WINDOWS API
VOLUME II

REFERENCE GUIDE

BORLAND

Windows API Guide

Reference

Volume 2

Version 3.0
for the MS-DOS and PC-DOS
Operating Systems

BORLAND INTERNATIONAL, INC. 1800 GREEN HILLS ROAD
P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001

PRINTED IN THE USA.
R1 10 9 8 7 6 5 4 3

CONTENTS

Chapter 13 Windows DDE protocol definition 205

Appendix A Virtual-key codes 221

Appendix B RC diagnostic messages 225

Index 235

TABLES

INTRODUCTION

This manual gives the Windows-application developer general as well as detailed information about Windows functions, messages, data types, Resource Compiler statements, assembly language macros, and file formats. This manual provides detailed descriptions of each component of the Windows application program interface (API) for readers who already have a basic understanding of Windows programming.

This manual is divided into two volumes. Volume 1 contains reference information describing the Windows functions and messages.

Volume 2 contains reference material for other components of the Windows API. It contains the following nine chapters and five appendixes:

Chapter 7, "Data types and structures," contains a table of data types and an alphabetical list of structures found in Windows.

Chapter 8, "Resource script statements," describes the statements that define resources which the Resource Compiler adds to an application's executable file. The statements are arranged according to functional groups.

Chapter 9, "File formats," describes the formats of five types of files: bitmap files, icon resource files, cursor resource files, clipboard files, and metafiles. Each description gives the general file structure and information about specific parts of the file.

Chapter 10, "Module-definition statements," describes the statements contained in the module-definition file that defines the application's contents and system requirements for the LINK program.

Chapter 11, "Binary and ternary raster-operation codes," describes the raster operations used for line output and those used for bitmap output.

Chapter 12, "Printer escapes," lists the printer escapes that are available in Windows.

Chapter 13, "Windows DDE protocol definition," contains an alphabetical listing and description of the Windows messages that comprise the Windows Dynamic Data Exchange protocol.

Appendix A, "Virtual-key codes," lists the symbolic names and hexadecimal values of Windows virtual-key codes and includes a brief description of each key.

Appendix B, "RC Diagnostic messages," contains a listing of Resource Compiler error messages and provides a brief description of each message.

Document conventions

Throughout this manual, the term "DOS" refers to both MS-DOS® and PC-DOS, except when noting features that are unique to one or the other.

The following document conventions are used throughout this manual:

Convention	**Description**
Bold text	Bold letters indicate a specific term or punctuation mark intended to be used literally: language key words or functions (such as **EXETYPE** or **CreateWindow**), DOS commands, and command-line options (such as **/Zi**). You must type these terms and punctuation marks exactly as shown. However, the use of uppercase or lowercase letters is not always significant. For instance, you can invoke the linker by typing either **LINK**, **link**, or **Link** at the DOS prompt.
( )	In syntax statements, parentheses enclose one or more parameters that you pass to a function.
Italic text	Words in italics indicate a placeholder; you are expected to provide the actual value. For example, the following syntax for the **SetCursorPos** function indicates that you must substitute values for the *X* and *Y* coordinates, separated by a comma: **SetCursorPos(*X*, *Y*)**
`Monospaced type`	Code examples are displayed in a nonproportional typeface.
⋮	Vertical ellipses in program examples indicate that a portion of the program is omitted.
. . .	Ellipses following an item indicate that more items having the same form may appear. In the following example, the

horizontal ellipses indicate that you can specify more than one *breakaddress* for the **g** command:

g [[=*startaddress*]] [[*breakaddress*]]...

[[ ]]

Double brackets enclose optional fields or parameters in command lines and syntax statements. In the following example, *option* and *executable-file* are optional parameters of the **RC** command:

RC [[*option*]] *filename* [[*executable-file*]]

|

A vertical bar indicates that you may enter one of the entries shown on either side of the bar. The following command-line syntax illustrates the use of a vertical bar:

DB [[*address* | *range*]]

The bar indicates that following the **DB** (dump bytes) command, you can specify either an *address* or a *range*.

{ }

Curly braces indicate that you must specify one of the enclosed items.

SMALL CAPITAL LETTERS

Small capital letters indicate the names of keys and key sequences, such as:

ALT + SPACEBAR

3.0

The Microsoft Windows version number indicates that a function, message, or data structure is compatible only with the specified version and later versions.

PART 3

General reference

Part 3 provides general reference information on components of the Windows application programming interface that are in addition to the functions and messages described in the preceding parts.

CHAPTER

7

Data types and structures

This chapter describes the data types and structures used by Microsoft Windows functions and messages. It contains two parts: a table of data types and a list of Windows data structures, each arranged alphabetically.

Data types

The data types in the following list are key words that define the size and meaning of parameters and return values associated with Windows functions. This list contains character, integer, and Boolean types, pointer types, and handles. The character, integer, and Boolean types are common to most C compilers. Most of the pointer-type names begin with either a P prefix (for short pointers) or an LP prefix (for long pointers). A short pointer accesses data within the current data segment; a long pointer contains a 32-bit segment/offset value. A Windows application uses a handle to refer to a resource that has been loaded into memory. Windows provides access to these resources through internally maintained tables that contain individual entries for each handle. Each entry in the handle table contains the address of the resource and a means of identifying the resource type. The Windows data types are defined in the following list:

Data type	Description
BOOL	16-bit Boolean value.
BYTE	Unsigned 8-bit integer.
char	ASCII character or a signed 8-bit integer.

