **CreateBrushIndirect** function. After creating a brush, the application can select it into its DC by calling the **SelectObject** function. Windows provides a complete set of functions to create, select, and alter the brush in an application's DC. For more information about these functions and about brushes in general, see Brushes.

The default pen is a cosmetic, solid black pen that is one pixel wide. An application can create a pen by using the **ExtCreatePen** function. After creating a pen, your application can select it into its DC by calling the **SelectObject** function. Windows provides a complete set of functions to create, select, and alter the pen in an application's DC. For more information about these functions and about pens in general, see Pens.

The default transformation is the unity transformation (specified by the identity matrix). An application can specify a new transformation by calling the **SetWorldTransform** function. Windows provides a complete set of functions to transform lines and curves by altering their width, location, and general appearance. For more information about these functions, see Coordinate Spaces and Transformations.

Using Lines and Curves

4/20/2022 • 2 minutes to read • Edit Online

You can use the line and curve functions to draw virtually any shape or object in the client area of an application window. This section illustrates how these functions can be used to draw markers or a pie chart.

- Drawing markers
- Drawing a pie chart

Drawing Markers

4/20/2022 • 2 minutes to read • Edit Online

You can use the line functions to draw markers. A marker is a symbol centered over a point. Drawing applications use markers to designate starting points, ending points, and control points. Spreadsheet applications use markers to designate points of interest on a chart or graph.

In the following code sample, the application-defined Marker function creates a marker by using the **MoveToEx** and **LineTo** functions. These functions draw two intersecting lines, 20 pixels in length, centered over the cursor coordinates.

```
void Marker(LONG x, LONG y, HWND hwnd)
{
    HDC hdc;

    hdc = GetDC(hwnd);
        MoveToEx(hdc, (int) x - 10, (int) y, (LPPOINT) NULL);
        LineTo(hdc, (int) x + 10, (int) y);
        MoveToEx(hdc, (int) x, (int) y - 10, (LPPOINT) NULL);
        LineTo(hdc, (int) x, (int) y + 10);

    ReleaseDC(hwnd, hdc);
}
```

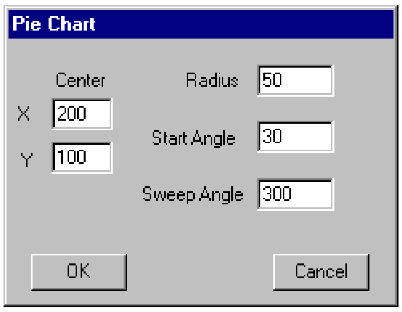
The system stores the coordinates of the cursor in the *lParam* parameter of the **WM_LBUTTONDOWN** message when the user presses the left mouse button. The following code demonstrates how an application gets these coordinates, determines whether they lie within its client area, and passes them to the Marker function to draw the marker.

```
// Line- and arc-drawing variables

static BOOL bCollectPoints;
static POINT ptMouseDown[32];
static int index;
POINTS ptTmp;
RECT rc;

    case WM_LBUTTONDOWN:


        if (bCollectPoints && index < 32)
        {
            // Create the region from the client area.

            GetClientRect(hwnd, &rc);
            hrgn = CreateRectRgn(rc.left, rc.top,
                rc.right, rc.bottom);

            ptTmp = MAKEPOINTS((POINTS FAR *) lParam);
            ptMouseDown[index].x = (LONG) ptTmp.x;
            ptMouseDown[index].y = (LONG) ptTmp.y;

            // Test for a hit in the client rectangle.

            if (PtInRegion(hrgn, ptMouseDown[index].x,
                    ptMouseDown[index].y))
            {
                // If a hit occurs, record the mouse coords.

                Marker(ptMouseDown[index].x, ptMouseDown[index].y,
                    hwnd);
                index++;
            }
        }
        break;
```

Drawing a Pie Chart

4/20/2022 • 3 minutes to read • Edit Online

You can use the line and curve functions to draw a pie chart. The primary function used to draw pie charts is the **AngleArc** function, which requires you to supply the coordinates of the center of the pie, the radius of the pie, a start angle, and a sweep angle. The following screen shot shows a dialog box that the user can use to enter these values.


Pie Chart

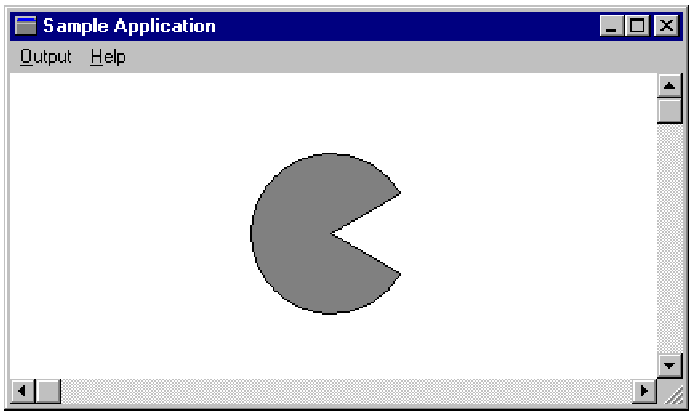
Center: X 200, Y 100

Radius: 50

Start Angle: 30

Sweep Angle: 300

The values shown above produce the following pie chart.



The dialog box template found in the application's resource script (.RC) file specifies characteristics of the preceding dialog box (its height, the controls it contains, and its style), as follows.

```
AngleArc DIALOG 6, 18, 160, 100
STYLE WS_DLGFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION
CAPTION "Pie Chart"
FONT 8, "MS Sans Serif"
BEGIN
    EDITTEXT   IDD_X, 18, 22, 25, 12, ES_AUTOHSCROLL
    LTEXT      "X", 102, 4, 24, 9, 8
    EDITTEXT   IDD_Y, 18, 39, 25, 12, ES_AUTOHSCROLL
    LTEXT      "Y", 104, 5, 42, 12, 8
    LTEXT      "Center", 105, 19, 11, 23, 8
    EDITTEXT   IDD_RADIUS, 103, 9, 32, 12, ES_AUTOHSCROLL
    EDITTEXT   IDD_STARTANGLE, 103, 31, 32, 12, ES_AUTOHSCROLL
    EDITTEXT   IDD_SWEEPANGLE, 103, 53, 32, 12, ES_AUTOHSCROLL
    LTEXT      "Radius", 109, 73, 11, 25, 8
    LTEXT      "Start Angle", 110, 59, 33, 42, 8
    LTEXT      "Sweep Angle", 111, 55, 55, 43, 8
    PUSHBUTTON "OK", IDD_OK, 9, 82, 40, 14
    PUSHBUTTON "Cancel", IDD_CANCEL, 110, 82, 40, 14
END
```

The dialog box procedure, found in the application's source file, retrieves data (center coordinates, arc radius, and start and sweep angles) by following these steps:

1. The application-defined ClearBits function initializes the array that receives the user-input to zero.
2. The application-defined GetStrLngth function retrieves the length of the string entered by the user.
3. The application-defined RetrieveInput function retrieves the value entered by the user.

The following sample code shows the dialog box procedure.

```
void ClearBits(LPTSTR, int);
int GetStrLngth(LPTSTR);
DWORD RetrieveInput(LPTSTR, int);

BOOL CALLBACK ArcDlgProc(HWND hdlg, UINT uMsg, WPARAM wParam,
LPARAM lParam)
{
    CHAR chInput[4];    // receives control-window input
    int cch;            // array-size and count variable

    switch (uMsg)
    {
        case WM_INITDIALOG:
            return FALSE;

        case WM_COMMAND:
            switch (wParam)
            {
                // If the user pressed the OK button, retrieve the
                // data that was entered in the various AngleArc
                // controls.

                case IDD_OK:
                    // Retrieve the x-coordinate of the arc's center.

                    ClearBits(chInput, sizeof(chInput));
                    GetDlgItemText(hdlg, IDD_X, chInput,
                        sizeof(chInput));
                    cch = GetStrLngth(chInput);
                    nX = (int)RetrieveInput(chInput, cch);

                    // Retrieve the y-coordinate of the arc's center.

                    ClearBits(chInput, sizeof(chInput));
                    GetDlgItemText(hdlg, IDD_Y, chInput,
                        sizeof(chInput));
                    cch = GetStrLngth(chInput);
                    nY = (int)RetrieveInput(chInput, cch);

                    // Retrieve the radius of the arc.

                    ClearBits(chInput, sizeof(chInput));
                    GetDlgItemText(hdlg, IDD_RADIUS, chInput,
                        sizeof(chInput));
                    cch = GetStrLngth(chInput);
                    dwRadius = (DWORD) RetrieveInput(chInput, cch);

                    // Retrieve the start angle.

                    ClearBits(chInput, sizeof(chInput));
                    GetDlgItemText(hdlg, IDD_STARTANGLE, chInput,
                        sizeof(chInput));
                    cch = GetStrLngth(chInput);
                    xStartAngle = (float) RetrieveInput(chInput, cch);

                    // Retrieve the sweep angle.

                    ClearBits(chInput, sizeof(chInput));
                    GetDlgItemText(hdlg, IDD_SWEEPANGLE, chInput,
                        sizeof(chInput));
                    cch = GetStrLngth(chInput);
                    xSweepAngle = (float) RetrieveInput(chInput, cch);

                    EndDialog(hdlg, FALSE);
                    return TRUE;

                // If user presses the CANCEL button, close the
                // dialog box.

                case IDD_CANCEL:
                    EndDialog(hdlg, FALSE);
                    return TRUE;
            } // end switch (wParam)

            break;

        default:
            return FALSE;
    } // end switch (message)

    UNREFERENCED_PARAMETER(lParam);
}


void ClearBits(LPTSTR cArray, int iLength)
{
    int i;

    for (i = 0; i < iLength; i++)
        cArray[i] = 0;
}

int GetStrLngth(LPTSTR cArray)
{
    int i = 0;

    while (cArray[i++] != 0);
        return i - 1;
}

DWORD RetrieveInput(LPTSTR cArray, int iLength)
{
    int i, iTmp;
    double dVal, dCount;

    dVal = 0.0;
    dCount = (double) (iLength - 1);

    // Convert ASCII input to a floating-point value.

    for (i = 0; i < iLength; i++)
    {
        iTmp = cArray[i] - 0x30;
        dVal = dVal + (((double)iTmp) * pow(10.0, dCount--));
    }

    return (DWORD) dVal;
}
```

To draw each section of the pie chart, pass the values entered by the user to the AngleArc function. To fill the pie chart using the current brush, embed the call to **AngleArc** in a path bracket. The following code sample shows the defined path bracket and the call to **AngleArc**.

```
    int nX;
    int nY;
    DWORD dwRadius;
    float xStartAngle;
    float xSweepAngle;

    hdc = GetDC(hwnd);
    BeginPath(hdc);
    SelectObject(hdc, GetStockObject(GRAY_BRUSH));
    MoveToEx(hdc, nX, nY, (LPPOINT) NULL);
    AngleArc(hdc, nX, nY, dwRadius, xStartAngle, xSweepAngle);
    LineTo(hdc, nX, nY);
    EndPath(hdc);
    StrokeAndFillPath(hdc);
    ReleaseDC(hwnd, hdc);
```

Line and Curve Reference

4/20/2022 • 2 minutes to read • Edit Online

The following elements are used with lines and curves.

- Line and Curve Functions

Line and Curve Functions

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used with lines and curves.

FUNCTION	DESCRIPTION
AngleArc	Draws a line segment and an arc.
Arc	Draws an elliptical arc.
ArcTo	Draws an elliptical arc.
GetArcDirection	Retrieves the current arc direction for the specified device context.
LineDDA	Determines which pixels should be highlighted for a line defined by the specified starting and ending points.
LineDDAProc	An application-defined callback function used with the LineDDA function.
LineTo	Draws a line from the current position up to, but not including, the specified point.
MoveToEx	Updates the current position to the specified point and optionally returns the previous position.
PolyBezier	Draws one or more Bézier curves.
PolyBezierTo	Draws one or more Bézier curves.
PolyDraw	Draws a set of line segments and Bézier curves.
Polyline	Draws a series of line segments by connecting the points in the specified array.
PolylineTo	Draws one or more straight lines.
PolyPolyline	Draws multiple series of connected line segments.
SetArcDirection	Sets the drawing direction to be used for arc and rectangle functions.

Metafiles (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

A *metafile* is a collection of structures that store a picture in a device-independent format. Device independence is the one feature that sets metafiles apart from bitmaps. Unlike a bitmap, a metafile guarantees device independence. There is a drawback to metafiles however, they are generally drawn more slowly than bitmaps. Therefore, if an application requires fast drawing and device independence is not an issue, it should use bitmaps instead of metafiles.

- About Metafiles
- Using Metafiles
- Metafile Reference

For information about bitmaps, see Bitmaps.

About Metafiles

4/20/2022 • 2 minutes to read • Edit Online

Internally, a metafile is an array of variable-length structures called *metafile records*. The first records in the metafile specify general information such as the resolution of the device on which the picture was created, the dimensions of the picture, and so on. The remaining records, which constitute the bulk of any metafile, correspond to the graphics device interface (GDI) functions required to draw the picture. These records are stored in the metafile after a special metafile device context is created. This metafile device context is then used for all drawing operations required to create the picture. When the system processes a GDI function associated with a metafile DC, it converts the function into the appropriate data and stores this data in a record appended to the metafile.

After a picture is complete and the last record is stored in the metafile, you can pass the metafile to another application by:

- Using the clipboard
- Embedding it within another file
- Storing it on disk
- Playing it repeatedly

A metafile is *played* when its records are converted to device commands and processed by the appropriate device.

There are two types of metafiles:

- Enhanced-format metafiles
- Windows-format metafiles

Enhanced-Format Metafiles

4/20/2022 • 2 minutes to read • Edit Online

The *enhanced-format metafile* consists of the following elements:

- A header
- A table of handles to GDI objects
- A private palette
- An array of metafile records

Enhanced metafiles provide true device independence. You can think of the picture stored in an enhanced metafile as a "snapshot" of the video display taken at a particular moment. This "snapshot" maintains its dimensions no matter where it appears on a printer, a plotter, the desktop, or in the client area of any application.

You can use enhanced metafiles to store a picture created by using the GDI functions (including new path and transformation functions). Because the enhanced metafile format is standardized, pictures that are stored in this format can be copied from one application to another; and, because the pictures are truly device independent, they are guaranteed to maintain their shape and proportion on any output device.

- Enhanced Metafile Records
- Enhanced Metafile Creation
- Enhanced Metafile Operations

Enhanced Metafile Records

4/20/2022 • 2 minutes to read • Edit Online

An enhanced metafile is an array of records. A metafile record is a variable-length ENHMETARECORD structure. At the beginning of every enhanced metafile record is an EMR structure, which contains two members. The first member, iType, identifies the record type that is, the GDI function whose parameters are contained in the record. Because the structures are variable in length, the other member, nSize, contains the size of the record. Immediately following the nSize member are the remaining parameters, if any, of the GDI function. The remainder of the structure contains additional data that is dependent on the record type.

The first record in an enhanced metafile is always the ENHMETAHEADER structure, which is the enhanced-metafile header. The header specifies the following information:

- Size of the metafile, in bytes
- Dimensions of the picture frame, in device units
- Dimensions of the picture frame, in .01-millimeter units
- Number of records in the metafile
- Offset to an optional text description
- Size of the optional palette
- Resolution of the original device, in pixels
- Resolution of the original device, in millimeters

An optional text description can follow the header record. The text description describes the picture and the author's name. The optional palette specifies the colors used to create the enhanced metafile. The remaining records identify the GDI functions used to create the picture. The following hexadecimal output corresponds to a record generated for a call to the SetMapMode function.

```
00000011 0000000C 00000004
```

The value 0x00000011 specifies the record type (corresponds to the EMR_SETMAPMODE constant defined in the file Wingdi.h). The value 0x0000000C specifies the length of the record, in bytes. The value 0x00000004 identifies the mapping mode (corresponds to the MM_LOENGLISH constant defined in the SetMapMode function).

For a list of additional record types, see Metafile Structures.

Enhanced Metafile Creation

4/20/2022 • 2 minutes to read • Edit Online

You create an enhanced metafile by using the **CreateEnhMetaFile** function, supplying the appropriate arguments. The system uses these arguments to maintain picture dimensions, determine whether the metafile should be stored on a disk or in memory, and so on.

To maintain picture dimensions across output devices, **CreateEnhMetaFile** requires the resolution of the reference device. This *reference device* is the device on which the picture first appeared, and the *reference DC* is the *device context* associated with the reference device. When calling the **CreateEnhMetaFile** function, you must supply a handle that identifies this DC. You can get this handle by calling the **GetDC** or **CreateDC** function. You can also specify **NULL** as the handle to use the current display device for the reference device.

Most applications store pictures permanently and therefore create an enhanced metafile that is stored on a disk; however, there are some instances when this is not necessary. For example, a word-processing application that provides chart-drawing capabilities could store a user-defined chart in memory as an enhanced metafile and then copy the enhanced metafile bits from memory into the user's document file. An application that requires a metafile that is stored permanently on a disk must supply the file name when it calls **CreateEnhMetaFile**. If you do not supply a file name, the system automatically treats the metafile as a temporary file and stores it in memory.

You can add an optional text description to a metafile containing information about the picture and the author. An application can display these strings in the File Open dialog box to provide the user with information about metafile content that will help in selecting the appropriate file. If an application includes the text description, it must supply a pointer to the string when it calls **CreateEnhMetaFile**.

When **CreateEnhMetaFile** succeeds, it returns a handle that identifies a special metafile device context. A metafile device context is unique in that it is associated with a file rather than with an output device. When the system processes a GDI function that received a handle to a metafile device context, it converts the GDI function into an enhanced-metafile record and appends the record to the end of the enhanced metafile.

After a picture is complete and the last record is appended to the enhanced metafile, the application can close the file by calling the **CloseEnhMetaFile** function. This function closes and deletes the special metafile device context and returns a handle identifying the enhanced metafile.

To delete an enhanced-format metafile or an enhanced-format metafile handle, call the **DeleteEnhMetaFile** function.

Enhanced Metafile Operations

4/20/2022 • 2 minutes to read • Edit Online

You can use the handle to an enhanced metafile to accomplish the following tasks:

- Display the picture stored in an enhanced metafile.
- Create copies of an enhanced metafile.
- Edit an enhanced metafile.
- Retrieve the optional description stored in an enhanced metafile.
- Retrieve a copy of an enhanced-metafile header.
- Retrieve a binary version of an enhanced metafile.
- Enumerate the colors in the optional palette.

These tasks are discussed in the sections in the remainder of this topic.

Display the Picture Stored in an Enhanced Metafile

You can display the picture stored in an enhanced metafile using the **PlayEnhMetaFile** function. Pass the function a handle to the enhanced metafile, without being concerned with the format of the enhanced metafile records. However, it is sometimes desirable to enumerate the records in the enhanced metafile to search for a particular GDI function and modify the parameters of the function in some manner. To do this, you can use **EnumEnhMetaFile** and provide a callback function, **EnhMetaFileProc**, to process the enhanced metafile records. To modify the parameters for an enhanced metafile record, you must know the format of the parameters within the record.

Create Copies of an Enhanced Metafile

Some applications create temporary backup (or duplicate) copies of a file before enabling the user to alter the original. An application can create a backup copy of an enhanced metafile by calling the **CopyEnhMetaFile** function, supplying a handle that identifies the enhanced metafile, and supplying a pointer to the name of the new file.

To create a memory-based enhanced-format metafile, call the **SetEnhMetaFileBits** function.

Edit an Enhanced Metafile

Most drawing, illustration, and computer-aided design (CAD) applications require a means of editing a picture stored in an enhanced metafile. Although editing an enhanced metafile is a complex task, you can use the **EnumEnhMetaFile** function in combination with other functions to provide this capability in your application. The **EnumEnhMetaFile** function and its associated callback function, **EnhMetaFileProc**, enable the application to process individual records in an enhanced metafile.

Retrieve the Optional Description Stored in an Enhanced Metafile

Some applications display the text description of an enhanced metafile with the corresponding file name in the **Open** dialog box. You can determine whether this string exists in an enhanced metafile by retrieving the metafile header with the **GetEnhMetaFileHeader** function and examining one of its members. If the string exists, the application retrieves it by calling the **GetEnhMetaFileDescription** function.

Retrieve a Binary Version of an Enhanced Metafile

You can retrieve the contents of a metafile by calling the **GetEnhMetaFileBits** function; however, before retrieving the contents, you must specify the size of the file. To get the size, you can use the **GetEnhMetaFileHeader** function and examine the appropriate member.

Enumerate the Colors in the Optional Palette

To achieve consistent colors when a picture is displayed on various output devices, you can call the **CreatePalette** function and store a logical palette in an enhanced metafile. An application that displays the picture stored in the enhanced metafile retrieves this palette and calls the **RealizePalette** function before displaying the picture. To determine whether a palette is stored in an enhanced metafile, retrieve the metafile header and examine the appropriate member. If a palette exists, you can call the **GetEnhMetaFilePaletteEntries** function to retrieve the logical palette.

Windows-Format Metafiles

4/20/2022 • 2 minutes to read • Edit Online

Microsoft Windows-format metafiles are limited in their capabilities and should rarely be used. The Windows-format functions are supported to maintain backward compatibility with applications that were written to run as 16 bit Windows-based applications. Instead, you should use the enhanced-format functions.

A *Windows-format metafile* is used by 16-bit Windows-based applications. The format consists of a header and an array of metafile records.

The following are the limitations of this format:

- A Windows-format metafile is application and device dependent. Changes in the application's mapping modes or the device resolution affect the appearance of metafiles created in this format.
- A Windows-format metafile does not contain a comprehensive header that describes the original picture dimensions, the resolution of the device on which the picture was created, an optional text description, or an optional palette.
- A Windows-format metafile does not support the new curve, path, and transformation functions. See the list of supported functions in the table that follows.
- Some Windows-format metafile records cannot be scaled.
- The metafile device context associated with a Windows-format metafile cannot be queried (that is, an application cannot retrieve device-resolution data, font metrics, and so on).

The following are the only functions that are supported by Windows-format metafiles.

AnimatePaletteArc BitBlt Chord CreateBrushIndirect CreateDIBPatternBrush CreateFontIndirect CreatePalette CreatePatternBrush CreatePenIndirect DeleteObject Ellipse Escape ExcludeClipRect ExtFloodFill ExtTextOut FillRgn FloodFill FrameRgn IntersectClipRect InvertRgn	LineToMoveToEx OffsetClipRgn OffsetViewportOrgEx OffsetWindowOrgEx PaintRgn PatBlt Pie Polygon Polyline PolyPolygon RealizePalette Rectangle ResizePalette RestoreDC RoundRect SaveDC ScaleViewportExtEx ScaleWindowExtEx SelectClipRgn SelectObject	SelectPaletteSetBkColor SetBkMode SetDIBitsToDevice SetMapMode SetMapperFlags SetPaletteEntries SetPixel SetPolyFillMode SetROP2 SetStretchBltMode SetTextAlign SetTextCharacterExtra SetTextColor SetTextJustification SetViewportOrgEx SetWindowExtEx SetWindowOrgEx StretchBlt StretchDIBits TextOut

To convert a Windows-format metafile to an enhanced-format metafile, call the GetMetaFileBitsEx function to retrieve the data from the Windows-format metafile and then call the SetWinMetaFileBits function to convert this data into an enhanced-format metafile. To convert an enhanced-format record into a Windows-format record, call the GetWinMetaFileBits function.

Using Metafiles

4/20/2022 • 2 minutes to read • Edit Online

This section explains how to perform the following tasks:

- Creating an enhanced metafile
- Displaying a picture and storing it in an enhanced metafile
- Opening an enhanced metafile and displaying its contents
- Editing an enhanced metafile

Creating an Enhanced Metafile

4/20/2022 • 2 minutes to read • Edit Online

This section contains an example that demonstrates the creation of an enhanced metafile that is stored on a disk, using a file name specified by the user.

The example uses a device context for the application window as the reference device context. (The system stores the resolution data for this device in the enhanced-metafile's header.) The application retrieves a handle identifying this device context by calling the GetDC function.

The example uses the dimensions of the application's client area to define the dimensions of the picture frame. Using the rectangle dimensions returned by the GetClientRect function, the application converts the device units to .01-millimeter units and passes the converted values to the CreateEnhMetaFile function.

The example displays a **Save As** common dialog box that enables the user to specify the file name of the new enhanced metafile. The system appends the three-character .emf extension to this file name and passes the name to the CreateEnhMetaFile function.

The example also embeds a text description of the picture in the enhanced-metafile header. This description is specified as a resource in the string table of the application's resource file. However, in a working application, this string would be retrieved from a custom control in a common dialog box or from a separate dialog box displayed solely for this purpose.

```
// Obtain a handle to a reference device context.

hdcRef = GetDC(hWnd);

// Determine the picture frame dimensions.
// iWidthMM is the display width in millimeters.
// iHeightMM is the display height in millimeters.
// iWidthPels is the display width in pixels.
// iHeightPels is the display height in pixels

iWidthMM = GetDeviceCaps(hdcRef, HORZSIZE);
iHeightMM = GetDeviceCaps(hdcRef, VERTSIZE);
iWidthPels = GetDeviceCaps(hdcRef, HORZRES);
iHeightPels = GetDeviceCaps(hdcRef, VERTRES);

// Retrieve the coordinates of the client
// rectangle, in pixels.

GetClientRect(hWnd, &rect);

// Convert client coordinates to .01-mm units.
// Use iWidthMM, iWidthPels, iHeightMM, and
// iHeightPels to determine the number of
// .01-millimeter units per pixel in the x-
//  and y-directions.

rect.left = (rect.left * iWidthMM * 100)/iWidthPels;
rect.top = (rect.top * iHeightMM * 100)/iHeightPels;
rect.right = (rect.right * iWidthMM * 100)/iWidthPels;
rect.bottom = (rect.bottom * iHeightMM * 100)/iHeightPels;

// Load the filename filter from the string table.

LoadString(hInst, IDS_FILTERSTRING,
     (LPSTR)szFilter, sizeof(szFilter));
```

```
// Replace the '%' separators that are embedded
// between the strings in the string-table entry
// with '\0'.

for (i=0; szFilter[i]!='\0'; i++)
    if (szFilter[i] == '%')
            szFilter[i] = '\0';

// Load the dialog title string from the table.

LoadString(hInst, IDS_TITLESTRING,
     (LPSTR)szTitle, sizeof(szTitle));

// Initialize the OPENFILENAME members.

szFile[0] = '\0';

Ofn.lStructSize = sizeof(OPENFILENAME);
Ofn.hwndOwner = hWnd;
Ofn.lpstrFilter = szFilter;
Ofn.lpstrFile= szFile;
Ofn.nMaxFile = sizeof(szFile)/ sizeof(*szFile);
Ofn.lpstrFileTitle = szFileTitle;
Ofn.nMaxFileTitle = sizeof(szFileTitle);
Ofn.lpstrInitialDir = (LPSTR)NULL;
Ofn.Flags = OFN_SHOWHELP | OFN_OVERWRITEPROMPT;
Ofn.lpstrTitle = szTitle;

// Display the Filename common dialog box. The
// filename specified by the user is passed
// to the CreateEnhMetaFile function and used to
// store the metafile on disk.

GetSaveFileName(&Ofn);

// Load the description from the string table.

LoadString(hInst, IDS_DESCRIPTIONSTRING,
     (LPSTR)szDescription, sizeof(szDescription));

// Replace the '%' string separators that are
// embedded between strings in the string-table
// entry with '\0'.

for (i=0; szDescription[i]!='\0'; i++)
{
    if (szDescription[i] == '%')
            szDescription[i] = '\0';
}

// Create the metafile device context.

hdcMeta = CreateEnhMetaFile(hdcRef,
          (LPTSTR) Ofn.lpstrFile,
          &rect, (LPSTR)szDescription);

if (!hdcMeta)
    errhandler("CreateEnhMetaFile", hWnd);

// Release the reference device context.

ReleaseDC(hWnd, hdcRef);
```

Displaying a Picture and Storing It in an Enhanced Metafile

4/20/2022 • 2 minutes to read • Edit Online

This section contains an example demonstrating the creation of a picture and the process of storing the corresponding records in a metafile. The example draws a picture to the display or stores it in a metafile. If a display device context handle is given, it draws a picture to the screen using various GDI functions. If an enhanced metafile device context is given, it stores the same picture in the enhanced metafile.

```
void DrawOrStore(HWND hwnd, HDC hdcMeta, HDC hdcDisplay)
{

RECT rect;
HDC hDC;
int fnMapModeOld;
HBRUSH hbrOld;

// Draw it to the display DC or store it in the metafile device context.

if (hdcMeta)
    hDC = hdcMeta;
else
    hDC = hdcDisplay;

// Set the mapping mode in the device context.

fnMapModeOld = SetMapMode(hDC, MM_LOENGLISH);

// Find the midpoint of the client area.

GetClientRect(hwnd, (LPRECT)&rect);
DPtoLP(hDC, (LPPOINT)&rect, 2);

// Select a gray brush.

hbrOld = SelectObject(hDC, GetStockObject(GRAY_BRUSH));

// Draw a circle with a one inch radius.

Ellipse(hDC, (rect.right/2 - 100), (rect.bottom/2 + 100),
       (rect.right/2 + 100), (rect.bottom/2 - 100));

// Perform additional drawing here.



// Set the device context back to its original state.

SetMapMode(hDC, fnMapModeOld);
SelectObject(hDC, hbrOld);
}
```

Opening an Enhanced Metafile and Displaying Its Contents

4/20/2022 • 2 minutes to read • Edit Online

This section contains an example demonstrating how an application opens an enhanced metafile stored on disk and displays the associated picture in the client area.

The example uses the **Open** common dialog box to enable the user to select an enhanced metafile from a list of existing files. It then passes the name of the selected file to the **GetEnhMetaFile** function, which returns a handle identifying the file. This handle is passed to the **PlayEnhMetaFile** function in order to display the picture.

```
LoadString(hInst, IDS_FILTERSTRING,
     (LPSTR)szFilter, sizeof(szFilter));

// Replace occurrences of '%' string separator
// with '\0'.

for (i=0; szFilter[i]!='\0'; i++)
{
    if (szFilter[i] == '%')
            szFilter[i] = '\0';
}

LoadString(hInst, IDS_DEFEXTSTRING,
     (LPSTR)szDefExt, sizeof(szFilter));


// Use the OpenFilename common dialog box
// to obtain the desired filename.

szFile[0] = '\0';
OPENFILENAME Ofn;
Ofn.lStructSize = sizeof(OPENFILENAME);
Ofn.hwndOwner = hWnd;
Ofn.lpstrFilter = szFilter;
Ofn.lpstrCustomFilter = (LPSTR)NULL;
Ofn.nMaxCustFilter = 0L;
Ofn.nFilterIndex = 1L;
Ofn.lpstrFile = szFile;
Ofn.nMaxFile = sizeof(szFile);
Ofn.lpstrFileTitle = szFileTitle;
Ofn.nMaxFileTitle = sizeof(szFileTitle);
Ofn.lpstrInitialDir = (LPSTR) NULL;
Ofn.lpstrTitle = (LPSTR)NULL;
Ofn.Flags = OFN_SHOWHELP | OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;
Ofn.nFileOffset = 0;
Ofn.nFileExtension = 0;
Ofn.lpstrDefExt = szDefExt;

GetOpenFileName(&Ofn);

// Open the metafile.

HENHMETAFILE hemf = GetEnhMetaFile(Ofn.lpstrFile);

// Retrieve a handle to a window device context.

HDC hDC = GetDC(hWnd);

// Retrieve the client rectangle dimensions.

GetClientRect(hWnd, &rct);

// Draw the picture.

PlayEnhMetaFile(hDC, hemf, &rct);

// Release the metafile handle.

DeleteEnhMetaFile(hemf);

// Release the window DC.

ReleaseDC(hWnd, hDC);
```

Editing an Enhanced Metafile

4/20/2022 • 2 minutes to read • Edit Online

To edit a picture stored in an enhanced metafile, an application must perform the tasks described in the following procedure.

To edit a picture stored in an enhanced metafile

1. Use hit-testing to capture the cursor coordinates and retrieve the position of the object (line, arc, rectangle, ellipse, polygon, or irregular shape) that the user wants to alter.
2. Convert these coordinates to logical (or world) units.
3. Call the **EnumEnhMetaFile** function and examine each metafile record.
4. Determine whether a given record corresponds to a GDI drawing function.
5. If it does, determine whether the coordinates stored in the record correspond to the line, arc, ellipse, or other graphics element that intersects the coordinates specified by the user.
6. Upon finding the record that corresponds to the output that the user wants to alter, erase the object on the screen that corresponds to the original record.
7. Delete the corresponding record from the metafile, saving a pointer to its location.
8. Permit the user to redraw or replace the object.
9. Convert the GDI functions used to draw the new object into one or more enhanced-metafile records.
10. Store these records in the enhanced metafile.

Metafile Reference

4/20/2022 • 2 minutes to read • Edit Online

The following elements are used with metafiles:

- Metafile Functions
- Metafile Structures

Metafile Functions

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used with enhanced-format metafiles.

FUNCTION	DESCRIPTION
CloseEnhMetaFile	Closes an enhanced-metafile device context.
CopyEnhMetaFile	Copies the contents of an enhanced-format metafile to a specified file.
CreateEnhMetaFile	Creates a device context for an enhanced-format metafile.
DeleteEnhMetaFile	Deletes an enhanced-format metafile or an enhanced-format metafile handle.
EnhMetaFileProc	An application-defined callback function used with the EnumEnhMetaFile function.
EnumEnhMetaFile	Enumerates the records within an enhanced-format metafile.
GdiComment	Copies a comment from a buffer into a specified enhanced-format metafile.
GetEnhMetaFile	Creates a handle that identifies the enhanced-format metafile stored in the specified file.
GetEnhMetaFileBits	Retrieves the contents of the specified enhanced-format metafile and copies them into a buffer.
GetEnhMetaFileDescription	Retrieves an optional text description from an enhanced-format metafile and copies the string to the specified buffer.
GetEnhMetaFileHeader	Retrieves the record containing the header for the specified enhanced-format metafile.
GetEnhMetaFilePaletteEntries	Retrieves optional palette entries from the specified enhanced metafile.
GetMetaFile	GetMetaFile is no longer available for use as of Windows 2000. Instead, use GetEnhMetaFile.
GetWinMetaFileBits	Converts the enhanced-format records from a metafile into Windows-format records.
PlayEnhMetaFile	Displays the picture stored in the specified enhanced-format metafile.
PlayEnhMetaFileRecord	Plays an enhanced-metafile record by executing the graphics device interface (GDI) functions identified by the record.

FUNCTION	DESCRIPTION
SetEnhMetaFileBits	Creates a memory-based enhanced-format metafile from the specified data.
SetWinMetaFileBits	Converts a metafile from the older Windows format to the new enhanced format.

Obsolete Functions

The following functions are obsolete. The are provided for compatibility with Windows-format metafiles.

- CloseMetaFile
- CopyMetaFile
- CreateMetaFile
- DeleteMetaFile
- EnumMetaFile
- EnumMetaFileProc
- GetMetaFileBitsEx
- PlayMetaFile
- PlayMetaFileRecord
- SetMetaFileBitsEx

Metafile Structures

4/20/2022 • 2 minutes to read • Edit Online

The following structures are used with enhanced-format metafiles.

EMRTEXT
EMRTRANSPARENTBLT
EMRWIDENPATH
ENHMETAHEADER
ENHMETARECORD
HANDLETABLE
POINTL
RECTL

Note that the EMR structure is used as the first member of the remaining structures.

EMR
EMRABORTPATH
EMRALPHABLEND
EMRANGLEARC
EMRARC
EMRARCTO
EMRBEGINPATH
EMRBITBLT
EMRCHORD
EMRCLOSEFIGURE
EMRCOLORCORRECTPALETTE
EMRCOLORMATCHTOTARGET
EMRCREATEBRUSHINDIRECT
EMRCREATECOLORSPACE
EMRCREATECOLORSPACEW
EMRCREATEDIBPATTERNBRUSHPT
EMRCREATEMONOBRUSH
EMRCREATEPALETTE
EMRCREATEPEN
EMRDELETECOLORSPACE
EMRDELETEOBJECT
EMRELLIPSE
EMRENDPATH
EMREOF
EMREXCLUDECLIPRECT
EMREXTCREATEFONTINDIRECTW
EMREXTCREATEPEN
EMREXTFLOODFILL
EMREXTSELECTCLIPRGN
EMREXTTEXTOUTA
EMREXTTEXTOUTW
EMRFILLPATH
EMRFILLRGN

EMRFLATTENPATH
EMRFORMAT
EMRFRAMERGN
EMRGDICOMMENT
EMRGLSBOUNDEDRECORD
EMRGLSRECORD
EMRGRADIENTFILL
EMRINTERSECTCLIPRECT
EMRINVERTRGN
EMRLINETO
EMRMASKBLT
EMRMODIFYWORLDTRANSFORM
EMRMOVETOEX
EMROFFSETCLIPRGN
EMRPAINTRGN
EMRPIE
EMRPIXELFORMAT
EMRPLGBLT
EMRPOLYBEZIER
EMRPOLYBEZIER16
EMRPOLYBEZIERTO
EMRPOLYBEZIERTO16
EMRPOLYDRAW
EMRPOLYDRAW16
EMRPOLYGON
EMRPOLYGON16
EMRPOLYLINE
EMRPOLYLINE16
EMRPOLYLINETO
EMRPOLYLINETO16
EMRPOLYPOLYGON
EMRPOLYPOLYGON16
EMRPOLYPOLYLINE
EMRPOLYPOLYLINE16
EMRPOLYTEXTOUTA
EMRPOLYTEXTOUTW
EMRREALIZEPALETTE
EMRRECTANGLE
EMRRESIZEPALETTE
EMRRESTOREDC
EMRROUNDRECT
EMRSAVEDC
EMRSCALEVIEWPORTEXTEX
EMRSCALEWINDOWEXTEX
EMRSELECTCLIPPATH
EMRSELECTCOLORSPACE
EMRSELECTOBJECT
EMRSELECTPALETTE
EMRSETARCDIRECTION
EMRSETBKCOLOR
EMRSETBKMODE
EMRSETBRUSHORGEX

EMRSETCOLORADJUSTMENT
EMRSETCOLORSPACE
EMRSETDIBITSTODEVICE
EMRSETICMMODE
EMRSETICMPROFILE
EMRSETLAYOUT
EMRSETMAPMODE
EMRSETMAPPERFLAGS
EMRSETMETARGN
EMRSETMITERLIMIT
EMRSETPALETTEENTRIES
EMRSETPIXELV
EMRSETPOLYFILLMODE
EMRSETROP2
EMRSETSTRETCHBLTMODE
EMRSETTEXTALIGN
EMRSETTEXTCOLOR
EMRSETVIEWPORTEXTEX
EMRSETVIEWPORTORGEX
EMRSETWINDOWEXTEX
EMRSETWINDOWORGEX
EMRSETWORLDTRANSFORM
EMRSTRETCHBLT
EMRSTRETCHDIBITS
EMRSTROKEANDFILLPATH
EMRSTROKEPATH

Obsolete Structures

The following structures are obsolete. The are provided for compatibility with Windows-format metafiles:

METAHEADERMETARECORD

Multiple Display Monitors

4/20/2022 • 2 minutes to read • Edit Online

Multiple Display Monitors is a set of related features that allow applications to make use of multiple display devices at the same time. There are two ways that multiple monitors may be used: as one large desktop or as a number of independent displays. When used as one large desktop, the monitors create more screen space for applications. This is useful whenever you need to maximize your onscreen workspace, and is especially useful in desktop publishing, Web development, or video editing. When used as independent displays, the monitors can be used for playing games, debugging full-screen applications, teaching, or making presentations.

This chapter covers the following topics:

- About Multiple Display Monitors
- Using Multiple Display Monitors
- Multiple Display Monitors Reference

About Multiple Display Monitors

4/20/2022 • 2 minutes to read • Edit Online

When multiple monitors are part of the desktop, objects can travel seamlessly between monitors. That is, you can drag windows or shortcuts from one monitor to another, and you can size windows to cover more than one monitor. Also, if one monitor is installed above another, a cursor that leaves the bottom of the upper monitor appears at the top of the lower monitor.

Typically, a user arranges the monitors in the system to reflect the arrangement of the physical display units; for example, side-by-side or one-on-top-of-the-other. The user does this with the Monitors tab, which replaces the **Settings** tab in the **Display Properties** dialog box through Control Panel. The monitors must form one contiguous region, that is, each monitor touches another monitor on at least part of one edge.

When a window is moved or resized, some part of the caption is always visible so the user can move and resize the window using the mouse. Cursor movement is restricted to the area of the monitors, so it is always visible. Shell icons are positioned on the same monitor as the taskbar, and the taskbar can be on any monitor, see Multiple Monitor Considerations for Older Programs.

A multiple monitor system affects certain key combinations. The ALT+PRINTSCRN key combination takes a snapshot of the foreground window, as always. However, the PRINTSCRN key takes a snapshot of the monitor that has the mouse. The CTRL+PRINTSCRN key combination takes a snapshot of the entire virtual screen, see The Virtual Screen.

The support for multiple monitors does not affect the performance of applications when running in a single display environment. That is, when running on a single display system, no additional overhead will be present in the high-performance graphics operations code. However, in a multiple monitor system, performance is slightly affected if an application runs only on one of the graphics devices. Also, performance may be greatly affected if an application spans multiple displays, especially for graphics-intensive operations.

Full-screen is a feature provided by the operating system that allows a user to toggle an application into a special state where the application can access VGA graphics hardware directly. This is a key feature for games and other graphics-focused applications that require high performance. Also, it is often used by developers for text editing since it enables very fast text scrolling.

In a multiple monitor environment, only one graphics device can be VGA compatible. This is a limitation of computer hardware which requires that only one device respond to any hardware address. Because the VGA hardware compatibility standard requires specific hardware addresses, only one VGA graphics device can be present in a machine and only this device can physically respond to VGA addresses. Thus, applications that require going full screen will only run on the particular device that supports VGA hardware compatibility.

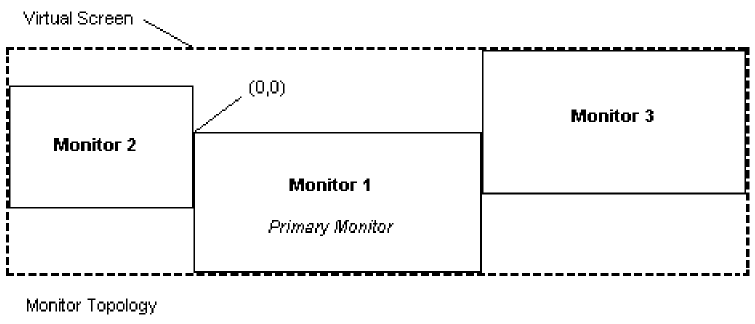
This overview provides information on the following topics.

- The Virtual Screen
- HMONITOR and the Device Context
- Enumeration and Display Control
- Multiple Monitor System Metrics
- Using Multiple Monitors as Independent Displays
- Colors on Multiple Display Monitors
- Positioning Objects on Multiple Display Monitors
- Multiple Monitor Applications on Different Systems
- Multiple Monitor Considerations for Older Programs

The Virtual Screen

4/20/2022 • 2 minutes to read • Edit Online

The bounding rectangle of all the monitors is the *virtual screen*. The desktop covers the virtual screen instead of a single monitor. The following illustration shows a possible arrangement of three monitors.



The *primary monitor* contains the origin (0,0). This is for compatibility with existing applications that expect a monitor with an origin. However, the primary monitor does not have to be in the upper left of the virtual screen. In Figure 1, it is near the center. When the primary monitor is not in the upper left of the virtual screen, parts of the virtual screen have negative coordinates. Because the arrangement of monitors is set by the user, all applications should be designed to work with negative coordinates. For more information, see Multiple Monitor Considerations for Older Programs.

The coordinates of the virtual screen are represented by a signed 16-bit value because of the 16-bit values contained in many existing messages. Thus, the bounds of the virtual screen are:

```
SHORT_MIN    <= rcVirtualScreen.left   <= SHORT_MAX - 1
SHORT_MIN +1 <= rcVirtualScreen.right  <= SHORT_MAX
SHORT_MIN    <= rcVirtualScreen.top    <= SHORT_MAX - 1
SHORT_MIN +1 <= rcVirtualScreen.bottom <= SHORT_MAX
```

HMONITOR and the Device Context

4/20/2022 • 2 minutes to read • Edit Online

Each physical display is represented by a monitor handle of type **HMONITOR**. A valid **HMONITOR** is guaranteed to be non-NULL. A physical display has the same **HMONITOR** as long as it is part of the desktop. When a **WM_DISPLAYCHANGE** message is sent, any monitor may be removed from the desktop and thus its **HMONITOR** becomes invalid or has its settings changed. Therefore, an application should check whether all **HMONITORS** are valid when this message is sent.

Any function that returns a display device context (DC) normally returns a DC for the primary monitor. To obtain the DC for another monitor, use the **EnumDisplayMonitors** function. Or, you can use the device name from the **GetMonitorInfo** function to create a DC with **CreateDC**. However, if the function, such as **GetWindowDC** or **BeginPaint**, gets a DC for a window that spans more than one display, the DC will also span the two displays.

Enumeration and Display Control

4/20/2022 • 2 minutes to read • Edit Online

To enumerate all the devices on the computer, call the EnumDisplayDevices function. The information that is returned also indicates which monitor is part of the desktop.

To enumerate the devices on the desktop that intersect a clipping region, call EnumDisplayMonitors. This returns the HMONITOR handle to each monitor, which is used with GetMonitorInfo. To enumerate all the devices in the virtual screen, use **EnumDisplayMonitors**. as shown in the following code:

```
EnumDisplayMonitors(NULL, NULL, MyInfoEnumProc, 0);
```

To get information about a display device, use EnumDisplaySettings or EnumDisplaySettingsEx.

The ChangeDisplaySettingsEx function is used to control the display devices on the computer. It can modify the configuration of the devices, such as specifying the position of a monitor on the virtual desktop and changing the bit depth of any display. Typically, an application does not use this function. To add a display monitor to a multiple-monitor system programmatically, set **DEVMODE.dmFields** to DM_POSITION and specify a position (using **DEVMODE.dmPosition**) for the monitor you are adding that is adjacent to at least one pixel of the display area of an existing monitor. To detach the monitor, set **DEVMODE.dmFields** to DM_POSITION and set **DEVMODE.dmPelsWidth** and **DEVMODE.dmPelsHeight** to zero.

The following code demonstrates how to detach all secondary display devices from the desktop:

```
void DetachDisplay()
{
    BOOL            FoundSecondaryDisp = FALSE;
    DWORD           DispNum = 0;
    DISPLAY_DEVICE  DisplayDevice;
    LONG            Result;
    TCHAR           szTemp[200];
    int             i = 0;
    DEVMODE   defaultMode;

    // initialize DisplayDevice
    ZeroMemory(&DisplayDevice, sizeof(DisplayDevice));
    DisplayDevice.cb = sizeof(DisplayDevice);

    // get all display devices
    while (EnumDisplayDevices(NULL, DispNum, &DisplayDevice, 0))
        {
        ZeroMemory(&defaultMode, sizeof(DEVMODE));
        defaultMode.dmSize = sizeof(DEVMODE);
        if ( !EnumDisplaySettings((LPSTR)DisplayDevice.DeviceName, ENUM_REGISTRY_SETTINGS, &defaultMode) )
                  OutputDebugString("Store default failed\n");

        if ((DisplayDevice.StateFlags & DISPLAY_DEVICE_ATTACHED_TO_DESKTOP) &&
            !(DisplayDevice.StateFlags & DISPLAY_DEVICE_PRIMARY_DEVICE))
            {
            DEVMODE    DevMode;
            ZeroMemory(&DevMode, sizeof(DevMode));
            DevMode.dmSize = sizeof(DevMode);
            DevMode.dmFields = DM_PELSWIDTH | DM_PELSHEIGHT | DM_BITSPERPEL | DM_POSITION
                        | DM_DISPLAYFREQUENCY | DM_DISPLAYFLAGS ;
            Result = ChangeDisplaySettingsEx((LPSTR)DisplayDevice.DeviceName,
                                            &DevMode,
                                            NULL,
                                            CDS_UPDATEREGISTRY,
                                            NULL);

            //The code below shows how to re-attach the secondary displays to the desktop

            //ChangeDisplaySettingsEx((LPSTR)DisplayDevice.DeviceName,
            //                       &defaultMode,
            //                       NULL,
            //                       CDS_UPDATEREGISTRY,
            //                       NULL);

            }

        // Reinit DisplayDevice just to be extra clean

        ZeroMemory(&DisplayDevice, sizeof(DisplayDevice));
        DisplayDevice.cb = sizeof(DisplayDevice);
        DispNum++;
        } // end while for all display devices
}
```

For each display device, the application can save information in the registry that describes the configuration parameters for the device, as well as location parameters. The application can also determine which displays are part of the desktop, and which are not, through the DISPLAY_DEVICE_ATTACHED_TO_DESKTOP flag in the DISPLAY_DEVICE structure. Once all the configuration information is stored in the registry, the application can call ChangeDisplaySettingsEx again to dynamically change the settings, with no restart required.

Multiple Monitor System Metrics

4/20/2022 • 2 minutes to read • Edit Online

The GetSystemMetrics function returns values for the primary monitor, except for SM_CXMAXTRACK and SM_CYMAXTRACK, which refer to the entire desktop. The following metrics are the same for all device drivers: SM_CXCURSOR, SM_CYCURSOR, SM_CXICON, SMCYICON. The following display capabilities are the same for all monitors: LOGPIXELSX, LOGPIXELSY, DESTOPHORZRES, DESKTOPVERTRES.

GetSystemMetrics also has constants that refer only to a Multiple Monitor system. SM_XVIRTUALSCREEN and SM_YVIRTUALSCREEN identify the upper-left corner of the virtual screen, SM_CXVIRTUALSCREEN and SM_CYVIRTUALSCREEN are the vertical and horizontal measurements of the virtual screen, SM_CMONITORS is the number of monitors attached to the desktop, and SM_SAMEDISPLAYFORMAT indicates whether all the monitors on the desktop have the same color format.

To get information about a single display monitor or all of the display monitors in a desktop, use EnumDisplayMonitors. The rectangle of the desktop window returned by GetWindowRect or GetClientRect is always equal to the rectangle of the primary monitor, for compatibility with existing applications. Thus, the result of

```
GetWindowRect(GetDesktopWindow(), &rc);
```

will be:

```
rc.left = 0;
rc.top = 0;
rc.right = GetSystemMetrics (SM_CXSCREEN);
rc.bottom = GetSystemMetrics (SM_CYSCREEN);
```

To change the work area of a monitor, call SystemParametersInfo with SPI_SETWORKAREA and *pvParam* pointing to a RECT structure that is on the desired monitor. If *pvParam* is **NULL**, the work area of the primary monitor is modified. Using SPI_GETWORKAREA always returns the work area of the primary monitor. To get the work area of a monitor other than the primary monitor, call GetMonitorInfo.

Using Multiple Monitors as Independent Displays

4/20/2022 • 2 minutes to read • Edit Online

When using multiple monitors as independent displays, the desktop contains one display or set of displays. This set of displays always includes the primary monitor and behaves as mentioned in the other sections of this topic. An application can use any other monitor as an independent display.

NOTE

Using other monitors as independent displays isn't supported on drivers that are implemented to the Windows Display Driver Model (WDDM).

The window manager knows nothing about the independent displays. They are completely controlled by the application, and no window manager functions are available to the application (all window manager calls automatically go to the primary display). Each independent display has its own origin and horizontal and vertical coordinates, and is accessed through the GDI functions such as **CreateDC** or the DirectX functions such as **DirectDrawCreate**.

To locate the independent displays, call **EnumDisplayDevices** and look for the displays that do not have DISPLAY_DEVICE_ATTACHED_TO_DESKTOP flag in the **DISPLAY_DEVICE** structure.

An application can open a display by calling

```
hdc = CreateDC(lpszDisplayName, NULL, NULL, lpDevMode);
```

In this call, the *lpszDisplayName* parameter is one of the device names returned by **EnumDisplayDevices** and *lpDevMode* is a description of the graphics mode for this device. The resulting hdc can be used to perform any graphics operation to the device.

Colors on Multiple Display Monitors

4/20/2022 • 2 minutes to read • Edit Online

Each monitor can have its own color depth. The system automatically adjusts colors as windows move across monitors with different color depths. In general, this produces good results. However, this is not always optimal. To take advantage of the color capabilities of different monitors, see the Painting on Multiple Display Monitors section that follows.

To determine if all monitors have the same color format, call **GetSystemMetrics** with SM_SAMEDISPLAYFORMAT.

If the primary monitor is palettized, **SelectPalette** and **RealizePalette** work the same as before, but across all monitors. In addition, the palettes of all palettized devices are synchronized. If the primary monitor is not palettized, **SelectPalette** and **RealizePalette** will select the palette into the background and palettized devices will not be synchronized.

Painting on Multiple Display Monitors

4/20/2022 • 2 minutes to read • Edit Online

The system automatically handles painting into a device context (DC) that spans more than one monitor, even when the monitors have different color depths. Usually this produces good results, but it may not be optimal. For example, a window on two monitors of vastly different color depths could have poor color rendition. Also, monitors with the same color depth may have different color formatsfor example, colors could be encoded with different numbers of bits, or be located in different places in a pixel's color value.

To get the best results for each of the monitors in a DC that spans more than one display, call **EnumDisplayMonitors** to enumerate the monitors that intersect your DC and paint the intersecting area in each of them separately according to the display attributes for that monitor. See the example in Painting on a DC That Spans Multiple Displays.

If you do all of your drawing in your **WM_PAINT** code and if your **WM_PAINT** code handles all of the various video modes, then you should be able to place your **WM_PAINT** code in the **MonitorEnumProc** of **EnumDisplayMonitors** with only a few modifications.

Positioning Objects on Multiple Display Monitors

4/20/2022 • 3 minutes to read • Edit Online

A window or menu that is on more than one monitor causes visual disruption for a viewer. To minimize this problem, the system displays menus and new and maximized windows on one monitor. The following table shows how the monitor is chosen.

OBJECT	LOCATION
window	CreateWindow(Ex) displays a window on the monitor that contains the largest part of the window.Maximizes on the monitor that contains the largest part of the window before it was minimized. The ALT-TAB key combination displays a window on the monitor that has the currently active window.
owned window	on the same monitor as its owner.
submenu	Appears on the monitor that contains the largest part of the corresponding menu item.
context menu	Appears on the monitor where the right click occurred.
drop-down list	Appears on the monitor that contains the rectangle of the combo box.
dialog box	Appears on the monitor of the window that owns it.If it is defined with DS_CENTERMOUSE style, it appears on the monitor with the mouse. If it has no owner and the active window and the dialog box are in the same application, the dialog box appears on the monitor of the currently active window. If the dialog box has no owner and the active window is not in the same application as the dialog box, the dialog box appears on the primary monitor.
message box	Appears on the monitor of the window that owns it.

If a window straddles two monitors and one of the monitors is repositioned, the system positions the window on the monitor that contains the largest part of the original window.

An application will also typically need to position objects. For example, a window may need to be created on the same monitor as another window.

To position an object on a multiple monitor system

1. Determine the appropriate monitor.
2. Get the coordinates to the monitor.
3. Position the object using the coordinates.

Typically, you will position an object either on the primary monitor or on a monitor that has an object already on it. To identify the monitor for a given point, rectangle, or window, use MonitorFromPoint, MonitorFromRect, and MonitorFromWindow.

To get the coordinates for the monitor, use GetMonitorInfo, which provides both the work area and the entire monitor rectangle. Note that SM_CXSCREEN and SM_CYSCREEN always refer to the primary monitor, not necessarily the monitor that displays your application. Also, avoid SM_xxVIRTUALSCREEN because this centers your window on the virtual screen, not a monitor.

To center dialog boxes in a window's work area, use the DS_CENTER style. To center the dialog box to an application window, use GetWindowRect. Windows automatically restricts menus and dialog boxes to a monitor. However, there can be a problem for custom menus, custom drop-down boxes, custom tool palettes, and the saved application position.

For an example of how to position objects correctly, see Positioning Objects on a Multiple Display Setup.

Using SM_CXSCREEN and SM_CYSCREEN to determine the location of an application desktop toolbar (also called *appbar*) restricts the appbar to the primary monitor. To allow an appbar to be on any edge of any monitor, use the appropriate system metrics to calculate the edges of the monitors. Also, use the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the coordinates, otherwise the sign of the coordinates may be wrong. These macros are included in Windowsx.h.

The size of a full-screen window needs to change as it moves among monitors with different resolutions. To do this, the application must check what window it is on, using MonitorFromWindow or MonitorFromPoint , and then use GetMonitorInfo to get the size of the monitor. As an alternative, you could use the **HMONITOR** from the DirectX **DirectDrawEnumerateEx** function. Then use SetWindowPos to position and size the window to cover the monitor.

A maximized window does not cover a taskbar that has the "Always on top" property. However, a full-screen window covers the taskbar--for example, in Microsoft PowerPoint slide shows and games.

To save, and later restore, the position of a window when an application exits, use the GetWindowPlacement and SetWindowPlacement functions. However, check that the position is still valid before using it because the monitor could have been moved or removed from the system. The application displays the window on the primary monitor if the **HMONITOR** of a window is invalid.

The system tries to start an application on the monitor that contains its shortcut. So, one way to position an application is to have its shortcut on a desired monitor.

If you use ShellExecute or ShellExecuteEx , supply an *hWnd* so the system will open any new windows on the same monitor as the calling application.

Note that the values for the MINMAXINFO structure are slightly altered for a system with multiple monitors.

Multiple Monitor Applications on Different Systems

4/20/2022 • 2 minutes to read • Edit Online

To have your multiple monitoraware application work both on systems with and without multiple monitor support, link your application with Multimon.h. You must also define COMPILE_MULTIMON_STUBS in exactly one C file. If the system does not support multiple monitors, this returns default values from **GetSystemMetrics** , and the multiple monitor functions act as if there is only one display. On multiple monitor systems, your application will work normally.

Because negative coordinates can happen easily in a multimonitor system, you should retrieve coordinates that are packed in the lParam by using the **GET_X_LPARAM** and **GET_Y_LPARAM** macros.

Do not use negative coordinates or coordinates larger than SM_CXSCREEN and SM_CYSCREEN to hide a window. Windows that use these limits to hide may appear on another monitor. Likewise, do not use these limits to keep a window visible because this can cause a window to snap to the primary monitor. It is best to reexamine existing applications for these problems. However, you can minimize problems in existing applications by running the application on the primary monitor or by keeping the primary monitor in the upper-left corner of the virtual screen.

Note that SM_CXMAXTRACK and SM_CYMAXTRACK are defined for the desktop, not just one monitor. Windows using these limits may need to be redefined.

A parent or related window might not be on the same monitor as a child window. To locate the monitor of a window, applications should use the **MonitorFromWindow** function.

To have a screen saver display on all monitors, link with the latest version of Scrnsave.lib. Otherwise, the screen saver may only appear on the primary monitor and leave the other monitors untouched. Screen savers linked with the latest Scrnsave.lib will work on both single and multiple monitor systems. To have a different screen saver on each monitor, use the multiple monitor functions to handle each monitor separately.

Input devices that deliver coordinates to the system in absolute coordinates, such as tablets, have their cursor input restricted to the primary monitor. To switch tablet input between monitors, see the instructions from the OEM.

To map mouse input that is sent in absolute coordinates to the entire virtual screen, use the **INPUT** structure with MOUSEEVENTF_ABSOLUTE and MOUSEEVENTF_VIRTUALDESKTOP.

The **BitBlt** function works well for multiple monitor systems. However, the **MaskBlt**, **PlgBlt**, **StretchBlt**, and **TransparentBlt** functions will fail if the source and destination device contexts are different.

Multiple Monitor Considerations for Older Programs

4/20/2022 • 2 minutes to read • Edit Online

Generally, older applications do not need changes to work on multiple monitor systems. For most, the system will appear to have only one display. System metrics reflect the primary display and fullscreen applications show up on the primary. The system handles minimization, maximization, menus, and dialog boxes.

However, some programs will have problems. Some that could have problems are remote control applications, those that have direct hardware access, and those that have patched the GDI or DISPLAY driver. In these cases, the user may be required to disable any monitor beyond the primary monitor.

Using Multiple Display Monitors

4/20/2022 • 2 minutes to read • Edit Online

The examples in this section deal with different aspects of using Multiple Display Monitors.

- Painting on a DC That Spans Multiple Displays
- Positioning Objects on a Multiple Display Setup

Painting on a DC That Spans Multiple Displays

4/20/2022 • 2 minutes to read • Edit Online

To respond to a **WM_PAINT** message, use code like the following.

```
case WM_PAINT:
hdc = BeginPaint(hwnd, &ps);
EnumDisplayMonitors(hdc, NULL, MyPaintEnumProc, 0);
EndPaint(hwnd, &ps);
```

To paint the top half of a window, use code like the following.

```
GetClient Rect(hwnd, &rc);
rc.bottom = (rc.bottom - rc.top) / 2;
hdc = GetDC(hwnd);
EnumDisplayMonitors(hdc, &rc, MyPaintEnumProc, 0);
ReleaseDC(hwnd, hdc);
```

To paint the entire virtual screen optimally for each monitor, use code like the following.

```
hdc = GetDC(NULL);
EnumDisplayMonitors(hdc, NULL, MyPaintScreenEnumProc, 0);
ReleaseDC(NULL, hdc);
```

Positioning Objects on a Multiple Display Setup

4/20/2022 • 2 minutes to read • Edit Online

The following sample code demonstrates how applications can correctly position objects on multiple displays. Note, do not assume that the RECT is based on the origin (0,0).

```
#include <windows.h>
#include "multimon.h"

#define MONITOR_CENTER     0x0001        // center rect to monitor
#define MONITOR_CLIP     0x0000        // clip rect to monitor
#define MONITOR_WORKAREA 0x0002        // use monitor work area
#define MONITOR_AREA     0x0000        // use monitor entire area

//
//  ClipOrCenterRectToMonitor
//
//  The most common problem apps have when running on a
//  multimonitor system is that they "clip" or "pin" windows
//  based on the SM_CXSCREEN and SM_CYSCREEN system metrics.
//  Because of app compatibility reasons these system metrics
//  return the size of the primary monitor.
//
//  This shows how you use the multi-monitor functions
//  to do the same thing.
//
void ClipOrCenterRectToMonitor(LPRECT prc, UINT flags)
{
    HMONITOR hMonitor;
    MONITORINFO mi;
    RECT        rc;
    int         w = prc->right  - prc->left;
    int         h = prc->bottom - prc->top;

    //
    // get the nearest monitor to the passed rect.
    //
    hMonitor = MonitorFromRect(prc, MONITOR_DEFAULTTONEAREST);

    //
    // get the work area or entire monitor rect.
    //
    mi.cbSize = sizeof(mi);
    GetMonitorInfo(hMonitor, &mi);

    if (flags & MONITOR_WORKAREA)
        rc = mi.rcWork;
    else
        rc = mi.rcMonitor;

    //
    // center or clip the passed rect to the monitor rect
    //
    if (flags & MONITOR_CENTER)
    {
        prc->left   = rc.left + (rc.right  - rc.left - w) / 2;
        prc->top    = rc.top  + (rc.bottom - rc.top  - h) / 2;
        prc->right  = prc->left + w;
        prc->bottom = prc->top  + h;
    }
    else
    {
        prc->left   = max(rc.left, min(rc.right-w,  prc->left));
        prc->top    = max(rc.top,  min(rc.bottom-h, prc->top));
        prc->right  = prc->left + w;
        prc->bottom = prc->top  + h;
    }
}

void ClipOrCenterWindowToMonitor(HWND hwnd, UINT flags)
{
    RECT rc;
    GetWindowRect(hwnd, &rc);
    ClipOrCenterRectToMonitor(&rc, flags);
    SetWindowPos(hwnd, NULL, rc.left, rc.top, 0, 0, SWP_NOSIZE | SWP_NOZORDER | SWP_NOACTIVATE);
}
```

Multiple Display Monitors Reference

4/20/2022 • 2 minutes to read • Edit Online

The following elements provide support for multiple monitors:

- Multiple Display Monitors Functions
- Multiple Display Monitors Structures

Multiple Display Monitors Functions

4/20/2022 • 2 minutes to read • Edit Online

The following functions provide support for multiple monitors.

FUNCTION	DESCRIPTION
EnumDisplayMonitors	Enumerates display monitors that intersect a region formed by the intersection of a specified clipping rectangle and the visible region of a device context.
GetMonitorInfo	Retrieves information about a display monitor.
MonitorEnumProc	An application-defined callback function that is called by the EnumDisplayMonitors function.
MonitorFromPoint	Retrieves a handle to the display monitor that contains a specified point.
MonitorFromRect	Retrieves a handle to the display monitor that has the largest area of intersection with a specified rectangle.
MonitorFromWindow	Retrieves a handle to the display monitor that has the largest area of intersection with the bounding rectangle of a specified window.

Multiple Display Monitors Structures

4/20/2022 • 2 minutes to read • Edit Online

The following structures are used with the monitor functions:

- MONITORINFO
- MONITORINFOEX

Painting and Drawing

4/20/2022 • 2 minutes to read • Edit Online

This overview describes how the system manages output to the screen and explains what applications must do to draw in a window. In particular, this overview describes *display device contexts* (or, more simply, *display DCs*) and how to prepare and use them. The overview does not explain how to use graphics device interface (GDI) functions to generate output, nor does it explain how to print.

About Painting and Drawing

4/20/2022 • 2 minutes to read • Edit Online

Nearly all applications use the screen to display the data they manipulate. An application paints images, draws figures, and writes text so that the user can view data as it is created, edited, and printed. Microsoft Windows provides rich support for painting and drawing, but, because of the nature of multitasking operating systems, applications must cooperate with one another when accessing the screen.

To keep all applications functioning smoothly and cooperatively, the system manages all output to the screen. Applications use windows as their primary output device rather than the screen itself. The system supplies display device contexts that uniquely correspond to the windows. Applications use display device contexts to direct their output to the specified windows. Drawing in a window (directing output to it) prevents an application from interfering with the output of other applications and allows applications to coexist with one another while still taking full advantage of the graphics capabilities of the system.

- When to Draw in a Window
- The WM_PAINT Message
- Drawing Without the WM_PAINT Message
- Window Coordinate System
- Window Regions
- Window Background
- Minimized Windows
- Resized Windows
- Nonclient Area
- Child Window Update Region
- Display Devices
- Window Update Lock
- Accumulated Bounding Rectangle

When to Draw in a Window

4/20/2022 • 2 minutes to read • Edit Online

An application draws in a window at a variety of times: when first creating a window, when changing the size of the window, when moving the window from behind another window, when minimizing or maximizing the window, when displaying data from an opened file, and when scrolling, changing, or selecting a portion of the displayed data.

The system manages actions such as moving and sizing a window. If an action affects the content of the window, the system marks the affected portion of the window as ready for updating and, at the next opportunity, sends a **WM_PAINT** message to the window procedure of the window. The message is a signal to the application to determine what must be updated and to carry out the necessary drawing.

Some actions are managed by the application, such as displaying open files and selecting displayed data. For these actions, an application can mark for updating the portion of the window affected by the action, causing a **WM_PAINT** message to be sent at the next opportunity. If an action requires immediate feedback, the application can draw while the action takes place, without waiting for **WM_PAINT**. For example, a typical application highlights the area the user selects rather than waiting for the next **WM_PAINT** message to update the area.

In all cases, an application can draw in a window as soon as it is created. To draw in the window, the application must first retrieve a handle to a display device context for the window. Ideally, an application carries out most of its drawing operations during the processing of **WM_PAINT** messages. In this case, the application retrieves a display device context by calling the **BeginPaint** function. If an application draws at any other time, such as from within **WinMain** or during the processing of keyboard or mouse messages, it calls the **GetDC** or **GetDCEx** function to retrieve the display DC.

The WM_PAINT Message

4/20/2022 • 2 minutes to read • Edit Online

Typically, an application draws in a window in response to a **WM_PAINT** message. The system sends this message to a window procedure when changes to the window have altered the content of the client area. The system sends the message only if there are no other messages in the application message queue.

Upon receiving a **WM_PAINT** message, an application can call **BeginPaint** to retrieve the display device context for the client area and use it in calls to GDI functions to carry out whatever drawing operations are necessary to update the client area. After completing the drawing operations, the application calls the **EndPaint** function to release the display device context.

Before **BeginPaint** returns the display device context, the system prepares the device context for the specified window. It first sets the clipping region for the device context to be equal to the intersection of the portion of the window that needs updating and the portion that is visible to the user. Only those portions of the window that have changed are redrawn. Attempts to draw outside this region are clipped and do not appear on the screen.

The system can also send **WM_NCPAINT** and **WM_ERASEBKGND** messages to the window procedure before **BeginPaint** returns. These messages direct the application to draw the nonclient area and window background. The *nonclient area* is the part of a window that is outside of the client area. The area includes features such as the title bar, window menu (also known as the **System** menu), and scroll bars. Most applications rely on the default window function, **DefWindowProc**, to draw this area and therefore pass the **WM_NCPAINT** message to this function. The *window background* is the color or pattern that a window is filled with before other drawing operations begin. The background covers any images previously in the window or on the screen under the window. If a window belongs to a window class having a class background brush, the **DefWindowProc** function draws the window background automatically.

BeginPaint fills a **PAINTSTRUCT** structure with information such as the dimensions of the portion of the window to be updated and a flag indicating whether the window background has been drawn. The application can use this information to optimize drawing. For example, it can use the dimensions of the update region, specified by the **rcPaint** member, to limit drawing to only those portions of the window that need updating. If an application has very simple output, it can ignore the update region and draw in the entire window, relying on the system to discard (clip) any unneeded output. Because the system clips drawing that extends outside the clipping region, only drawing that is in the update region is visible.

BeginPaint sets the update region of a window to **NULL**. This clears the region, preventing it from generating subsequent **WM_PAINT** messages. If an application processes a **WM_PAINT** message but does not call **BeginPaint** or otherwise clear the update region, the application continues to receive **WM_PAINT** messages as long as the region is not empty. In all cases, an application must clear the update region before returning from the **WM_PAINT** message.

After the application finishes drawing, it should call **EndPaint**. For most windows, **EndPaint** releases the display device context, making it available to other windows. **EndPaint** also shows the caret, if it was previously hidden by **BeginPaint**. **BeginPaint** hides the caret to prevent drawing operations from corrupting it.

- The Update Region
- Invalidating and Validating the Update Region
- Retrieving the Update Region
- Excluding the Update Region
- Synchronous and Asynchronous Drawing

The Update Region

4/20/2022 • 2 minutes to read • Edit Online

The *update region* identifies the portion of a window that is out-of-date or invalid and in need of redrawing. The system uses the update region to generate **WM_PAINT** messages for applications and to minimize the time applications spend bringing the contents of their windows up to date. The system adds only the invalid portion of the window to the update region, requiring only that portion to be drawn.

When the system determines that a window needs updating, it sets the dimensions of the update region to the invalid portion of the window. Setting the update region does not immediately cause the application to draw. Instead, the application continues retrieving messages from the application message queue until no messages remain. The system then checks the update region, and if the region is not empty (non-NULL), it sends a **WM_PAINT** message to the window procedure.

An application can use the update region to generate its **WM_PAINT** messages. For example, an application that loads data from open files typically sets the update region while loading so that new data is drawn during processing of the next **WM_PAINT** message. In general, an application should not draw at the time its data changes, but route all drawing operations through the **WM_PAINT** message.

Invalidating and Validating the Update Region

4/20/2022 • 2 minutes to read • Edit Online

An application invalidates a portion of a window and sets the update region by using the **InvalidateRect** or **InvalidateRgn** function. These functions add the specified rectangle or region (in client coordinates) to the update region, combining the rectangle or region with anything the system or the application may have previously added to the update region.

The **InvalidateRect** and **InvalidateRgn** functions do not generate **WM_PAINT** messages. Instead, the system accumulates the changes made by these functions and its own changes while a window processes other messages in its message queue. By accumulating changes, a window processes all changes at once instead of updating bits and pieces one step at a time.

The **ValidateRect** and **ValidateRgn** functions validate a portion of the window by removing a specified rectangle or region from the update region. These functions are typically used when the window has updated a specific part of the screen in the update region before receiving the **WM_PAINT** message.

Retrieving the Update Region

4/20/2022 • 2 minutes to read • Edit Online

The **GetUpdateRect** and **GetUpdateRgn** functions retrieve the current update region for the window. GetUpdateRect retrieves the smallest rectangle (in logical coordinates) that encloses the entire update region. GetUpdateRgn retrieves the update region itself. These functions can be used to calculate the current size of the update region to determine where to carry out a drawing operation.

BeginPaint also retrieves the dimensions of the smallest rectangle enclosing the current update region, copying the dimensions to the **rcPaint** member in the **PAINTSTRUCT** structure. Because **BeginPaint** validates the update region, any call to **GetUpdateRect** and **GetUpdateRgn** immediately after a call to **BeginPaint** returns an empty update region.

Excluding the Update Region

4/20/2022 • 2 minutes to read • Edit Online

The **ExcludeUpdateRgn** function excludes the update region from the clipping region for the display device context. This function is useful when drawing in a window other than when a WM_PAINT message is processing. It prevents drawing in the areas that will be updated during the next WM_PAINT message.

Synchronous and Asynchronous Drawing

4/20/2022 • 2 minutes to read • Edit Online

Most drawing carried out during processing of the **WM_PAINT** message is asynchronous; that is, there is a delay between the time a portion of the window is invalidated and the time WM_PAINT is sent. During the delay, the application typically retrieves messages from the queue and carries out other tasks. The reason for the delay is that the system generally treats drawing in a window as a low-priority operation and works as though user-input messages and messages that may affect the position or size of a window will be processed before WM_PAINT .

In some cases, it is necessary for an application to draw synchronously that is, draw in the window immediately after invalidating a portion of the window. A typical application draws its main window immediately after creating the window to signal the user that the application has started successfully. The system draws some control windows synchronously, such as buttons, because such windows serve as the focus for user input. Although any window with a simple drawing routine can be drawn synchronously, all such drawing should be done quickly and should not interfere with the application's ability to respond to user input.

The **UpdateWindow** and **RedrawWindow** functions allow for synchronous drawing. **UpdateWindow** sends a **WM_PAINT** message directly to the window if the update region is not empty. **RedrawWindow** also sends a **WM_PAINT** message, but gives the application greater control over how to draw the window, such as whether to draw the nonclient area and window background or whether to send the message regardless of whether the update region is empty. These functions send the **WM_PAINT** message directly to the window, regardless of the number of other messages in the application message queue.

Any window requiring time-consuming drawing operations should be drawn asynchronously to prevent pending messages from being blocked as the window is drawn. Also, any application that frequently invalidates small portions of a window should allow these invalid portions to consolidate into a single asynchronous **WM_PAINT** message, rather than a series of synchronous **WM_PAINT** messages.

Drawing Without the WM_PAINT Message

4/20/2022 • 2 minutes to read • Edit Online

Although applications carry out most drawing operations while the **WM_PAINT** message is processing, it is sometimes more efficient for an application to draw directly in a window without relying on the **WM_PAINT** message. This can be useful when the user needs immediate feedback, such as when selecting text and dragging or sizing an object. In such cases, the application usually draws while processing keyboard or mouse messages.

To draw in a window without using a **WM_PAINT** message, the application uses the **GetDC** or **GetDCEx** function to retrieve a display device context for the window. With the display device context, the application can draw in the window and avoid intruding into other windows. When the application has finished drawing, it calls the **ReleaseDC** function to release the display device context for use by other applications.

When drawing without using a **WM_PAINT** message, the application usually does not invalidate the window. Instead, it draws in such a fashion that it can easily restore the window and remove the drawing. For example, when the user selects text or an object, the application typically draws the selection by inverting whatever is already in the window. The application can remove the selection and restore the original contents of the window by simply inverting again.

The application is responsible for carefully managing any changes it makes to the window. In particular, if an application draws a selection and an intervening **WM_PAINT** message occurs, the application must ensure that any drawing done during the message does not corrupt the selection. To avoid this, many applications remove the selection, carry out usual drawing operations, and then restore the selection when drawing is complete.

Window Coordinate System

4/20/2022 • 2 minutes to read • Edit Online

The coordinate system for a window is based on the coordinate system of the display device. The basic unit of measure is the device unit (typically, the pixel). Points on the screen are described by x- and y-coordinate pairs. The x-coordinates increase to the right; y-coordinates increase from top to bottom. The origin (0,0) for the system depends on the type of coordinates being used.

The system and applications specify the position of a window on the screen in *screen coordinates*. For screen coordinates, the origin is the upper-left corner of the screen. The full position of a window is often described by a RECT structure containing the screen coordinates of two points that define the upper-left and lower-right corners of the window.

The system and applications specify the position of points in a window by using *client coordinates*. The origin in this case is the upper-left corner of the window or client area. Client coordinates ensure that an application can use consistent coordinate values while drawing in the window, regardless of the position of the window on the screen.

The dimensions of the client area are also described by a RECT structure that contains client coordinates for the area. In all cases, the upper-left coordinate of the rectangle is included in the window or client area, while the lower-right coordinate is excluded. Graphics operations in a window or client area are excluded from the right and lower edges of the enclosing rectangle.

Occasionally, applications may be required to map coordinates in one window to those of another window. An application can map coordinates by using the MapWindowPoints function. If one of the windows is the desktop window, the function effectively converts screen coordinates to client coordinates and vice versa; the desktop window is always specified in screen coordinates.

Window Regions

4/20/2022 • 2 minutes to read • Edit Online

In addition to the update region, every window has a *visible region* that defines the window portion visible to the user. The system changes the visible region for the window whenever the window changes size or whenever another window is moved such that it obscures or exposes a portion of the window. Applications cannot change the visible region directly, but the system automatically uses the visible region to create the clipping region for any display device context retrieved for the window.

The *clipping region* determines where the system permits drawing. When the application retrieves a display device context using the **BeginPaint**, **GetDC**, or **GetDCEx** function, the system sets the clipping region for the device context to the intersection of the visible region and the update region. Applications can change the clipping region by using functions such as **SetWindowRgn**, **SelectClipPath** and **SelectClipRgn**, to further limit drawing to a particular portion of the update area.

The WS_CLIPCHILDREN and WS_CLIPSIBLINGS styles further specify how the system calculates the visible region for a window. If a window has one or both of these styles, the visible region excludes any child window or sibling windows (windows having the same parent window). Therefore, drawing that would otherwise intrude in these windows will always be clipped.

Window Background

4/20/2022 • 2 minutes to read • Edit Online

The window background is the color or pattern used to fill the client area before a window begins drawing. The window background covers whatever was on the screen before the window was moved there, erasing existing images and preventing the application's new output from being mixed with unrelated information.

The system paints the background for a window or gives the window the opportunity to do so by sending it a **WM_ERASEBKGND** message when the application calls **BeginPaint**. If an application does not process the message but passes it to **DefWindowProc**, the system erases the background by filling it with the pattern in the background brush specified by the window's class. If the brush is not valid or the class has no background brush, the system sets the **fErase** member in the **PAINTSTRUCT** structure that **BeginPaint** returns, but carries out no other action. The application then has a second chance to draw the window background, if necessary.

If it processes **WM_ERASEBKGND**, the application should use the message's *wParam* parameter to draw the background. This parameter contains a handle to the display device context for the window. After drawing the background, the application should return a nonzero value. This ensures that **BeginPaint** does not erroneously set the **fErase** member of the **PAINTSTRUCT** structure to a nonzero value (indicating the background should be erased) when the application processes the subsequent **WM_PAINT** message.

An application can define a class background brush by assigning a brush handle or a system color value to the **hbrBackground** member of the **WNDCLASS** structure when registering the class with the **RegisterClass** function. The **GetStockObject** or **CreateSolidBrush** function can be used to create a brush handle. A system color value can be one of those defined for the **SetSysColors** function. (The value must be increased by one before it is assigned to the member.)

An application can process the **WM_ERASEBKGND** message even though a class background brush is defined. This is typical in applications that enable the user to change the window background color or pattern for a specified window without affecting other windows in the class. In such cases, the application must not pass the message to **DefWindowProc**.

It is not necessary for an application to align brushes, because the system draws the brush using the window origin as the point of reference. Given this, the user can move the window without affecting the alignment of pattern brushes.

Minimized Windows

4/20/2022 • 2 minutes to read • Edit Online

The system reduces an application's main window (overlapping style) to a minimized window when the user clicks Minimize from the window menu or the application calls the **ShowWindow** function and specifies a value such as SW_MINIMIZE. Minimizing a window speeds up system performance by reducing the amount of work an application must do when updating its main window.

For a typical application, the system draws an icon, called the class icon, when the window is minimized, labeling the icon with the name of the window. The class icon, a static image that represents the application, is specified by the application when it registers the window class. The application assigns a handle to the class icon to the **hIcon** member of **WNDCLASS** before calling **RegisterClass**. The application can use the **LoadIcon** function to retrieve the icon handle.

Resized Windows

4/20/2022 • 2 minutes to read • Edit Online

The system changes the size of a window when the user chooses window menu commands, such as Size and Maximize, or when the application calls functions, such as the **SetWindowPos** function. When a window changes size, the system assumes that the contents of the previously exposed portion of the window are not affected and need not be redrawn. The system invalidates only the newly exposed portion of the window, which saves time when the eventual **WM_PAINT** message is processed by the application. In this case, **WM_PAINT** is not generated when the size of the window is reduced.

For some windows, any change to the size of the window invalidates the contents. For example, a clock application that adapts the face of the clock to fit neatly within its window must redraw the clock whenever the window changes size. To force the system to invalidate the entire client area of the window when a vertical, horizontal, or both vertical and horizontal change is made, an application must specify the CS_VREDRAW or CS_HREDRAW style, or both, when registering the window class. Any window belonging to a window class having these styles is invalidated each time the user or the application changes the size of the window.

Nonclient Area

4/20/2022 • 2 minutes to read • Edit Online

The system sends a WM_NCPAINT message to the window whenever a part of the nonclient area of the window, such as the title bar, menu bar, or window frame, must be updated. The system may also send other messages to direct a window to update a portion of its client area; for example, when a window becomes active or inactive, it sends the WM_NCACTIVATE message to update its title bar. In general, processing these messages for standard windows is not recommended, because the application must be able to draw all the required parts of the nonclient area for the window. For this reason, most applications pass these messages to DefWindowProc for default processing.

An application that creates custom nonclient areas for its windows must process these messages. When doing so, the application must use a window device context to carry out drawing in the window. The *window device context* enables the application to draw in all portions of the window, including the nonclient area. An application retrieves a window device context by using the GetWindowDC or GetDCEx function and, when drawing is complete, must release the window device context by using the ReleaseDC function.

The system maintains an update region for the nonclient area. When an application receives a WM_NCPAINT message, the *wParam* parameter contains a handle to a region defining the dimensions of the update region. The application can use the handle to combine the update region with the clipping region for the window device context. The system does not automatically combine the update region when retrieving the window device context unless the application uses GetDCEx and specifies both the region handle and the DCX_INTERSECTRGN flag. If the application does not combine the update region, only drawing operations that would otherwise extend outside the window are clipped. The application is not responsible for clearing the update region, regardless of whether it uses the region.

If an application processes the WM_NCACTIVATE message, after processing it must return **TRUE** to direct the system to complete the change of active window. If the window is minimized when the application receives the **WM_NCACTIVATE** message, it should pass the message to DefWindowProc. In such cases, the default function redraws the label for the icon.

Child Window Update Region

4/20/2022 • 2 minutes to read • Edit Online

A child window is a window with the WS_CHILD or WS_CHILDWINDOW style. Like other window styles, child windows receive **WM_PAINT** messages to prompt updating. Each child window has an update region, which either the system or the application can set to generate eventual **WM_PAINT** messages.

A child window's update and visible regions are affected by the child's parent window; this is not true for windows of other styles. The system often sets the child window's update region when it sets the parent window's update region, causing the child window to receive **WM_PAINT** messages when the parent window receives them. The system limits the location of the child window's visible region to within the client area of the parent window and clips any portion of the child window moved outside the parent window.

The system sets the update region for a child window whenever part of the parent window's update region includes a portion of the child window. In such cases, the system first sends a **WM_PAINT** message to the parent window and then sends a message to the child window, allowing the child to restore any portions of the window that the parent may have drawn over.

The system does not set the parent's update region when the child's is set. An application cannot generate a **WM_PAINT** message for the parent window by invalidating the child window. Similarly, an application cannot generate a **WM_PAINT** message for the child by invalidating a portion of the parent's client area that lies entirely under the child window. In such cases, neither window receives a **WM_PAINT** message.

An application can prevent a child window's update region from being set when the parent window's is set by specifying the WS_CLIPCHILDREN style when creating the parent window. When this style is set, the system excludes the child windows from the parent's visible region and therefore ignores any portion of the update region that may contain the child windows. When the application paints in the parent window, any drawing that would cover the child window is clipped, making a subsequent **WM_PAINT** message to the child window unnecessary.

The update and visible regions of a child window are also affected by the child window's siblings. Sibling windows are any windows that have a common parent window. If sibling windows overlap, then setting the update region for one affects the update region of another, causing **WM_PAINT** messages to be sent to both windows. If a window in the parent chain is composited (a window with WX_EX_COMPOSITED), sibling windows receive **WM_PAINT** messages in the reverse order of their position in the Z order. Given this, the window highest in the Z order (on the top) receives its **WM_PAINT** message last, and vice versa. If a window in the parent chain is not composited, sibling windows receive **WM_PAINT** messages in Z order.

Sibling windows are not automatically clipped. One sibling can draw over another overlapping sibling even if the window that is drawing has a lower position in the Z order. An application can prevent this by specifying the WS_CLIPSIBLINGS style when creating the windows. When this style is set, the system excludes all portions of an overlapping sibling window from a window's visible region if the overlapping sibling window has a higher position in the Z order.

NOTE

The update and visible regions for windows that have the WS_POPUP or WS_POPUPWINDOW style are not affected by their parent windows.

Display Devices

4/20/2022 • 2 minutes to read • Edit Online

Before painting, the system must prepare the display device for drawing operations. A display device context defines a set of graphic objects and their associated attributes, and the graphic modes that affect output. The system prepares each display device context for output to a window, setting the drawing objects, colors, and modes for the window instead of the display device. When the application supplies the display device context through calls to GDI functions, GDI uses the information in the context to generate output in the specified window without intruding on other windows or other parts of the screen.

The system provides five kinds of display device contexts.

TYPE	MEANING
common	Permits drawing in the client area of a specified window.
class	Permits drawing in the client area of a specified window.
parent	Permits drawing anywhere in the window. Although the parent device context also permits drawing in the parent window, it is not intended to be used in this way.
private	Permits drawing in the client area of a specified window.
window	Permits drawing anywhere in the window.

The system supplies a common, class, parent, or private device context to a window based on the type of display device context specified in that window's class style. The system supplies a window device context only when the application explicitly requests one for example, by calling the **GetWindowDC** or **GetDCEx** function. In all cases, an application can use the **WindowFromDC** function to determine which window a display DC currently represents.

This section provides information on the following topics.

- Display Device Context Cache
- Display Device Context Defaults
- Common Display Device Contexts
- Private Display Device Contexts
- Parent Display Device Contexts
- Class Display Device Contexts
- Window Display Device Contexts
- Parent Display Device Contexts

Display Device Context Cache

4/20/2022 • 2 minutes to read • Edit Online

The system maintains a cache of display device contexts that it uses for common, parent, and window device contexts. The system retrieves a device context from the cache whenever an application calls the **GetDC** or **BeginPaint** function; the system returns the DC to the cache when the application subsequently calls the **ReleaseDC** or **EndPaint** function.

There is no predetermined limit on the amount of device contexts that a cache can hold; the system creates a new display device context for the cache if none is available. Given this, an application can have more than five active device contexts from the cache at a time. However, the application must continue to release these device contexts after use. Because new display device contexts for the cache are allocated in the application's heap space, failing to release the device contexts eventually consumes all available heap space. The system indicates this failure by returning an error when it cannot allocate space for the new device context. Other functions unrelated to the cache may also return errors.

Display Device Context Defaults

4/20/2022 • 2 minutes to read • Edit Online

Upon first creating a display device context, the system assigns default values for the attributes (that is, drawing objects, colors, and modes) that comprise the device context. The following table shows the default values for the attributes of a display device context.

ATTRIBUTE	DEFAULT VALUE
Background color	Background color setting from Control Panel (typically, white).
Background mode	OPAQUE
Bitmap	None
Brush	WHITE_BRUSH
Brush origin	(0,0)
Clipping region	Entire window or client area with the update region clipped, as appropriate. Child and pop-up windows in the client area may also be clipped.
Palette	DEFAULT_PALETTE
Current pen position	(0,0)
Device origin	Upper left corner of the window or the client area.
Drawing mode	R2_COPYPEN
Font	SYSTEM_FONT
Intercharacter spacing	0
Mapping mode	MM_TEXT
Pen	BLACK_PEN
Polygon -fill mode	ALTERNATE
Stretch mode	BLACKONWHITE
Text color	Text color setting from Control Panel (typically, black).
Viewport extent	(1,1)
Viewport origin	(0,0)

ATTRIBUTE	DEFAULT VALUE
Window extent	(1,1)
Window origin	(0,0)

An application can modify the values of the display device context attributes by using selection and attribute functions, such as SelectObject, SetMapMode, and SetTextColor. For example, an application can modify the default units of measure in the coordinate system by using **SetMapMode** to change the mapping mode.

Changes to the attribute values of a common, parent, or window device context are not permanent. When an application releases these device contexts, the current selections, such as mapping mode and clipping region, are lost as the context is returned to the cache. Changes to a class or private device context persist indefinitely. To restore them to their original defaults, an application must explicitly set each attribute.

Common Display Device Contexts

4/20/2022 • 2 minutes to read • Edit Online

A *common device context* is used for drawing in the client area of the window. The system provides a common device context by default for any window whose window class does not explicitly specify a display device context style. Common device contexts are typically used with windows that can be drawn without extensive changes to the device context attributes. Common device contexts are convenient because they do not require additional memory or system resources, but they can be inconvenient if the application must set up many attributes before using them.

The system retrieves all common device contexts from the display device context cache. An application can retrieve a common device context immediately after the window is created. Because the common device context is from the cache, the application must always release the device context as soon as possible after drawing. After the common device context is released, it is no longer valid and the application must not attempt to draw with it. To draw again, the application must retrieve a new common device context, and continue to retrieve and release a common device context each time it draws in the window. If the application retrieves the device context handle by using the **GetDC** function, it must use the **ReleaseDC** function to release the handle. Similarly, for each **BeginPaint** function, the application must use a corresponding **EndPaint** function.

When the application retrieves the device context, the system adjusts the origin so that it aligns with the upper left corner of the client area. It also sets the clipping region so that output to the device context is clipped to the client area. Any output that would otherwise appear outside the client area is clipped. If the application retrieves the common device context by using **BeginPaint**, the system also includes the update region in the clipping region to further restrict the output.

When an application releases a common device context, the system restores the default values for the attributes of the device context. An application that modifies attribute values must do so each time it retrieves a common device context. Releasing the device context releases any drawing objects the application may have selected into it, so the application need not release these objects before releasing the device context. In all cases, an application must never assume that the common device context retains nondefault selections after being released.

Private Display Device Contexts

4/20/2022 • 2 minutes to read • Edit Online

A *private device context* enables an application to avoid retrieving and initializing a display device context each time the application must draw in a window. Private device contexts are useful for windows that require many changes to the values of the attributes of the device context to prepare it for drawing. Private device contexts reduce the time required to prepare the device context and therefore the time needed to carry out drawing in the window.

An application directs the system to create a private device context for a window by specifying the CS_OWNDC style in the window class. The system creates a unique private device context each time it creates a new window belonging to the class. Initially, the private device context has the same default values for attributes as a common device context, but the application can modify these at any time. The system preserves changes to the device context for the life of the window or until the application makes additional changes.

An application can retrieve a handle to the private device context by using the **GetDC** function any time after the window is created. The application must retrieve the handle only once. Thereafter, it can keep and use the handle any number of times. Because a private device context is not part of the display device context cache, an application need never release the device context by using the **ReleaseDC** function.

The system automatically adjusts the device context to reflect changes to the window, such as moving or sizing. This ensures that any overlapping windows are always properly clipped; that is, no action is required by the application to ensure clipping. However, the system does not revise the device context to include the update region. Therefore, when processing a **WM_PAINT** message, the application must incorporate the update region either by calling **BeginPaint** or by retrieving the update region and intersecting it with the current clipping region. If the application does not call **BeginPaint**, it must explicitly validate the update region by using the **ValidateRect** or **ValidateRgn** function. If the application does not validate the update region, the window receives an endless series of **WM_PAINT** messages.

Because **BeginPaint** hides the caret if a window is showing it, an application that calls **BeginPaint** should also call the **EndPaint** function to restore the caret. **EndPaint** has no other effect on a private device context.

Although a private device context is convenient to use, it is memory-intensive in terms of system resources, requiring 800 or more bytes to store. Private device contexts are recommended when performance considerations outweigh storage costs.

The system includes the private device context when sending the **WM_ERASEBKGND** message to the application. The current selections of the private device context, including mapping mode, are in effect when the application or the system processes these messages. To avoid undesirable effects, the system uses logical coordinates when erasing the background; for example, it uses the **GetClipBox** function to retrieve the logical coordinates of the area to erase and passes these coordinates to the **FillRect** function. Applications that process these messages can use similar techniques.

An application can use the **GetDCEx** function to force the system to return a common device context for the window that has a private device context. This is useful for carrying out quick touch-ups to a window without changing the current values of the attributes of the private device context.

Class Display Device Contexts

4/20/2022 • 2 minutes to read • Edit Online

By using a *class device context*, an application can use a single display device context for every window belonging to a specified class. Class device contexts are often used with control windows that are drawn using the same attribute values. Like private device contexts, class device contexts minimize the time required to prepare a device context for drawing.

The system supplies a class device context for a window if it belongs to a window class having the CS_CLASSDC style. The system creates the device context when creating the first window belonging to the class and then uses the same device context for all subsequently created windows in the class. Initially, the class device context has the same default values for attributes as a common device context, but the application can modify these at any time. The system preserves all changes, except for the clipping region and device origin, until the last window in the class has been destroyed. A change made for one window applies to all windows in that class.

An application can retrieve the handle to the class device context by using the **GetDC** function any time after the first window has been created. The application can keep and use the handle without releasing it because the class device context is not part of the display device context cache. If the application creates another window in the same window class, the application must retrieve the class device context again. Retrieving the device context sets the correct device origin and clipping region for the new window. After the application retrieves the class device context for a new window in the class, the device context can no longer be used to draw in the original window without again retrieving it for that window. In general, each time it must draw in a window, an application must explicitly retrieve the class device context for the window.

Applications that use class device contexts should always call **BeginPaint** when processing a **WM_PAINT** message. The function sets the correct device origin and clipping region for the window, and incorporates the update region. The application should also call **EndPaint** to restore the caret if **BeginPaint** hid it. **EndPaint** has no other effect on a class device context.

The system passes the class device context when sending the **WM_ERASEBKGND** message to the application, permitting the current attribute values to affect any drawing carried out by the application or the system when processing this message. As it could with a window having a private device context, an application can use **GetDCEx** to force the system to return a common device context for the window that has a class device context.

Using class device contexts is not recommended.

Window Display Device Contexts

4/20/2022 • 2 minutes to read • Edit Online

A *window device context* enables an application to draw anywhere in a window, including the nonclient area. Window device contexts are typically used by applications that process the **WM_NCPAINT** and **WM_NCACTIVATE** messages for windows with custom nonclient areas. Using a window device context is not recommended for any other purpose.

An application can retrieve a window device context by using the **GetWindowDC** or **GetDCEx** function with the DCX_WINDOW option specified. The function retrieves a window device context from the display device context cache. A window that uses a window device context must release it after drawing by using the **ReleaseDC** function as soon as possible. Window device contexts are always from the cache; the CS_OWNDC and CS_CLASSDC class styles do not affect the device context.

When an application retrieves a window device context, the system sets the device origin to the upper left corner of the window instead of the upper left corner of the client area. It also sets the clipping region to include the entire window, not just the client area. The system sets the current attribute values of a window device context to the same default values as a common device context. An application can change the attribute values, but the system does not preserve any changes when the device context is released.

Parent Display Device Contexts

4/20/2022 • 2 minutes to read • Edit Online

A *parent device context* enables an application to minimize the time necessary to set up the clipping region for a window. An application typically uses parent device contexts to speed up drawing for control windows without requiring a private or class device context. For example, the system uses parent device contexts for push button and edit controls. Parent device contexts are intended for use with child windows only, never with top-level or pop-up windows.

An application can specify the CS_PARENTDC style to set the clipping region of the child window to that of the parent window so that the child can draw in the parent. Specifying CS_PARENTDC enhances an application's performance because the system doesn't need to keep recalculating the visible region for each child window.

Attribute values set by the parent window are not preserved for the child window; for example, the parent window cannot set the brush for its child windows. The only property preserved is the clipping region. The window must clip its own output to the limits of the window. Because the clipping region for the parent device context is identical to the parent window, the child window can potentially draw over the entire parent window, but the parent device context must not be used in this way.

The system ignores the CS_PARENTDC style if the parent window uses a private or class device context, if the parent window clips its child windows, or if the child window clips its child windows or sibling windows.

Window Update Lock

4/20/2022 • 2 minutes to read • Edit Online

A *window update lock* is a temporary suspension of drawing in a window. The system uses the lock to prevent other windows from drawing over the tracking rectangle whenever the user moves or sizes a window. Applications can use the lock to prevent drawing if they carry out similar moving or sizing operations with their own windows.

An application uses the **LockWindowUpdate** function to set or clear a window update lock, specifying the window to lock. The lock applies to the specified window and all of its child windows. When the lock is set, the **GetDC** and **BeginPaint** functions return a display device context with a visible region that is empty. Given this, the application can continue to draw in the window, but all output is clipped. The lock persists until the application clears it by calling **LockWindowUpdate**, specifying **NULL** for the window. Although **LockWindowUpdate** forces a window's visible region to be empty, the function does not make the specified window invisible and does not clear the WS_VISIBLE style bit.

After the lock is set, the application can use the **GetDCEx** function, with the DCX_LOCKWINDOWUPDATE value, to retrieve a display device context to draw over the locked window. This allows the application to draw a tracking rectangle when processing keyboard or mouse messages. The system uses this method when the user moves and sizes windows. **GetDCEx** retrieves the display device context from the display device context cache, so the application must release the device context as soon as possible after drawing.

While a window update lock is set, the system creates an accumulated bounding rectangle for each locked window. When the lock is cleared, the system uses this bounding rectangle to set the update region for the window and its child windows, forcing an eventual **WM_PAINT** message. If the accumulated bounding rectangle is empty (that is, if no drawing has occurred while the lock was set), the update region is not set.

Accumulated Bounding Rectangle

4/20/2022 • 2 minutes to read • Edit Online

The *accumulated bounding rectangle* is the smallest rectangle enclosing the portion of a window or client area affected by recent drawing operations. An application can use this rectangle to conveniently determine the extent of changes caused by drawing operations. It is sometimes used in conjunction with **LockWindowUpdate** to determine which portion of the client area must be redrawn after the update lock is cleared.

An application uses the **SetBoundsRect** function (specifying DCB_ENABLE) to begin accumulating the bounding rectangle. The system subsequently accumulates points for the bounding rectangle as the application uses the specified display device context. The application can retrieve the current bounding rectangle at any time by using the **GetBoundsRect** function. The application stops the accumulation by calling **SetBoundsRect** again, specifying the DCB_DISABLE value.

Using the WM_PAINT Message

4/20/2022 • 2 minutes to read • Edit Online

You can use the **WM_PAINT** message to carry out the drawing necessary for displaying information. Because the system sends WM_PAINT messages to your application when your window must be updated or when you explicitly request an update, you can consolidate the code for drawing in your application's window procedure. You can then use this code whenever your application must draw either new or existing information.

The following sections show a variety of ways to use the WM_PAINT message to draw in a window:

- Drawing in the Client Area
- Redrawing the Entire Client Area
- Redrawing in the Update Region
- Invalidating the Client Area
- Drawing a Minimized Window
- Drawing a Custom Window Background

Drawing in the Client Area

4/20/2022 • 2 minutes to read • Edit Online

You use the BeginPaint and EndPaint functions to prepare for and complete the drawing in the client area. **BeginPaint** returns a handle to the display device context used for drawing in the client area; **EndPaint** ends the paint request and releases the device context.

In the following example, the window procedure writes the message "Hello, Windows!" in the client area. To make sure the string is visible when the window is first created, the WinMain function calls UpdateWindow immediately after creating and showing the window. This causes a WM_PAINT message to be sent immediately to the window procedure.

```
LRESULT APIENTRY WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);
            TextOut(hdc, 0, 0, "Hello, Windows!", 15);
            EndPaint(hwnd, &ps);
            return 0L;

        // Process other messages.
    }
}

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    HWND hwnd;

    hwnd = CreateWindowEx(
        // parameters
        );

    ShowWindow(hwnd, SW_SHOW);
    UpdateWindow(hwnd);

    return msg.wParam;
}
```

Redrawing the Entire Client Area

4/20/2022 • 2 minutes to read • Edit Online

You can have your application redraw the entire contents of the client area whenever the window changes size by setting the CS_HREDRAW and CS_VREDRAW styles for the window class. Applications that adjust the size of the drawing based on the size of the window use these styles to ensure that they start with a completely empty client area when drawing.

In the following example, the window procedure draws a five-pointed star that fits neatly in the client area. It uses a common device context and must set the mapping mode as well as window and viewport extents each time the **WM_PAINT** message is processed.

```
LRESULT APIENTRY WndProc(HWMD hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdc;
    RECT rc;
    POINT aptStar[6] = {50,2, 2,98, 98,33, 2,33, 98,98, 50,2};

    .
    .
    .

        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);
            GetClientRect(hwnd, &rc);
            SetMapMode(hdc, MM_ANISOTROPIC);
            SetWindowExtEx(hdc, 100, 100, NULL);
            SetViewportExtEx(hdc, rc.right, rc.bottom, NULL);
            Polyline(hdc, aptStar, 6);
            EndPaint(hwnd, &ps);
            return 0L;

        .
        .
        .
}


int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASS wc;

    .
    .
    .

        wc.style = CS_HREDRAW | CS_VREDRAW;
        wc.lpfnWndProc = (WNDPROC) WndProc;

    .
    .
    .

        RegisterClass(&wc);

    .
    .
    .

    return msg.wParam;
}
```

Redrawing in the Update Region

4/20/2022 • 2 minutes to read • Edit Online

You can limit the amount of drawing your application carries out when processing the WM_PAINT message by determining the size and location of the update region. Because the system uses the update region when creating the clipping region for the window's display device context, you can indirectly determine the update region by examining the clipping region.

In the following example, the window procedure draws a triangle, a rectangle, a pentagon, and a hexagon, but only if all or a portion of each figure lies within the update region. The window procedure uses the RectVisible function and a 100-by-100 rectangle to determine whether a figure is within the clipping region (and therefore the update region) for the common device context retrieved by BeginPaint.

```
POINT aptTriangle[4]  = {50,2, 98,86,  2,86, 50,2},
      aptRectangle[5] = { 2,2, 98,2,  98,98,  2,98, 2,2},
      aptPentagon[6]  = {50,2, 98,35, 79,90, 21,90, 2,35, 50,2},
      aptHexagon[7]   = {50,2, 93,25, 93,75, 50,98, 7,75, 7,25, 50,2};
  .
  .
  .

        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);
            SetRect(&rc, 0, 0, 100, 100);

            if (RectVisible(hdc, &rc))
                Polyline(hdc, aptTriangle, 4);

            SetViewportOrgEx(hdc, 100, 0, NULL);
            if (RectVisible(hdc, &rc))
                Polyline(hdc, aptRectangle, 5);

            SetViewportOrgEx(hdc, 0, 100, NULL);
            if (RectVisible(hdc, &rc))
                Polyline(hdc, aptPentagon, 6);

            SetViewportOrgEx(hdc, 100, 100, NULL);
            if (RectVisible(hdc, &rc))
                Polyline(hdc, aptHexagon, 7);
            EndPaint(hwnd, &ps);
            return 0L;

  .
  .
  .
```

The coordinates of each figure in this example lie within the same 100-by-100 rectangle. Before drawing a figure, the window procedure sets the viewport origin to a different part of the client area by using the SetViewportOrgEx function. This prevents figures from being drawn one on top of the other. Changing the viewport origin does not affect the clipping region, but does affect how the coordinates of the rectangle passed to RectVisible are interpreted. Changing the origin also allows you to use a single rectangle to check the update region rather than individual rectangles for each figure.

Invalidating the Client Area

4/20/2022 • 2 minutes to read • Edit Online

The system is not the only source of WM_PAINT messages. The InvalidateRect or InvalidateRgn function can indirectly generate **WM_PAINT** messages for your windows. These functions mark all or part of a client area as invalid (that must be redrawn).

In the following example, the window procedure invalidates the entire client area when processing WM_CHAR messages. This allows the user to change the figure by typing a number and view the results; these results are drawn as soon as there are no other messages in the application's message queue.

```
RECT rc;
POINT aptPentagon[6] = {50,2, 98,35, 79,90, 21,90, 2,35, 50,2},
      aptHexagon[7]  = {50,2, 93,25, 93,75, 50,98, 7,75, 7,25, 50,2};
POINT *ppt = aptPentagon;
int cpt = 6;

  .
  .
  .

case WM_CHAR:
    switch (wParam)
    {
        case '5':
            ppt = aptPentagon;
            cpt = 6;
            break;
        case '6':
            ppt = aptHexagon;
            cpt = 7;
            break;
    }
    InvalidateRect(hwnd, NULL, TRUE);
    return 0L;

case WM_PAINT:
    hdc = BeginPaint(hwnd, &ps);
    GetClientRect(hwnd, &rc);
    SetMapMode(hdc, MM_ANISOTROPIC);
    SetWindowExtEx(hdc, 100, 100, NULL);
    SetViewportExtEx(hdc, rc.right, rc.bottom, NULL);
    Polyline(hdc, ppt, cpt);
    EndPaint(hwnd, &ps);
    return 0L;
```

In this example, the **NULL** argument used by InvalidateRect specifies the entire client area; the **TRUE** argument causes the background to be erased. If you do not want the application to wait until the application's message queue has no other messages, use the UpdateWindow function to force the WM_PAINT message to be sent immediately. If there is any invalid part of the client area, **UpdateWindow** sends the **WM_PAINT** message for the specified window directly to the window procedure.

Drawing a Minimized Window

4/20/2022 • 2 minutes to read • Edit Online

You can draw your own minimized windows rather than having the system draw them for you. Most applications define a class icon when registering the window class for the window, and the system draws the icon when the window is minimized. If you set the class icon to **NULL**, however, the system sends a **WM_PAINT** message to your window procedure whenever the window is minimized, enabling the window procedure to draw in the minimized window.

In the following example, the window procedure draws a star in the minimized window. The procedure uses the **IsIconic** function to determine when the window is minimized. This ensures that the star is drawn only when the window is minimized.

```
POINT aptStar[6] = {50,2, 2,98, 98,33, 2,33, 98,98, 50,2};

  .
  .
  .

case WM_PAINT:
    hdc = BeginPaint(hwnd, &ps);

    // Determine whether the window is minimized.

    if (IsIconic(hwnd))
    {
        GetClientRect(hwnd, &rc);
        SetMapMode(hdc, MM_ANISOTROPIC);
        SetWindowExtEx(hdc, 100, 100, NULL);
        SetViewportExtEx(hdc, rc.right, rc.bottom, NULL);
        Polyline(hdc, aptStar, 6);
    }
    else
    {
        TextOut(hdc, 0,0, "Hello, Windows!", 15);
    }
    EndPaint(hwnd, &ps);
    return 0L;
```

You set the class icon to **NULL** by setting the **hIcon** member of the **WNDCLASS** structure to **NULL** before calling the **RegisterClass** function for the window class.

Drawing a Custom Window Background

4/20/2022 • 2 minutes to read • Edit Online

You can draw your own window background rather than having the system draw it for you. Most applications specify a brush handle or system color value for the class background brush when registering the window class; the system uses the brush or color to draw the background. If you set the class background brush to **NULL**, however, the system sends a WM_ERASEBKGND message to your window procedure whenever the window background must be drawn, letting you draw a custom background.

In the following example, the window procedure draws a large checkerboard pattern that fits neatly in the window. The procedure fills the client area with a white brush and then draws thirteen 20-by-20 rectangles using a gray brush. The display device context to use when drawing the background is specified in the *wParam* parameter for the message.

```
HBRUSH hbrWhite, hbrGray;

  .
  .
  .

case WM_CREATE:
    hbrWhite = GetStockObject(WHITE_BRUSH);
    hbrGray  = GetStockObject(GRAY_BRUSH);
    return 0L;

case WM_ERASEBKGND:
    hdc = (HDC) wParam;
    GetClientRect(hwnd, &rc);
    SetMapMode(hdc, MM_ANISOTROPIC);
    SetWindowExtEx(hdc, 100, 100, NULL);
    SetViewportExtEx(hdc, rc.right, rc.bottom, NULL);
    FillRect(hdc, &rc, hbrWhite);

    for (i = 0; i < 13; i++)
    {
        x = (i * 40) % 100;
        y = ((i * 40) / 100) * 20;
        SetRect(&rc, x, y, x + 20, y + 20);
        FillRect(hdc, &rc, hbrGray);
    }
  return 1L;
```

If the application draws its own minimized window, the system also sends the WM_ERASEBKGND message to the window procedure to draw the background for the minimized window. You can use the same technique used by WM_PAINT to determine whether the window is minimized that is, call the IsIconic function and check for the return value **TRUE**.

Using the GetDC Function

4/20/2022 • 2 minutes to read • Edit Online

You use the **GetDC** function to carry out drawing that must occur instantly rather than when a **WM_PAINT** message is processing. Such drawing is usually in response to an action by the user, such as making a selection or drawing with the mouse. In such cases, the user should receive instant feedback and must not be forced to stop selecting or drawing in order for the application to display the result. The following sections show how to use GetDC to draw in a window:

- Drawing with the Mouse
- Drawing at Timed Intervals

Drawing with the Mouse

4/20/2022 • 2 minutes to read • Edit Online

You can permit the user to draw lines with the mouse by having your window procedure draw while processing the WM_MOUSEMOVE message. The system sends the **WM_MOUSEMOVE** message to the window procedure whenever the user moves the cursor within the window. To draw lines, the window procedure can retrieve a display device context and draw a line in the window between the current and previous cursor positions.

In the following example, the window procedure prepares for drawing when the user presses and holds the left mouse button (sending the WM_LBUTTONDOWN message). As the user moves the cursor within the window, the window procedure receives a series of WM_MOUSEMOVE messages. For each message, the window procedure draws a line connecting the previous position and the current position. To draw the line, the procedure uses GetDC to retrieve a display device context; then, as soon as drawing is complete and before returning from the message, the procedure uses the ReleaseDC function to release the display device context. As soon as the user releases the mouse button, the window procedure clears the flag, and the drawing stops (which sends the WM_LBUTTONUP message).

```
BOOL fDraw = FALSE;
POINT ptPrevious;

  .
  .
  .

case WM_LBUTTONDOWN:
    fDraw = TRUE;
    ptPrevious.x = LOWORD(lParam);
    ptPrevious.y = HIWORD(lParam);
    return 0L;

case WM_LBUTTONUP:
    if (fDraw)
    {
        hdc = GetDC(hwnd);
        MoveToEx(hdc, ptPrevious.x, ptPrevious.y, NULL);
        LineTo(hdc, LOWORD(lParam), HIWORD(lParam));
        ReleaseDC(hwnd, hdc);
    }
    fDraw = FALSE;
    return 0L;

case WM_MOUSEMOVE:
    if (fDraw)
    {
        hdc = GetDC(hwnd);
        MoveToEx(hdc, ptPrevious.x, ptPrevious.y, NULL);
        LineTo(hdc, ptPrevious.x = LOWORD(lParam),
          ptPrevious.y = HIWORD(lParam));
        ReleaseDC(hwnd, hdc);
    }
    return 0L;
```

An application that enables drawing, as in this example, typically records either the points or lines so that the lines can be redrawn whenever the window is updated. Drawing applications often use a memory device context and an associated bitmap to store lines that were drawn by using a mouse.

Drawing at Timed Intervals

4/20/2022 • 3 minutes to read • Edit Online

You can draw at timed intervals by creating a timer with the **SetTimer** function. By using a timer to send **WM_TIMER** messages to the window procedure at regular intervals, an application can carry out simple animation in the client area while other applications continue running.

In the following example, the application bounces a star from side to side in the client area. Each time the window procedure receives a **WM_TIMER** message, the procedure erases the star at the current position, calculates a new position, and draws the star within the new position. The procedure starts the timer by calling **SetTimer** while processing the **WM_CREATE** message.

```
RECT rcCurrent = {0,0,20,20};
POINT aptStar[6] = {10,1, 1,19, 19,6, 1,6, 19,19, 10,1};
int X = 2, Y = -1, idTimer = -1;
BOOL fVisible = FALSE;
HDC hdc;

LRESULT APIENTRY WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    RECT rc;

    switch (message)
    {
        case WM_CREATE:

            // Calculate the starting point.

            GetClientRect(hwnd, &rc);
            OffsetRect(&rcCurrent, rc.right / 2, rc.bottom / 2);

            // Initialize the private DC.

            hdc = GetDC(hwnd);
            SetViewportOrgEx(hdc, rcCurrent.left,
                rcCurrent.top, NULL);
            SetROP2(hdc, R2_NOT);

            // Start the timer.

            SetTimer(hwnd, idTimer = 1, 10, NULL);
            return 0L;

        case WM_DESTROY:
            KillTimer(hwnd, 1);
            PostQuitMessage(0);
            return 0L;

        case WM_SIZE:
            switch (wParam)
            {
                case SIZE_MINIMIZED:

                    // Stop the timer if the window is minimized.

                    KillTimer(hwnd, 1);
                    idTimer = -1;
                    break;

                case SIZE_RESTORED:
```

```
            case SIZE_RESTORED:

                // Move the star back into the client area
                // if necessary.

                if (rcCurrent.right > (int) LOWORD(lParam))
                {
                    rcCurrent.left =
                        (rcCurrent.right =
                            (int) LOWORD(lParam)) - 20;
                }
                if (rcCurrent.bottom > (int) HIWORD(lParam))
                {
                    rcCurrent.top =
                        (rcCurrent.bottom =
                            (int) HIWORD(lParam)) - 20;
                }

                // Fall through to the next case.

            case SIZE_MAXIMIZED:

                // Start the timer if it had been stopped.

                if (idTimer == -1)
                    SetTimer(hwnd, idTimer = 1, 10, NULL);
                break;
        }
        return 0L;

    case WM_TIMER:

        // Hide the star if it is visible.

        if (fVisible)
            Polyline(hdc, aptStar, 6);

        // Bounce the star off a side if necessary.

        GetClientRect(hwnd, &rc);
        if (rcCurrent.left + X < rc.left ||
            rcCurrent.right + X > rc.right)
            X = -X;
        if (rcCurrent.top + Y < rc.top ||
            rcCurrent.bottom + Y > rc.bottom)
            Y = -Y;

        // Show the star in its new position.

        OffsetRect(&rcCurrent, X, Y);
        SetViewportOrgEx(hdc, rcCurrent.left,
            rcCurrent.top, NULL);
        fVisible = Polyline(hdc, aptStar, 6);

        return 0L;

    case WM_ERASEBKGND:

        // Erase the star.

        fVisible = FALSE;
        return DefWindowProc(hwnd, message, wParam, lParam);

    case WM_PAINT:

        // Show the star if it is not visible. Use BeginPaint
        // to clear the update region.

        BeginPaint(hwnd, &ps);
        if (!fVisible)
```

```
            if (!fVisible)
                fVisible = Polyline(hdc, aptStar, 6);
            EndPaint(hwnd, &ps);
            return 0L;
    }
    return DefWindowProc(hwnd, message, wParam, lParam);
}
```

This application uses a private device context to minimize the time required to prepare the device context for drawing. The window procedure retrieves and initializes the private device context when processing the **WM_CREATE** message, setting the binary raster operation mode to allow the star to be erased and drawn using the same call to the **Polyline** function. The window procedure also sets the viewport origin to allow the star to be drawn using the same set of points regardless of the star's position in the client area.

The application uses the **WM_PAINT** message to draw the star whenever the window must be updated. The window procedure draws the star only if it is not visible; that is, only if it has been erased by the **WM_ERASEBKGND** message. The window procedure intercepts the **WM_ERASEBKGND** message to set the *fVisible* variable, but passes the message to **DefWindowProc** so that the system can draw the window background.

The application uses the **WM_SIZE** message to stop the timer when the window is minimized and to restart the timer when the minimized window is restored. The window procedure also uses the message to update the current position of the star if the size of the window has been reduced so that the star is no longer in the client area. The application keeps track of the star's current position by using the structure specified by rcCurrent, which defines the bounding rectangle for the star. Keeping all corners of the rectangle in the client area keeps the star in the area. The window procedure initially centers the star in the client area when processing the **WM_CREATE** message.

Painting and Drawing Reference

4/20/2022 • 2 minutes to read • Edit Online

The following elements are associated with painting and drawing:

- Painting and Drawing Functions
- Painting and Drawing Structures
- Painting and Drawing Messages
- Raster Operation Codes

Painting and Drawing Functions

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used with painting and drawing.

FUNCTION	DESCRIPTION
BeginPaint	Prepares a window for painting.
DrawAnimatedRects	Draws a rectangle and animates it to indicate icon or window activity.
DrawCaption	Draws a window caption.
DrawEdge	Draws one or more edges of rectangle.
DrawFocusRect	Draws a rectangle in the style that indicates the rectangle has the focus.
DrawFrameControl	Draws a frame control.
DrawState	Displays an image and applies a visual effect to indicate a state.
DrawStateProc	A callback function that renders a complex image for DrawState.
EndPaint	Marks the end of painting in a window.
ExcludeUpdateRgn	Prevents drawing within invalid areas of a window.
GdiFlush	Flushes the calling thread's current batch.
GdiGetBatchLimit	Returns the maximum number of function calls that can be accumulated in the calling thread's current batch.
GdiSetBatchLimit	Sets the maximum number of function calls that can be accumulated in the calling thread's current batch.
GetBkColor	Returns the background color for a device context.
GetBkMode	Returns the background mix mode for a device context.
GetBoundsRect	Gets the accumulated bounding rectangle for a device context.
GetROP2	Gets the foreground mix mode of a device context.
GetUpdateRect	Gets the coordinates of the smallest rectangle that encloses the update region of a window.

FUNCTION	DESCRIPTION
GetUpdateRgn	Gets the update region of a window.
GetWindowDC	Gets the device context for a window, including title bar, menus, and scroll bars.
GetWindowRgn	Gets a copy of the window region of a window.
GetWindowRgnBox	Gets the dimensions of the tightest bounding rectangle for the window region of a window.
GrayString	Draws gray text at a location.
InvalidateRect	Adds a rectangle to a window's update region.
InvalidateRgn	Invalidates the client area within a region.
LockWindowUpdate	Disables or enables drawing in a window.
OutputProc	A callback function used with the GrayString function. It is used to draw a string.
PaintDesktop	Fills the clipping region in a device context with a pattern.
RedrawWindow	Updates a region in a window's client area.
SetBkColor	Sets the background to a color value.
SetBkMode	Sets the background mix mode of a device context.
SetBoundsRect	Controls the accumulation of bounding rectangle information for a device context.
SetROP2	Sets the foreground mix mode.
SetWindowRgn	Sets the window region of a window.
UpdateWindow	Updates the client area of a window.
ValidateRect	Validates the client area within a rectangle.
ValidateRgn	Validates the client area within a region.
WindowFromDC	Returns a handle to the window associated with a device context.

Painting and Drawing Structures

4/20/2022 • 2 minutes to read • Edit Online

The following structure is used with painting and drawing.

PAINTSTRUCT

Painting and Drawing Messages

4/20/2022 • 2 minutes to read • Edit Online

The following messages are used with painting and drawing:

- WM_DISPLAYCHANGE
- WM_ERASEBKGND
- WM_NCPAINT
- WM_PAINT
- WM_PRINT
- WM_PRINTCLIENT
- WM_SETREDRAW
- WM_SYNCPAINT

WM_DISPLAYCHANGE message

4/20/2022 • 2 minutes to read • Edit Online

The **WM_DISPLAYCHANGE** message is sent to all windows when the display resolution has changed.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,
  UINT  uMsg,
  WPARAM wParam,
  LPARAM lParam
);
```

Parameters

wParam

The new image depth of the display, in bits per pixel.

lParam

The low-order word specifies the horizontal resolution of the screen.

The high-order word specifies the vertical resolution of the screen.

Remarks

This message is only sent to top-level windows. For all other windows it is posted.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

See also

Painting and Drawing Overview

Painting and Drawing Messages

HIWORD

LOWORD

WM_NCPAINT message

4/20/2022 • 2 minutes to read • Edit Online

The **WM_NCPAINT** message is sent to a window when its frame must be painted.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,
  UINT  uMsg,
  WPARAM wParam,
  LPARAM lParam
);
```

Parameters

wParam

A handle to the update region of the window. The update region is clipped to the window frame.

lParam

This parameter is not used.

Return value

An application returns zero if it processes this message.

Remarks

The **DefWindowProc** function paints the window frame.

An application can intercept the **WM_NCPAINT** message and paint its own custom window frame. The clipping region for a window is always rectangular, even if the shape of the frame is altered.

The *wParam* value can be passed to **GetDCEx** as in the following example.

```
case WM_NCPAINT:
{
    HDC hdc;
    hdc = GetDCEx(hwnd, (HRGN)wParam, DCX_WINDOW|DCX_INTERSECTRGN);
    // Paint into this DC
    ReleaseDC(hwnd, hdc);
}
```

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]

REQUIREMENT	VALUE
Header	Winuser.h (include Windows.h)

See also

Painting and Drawing Overview

Painting and Drawing Messages

DefWindowProc

GetWindowDC

WM_PAINT

GetDCEx

WM_PAINT message

4/20/2022 • 2 minutes to read • Edit Online

The **WM_PAINT** message is sent when the system or another application makes a request to paint a portion of an application's window. The message is sent when the UpdateWindow or RedrawWindow function is called, or by the DispatchMessage function when the application obtains a **WM_PAINT** message by using the GetMessage or PeekMessage function.

A window receives this message through its WindowProc function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,
  UINT  uMsg,
  WPARAM wParam,
  LPARAM lParam
);
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

An application returns zero if it processes this message.

Example

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;

    case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hwnd, &ps);

            // All painting occurs here, between BeginPaint and EndPaint.
            FillRect(hdc, &ps.rcPaint, (HBRUSH) (COLOR_WINDOW+1));
            EndPaint(hwnd, &ps);
        }
        return 0;
    }

    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

Example from Windows Classic Samples on GitHub.

Remarks

The **WM_PAINT** message is generated by the system and should not be sent by an application. To force a window to draw into a specific device context, use the WM_PRINT or WM_PRINTCLIENT message. Note that this requires the target window to support the **WM_PRINTCLIENT** message. Most common controls support the **WM_PRINTCLIENT** message.

The DefWindowProc function validates the update region. The function may also send the WM_NCPAINT message to the window procedure if the window frame must be painted and send the WM_ERASEBKGND message if the window background must be erased.

The system sends this message when there are no other messages in the application's message queue. DispatchMessage determines where to send the message; GetMessage determines which message to dispatch. **GetMessage** returns the **WM_PAINT** message when there are no other messages in the application's message queue, and **DispatchMessage** sends the message to the appropriate window procedure.

A window may receive internal paint messages as a result of calling RedrawWindow with the RDW_INTERNALPAINT flag set. In this case, the window may not have an update region. An application may call the GetUpdateRect function to determine whether the window has an update region. If **GetUpdateRect** returns zero, the application need not call the BeginPaint and EndPaint functions.

An application must check for any necessary internal painting by looking at its internal data structures for each **WM_PAINT** message, because a **WM_PAINT** message may have been caused by both a non-NULL update region and a call to RedrawWindow with the RDW_INTERNALPAINT flag set.

The system sends an internal **WM_PAINT** message only once. After an internal **WM_PAINT** message is returned from GetMessage or PeekMessage or is sent to a window by UpdateWindow, the system does not post or send further **WM_PAINT** messages until the window is invalidated or until RedrawWindow is called again with the RDW_INTERNALPAINT flag set.

For some common controls, the default **WM_PAINT** message processing checks the *wParam* parameter. If *wParam* is non-NULL, the control assumes that the value is an HDC and paints using that device context.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

See also

Painting and Drawing Overview

Painting and Drawing Messages

BeginPaint

DefWindowProc

DispatchMessage

EndPaint

GetMessage

GetUpdateRect

PeekMessage

RedrawWindow

UpdateWindow

WM_ERASEBKGND

WM_NCPAINT

WM_PRINT

WM_PRINTCLIENT

WM_PRINT message

4/20/2022 • 2 minutes to read • Edit Online

The **WM_PRINT** message is sent to a window to request that it draw itself in the specified device context, most commonly in a printer device context.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,
  UINT  uMsg,
  WPARAM wParam,
  LPARAM lParam
);
```

Parameters

wParam

A handle to the device context to draw in.

lParam

The drawing options. This parameter can be one or more of the following values.

VALUE	MEANING
PRF_CHECKVISIBLE	Draws the window only if it is visible.
PRF_CHILDREN	Draws all visible children windows.
PRF_CLIENT	Draws the client area of the window.
PRF_ERASEBKGND	Erases the background before drawing the window.
PRF_NONCLIENT	Draws the nonclient area of the window.
PRF_OWNED	Draws all owned windows.

Remarks

The **DefWindowProc** function processes this message based on which drawing option is specified: if

PRF_CHECKVISIBLE is specified and the window is not visible, do nothing, if PRF_NONCLIENT is specified, draw the nonclient area in the specified device context, if PRF_ERASEBKGND is specified, send the window a **WM_ERASEBKGND** message, if PRF_CLIENT is specified, send the window a **WM_PRINTCLIENT** message, if PRF_CHILDREN is set, send each visible child window a **WM_PRINT** message, if PRF_OWNED is set, send each visible owned window a **WM_PRINT** message.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

See also

Painting and Drawing Overview

Painting and Drawing Messages

DefWindowProc

WM_ERASEBKGND

WM_PRINTCLIENT

WM_PRINTCLIENT message

4/20/2022 • 2 minutes to read • Edit Online

The **WM_PRINTCLIENT** message is sent to a window to request that it draw its client area in the specified device context, most commonly in a printer device context.

Unlike **WM_PRINT**, **WM_PRINTCLIENT** is not processed by **DefWindowProc**. A window should process the **WM_PRINTCLIENT** message through an application-defined **WindowProc** function for it to be used properly.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,
  UINT  uMsg,
  WPARAM wParam,
  LPARAM lParam
);
```

Parameters

wParam

A handle to the device context to draw in.

lParam

The drawing options. This parameter can be one or more of the following values.

VALUE	MEANING
PRF_CHECKVISIBLE	Draws the window only if it is visible.
PRF_CHILDREN	Draws all visible children windows.
PRF_CLIENT	Draws the client area of the window.
PRF_ERASEBKGND	Erases the background before drawing the window.
PRF_NONCLIENT	Draws the nonclient area of the window.
PRF_OWNED	Draws all owned windows.

Remarks

A window can process this message in much the same manner as **WM_PAINT**, except that **BeginPaint** and **EndPaint** need not be called (a device context is provided), and the window should draw its entire client area rather than just the invalid region.

Windows that can be used anywhere in the system, such as controls, should process this message. It is probably worthwhile for other windows to process this message as well because it is relatively easy to implement.

The AnimateWindow function requires that the window being animated implements the **WM_PRINTCLIENT** message.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

See also

Painting and Drawing Overview

Painting and Drawing Messages

AnimateWindow

BeginPaint

EndPaint

WM_PAINT

WM_SETREDRAW message

4/20/2022 • 2 minutes to read • Edit Online

You send the **WM_SETREDRAW** message to a window to allow changes in that window to be redrawn, or to prevent changes in that window from being redrawn.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
  (HWND) hWnd,
  WM_SETREDRAW,
  (WPARAM) wParam,
  (LPARAM) lParam
);
```

Parameters

`wParam`

The redraw state. If this parameter is **TRUE**, then the content can be redrawn after a change. If this parameter is **FALSE**, then the content can't be redrawn after a change.

`lParam`

This parameter isn't used.

Return value

Your application should return 0 if it processes this message.

Remarks

This message can be useful if your application must add several items to a list box. Your application can call this message with *wParam* set to **FALSE**, add the items, and then call the message again with *wParam* set to **TRUE**. Finally, your application can call **RedrawWindow**(*hWnd*, **NULL**, **NULL**, RDW_ERASE | RDW_FRAME | RDW_INVALIDATE | RDW_ALLCHILDREN) to cause the list box to be repainted.

NOTE

You should use **RedrawWindow** with the specified flags, instead of **InvalidateRect**, because the former is necessary for some controls that have nonclient area of their own, or have window styles that cause them to be given a nonclient area (such as **WS_THICKFRAME**, **WS_BORDER**, or **WS_EX_CLIENTEDGE**). If the control does not have a nonclient area, then **RedrawWindow** with these flags will do only as much invalidation as **InvalidateRect** would.

Passing a **WM_SETREDRAW** message to the **DefWindowProc** function removes the **WS_VISIBLE** style from the window when *wParam* is set to **FALSE**. Although the window content remains visible on screen, the **IsWindowVisible** function returns **FALSE** when called on a window in this state.

Passing a **WM_SETREDRAW** message to the **DefWindowProc** function adds the **WS_VISIBLE** style to the window, if not set, when *wParam* is set to **TRUE**. If your application sends the **WM_SETREDRAW** message with *wParam* set to **TRUE** to a hidden window, then the window becomes visible.

Windows 10 and later; Windows Server 2016 and later. The system sets a property named *SysSetRedraw* on a window whose window procedure passes **WM_SETREDRAW** messages to **DefWindowProc**. You can use the **GetProp** function to get the property value when it's available. **GetProp** returns a non-zero value when redraw is disabled. **GetProp** will return zero when redraw is enabled, or when the window property doesn't exist.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

See also

- Painting and drawing overview
- Painting and drawing messages
- RedrawWindow

WM_SYNCPAINT message

4/20/2022 • 2 minutes to read • Edit Online

The **WM_SYNCPAINT** message is used to synchronize painting while avoiding linking independent GUI threads.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,
  UINT  uMsg,
  WPARAM wParam,
  LPARAM lParam
);
```

Parameters

wParam

This parameter is not used.

lParam

This parameter is not used.

Return value

An application returns zero if it processes this message.

Remarks

When a window has been hidden, shown, moved, or sized, the system may determine that it is necessary to send a **WM_SYNCPAINT** message to the top-level windows of other threads. Applications must pass **WM_SYNCPAINT** to **DefWindowProc** for processing. The **DefWindowProc** function will send a **WM_NCPAINT** message to the window procedure if the window frame must be painted and send a **WM_ERASEBKGND** message if the window background must be erased.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Header	Winuser.h (include Windows.h)

See also

Painting and Drawing Overview

Painting and Drawing Messages

DefWindowProc

GetDCEx

GetWindowDC

WM_PAINT

Raster Operation Codes

4/20/2022 • 2 minutes to read • Edit Online

Raster-operation codes define how the graphics device interface (GDI) combines the bits from the selected pen with the bits in the destination bitmap.

This overview lists and describes the binary and ternary raster operations used by GDI. A binary raster operation involves two operands: a pen and a destination bitmap. A ternary raster operation involves three operands: a source bitmap, a brush, and a destination bitmap. Both binary and ternary raster operations use Boolean operators.

Binary Raster Operations

4/20/2022 • 2 minutes to read • Edit Online

This section lists the binary raster-operation codes used by the GetROP2 and SetROP2 functions. Raster-operation codes define how GDI combines the bits from the selected pen with the bits in the destination bitmap.

Each raster-operation code represents a Boolean operation in which the values of the pixels in the selected pen and the destination bitmap are combined. The following are the two operands used in these operations.

OPERAND	MEANING
P	Selected pen
D	Destination bitmap

The Boolean operators used in these operations follow.

OPERATOR	MEANING
a	Bitwise AND
n	Bitwise NOT (inverse)
o	Bitwise OR
x	Bitwise exclusive OR (XOR)

All Boolean operations are presented in reverse Polish notation. For example, the following operation replaces the values of the pixels in the destination bitmap with a combination of the pixel values of the pen and the selected brush:

```
DPo
```

Each raster-operation code is a 32-bit integer whose high-order word is a Boolean operation index and whose low-order word is the operation code. The 16-bit operation index is a zero-extended 8-bit value that represents all possible outcomes resulting from the Boolean operation on two parameters (in this case, the pen and destination values). For example, the operation indexes for the DPo and DPan operations are shown in the following list.

P	D	DPO	DPAN
0	0	0	1
0	1	1	1
1	0	1	1

P	D	DPO	DPAN
1	1	1	0

The following list outlines the drawing modes and the Boolean operations that they represent.

RASTER OPERATION	BOOLEAN OPERATION
R2_BLACK	0
R2_COPYPEN	P
R2_MASKNOTPEN	DPna
R2_MASKPEN	DPa
R2_MASKPENNOT	PDna
R2_MERGENOTPEN	DPno
R2_MERGEPEN	DPo
R2_MERGEPENNOT	PDno
R2_NOP	D
R2_NOT	Dn
R2_NOTCOPYPEN	Pn
R2_NOTMASKPEN	DPan
R2_NOTMERGEPEN	DPon
R2_NOTXORPEN	DPxn
R2_WHITE	1
R2_XORPEN	DPx

For a monochrome device, GDI maps the value zero to black and the value 1 to white. If an application attempts to draw with a black pen on a white destination by using the available binary raster operations, the following results occur.

RASTER OPERATION	RESULT
R2_BLACK	Visible black line
R2_COPYPEN	Visible black line

RASTER OPERATION	RESULT
R2_MASKNOTPEN	No visible line
R2_MASKPEN	Visible black line
R2_MASKPENNOT	Visible black line
R2_MERGENOTPEN	No visible line
R2_MERGEPEN	Visible black line
R2_MERGEPENNOT	Visible black line
R2_NOP	No visible line
R2_NOT	Visible black line
R2_NOTCOPYPEN	No visible line
R2_NOTMASKPEN	No visible line
R2_NOTMERGEPEN	Visible black line
R2_NOTXORPEN	Visible black line
R2_WHITE	No visible line
R2_XORPEN	No visible line

For a color device, GDI uses RGB values to represent the colors of the pen and the destination. An RGB color value is a long integer that contains a red, a green, and a blue color field, each specifying the intensity of the specified color. Intensities range from 0 through 255. The values are packed in the three low-order bytes of the long integer. The color of a pen is always a solid color, but the color of the destination may be a mixture of any two or three colors. If an application attempts to draw with a white pen on a blue destination by using the available binary raster operations, the following results occur.

RASTER OPERATION	RESULT
R2_BLACK	Visible black line
R2_COPYPEN	Visible white line
R2_MASKNOTPEN	Visible black line
R2_MASKPEN	Invisible blue line
R2_MASKPENNOT	Visible red/green line
R2_MERGENOTPEN	Invisible blue line

RASTER OPERATION	RESULT
R2_MERGEPEN	Visible white line
R2_MERGEPENNOT	Visible white line
R2_NOP	Invisible blue line
R2_NOT	Visible red/green line
R2_NOTCOPYPEN	Visible black line
R2_NOTMASKPEN	Visible red/green line
R2_NOTMERGEPEN	Visible black line
R2_NOTXORPEN	Invisible blue line
R2_WHITE	Visible white line
R2_XORPEN	Visible red/green line

Ternary Raster Operations

4/20/2022 • 6 minutes to read • Edit Online

This section lists the ternary raster-operation codes used by the **BitBlt**, **PatBlt**, and **StretchBlt** functions. Ternary raster-operation codes define how GDI combines the bits in a source bitmap with the bits in the destination bitmap.

Each raster-operation code represents a Boolean operation in which the values of the pixels in the source, the selected brush, and the destination are combined. The following are the three operands used in these operations.

OPERAND	MEANING
D	Destination bitmap
P	Selected brush (also called pattern)
S	Source bitmap

Boolean operators used in these operations follow.

OPERATOR	MEANING
a	Bitwise AND
n	Bitwise NOT (inverse)
o	Bitwise OR
x	Bitwise exclusive OR (XOR)

All Boolean operations are presented in reverse Polish notation. For example, the following operation replaces the values of the pixels in the destination bitmap with a combination of the pixel values of the source and brush:

```
PSo
```

The following operation combines the values of the pixels in the source and brush with the pixel values of the destination bitmap (there are alternative spellings of the same function, so although a particular spelling may not be in the list, an equivalent form would be):

```
DPSoo
```

Each raster-operation code is a 32-bit integer whose high-order word is a Boolean operation index and whose low-order word is the operation code. The 16-bit operation index is a zero-extended, 8-bit value that represents the result of the Boolean operation on predefined brush, source, and destination values. For example, the operation indexes for the PSo and DPSoo operations are shown in the following list.

P	S	D	PSO	DPSOO
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1
Operation index:			00FCh	00FEh

In this case, PSo has the operation index 00FC (read from the bottom up); DPSoo has the operation index 00FE. These values define the location of the corresponding raster-operation codes, as shown in Table A.1, "Raster-Operation Codes." The PSo operation is in line 252 (00FCh) of the table; DPSoo is in line 254 (00FEh).

The most commonly used raster operations have been given special names in the SDK header file, WINDOWS.H. You should use these names whenever possible in your applications.

When the source and destination bitmaps are monochrome, a bit value of zero represents a black pixel and a bit value of 1 represents a white pixel. When the source and the destination bitmaps are color, those colors are represented with RGB values. For more information about RGB values, see RGB.

Raster-Operation Codes

BOOLEAN FUNCTION (HEXADECIMAL)	RASTER OPERATION (HEXADECIMAL)	BOOLEAN FUNCTION IN REVERSE POLISH	COMMON NAME
00	00000042	0	BLACKNESS
01	00010289	DPSoon	
02	00020C89	DPSona	
03	000300AA	PSon	
04	00040C88	SDPona	
05	000500A9	DPon	
06	00060865	PDSxnon	
07	000702C5	PDSaon	

BOOLEAN FUNCTION (HEXADECIMAL)	RASTER OPERATION (HEXADECIMAL)	BOOLEAN FUNCTION IN REVERSE POLISH	COMMON NAME
08	00080F08	SDPnaa	
09	00090245	PDSxon	
0A	000A0329	DPna	
0B	000B0B2A	PSDnaon	
0C	000C0324	SPna	
0D	000D0B25	PDSnaon	
0E	000E08A5	PDSonon	
0F	000F0001	Pn	
10	00100C85	PDSona	
11	001100A6	DSon	NOTSRCERASE
12	00120868	SDPxnon	
13	001302C8	SDPaon	
14	00140869	DPSxnon	
15	001502C9	DPSaon	
16	00165CCA	PSDPSanaxx	
17	00171D54	SSPxDSxaxn	
18	00180D59	SPxPDxa	
19	00191CC8	SDPSanaxn	
1A	001A06C5	PDSPaox	
1B	001B0768	SDPSxaxn	
1C	001C06CA	PSDPaox	
1D	001D0766	DSPDxaxn	
1E	001E01A5	PDSox	
1F	001F0385	PDSoan	
20	00200F09	DPSnaa	

BOOLEAN FUNCTION (HEXADECIMAL)	RASTER OPERATION (HEXADECIMAL)	BOOLEAN FUNCTION IN REVERSE POLISH	COMMON NAME
21	00210248	SDPxon	
22	00220326	DSna	
23	00230B24	SPDnaon	
24	00240D55	SPxDSxa	
25	00251CC5	PDSPanaxn	
26	002606C8	SDPSaox	
27	00271868	SDPSxnox	
28	00280369	DPSxa	
29	002916CA	PSDPSaoxxn	
2A	002A0CC9	DPSana	
2B	002B1D58	SSPxPDxaxn	
2C	002C0784	SPDSoax	
2D	002D060A	PSDnox	
2E	002E064A	PSDPxox	
2F	002F0E2A	PSDnoan	
30	0030032A	PSna	
31	00310B28	SDPnaon	
32	00320688	SDPSoox	
33	00330008	Sn	NOTSRCCOPY
34	003406C4	SPDSaox	
35	00351864	SPDSxnox	
36	003601A8	SDPox	
37	00370388	SDPoan	
38	0038078A	PSDPoax	
39	00390604	SPDnox	

BOOLEAN FUNCTION (HEXADECIMAL)	RASTER OPERATION (HEXADECIMAL)	BOOLEAN FUNCTION IN REVERSE POLISH	COMMON NAME
3A	003A0644	SPDSxox	
3B	003B0E24	SPDnoan	
3C	003C004A	PSx	
3D	003D18A4	SPDSonox	
3E	003E1B24	SPDSnaox	
3F	003F00EA	PSan	
40	00400F0A	PSDnaa	
41	00410249	DPSxon	
42	00420D5D	SDxPDxa	
43	00431CC4	SPDSanaxn	
44	00440328	SDna	SRCERASE
45	00450B29	DPSnaon	
46	004606C6	DSPDaox	
47	0047076A	PSDPxaxn	
48	00480368	SDPxa	
49	004916C5	PDSPDaoxxn	
4A	004A0789	DPSDoax	
4B	004B0605	PDSnox	
4C	004C0CC8	SDPana	
4D	004D1954	SSPxDSxoxn	
4E	004E0645	PDSPxox	
4F	004F0E25	PDSnoan	
50	00500325	PDna	
51	00510B26	DSPnaon	
52	005206C9	DPSDaox	

BOOLEAN FUNCTION (HEXADECIMAL)	RASTER OPERATION (HEXADECIMAL)	BOOLEAN FUNCTION IN REVERSE POLISH	COMMON NAME
53	00530764	SPDSxaxn	
54	005408A9	DPSonon	
55	00550009	Dn	DSTINVERT
56	005601A9	DPSox	
57	00570389	DPSoan	
58	00580785	PDSPoax	
59	00590609	DPSnox	
5A	005A0049	DPx	PATINVERT
5B	005B18A9	DPSDonox	
5C	005C0649	DPSDxox	
5D	005D0E29	DPSnoan	
5E	005E1B29	DPSDnaox	
5F	005F00E9	DPan	
60	00600365	PDSxa	
61	006116C6	DSPDSaoxxn	
62	00620786	DSPDoax	
63	00630608	SDPnox	
64	00640788	SDPSoax	
65	00650606	DSPnox	
66	00660046	DSx	SRCINVERT
67	006718A8	SDPSonox	
68	006858A6	DSPDSonoxxn	
69	00690145	PDSxxn	
6A	006A01E9	DPSax	
6B	006B178A	PSDPSoaxxn	

BOOLEAN FUNCTION (HEXADECIMAL)	RASTER OPERATION (HEXADECIMAL)	BOOLEAN FUNCTION IN REVERSE POLISH	COMMON NAME
6C	006C01E8	SDPax	
6D	006D1785	PDSPDoaxxn	
6E	006E1E28	SDPSnoax	
6F	006F0C65	PDSxnan	
70	00700CC5	PDSana	
71	00711D5C	SSDxPDxaxn	
72	00720648	SDPSxox	
73	00730E28	SDPnoan	
74	00740646	DSPDxox	
75	00750E26	DSPnoan	
76	00761B28	SDPSnaox	
77	007700E6	DSan	
78	007801E5	PDSax	
79	00791786	DSPDSoaxxn	
7A	007A1E29	DPSDnoax	
7B	007B0C68	SDPxnan	
7C	007C1E24	SPDSnoax	
7D	007D0C69	DPSxnan	
7E	007E0955	SPxDSxo	
7F	007F03C9	DPSaan	
80	008003E9	DPSaa	
81	00810975	SPxDSxon	
82	00820C49	DPSxna	
83	00831E04	SPDSnoaxn	
84	00840C48	SDPxna	

BOOLEAN FUNCTION (HEXADECIMAL)	RASTER OPERATION (HEXADECIMAL)	BOOLEAN FUNCTION IN REVERSE POLISH	COMMON NAME
85	00851E05	PDSPnoaxn	
86	008617A6	DSPDSoaxx	
87	008701C5	PDSaxn	
88	008800C6	DSa	SRCAND
89	00891B08	SDPSnaoxn	
8A	008A0E06	DSPnoa	
8B	008B0666	DSPDxoxn	
8C	008C0E08	SDPnoa	
8D	008D0668	SDPSxoxn	
8E	008E1D7C	SSDxPDxax	
8F	008F0CE5	PDSanan	
90	00900C45	PDSxna	
91	00911E08	SDPSnoaxn	
92	009217A9	DPSDPoaxx	
93	009301C4	SPDaxn	
94	009417AA	PSDPSoaxx	
95	009501C9	DPSaxn	
96	00960169	DPSxx	
97	0097588A	PSDPSonoxx	
98	00981888	SDPSonoxn	
99	00990066	DSxn	
9A	009A0709	DPSnax	
9B	009B07A8	SDPSoaxn	
9C	009C0704	SPDnax	
9D	009D07A6	DSPDoaxn	

BOOLEAN FUNCTION (HEXADECIMAL)	RASTER OPERATION (HEXADECIMAL)	BOOLEAN FUNCTION IN REVERSE POLISH	COMMON NAME
9E	009E16E6	DSPDSaoxx	
9F	009F0345	PDSxan	
A0	00A000C9	DPa	
A1	00A11B05	PDSPnaoxn	
A2	00A20E09	DPSnoa	
A3	00A30669	DPSDxoxn	
A4	00A41885	PDSPonoxn	
A5	00A50065	PDxn	
A6	00A60706	DSPnax	
A7	00A707A5	PDSPoaxn	
A8	00A803A9	DPSoa	
A9	00A90189	DPSoxn	
AA	00AA0029	D	
AB	00AB0889	DPSono	
AC	00AC0744	SPDSxax	
AD	00AD06E9	DPSDaoxn	
AE	00AE0B06	DSPnao	
AF	00AF0229	DPno	
B0	00B00E05	PDSnoa	
B1	00B10665	PDSPxoxn	
B2	00B21974	SSPxDSxox	
B3	00B30CE8	SDPanan	
B4	00B4070A	PSDnax	
B5	00B507A9	DPSDoaxn	
B6	00B616E9	DPSDPaoxx	

BOOLEAN FUNCTION (HEXADECIMAL)	RASTER OPERATION (HEXADECIMAL)	BOOLEAN FUNCTION IN REVERSE POLISH	COMMON NAME
B7	00B70348	SDPxan	
B8	00B8074A	PSDPxax	
B9	00B906E6	DSPDaoxn	
BA	00BA0B09	DPSnao	
BB	00BB0226	DSno	MERGEPAINT
BC	00BC1CE4	SPDSanax	
BD	00BD0D7D	SDxPDxan	
BE	00BE0269	DPSxo	
BF	00BF08C9	DPSano	
C0	00C000CA	PSa	MERGECOPY
C1	00C11B04	SPDSnaoxn	
C2	00C21884	SPDSonoxn	
C3	00C3006A	PSxn	
C4	00C40E04	SPDnoa	
C5	00C50664	SPDSxoxn	
C6	00C60708	SDPnax	
C7	00C707AA	PSDPoaxn	
C8	00C803A8	SDPoa	
C9	00C90184	SPDoxn	
CA	00CA0749	DPSDxax	
CB	00CB06E4	SPDSaoxn	
CC	00CC0020	S	SRCCOPY
CD	00CD0888	SDPono	
CE	00CE0B08	SDPnao	
CF	00CF0224	SPno	

BOOLEAN FUNCTION (HEXADECIMAL)	RASTER OPERATION (HEXADECIMAL)	BOOLEAN FUNCTION IN REVERSE POLISH	COMMON NAME
D0	00D00E0A	PSDnoa	
D1	00D1066A	PSDPxoxn	
D2	00D20705	PDSnax	
D3	00D307A4	SPDSoaxn	
D4	00D41D78	SSPxPDxax	
D5	00D50CE9	DPSanan	
D6	00D616EA	PSDPSaoxx	
D7	00D70349	DPSxan	
D8	00D80745	PDSPxax	
D9	00D906E8	SDPSaoxn	
DA	00DA1CE9	DPSDanax	
DB	00DB0D75	SPxDSxan	
DC	00DC0B04	SPDnao	
DD	00DD0228	SDno	
DE	00DE0268	SDPxo	
DF	00DF08C8	SDPano	
E0	00E003A5	PDSoa	
E1	00E10185	PDSoxn	
E2	00E20746	DSPDxax	
E3	00E306EA	PSDPaoxn	
E4	00E40748	SDPSxax	
E5	00E506E5	PDSPaoxn	
E6	00E61CE8	SDPSanax	
E7	00E70D79	SPxPDxan	
E8	00E81D74	SSPxDSxax	

BOOLEAN FUNCTION (HEXADECIMAL)	RASTER OPERATION (HEXADECIMAL)	BOOLEAN FUNCTION IN REVERSE POLISH	COMMON NAME
E9	00E95CE6	DSPDSanaxxn	
EA	00EA02E9	DPSao	
EB	00EB0849	DPSxno	
EC	00EC02E8	SDPao	
ED	00ED0848	SDPxno	
EE	00EE0086	DSo	SRCPAINT
EF	00EF0A08	SDPnoo	
F0	00F00021	P	PATCOPY
F1	00F10885	PDSono	
F2	00F20B05	PDSnao	
F3	00F3022A	PSno	
F4	00F40B0A	PSDnao	
F5	00F50225	PDno	
F6	00F60265	PDSxo	
F7	00F708C5	PDSano	
F8	00F802E5	PDSao	
F9	00F90845	PDSxno	
FA	00FA0089	DPo	
FB	00FB0A09	DPSnoo	PATPAINT
FC	00FC008A	PSo	
FD	00FD0A0A	PSDnoo	
FE	00FE02A9	DPSoo	
FF	00FF0062	1	WHITENESS
8000	80000000		NOMIRRORBITMAP

Paths (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

A *path* is one or more figures (or shapes) that are filled, outlined, or both filled and outlined. Paths are used in drawing and painting applications. Computer-aided design (CAD) applications use paths to create unique clipping regions, to draw outlines of irregular shapes, and to fill the interiors of irregular shapes. An irregular shape is a shape composed of Bézier curves and straight lines. (Regular shapes include ellipses, circles, rectangles, and polygons.)

- About Paths
- Using Paths
- Path Reference

About Paths

4/20/2022 • 2 minutes to read • Edit Online

A path is one of the objects associated with a device context (DC). However, unlike the default objects (the pen, the brush, and the font) that are part of any new DC, there is no default path. For more information about DCs, see Device Contexts.

- Path Creation
- Outlined and Filled Paths
- Transformations of Paths
- Clip Paths and Graphic Effects
- Conversion of Paths to Regions
- Curved Paths

Path Creation

4/20/2022 • 2 minutes to read • Edit Online

To create a path and select it into a DC, it is first necessary to define the points that describe it. This is done by calling the **BeginPath** function, specifying the appropriate drawing functions, and then by calling the **EndPath** function. This combination of functions (**BeginPath**, drawing functions, and **EndPath**) constitute a *path bracket*. The following is the list of drawing functions that can be used.

- **AngleArc**
- **Arc**
- **ArcTo**
- **Chord**
- **CloseFigure**
- **Ellipse**
- **ExtTextOut**
- **LineTo**
- **MoveToEx**
- **Pie**
- **PolyBezier**
- **PolyBezierTo**
- **PolyDraw**
- **Polygon**
- **Polyline**
- **PolylineTo**
- **PolyPolygon**
- **PolyPolyline**
- **Rectangle**
- **RoundRect**
- **TextOut**

When an application calls **EndPath**, the system selects the associated path into the specified DC. (If another path had previously been selected into the DC, the system deletes that path without saving it.) After the system selects the path into the DC, an application can operate on the path in one of the following ways:

- Draw the outline of the path (using the current pen).
- Paint the interior of the path (using the current brush).
- Draw the outline and fill the interior of the path.
- Modify the path (converting curves to line segments).
- Convert the path into a clip path.
- Convert the path into a region.
- Flatten the path by converting each curve in the path into a series of line segments.
- Retrieve the coordinates of the lines and curves that compose a path.

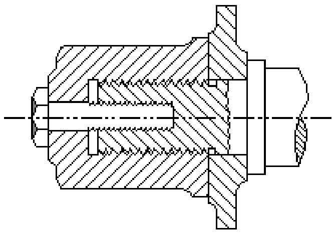
Outlined and Filled Paths

4/20/2022 • 2 minutes to read • Edit Online

An application can draw the outline of a path by calling the **StrokePath** function, it can fill the interior of a path by calling the **FillPath** function, and it can both outline and fill the path by calling the **StrokeAndFillPath** function.

Whenever an application fills a path, the system uses the DC's current fill mode. An application can retrieve this mode by calling the **GetPolyFillMode** function, and it can set a new fill mode by calling the **SetPolyFillMode** function. For a description of the two fill modes, see Regions.

The following illustration shows the cross-section of an object created by a computer-aided design (CAD) application using paths that were both outlined and filled.



Transformations of Paths

4/20/2022 • 2 minutes to read • Edit Online

Paths are defined using logical units and current transformations. (If the **SetWorldTransform** function has been called, the logical units are world units; otherwise, the logical units are page units.) An application can use world transformations to scale, rotate, shear, translate, and reflect the lines and Bézier curves that define a path.

> **NOTE**
>
> A world transformation within a path bracket only affects those lines and curves drawn after the transformation was set. It will have no affect on those lines and curves that were drawn before it was set. For a description of the world transformation, see Coordinate Spaces and Transformations.

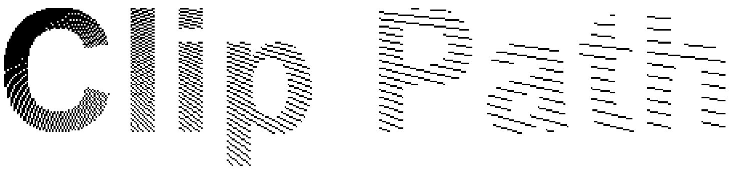
An application can also use **SetWorldTransform** to outline the shape of the pen used to outline a path if the pen is a geometric pen. For a description of geometric pens, see Pens.

Clip Paths and Graphic Effects

4/20/2022 • 2 minutes to read • Edit Online

An application can use clipping and paths to create special graphic effects. The following illustration shows a string of text drawn with a large Arial font.

The next illustration shows the result of selecting the text as a clip path and drawing radial lines for a circle whose center is located above and left of the string.



NOTE

Before graphics device interface (GDI) adds text created with a bitmapped font to a path, it converts the font to an outline or vector font.

An application creates a clip path by generating a path bracket and then calling the **SelectClipPath** function. After a clip path is selected into a DC, output only appears within the borders of the path.

In addition to creating special graphics effects, clip paths are also useful in applications that save images as enhanced metafiles. By using a clip path, an application is able to ensure device independence because the units used to specify a path are logical units.

Conversion of Paths to Regions

4/20/2022 • 2 minutes to read • Edit Online

An application can convert a path into a region by calling the **PathToRegion** function. Like **SelectClipPath**, **PathToRegion** is useful in the creation of special graphics effects. For example, there are no functions that allow an application to offset a path; however, there is a function that enables an application to offset a region (**OffsetRgn**). Using **PathToRegion** , an application can create the effect of animating a complex shape by creating a path that defines the shape, converting the path into a region (by calling **PathToRegion**), and then repeatedly painting, moving, and erasing the region (by calling functions in a sequence, such as **FillRgn**, **OffsetRgn** , and **FillRgn**).

Curved Paths

4/20/2022 • 2 minutes to read • Edit Online

An application can flatten the curves in a path by calling the **FlattenPath** function. This function is especially useful for applications that fit text onto the contour of a path which contains curves. To fit the text, the application must perform the following steps:

1. Create the path where the text appears.
2. Call the **FlattenPath** function to convert the curves in a path into line segments.
3. Call the **GetPath** function to retrieve those line segments.
4. Calculate the length of each line and the width of each character in the string.
5. Use line-width and character-width data to position each character along the curve.

Using Paths

4/20/2022 • 3 minutes to read • Edit Online

This section contains a code sample that enables the user to select a font of a particular point size (by using the Choose Font dialog box), select a clip path (by using text drawn with this font), and then view the result of clipping to the text.

This code sample was used to create the illustration that appears in Clip Paths.

```
CHOOSEFONT cf;          // common dialog box font structure
LOGFONT lf;             // logical font structure
HFONT hfont;            // new logical font handle
HFONT hfontOld;         // original logical font handle
HDC hdc;                // display DC handle
int nXStart, nYStart;   // drawing coordinates
RECT rc;                // structure for painting window
SIZE sz;                // structure that receives text extents
double aflSin[90];      // sine of 0-90 degrees
double aflCos[90];      // cosine of 0-90 degrees
double flRadius,a;      // radius of circle
int iMode;              // clipping mode
HRGN hrgn;              // clip region handle

LRESULT APIENTRY MainWndProc(
    HWND hwnd,                // window handle
    UINT message,             // type of message
    WPARAM wParam,            // additional information
    LPARAM lParam)            // additional information

{

    PAINTSTRUCT ps;

    switch (message)
    {
        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);
            EndPaint(hwnd, &ps);
            break;

        case WM_COMMAND:     // command from app's menu
            switch (wParam)
            {
                case IDM_VANISH:  // erases client area
                    hdc = GetDC(hwnd);
                    GetClientRect(hwnd, &rc);
                    FillRect(hdc, &rc, GetStockObject(WHITE_BRUSH));
                    ReleaseDC(hwnd, hdc);
                    break;

                case IDM_AND: // sets clip mode to RGN_AND
                    iMode = RGN_AND;
                    break;

                case IDM_COPY: // sets clip mode to RGN_COPY
                    iMode = RGN_COPY;
                    break;

                case IDM_DIFF: // sets clip mode to RGN_DIFF
                    iMode = RGN_DIFF;
                    break;
                case IDM_OR: // sets clip mode to RGN_OR
                    iMode = RGN_OR;
                    break;

                case IDM_XOR: // sets clip mode to RGN_XOR
                    iMode = RGN_XOR;
                    break;

                case IDM_CLIP_PATH:

                    // Retrieve a cached DC for the window.

                    hdc = GetDC(hwnd);

                    // Use the font requested by the user in the
                    // Choose Font dialog box to create a logical
                    // font, then select that font into the DC.

                    hfont = CreateFontIndirect(cf.lpLogFont);
                    hfontOld = SelectObject(hdc, hfont);

                    // Retrieve the dimensions of the rectangle
                    // that surrounds the text.

                    GetTextExtentPoint32(hdc, "Clip Path", 9, &sz);

                    // Set a clipping region using the rect that
                    // surrounds the text.

                    hrgn = CreateRectRgn(nXStart, nYStart,
                        nXStart + sz.cx,
                        nYStart + sz.cy);

                    SelectClipRgn(hdc, hrgn);

                    // Create a clip path using text drawn with
                    // the user's requested font.

                    BeginPath(hdc);
                    TextOut(hdc, nXStart, nYStart, "Clip Path", 9);
                    EndPath(hdc);
                    SelectClipPath(hdc, iMode);

                    // Compute the sine of 0, 1, ... 90 degrees.
                    for (i = 0; i < 90; i++)
                    {
                        aflSin[i] = sin( (((double)i) / 180.0)
                                    * 3.14159);
                    }

                    // Compute the cosine of 0, 1,... 90 degrees.
                    for (i = 0; i < 90; i++)
                    {
                        aflCos[i] = cos( (((double)i) / 180.0)
                                    * 3.14159);
                    }

                    // Set the radius value.

                    flRadius = (double)(2 * sz.cx);

                    // Draw the 90 rays extending from the
                    // radius to the edge of the circle.

                    for (i = 0; i < 90; i++)
                    {
                        MoveToEx(hdc, nXStart, nYStart,
                           (LPPOINT) NULL);
                        LineTo(hdc, nXStart + ((int) (flRadius
                            * aflCos[i])),
                               nYStart + ((int) (flRadius
                            * aflSin[i])));
                    }

                    // Reselect the original font into the DC.

                    SelectObject(hdc, hfontOld);

                    // Delete the user's font.

                    DeleteObject(hfont);

                    // Release the DC.

                    ReleaseDC(hwnd, hdc);

                    break;


                case IDM_FONT:

                    // Initialize necessary members.

                    cf.lStructSize = sizeof (CHOOSEFONT);
                    cf.hwndOwner = hwnd;
                    cf.lpLogFont = &lf;
                    cf.Flags = CF_SCREENFONTS | CF_EFFECTS;
                    cf.rgbColors = RGB(0, 255, 255);
                    cf.nFontType = SCREEN_FONTTYPE;

                    // Display the Font dialog box, allow the user
                    // to choose a font, and render text in the
                    // window with that selection.

                    if (ChooseFont(&cf))
                    {
                        hdc = GetDC(hwnd);
                        hfont = CreateFontIndirect(cf.lpLogFont);
                        hfontOld = SelectObject(hdc, hfont);
                        crOld = SetTextColor(hdc, cf.rgbColors);
                        TextOut(hdc, nXStart, nYStart,
                            "Clip Path", 9);
                        SetTextColor(hdc, crOld);
                        SelectObject(hdc, hfontOld);
                        DeleteObject(hfont);
                        ReleaseDC(hwnd, hdc);
                    }

                    break;

                default:
                    return DefWindowProc(hwnd, message, wParam,
                        lParam);
            }
            break;

        case WM_DESTROY:    // window is being destroyed
            PostQuitMessage(0);
            break;

        default:
            return DefWindowProc(hwnd, message, wParam, lParam);
    }
    return 0;
}
```

Path Reference

4/20/2022 • 2 minutes to read • Edit Online

The following elements are used to create, alter, or draw paths.

- Path Functions

Path Functions (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used to create, alter, or draw paths.

FUNCTION	DESCRIPTION
AbortPath	Closes and discards any paths in the specified device context.
BeginPath	Opens a path bracket in the specified device context.
CloseFigure	Closes an open figure in a path.
EndPath	Closes a path bracket and selects the path defined by the bracket into the specified device context.
FillPath	Closes any open figures in the current path and fills the path's interior by using the current brush and polygon-filling mode.
FlattenPath	Transforms any curves in the path that is selected into the current device context (DC), turning each curve into a sequence of lines.
GetMiterLimit	Retrieves the miter limit for the specified device context.
GetPath	Retrieves the coordinates defining the endpoints of lines and the control points of curves found in the path that is selected into the specified device context.
PathToRegion	Creates a region from the path that is selected into the specified device context.
SetMiterLimit	Sets the limit for the length of miter joins for the specified device context.
StrokeAndFillPath	Closes any open figures in a path, strokes the outline of the path by using the current pen, and fills its interior by using the current brush.
StrokePath	Renders the specified path by using the current pen.
WidenPath	Redefines the current path as the area that would be painted if the path were stroked using the pen currently selected into the given device context.

Pens

4/20/2022 • 2 minutes to read • Edit Online

A pen is a graphics tool that an application can use to draw lines and curves. Drawing applications use pens to draw freehand lines, straight lines, and curves. Computer-aided design (CAD) applications use pens to draw visible lines, hidden lines, section lines, center lines, and so on. Word processing and desktop publishing applications use pens to draw borders and rules. Spreadsheet applications use pens to designate trends in graphs and to outline bar graphs and pie charts.

- About Pens
- Using Pens
- Pen Reference

About Pens

4/20/2022 • 2 minutes to read • Edit Online

There are two types of pen: cosmetic and geometric. A *cosmetic pen* is used with applications requiring lines of fixed width and lines that are quickly drawn. A CAD application, for example, uses a cosmetic pen to generate hidden, section, center, and dimension lines that are between .015 and .022 inches wide regardless of the scale factor. A *geometric pen* is used with applications requiring scalable lines, lines with unique end or join styles, and lines that are wider than a single pixel. A spreadsheet application, for example, uses a geometric pen to define each of the bars in a bar graph as a wide line.

- Cosmetic Pens
- Geometric Pens
- Pen Attributes
- ICM-Enabled Pen Functions

Cosmetic Pens

4/20/2022 • 2 minutes to read • Edit Online

The dimensions of a cosmetic pen are specified in device units. Therefore, lines drawn with a cosmetic pen always have a fixed width. Lines drawn with a cosmetic pen are generally drawn 3 to 10 times faster than lines drawn with a geometric pen. Cosmetic pens have three attributes: width, style, and color. For more information about these attributes, see Pen Attributes.

To create a cosmetic pen, use the **CreatePen**, **CreatePenIndirect**, or **ExtCreatePen** function. To retrieve one of the three stock cosmetic pens managed by the system, use the **GetStockObject** function.

After you create a pen (or obtain a handle to one of the stock pens), select the pen into the application's device context (DC) using the **SelectObject** function. From this point on, the application uses this pen for any line-drawing operations in its client area.

Geometric Pens

4/20/2022 • 2 minutes to read • Edit Online

The dimensions of a geometric pen are specified in logical units. Therefore, lines drawn with a geometric pen can be scaled that is, they may appear wider or narrower, depending on the current world transformation. For more information about world transformation, see Coordinate Spaces and Transformations.

In addition to the three attributes shared with cosmetic pens (width, style, and color), geometric pens possess the following four attributes: pattern, optional hatch, end style, and join style. For more information about these attributes, see Pen Attributes.

To create a geometric pen, an application uses the **ExtCreatePen** function. As with cosmetic pens, the **SelectObject** function selects a geometric pen into the application's DC.

Pen Attributes

4/20/2022 • 2 minutes to read • Edit Online

There are seven pen attributes that define the type of pen and its characteristics: width, style, color, pattern, hatch, end style, and join style. Both cosmetic and geometric pens have the width, style, and color attributes. Only geometric pens have the pattern, hatch, end style, and join style attributes. The pattern and optional hatch attribute are usually associated with a brush but can also be used with geometric pens.

For more information, see the following topics:

- Pen Width
- Pen Style
- Pen Color
- Pen Pattern
- Pen Hatch
- Pen End Cap
- Pen Join

Pen Width

4/20/2022 • 2 minutes to read • Edit Online

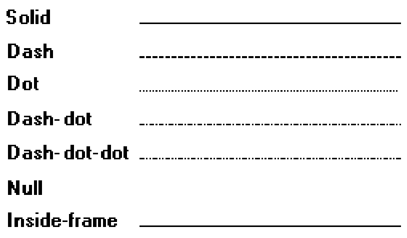
The width attribute specifies a cosmetic pen width in device units. When used with a geometric pen, however, it specifies the pen's width in logical units. For more information about device units, see Coordinate Spaces and Transformations.

Currently, the system limits the width of cosmetic pens to a single pixel; however, future versions may remove this limitation.

Pen Style

4/20/2022 • 2 minutes to read • Edit Online

The style attribute specifies the line pattern that appears when a particular cosmetic or geometric pen is used. There are eight predefined pen styles. The following illustration shows the seven of these styles that are defined by the system.



The inside-frame style is identical to the solid style for cosmetic pens. However, it operates differently when used with a geometric pen. If the geometric pen is wider than a single pixel and a drawing function uses the pen to draw a border around a filled object, the system draws the border inside the object's frame. By using the inside-frame style, an application can ensure that an object appears entirely within the specified dimensions, regardless of the geometric pen width.

In addition to the seven styles defined by the system, there is an eighth style that is user (or application) defined. A user-defined style generates lines with a customized series of dashes and dots.

Use the **CreatePen**, **CreatePenIndirect**, or **ExtCreatePen** function to create a pen that has the system-defined styles. Use the **ExtCreatePen** function to create a pen that has a user-defined style.

Pen Color

4/20/2022 • 2 minutes to read • Edit Online

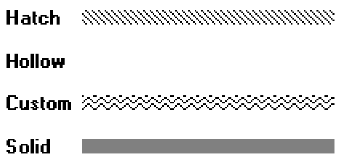
The color attribute specifies the pen's color. An application can create a cosmetic pen with a unique color by using the **RGB** macro to store the red, green, blue triplet that specifies the desired color in a COLORREF structure and passing this structure's address to the **CreatePen**, **CreatePenIndirect**, or **ExtCreatePen** function. (The stock pens are limited to black, white, and invisible.) For more information about RGB triplets and color, see Colors.

Pen Pattern

4/20/2022 • 2 minutes to read • Edit Online

The pattern attribute specifies the pattern of a geometric pen.

The following illustration shows lines drawn with different geometric pens. Each pen was created using a different pattern attribute.

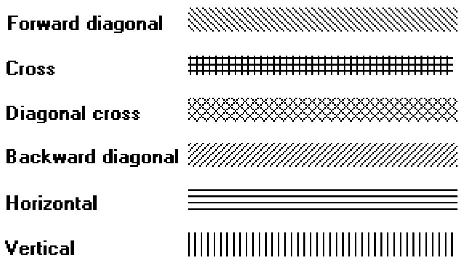


The first line in the previous illustration is drawn using one of the six available hatch patterns; for more information about hatch patterns, see Pen Hatch. The next line is drawn using the hollow pattern, identical to the null pattern. The third line is drawn using a custom pattern created from an 8-by-8-pixel bitmap. (For more information about bitmaps and their creation, see Bitmaps.) The last line is drawn using a solid pattern. Creating a brush and passing its handle to the **ExtCreatePen** function creates a pattern.

Pen Hatch

4/20/2022 • 2 minutes to read • Edit Online

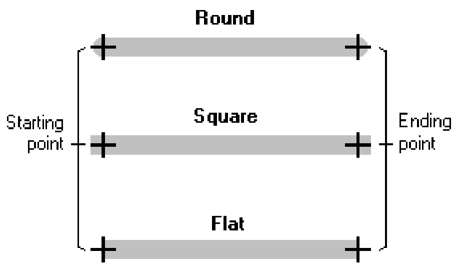
The hatch attribute specifies the hatch type of a geometric pen with the hatch pattern attribute. There are six patterns available. The following illustration shows lines drawn using different hatch patterns.



Pen End Cap

4/20/2022 • 2 minutes to read • Edit Online

The end cap attribute specifies the shape of a geometric pen: round, square, or flat. The following illustration shows parallel lines drawn using each type of end cap.

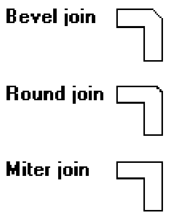


The round and square end caps extend past the starting and ending points of a line drawn with a geometric pen; the flat end cap does not.

Pen Join

4/20/2022 • 2 minutes to read • Edit Online

The join attribute specifies how the ends of two geometric lines are joined: beveled, mitered, or round. The following illustration shows pairs of connected lines drawn using each type of join.



ICM-Enabled Pen Functions

4/20/2022 • 2 minutes to read • Edit Online

Microsoft Image Color Management (ICM) ensures that a color image, graphic, or text object is rendered as close as possible to its original intent on any device, despite differences in imaging technologies and color capabilities among devices. Whether you are scanning an image or other graphic on a color scanner, downloading it over the Internet, viewing or editing it on the screen, or outputting it to paper, film, or other media, ICM version 2.0 helps you keep its colors consistent and accurate. For more information about ICM, see Windows Color System

There are various functions in the graphics device interface (GDI) that use or operate on color data. The following pen functions are enabled for use with ICM:

- **CreatePen**
- **ExtCreatePen**

Using Pens

4/20/2022 • 2 minutes to read • Edit Online

This section contains sample code that demonstrates the appearance of lines drawn using various pen styles and attributes.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam,
                             LPARAM lParam)
{
    PAINTSTRUCT ps;
    LOGBRUSH lb;
    RECT rc;
    HDC hdc;
    int i;
    HGDIOBJ hPen = NULL;
    HGDIOBJ hPenOld;
    DWORD dwPenStyle[] = {
                           PS_DASH,
                           PS_DASHDOT,
                           PS_DOT,
                           PS_INSIDEFRAME,
                           PS_NULL,
                           PS_SOLID
                        };
    UINT uHatch[] = {
                      HS_BDIAGONAL,
                      HS_CROSS,
                      HS_DIAGCROSS,
                      HS_FDIAGONAL,
                      HS_HORIZONTAL,
                      HS_VERTICAL
                   };

    switch (uMsg)
    {
        case WM_PAINT:
        {
            GetClientRect(hWnd, &rc);
            rc.left += 10;
            rc.top += 10;
            rc.bottom -= 10;

            // Initialize the pen's brush.
            lb.lbStyle = BS_SOLID;
            lb.lbColor = RGB(255,0,0);
            lb.lbHatch = 0;

            hdc = BeginPaint(hWnd, &ps);
            for (i = 0; i < 6; i++)
            {
                hPen = ExtCreatePen(PS_COSMETIC | dwPenStyle[i],
                                    1, &lb, 0, NULL);
                hPenOld = SelectObject(hdc, hPen);
                MoveToEx(hdc, rc.left + (i * 20), rc.top, NULL);
                LineTo(hdc, rc.left + (i * 20), rc.bottom);
                SelectObject(hdc, hPenOld);
                DeleteObject(hPen);
            }
            rc.left += 150;
            for (i = 0; i < 6; i++)
            {
                lb.lbStyle = BS_HATCHED;
                lb.lbColor = RGB(0,0,255);
                lb.lbHatch = uHatch[i];
                hPen = ExtCreatePen(PS_GEOMETRIC,
                                    5, &lb, 0, NULL);
                hPenOld = SelectObject(hdc, hPen);
                MoveToEx(hdc, rc.left + (i * 20), rc.top, NULL);
                LineTo(hdc, rc.left + (i * 20), rc.bottom);
                SelectObject(hdc, hPenOld);
                DeleteObject(hPen);
            }
            EndPaint(hWnd, &ps);

        }
        break;

        case WM_DESTROY:
            DeleteObject(hPen);
            PostQuitMessage(0);
            break;

        default:
            return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }

    return FALSE;
}
```

Pen Reference

4/20/2022 • 2 minutes to read • Edit Online

The following elements are used with pens:

- Pen Functions
- Pen Structures

Pen Functions (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used with pens.

FUNCTION	DESCRIPTION
CreatePen	Creates a logical pen that has the specified style, width, and color.
CreatePenIndirect	Creates a logical cosmetic pen that has the style, width, and color specified in a structure.
ExtCreatePen	Creates a logical cosmetic or geometric pen that has the specified style, width, and brush attributes.
SetDCPenColor	Sets the current device context pen color.

Pen Structures

4/20/2022 • 2 minutes to read • Edit Online

The following structures are used with pens:

- EXTLOGPEN
- LOGPEN

Rectangles

4/20/2022 • 2 minutes to read • Edit Online

Applications use *rectangles* to specify rectangular areas on the screen or in a window.

- About Rectangles
- Using Rectangles
- Rectangle Reference

About Rectangles

4/20/2022 • 2 minutes to read • Edit Online

Rectangles are used for the cursor clipping region, the invalid portion of the client area, an area for displaying formatted text, or the scroll area. Your applications can also use rectangles to fill, frame, or invert a portion of the client area with a given brush, and to retrieve the coordinates of a window or a window's client area.

- Rectangle Coordinates
- Rectangle Operations

Rectangle Coordinates

4/20/2022 • 2 minutes to read • Edit Online

An application must use a **RECT** structure to define a rectangle. The structure specifies the coordinates of two points: the upper left and lower right corners of the rectangle. The sides of the rectangle extend from these two points and are parallel to the x-axis and y-axis.

The coordinate values for a rectangle are expressed as signed integers. The coordinate value of a rectangle's right side must be greater than that of its left side. Likewise, the coordinate value of the bottom must be greater than that of the top.

Because applications can use rectangles for many different purposes, the rectangle functions do not use an explicit unit of measure. Instead, all rectangle coordinates and dimensions are given in signed, logical values. The mapping mode and the function in which the rectangle is used determine the units of measure. For more information about coordinates and mapping modes, see Coordinate Spaces and Transformations.

Rectangle Operations

4/20/2022 • 2 minutes to read • Edit Online

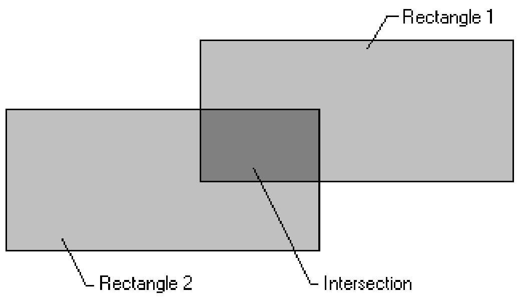
The SetRect function creates a rectangle, the CopyRect function makes a copy of a given rectangle, and the SetRectEmpty function creates an empty rectangle. An empty rectangle is any rectangle that has zero width, zero height, or both. The IsRectEmpty function determines whether a given rectangle is empty. The EqualRect function determines whether two rectangles are identical that is, whether they have the same coordinates.

The InflateRect function increases or decreases the width or height of a rectangle, or both. It can add or remove width from both ends of the rectangle; it can add or remove height from both the top and bottom of the rectangle.

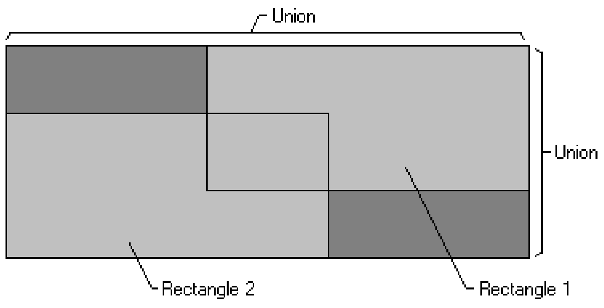
The OffsetRect function moves a rectangle by a given amount. It moves the rectangle by adding the given x-amount, y-amount, or x- and y-amounts to the corner coordinates.

The PtInRect function determines whether a given point lies within a given rectangle. The point is in the rectangle if it lies on the left or top side or is completely within the rectangle. The point is not in the rectangle if it lies on the right or bottom side.

The IntersectRect function creates a new rectangle that is the intersection of two existing rectangles, as shown in the following figure.



The UnionRect function creates a new rectangle that is the union of two existing rectangles, as shown in the following figure.



For information about functions that draw ellipses and polygons, see Filled Shapes.

Using Rectangles (Windows GDI)

4/20/2022 • 6 minutes to read • Edit Online

The example in this section illustrates how to use the rectangle functions. It consists of the main window procedure from an application that enables the user to move and size a bitmap.

When the application starts, it draws a 32-pixel by 32-pixel bitmap in the upper left corner of the screen. The user can move the bitmap by dragging it. To size the bitmap, the user creates a target rectangle by dragging the mouse, then drags the bitmap and "drops" it on the target rectangle. The application responds by copying the bitmap into the target rectangle.

The window procedure that allows the user to move and size the bitmap is given in the following example.

```
LRESULT CALLBACK MainWndProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;                 // device context (DC) for window
    RECT rcTmp;              // temporary rectangle
    PAINTSTRUCT ps;          // paint data for BeginPaint and EndPaint
    POINT ptClientUL;        // client area upper left corner
    POINT ptClientLR;        // client area lower right corner
    static HDC hdcCompat;    // DC for copying bitmap
    static POINT pt;         // x and y coordinates of cursor
    static RECT rcBmp;       // rectangle that encloses bitmap
    static RECT rcTarget;    // rectangle to receive bitmap
    static RECT rcClient;    // client-area rectangle
    static BOOL fDragRect;   // TRUE if bitmap rect. is dragged
    static HBITMAP hbmp;     // handle of bitmap to display
    static HBRUSH hbrBkgnd;  // handle of background-color brush
    static COLORREF crBkgnd; // color of client-area background
    static HPEN hpenDot;     // handle of dotted pen

    switch (uMsg)
    {
        case WM_CREATE:

            // Load the bitmap resource.

            hbmp = LoadBitmap(hinst, MAKEINTRESOURCE(1));

            // Create a device context (DC) to hold the bitmap.
            // The bitmap is copied from this DC to the window's DC
            // whenever it must be drawn.

            hdc = GetDC(hwnd);
            hdcCompat = CreateCompatibleDC(hdc);
            SelectObject(hdcCompat, hbmp);

            // Create a brush of the same color as the background
            // of the client area. The brush is used later to erase
            // the old bitmap before copying the bitmap into the
            // target rectangle.

            crBkgnd = GetBkColor(hdc);
            hbrBkgnd = CreateSolidBrush(crBkgnd);
            ReleaseDC(hwnd, hdc);

            // Create a dotted pen. The pen is used to draw the
            // bitmap rectangle as the user drags it.

            hpenDot = CreatePen(PS_DOT, 1, RGB(0, 0, 0));
            // Set the initial rectangle for the bitmap. Note that
            // this application supports only a 32- by 32-pixel
            // bitmap. The rectangle is slightly larger than the
            // bitmap.

            SetRect(&rcBmp, 1, 1, 34, 34);
            return 0;

        case WM_PAINT:

            // Draw the bitmap rectangle and copy the bitmap into
            // it. The 32-pixel by 32-pixel bitmap is centered
            // in the rectangle by adding 1 to the left and top
            // coordinates of the bitmap rectangle, and subtracting 2
            // from the right and bottom coordinates.

            BeginPaint(hwnd, &ps);
            Rectangle(ps.hdc, rcBmp.left, rcBmp.top,
                rcBmp.right, rcBmp.bottom);
            StretchBlt(ps.hdc, rcBmp.left + 1, rcBmp.top + 1,
                (rcBmp.right - rcBmp.left) - 2,
                (rcBmp.bottom - rcBmp.top) - 2, hdcCompat,
                0, 0, 32, 32, SRCCOPY);
            EndPaint(hwnd, &ps);
            break;

        case WM_MOVE:
        case WM_SIZE:

            // Convert the client coordinates of the client-area
            // rectangle to screen coordinates and save them in a
            // rectangle. The rectangle is passed to the ClipCursor
            // function during WM_LBUTTONDOWN processing.

            GetClientRect(hwnd, &rcClient);
            ptClientUL.x = rcClient.left;
            ptClientUL.y = rcClient.top;
            ptClientLR.x = rcClient.right;
            ptClientLR.y = rcClient.bottom;
            ClientToScreen(hwnd, &ptClientUL);
            ClientToScreen(hwnd, &ptClientLR);
            SetRect(&rcClient, ptClientUL.x, ptClientUL.y,
                ptClientLR.x, ptClientLR.y);
            return 0;

        case WM_LBUTTONDOWN:

            // Restrict the mouse cursor to the client area. This
            // ensures that the window receives a matching
            // WM_LBUTTONUP message.

            ClipCursor(&rcClient);

            // Save the coordinates of the mouse cursor.

            pt.x = (LONG) LOWORD(lParam);
            pt.y = (LONG) HIWORD(lParam);

            // If the user has clicked the bitmap rectangle, redraw
            // it using the dotted pen. Set the fDragRect flag to
            // indicate that the user is about to drag the rectangle.

            if (PtInRect(&rcBmp, pt))
            {
                hdc = GetDC(hwnd);
                SelectObject(hdc, hpenDot);
                Rectangle(hdc, rcBmp.left, rcBmp.top, rcBmp.right,
                    rcBmp.bottom);
                fDragRect = TRUE;
                fDragRect = TRUE;
                ReleaseDC(hwnd, hdc);
            }
            return 0;

        case WM_MOUSEMOVE:

            // Draw a target rectangle or drag the bitmap rectangle,
            // depending on the status of the fDragRect flag.

            if ((wParam && MK_LBUTTON)
                    && !fDragRect)
            {
                // Set the mix mode so that the pen color is the
                // inverse of the background color. The previous
                // rectangle can then be erased by drawing
                // another rectangle on top of it.

                hdc = GetDC(hwnd);
                SetROP2(hdc, R2_NOTXORPEN);

                // If a previous target rectangle exists, erase
                // it by drawing another rectangle on top of it.

                if (!IsRectEmpty(&rcTarget))
                {
                    Rectangle(hdc, rcTarget.left, rcTarget.top,
                        rcTarget.right, rcTarget.bottom);
                }

                // Save the coordinates of the target rectangle. Avoid
                // invalid rectangles by ensuring that the value of
                // the left coordinate is lesser than the
                // right coordinate, and that the value of the top
                // coordinate is lesser than the bottom coordinate.

                if ((pt.x < (LONG) LOWORD(lParam)) &&
                        (pt.y > (LONG) HIWORD(lParam)))
                {
                    SetRect(&rcTarget, pt.x, HIWORD(lParam),
                        LOWORD(lParam), pt.y);
                }
                else if ((pt.x > (LONG) LOWORD(lParam)) &&
                        (pt.y > (LONG) HIWORD(lParam)))
                {
                    SetRect(&rcTarget, LOWORD(lParam),
                        HIWORD(lParam), pt.x, pt.y);
                }
                else if ((pt.x > (LONG) LOWORD(lParam)) &&
                        (pt.y < (LONG) HIWORD(lParam)))
                {
                    SetRect(&rcTarget, LOWORD(lParam), pt.y,
                        pt.x, HIWORD(lParam));
                }
                else
                {
                    SetRect(&rcTarget, pt.x, pt.y, LOWORD(lParam),
                        HIWORD(lParam));
                }

                // Draw the new target rectangle.

                Rectangle(hdc, rcTarget.left, rcTarget.top,
                    rcTarget.right, rcTarget.bottom);
                ReleaseDC(hwnd, hdc);
            }
            else if ((wParam && MK_LBUTTON)
                    && fDragRect)
            {
                // Set the mix mode so that the pen color is the
                // inverse of the background color.

                hdc = GetDC(hwnd);
                SetROP2(hdc, R2_NOTXORPEN);

                // Select the dotted pen into the DC and erase
                // the previous bitmap rectangle by drawing
                // another rectangle on top of it.

                SelectObject(hdc, hpenDot);
                Rectangle(hdc, rcBmp.left, rcBmp.top,
                    rcBmp.right, rcBmp.bottom);

                // Set the new coordinates of the bitmap rectangle,
                // then redraw it.

                OffsetRect(&rcBmp, LOWORD(lParam) - pt.x,
                    HIWORD(lParam) - pt.y);
                Rectangle(hdc, rcBmp.left, rcBmp.top,
                    rcBmp.right, rcBmp.bottom);
                ReleaseDC(hwnd, hdc);

                // Save the coordinates of the mouse cursor.

                pt.x = (LONG) LOWORD(lParam);
                pt.y = (LONG) HIWORD(lParam);
            }
            return 0;

        case WM_LBUTTONUP:

            // If the bitmap rectangle and target rectangle
            // intersect, copy the bitmap into the target
            // rectangle. Otherwise, copy the bitmap into the
            // rectangle bitmap at its new location.

            if (IntersectRect(&rcTmp, &rcBmp, &rcTarget))
            {

                // Erase the bitmap rectangle by filling it with
                // the background color.

                hdc = GetDC(hwnd);
                FillRect(hdc, &rcBmp, hbrBkgnd);

                // Redraw the target rectangle because the part
                // that intersected with the bitmap rectangle was
                // erased by the call to FillRect.

                Rectangle(hdc, rcTarget.left, rcTarget.top,
                    rcTarget.right, rcTarget.bottom);

                // Copy the bitmap into the target rectangle.

                StretchBlt(hdc, rcTarget.left + 1, rcTarget.top + 1,
                    (rcTarget.right - rcTarget.left) - 2,
                    (rcTarget.bottom - rcTarget.top) - 2, hdcCompat,
                    0, 0, 32, 32, SRCCOPY);

                // Copy the target rectangle to the bitmap
                // rectangle, set the coordinates of the target
                // rectangle to 0, then reset the fDragRect flag.

                CopyRect(&rcBmp, &rcTarget);
                SetRectEmpty(&rcTarget);
                ReleaseDC(hwnd, hdc);
                fDragRect = FALSE;
            }

            else if (fDragRect)
            {

                // Draw the bitmap rectangle, copy the bitmap into
                // it, and reset the fDragRect flag.

                hdc = GetDC(hwnd);
                Rectangle(hdc, rcBmp.left, rcBmp.top,
                    rcBmp.right, rcBmp.bottom);
                StretchBlt(hdc, rcBmp.left + 1, rcBmp.top + 1,
                    (rcBmp.right - rcBmp.left) - 2,
                    (rcBmp.bottom - rcBmp.top) - 2, hdcCompat,
                    0, 0, 32, 32, SRCCOPY);
                ReleaseDC(hwnd, hdc);
                fDragRect = FALSE;
            }

            // Release the mouse cursor.

            ClipCursor((LPRECT) NULL);
            return 0;

        case WM_DESTROY:

            // Destroy the background brush, compatible bitmap,
            // and the bitmap.

            DeleteObject(hbrBkgnd);
            DeleteDC(hdcCompat);
            DeleteObject(hbmp);
            PostQuitMessage(0);
            break;

        default:
            return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
    return (LRESULT) NULL;
}
```

Rectangle Reference

4/20/2022 • 2 minutes to read • Edit Online

The following elements are used with rectangles:

- Rectangle Functions
- Rectangle Structures
- Rectangle Macros

Rectangle Functions

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used with rectangles.

FUNCTION	DESCRIPTION
CopyRect	Copies the coordinates of one rectangle to another.
EqualRect	Determines whether the two specified rectangles are equal by comparing the coordinates of their upper-left and lower-right corners.
InflateRect	Increases or decreases the width and height of the specified rectangle.
IntersectRect	Calculates the intersection of two source rectangles and places the coordinates of the intersection rectangle into the destination rectangle.
IsRectEmpty	Determines whether the specified rectangle is empty.
OffsetRect	Moves the specified rectangle by the specified offsets.
PtInRect	Determines whether the specified point lies within the specified rectangle.
SetRect	Sets the coordinates of the specified rectangle.
SetRectEmpty	Creates an empty rectangle in which all coordinates are set to zero.
SubtractRect	Determines the coordinates of a rectangle formed by subtracting one rectangle from another.
UnionRect	Creates the union of two rectangles.

Rectangle Structures

4/20/2022 • 2 minutes to read • Edit Online

The following structures are used with rectangles.

- POINT
- POINTS
- RECT

Rectangle Macros

4/20/2022 • 2 minutes to read • Edit Online

The following macros are used with rectangles.

- MAKEPOINTS
- POINTSTOPOINT
- POINTTOPOINTS

Regions (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

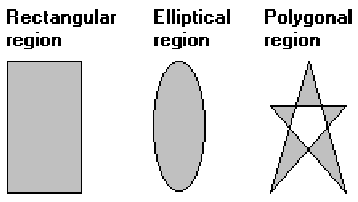
A *region* is a rectangle, polygon, or ellipse (or a combination of two or more of these shapes) that can be filled, painted, inverted, framed, and used to perform hit testing (testing for the cursor location).

- About Regions
- Using Regions
- Region Reference

About Regions

4/20/2022 • 2 minutes to read • Edit Online

Following are three types of regions that have been filled and framed.



- Region Creation and Selection
- Region Operations

Region Creation and Selection

4/20/2022 • 2 minutes to read • Edit Online

An application creates a region by calling a function associated with a specific shape. The following table shows the function(s) associated with each of the standard shapes.

SHAPE	FUNCTION
Rectangular region	CreateRectRgn, CreateRectRgnIndirect, SetRectRgn
Rectangular region with rounded corners	CreateRoundRectRgn
Elliptical region	CreateEllipticRgn, CreateEllipticRgnIndirect
Polygonal region	CreatePolygonRgn, CreatePolyPolygonRgn

Each region creation function returns a handle that identifies the new region. An application can use this handle to select the region into a device context by calling the SelectObject function and supplying this handle as the second argument. After a region is selected into a device context, the application can perform various operations on it.

Region Operations

4/20/2022 • 2 minutes to read • Edit Online

Applications can combine regions, compare them, paint or invert their interiors, draw a frame around them, retrieve their dimensions, and test whether the cursor lies within their boundaries.

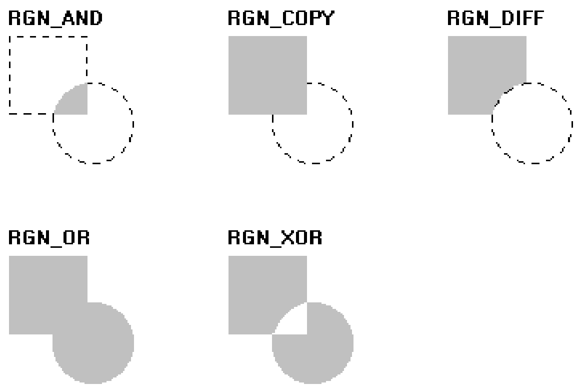
Combining Regions

4/20/2022 • 2 minutes to read • Edit Online

An application combines two regions by calling the **CombineRgn** function. Using this function, an application can combine the intersecting parts of two regions, all but the intersecting parts of two regions, the two original regions in their entirety, and so on. Following are five values that define the region combinations.

VALUE	MEANING
RGN_AND	The intersecting parts of two original regions define a new region.
RGN_COPY	A copy of the first (of the two original regions) defines a new region.
RGN_DIFF	The part of the first region that does not intersect the second defines a new region.
RGN_OR	The two original regions define a new region.
RGN_XOR	Those parts of the two original regions that do not overlap define a new region.

The following illustration shows the five possible combinations of a square and a circular region resulting from a call to **CombineRgn**.



Comparing Regions

4/20/2022 • 2 minutes to read • Edit Online

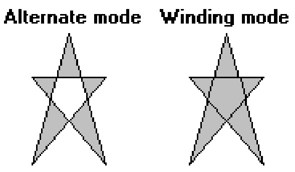
An application compares two regions to determine whether they are identical by calling the **EqualRgn** function. EqualRgn considers two regions identical if they are equal in size and shape.

Filling Regions

4/20/2022 • 2 minutes to read • Edit Online

An application fills the interior of a region by calling the **FillRgn** function and supplying a handle that identifies a specific brush. When an application calls FillRgn , the system fills the region with the brush by using the current fill mode for the specified device context. There are two fill modes: alternate and winding. The application can set the fill mode for a device context by calling the **SetPolyFillMode** function. The application can retrieve the current fill mode for a device context by calling the **GetPolyFillMode** function.

The following illustration shows two identical regions: one filled using alternate mode and the other filled using winding mode.



Alternate Mode

To determine which pixels the system highlights when alternate mode is specified, perform the following test:

1. Select a pixel within the region's interior.
2. Draw an imaginary ray, in the positive x-direction, from that pixel toward infinity.
3. Each time the ray intersects a boundary line, increment a count value.

The system highlights the pixel if the count value is an odd number.

Winding Mode

To determine which pixels the system highlights when winding mode is specified, perform the following test:

1. Determine the direction in which each boundary line is drawn.
2. Select a pixel within the region's interior.
3. Draw an imaginary ray, in the positive x-direction, from the pixel toward infinity.
4. Each time the ray intersects a boundary line with a positive y-component, increment a count value. Each time the ray intersects a boundary line with a negative y-component, decrement the count value.

The system highlights the pixel if the count value is nonzero.

Painting Regions

4/20/2022 • 2 minutes to read • Edit Online

An application fills the interior of a region by using the brush currently selected into a device context by the **PaintRgn** function. This function uses the current polygon fill modes (alternate and winding).

Inverting Regions

4/20/2022 • 2 minutes to read • Edit Online

An application inverts the colors that appear within a region by calling the **InvertRgn** function. On monochrome displays, **InvertRgn** makes white pixels black and black pixels white. On color screens, this inversion is dependent on the type of technology used to generate the colors for the screen.

Framing Regions

4/20/2022 • 2 minutes to read • Edit Online

An application draws a border around a region by calling the **FrameRgn** function and specifying the border width and brush pattern that the system uses when drawing the frame.

Retrieving a Bounding Rectangle

4/20/2022 • 2 minutes to read • Edit Online

An application retrieves the dimensions of a region's bounding rectangle by calling the **GetRgnBox** function. If the region is rectangular, **GetRgnBox** returns the dimensions of the region. If the region is elliptical, the function returns the dimensions of the smallest rectangle that can be drawn around the ellipse. The long sides of the rectangle are the same length as the ellipse's major axis, and the short sides of the rectangle are the same length as the ellipse's minor axis. If the region is polygonal, **GetRgnBox** returns the dimensions of the smallest rectangle that can be drawn around the entire polygon.

Moving Regions

4/20/2022 • 2 minutes to read • Edit Online

An application moves a region by calling the **OffsetRgn** function. The given offsets along the x-axis and y-axis determine the number of logical units to move left or right and up or down.

Hit Testing Regions

4/20/2022 • 2 minutes to read • Edit Online

An application performs hit testing on regions to determine the coordinates of the current cursor position. Then it passes these coordinates as well as a handle identifying the region to the **PtInRegion** function. The cursor coordinates can be retrieved by processing the various mouse messages, such as **WM_LBUTTONDOWN** , **WM_LBUTTONUP** , **WM_RBUTTONDOWN** , and **WM_RBUTTONUP**. The return value for PtInRegion indicates whether the cursor position is within the given region.

Using Regions (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

This topic has code examples for the following tasks.

- Using regions to clip output
- Using regions to perform hit testing

Using Regions to Clip Output

4/20/2022 • 2 minutes to read • Edit Online

This section contains a single example that demonstrates how you can use regions to enable the user to define how a part of client area output can appear. Regions used for this purpose are called clipping regions.

The example for this section is taken from an application that enables a user to capture the entire desktop as a bitmap and then isolate and save a part of this image as a .BMP file.

By clicking **Define Clip Region** from the application's menu, the user is able to select a clipping region by clicking the left mouse button and dragging the mouse. As the user drags the mouse, the application draws a rectangle that corresponds to the new clipping region.

By clicking **Clip** from the application's menu, the user is able to redraw the isolated part of the image within the boundaries of the specified rectangle.

This section provides information on the following topics.

- Defining the Clipping Region
- Clipping Output

Defining the Clipping Region

4/20/2022 • 2 minutes to read • Edit Online

When the user clicks Define Clip Region , the system issues a WM_COMMAND message. The *wParam* parameter of this message contains an application-defined constant, IDM_DEFINE, that indicates that the user selected this option from the menu. The application processes this input by setting a Boolean flag, fDefineRegion, as shown in the following code sample.

```
case WM_COMMAND:
    switch (wParam)
    {

        case IDM_DEFINE:
            fDefineRegion = TRUE;
            break;
```

After clicking **Define Clipping Region** , the user can begin drawing the rectangle by clicking and dragging the mouse while the cursor is in the application's client area.

When the user presses the left button, the system issues a WM_LBUTTONDOWN message. The *lParam* parameter of this message contains the cursor coordinates, which correspond to the upper left corner of a rectangle used to define the clipping region. The application processes the **WM_LBUTTONDOWN** message, as follows.

```
// These variables are required for clipping.

static POINT ptUpperLeft;
static POINT ptLowerRight;
static POINT aptRect[5];
static POINT ptTmp;
static POINTS ptsTmp;
static BOOL fDefineRegion;
static BOOL fRegionExists;
static HRGN hrgn;
static RECT rctTmp;
int i;

switch (message)
{

    case WM_LBUTTONDOWN:
        if (fDefineRegion)
        {

        // Retrieve the new upper left corner.

            ptsTmp = MAKEPOINTS(lParam);
            ptUpperLeft.x = (LONG) ptsTmp.x;
            ptUpperLeft.y = (LONG) ptsTmp.y;
        }

        if (fRegionExists)
        {

            // Erase the previous rectangle.

            hdc = GetDC(hwnd);
            SetROP2(hdc, R2_NOTXORPEN);
            if (!Polyline(hdc, (CONST POINT *) aptRect, 5))
                errhandler("Polyline Failed", hwnd);
            ReleaseDC(hwnd, hdc);

            // Clear the rectangle coordinates.

            for (i = 0; i < 4; i++)
            {
                aptRect[i].x = 0;
                aptRect[i].y = 0;
            }

            // Clear the temporary point structure.

            ptTmp.x = 0;
            ptTmp.y = 0;

            // Clear the lower right coordinates.

            ptLowerRight.x = 0;
            ptLowerRight.y = 0;

            // Reset the flag.

            fRegionExists = FALSE;
            fDefineRegion = TRUE;

            // Retrieve the new upper left corner.

            ptsTmp = MAKEPOINTS(lParam);
            ptUpperLeft.x = (LONG) ptsTmp.x;
            ptUpperLeft.y = (LONG) ptsTmp.y;
        }
    break;
}
```

As the user drags the mouse, the system issues WM_MOUSEMOVE messages and stores the new cursor coordinates in the *lParam* parameter. Each time the application receives a new **WM_MOUSEMOVE** message, it erases the previous rectangle (if one exists) and draws the new rectangle by calling the Polyline function, passing it the coordinates of the four corners of the rectangle. The application performs the following tasks.

```
// These variables are required for clipping.

static POINT ptUpperLeft;
static POINT ptLowerRight;
static POINT aptRect[5];
static POINT ptTmp;
static POINTS ptsTmp;
static BOOL fDefineRegion;
static BOOL fRegionExists;
static HRGN hrgn;
static RECT rctTmp;
int i;

switch (message)
{

    case WM_MOUSEMOVE:

    if (wParam & MK_LBUTTON && fDefineRegion)
    {

        // Get a window DC.

        hdc = GetDC(hwnd);
        if (!SetROP2(hdc, R2_NOTXORPEN))
            errhandler("SetROP2 Failed", hwnd);

        // If previous mouse movement occurred, store the original
        // lower right corner coordinates in a temporary structure.

        if (ptLowerRight.x)
        {
            ptTmp.x = ptLowerRight.x;
            ptTmp.y = ptLowerRight.y;
        }

        // Get the new coordinates of the clipping region's lower
        // right corner.

        ptsTmp = MAKEPOINTS(lParam);
        ptLowerRight.x = (LONG) ptsTmp.x;
        ptLowerRight.y = (LONG) ptsTmp.y;

        // If previous mouse movement occurred, erase the original
        // rectangle.

        if (ptTmp.x)
        {
            aptRect[0].x = ptUpperLeft.x;
            aptRect[0].y = ptUpperLeft.y;
            aptRect[1].x = ptTmp.x;
            aptRect[1].y = ptUpperLeft.y;
            aptRect[2].x = ptTmp.x;
            aptRect[2].y = ptTmp.y;
            aptRect[3].x = ptUpperLeft.x;
            aptRect[3].y = ptTmp.y;
            aptRect[4].x = aptRect[0].x;
            aptRect[4].y = aptRect[0].y;

            if (!Polyline(hdc, (CONST POINT *) aptRect, 5))
                errhandler("Polyline Failed", hwnd);
        }

        aptRect[0].x = ptUpperLeft.x;
        aptRect[0].y = ptUpperLeft.y;
        aptRect[1].x = ptLowerRight.x;
        aptRect[1].y = ptUpperLeft.y;
        aptRect[2].x = ptLowerRight.x;
        aptRect[2].y = ptLowerRight.y;
        aptRect[3].x = ptUpperLeft.x;
        aptRect[3].y = ptLowerRight.y;
        aptRect[4].x = aptRect[0].x;
        aptRect[4].y = aptRect[0].y;

        if (!Polyline(hdc, (CONST POINT *) aptRect, 5))
             errhandler("Polyline Failed", hwnd);

        ReleaseDC(hwnd, hdc);
    }
    break;
```

Clipping Output

4/20/2022 • 2 minutes to read • Edit Online

After the user clicks Clip from the menu, the application uses the coordinates of the rectangle the user created to define a clipping region. After defining the clipping region and selecting it into the application's device context, the application redraws the bitmapped image. The application performs these tasks, as follows.

```
// These variables are required for clipping.

static POINT ptUpperLeft;
static POINT ptLowerRight;
static POINT aptRect[5];
static POINT ptTmp;
static POINTS ptsTmp;
static BOOL fDefineRegion;
static BOOL fRegionExists;
static HRGN hrgn;
static RECT rctTmp;
int i;

case WM_COMMAND:
    switch (wParam)
    {

    case IDM_CLIP:

    hdc = GetDC(hwnd);

    // Retrieve the application's client rectangle and paint
    // with the default (white) brush.

    GetClientRect(hwnd, &rctTmp);
    FillRect(hdc, &rctTmp, GetStockObject(WHITE_BRUSH));

    // Use the rect coordinates to define a clipping region.

    hrgn = CreateRectRgn(aptRect[0].x, aptRect[0].y,
        aptRect[2].x, aptRect[2].y);
    SelectClipRgn(hdc, hrgn);

    // Transfer (draw) the bitmap into the clipped rectangle.

    BitBlt(hdc,
       0, 0,
       bmp.bmWidth, bmp.bmHeight,
       hdcCompatible,
       0, 0,
       SRCCOPY);

    ReleaseDC(hwnd, hdc);
    break;
    }
```

Using Regions to Perform Hit Testing

4/20/2022 • 2 minutes to read • Edit Online

The example in Brushes uses regions to simulate a "zoomed" view of an 8- by 8-pixel monochrome bitmap. By clicking on the pixels in this bitmap, the user creates a custom brush suitable for drawing operations. The example shows how to use the **PtInRegion** function to perform hit testing and the **InvertRgn** function to invert the colors in a region.

Region Reference

4/20/2022 • 2 minutes to read • Edit Online

The following elements are used with regions.

- Region Functions
- Region Structures

Region Functions (Windows GDI)

4/20/2022 • 2 minutes to read • Edit Online

The following functions are used with regions.

FUNCTION	DESCRIPTION
CombineRgn	Combines two regions and stores the result in a third region.
CreateEllipticRgn	Creates an elliptical region.
CreateEllipticRgnIndirect	Creates an elliptical region.
CreatePolygonRgn	Creates a polygonal region.
CreatePolyPolygonRgn	Creates a region consisting of a series of polygons.
CreateRectRgn	Creates a rectangular region.
CreateRectRgnIndirect	Creates a rectangular region.
CreateRoundRectRgn	Creates a rectangular region with rounded corners.
EqualRgn	Checks the two specified regions to determine whether they are identical.
ExtCreateRegion	Creates a region from the specified region and transformation data.
FillRgn	Fills a region by using the specified brush.
FrameRgn	Draws a border around the specified region by using the specified brush.
GetPolyFillMode	Retrieves the current polygon fill mode.
GetRegionData	Fills the specified buffer with data describing a region.
GetRgnBox	Retrieves the bounding rectangle of the specified region.
InvertRgn	Inverts the colors in the specified region.
OffsetRgn	Moves a region by the specified offsets.
PaintRgn	Paints the specified region by using the brush currently selected into the device context.
PtInRegion	Determines whether the specified point is inside the specified region.

FUNCTION	DESCRIPTION
RectInRegion	Determines whether any part of the specified rectangle is within the boundaries of a region.
SetPolyFillMode	Sets the polygon fill mode for functions that fill polygons.
SetRectRgn	Converts a region into a rectangular region with the specified coordinates.

Region Structures

4/20/2022 • 2 minutes to read • Edit Online

The following structures are used with regions:

- RGNDATA
- RGNDATAHEADER