DWORD	Unsigned 32-bit integer or a segment/offset address.
FAR	Data-type attribute that can be used to create a long pointer.
FARPROC	Long pointer to a function obtained by calling the **MakeProcInstance** function.
GLOBALHANDLE	Handle to global memory. It is a 16-bit index to a block of memory allocated from the system's global heap.
HANDLE	General handle. It represents a 16-bit index to a table entry that identifies program data.
HBITMAP	Handle to a physical bitmap. It is a 16-bit index to GDI's physical drawing objects.
HBRUSH	Handle to a physical brush. It is a 16-bit index to GDI's physical drawing objects.
HCURSOR	Handle to a cursor resource. It is a 16-bit index to a resource-table entry.
HDC	Handle to a display context. It is a 16-bit index to GDI's device-context tables.
HFONT	Handle to a physical font. It is a 16-bit index to GDI's physical drawing objects.
HICON	Handle to an icon resource. It is a 16-bit index to a resource-table entry.
HMENU	Handle to a menu resource. It is a 16-bit index to a resource-table entry.
HPALETTE	Handle to a logical palette. It is a 16-bit index to GDI's physical drawing objects.
HPEN	Handle to a physical pen. It is a 16-bit index to GDI's physical drawing objects.
HRGN	Handle to a physical region. It is a 16-bit index to GDI's physical drawing objects.
HSTR	Handle to a string resource. It is a 16-bit index to a resource-table entry.
int	Signed 16-bit integer.
LOCALHANDLE	Handle to local memory. It is a 16-bit index to a block of memory allocated from the application's local heap.
long	Signed 32-bit integer.
LONG	Signed 32-bit integer.
LPBITMAP	Long pointer to a **BITMAP** data structure.
LPBITMAPCOREHEADER	Long pointer to a **BITMAPCOREHEADER** data structure.
LPBITMAPCOREINFO	Long pointer to a **BITMAPCOREINFO** data structure.
LPBITMAPFILEHEADER	Long pointer to a **BITMAPFILEHEADER** data structure.
LPBITMAPINFO	Long pointer to a **BITMAPINFO** data structure.
LPBITMAPINFOHEADER	Long pointer to a **BITMAPINFOHEADER** data structure.
LPCOMPAREITEMSTRUCT	Long pointer to a **COMPAREITEMSTRUCT** data structure.
LPCREATESTRUCT	Long pointer to a **CREATESTRUCT** data structure.

LPDELETEITEMSTRUCT	Long pointer to a **DELETEITEMSTRUCT** data structure.
LPDRAWITEMSTRUCT	Long pointer to a **DRAWITEMSTRUCT** data structure.
LPHANDLETABLE	Long pointer to a **HANDLETABLE** data structure.
LPINT	Long pointer to a signed 16-bit integer.
LPLOGBRUSH	Long pointer to a **LOGBRUSH** data structure.
LPLOGFONT	Long pointer to a **LOGFONT** data structure.
LPLOGPALETTE	Long pointer to a **LOGPALETTE** data structure.
LPLOGPEN	Long pointer to a **LOGPEN** data structure.
LPMEASUREITEMSTRUCT	Long pointer to a **MEASUREITEMSTRUCT** data structure.
LPMETAFILEPICT	Long pointer to a **METAFILEPICT** data structure.
LPMSG	Long pointer to a **MSG** data structure.
LPOFSTRUCT	Long pointer to an **OFSTRUCT** data structure.
LPPAINTSTRUCT	Long pointer to a **PAINTSTRUCT** data structure.
LPPALETTEENTRY	Long pointer to a **PALETTEENTRY** data structure.
LPPOINT	Long pointer to a **POINT** data structure.
LPRECT	Long pointer to a **RECT** data structure.
LPRESOURCELIST	Long pointer to one or more **RESOURCESTRUCT** data structures.
LPSTR	Long pointer to a character string.
LPTEXTMETRIC	Long pointer to a **TEXTMETRIC** data structure.
LPVOID	Long pointer to an undefined data type.
LPWNDCLASS	Long pointer to a **WNDCLASS** data structure.
NEAR	Data-type attribute that can be used to create a short pointer.
NPSTR	Near pointer to a character string.
PINT	Pointer to a signed 16-bit integer.
PSTR	Pointer to a character string.
PWORD	Pointer to an unsigned 16-bit integer.
short	Signed 16-bit integer.
void	Empty value. It is used with a function to specify no return value.
WORD	Unsigned 16-bit integer.

Data structures

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

BITMAP

Bitmap data structure

The **BITMAP** structure defines the height, width, color format, and bit values of a logical bitmap.

```
typedef struct tagBITMAP {
   short      bmType;
   short      bmWidth;
   short      bmHeight;
   short      bmWidthBytes;
   BYTE       bmPlanes;
   BYTE       bmBitsPixel;
   LPSTR      bmBits;
} BITMAP;
```

```
TBitmap = record
  bmType: Integer;
  bmWidth: Integer;
  bmHeight: Integer;
  bmWidthBytes: Integer;
  bmPlanes: Byte;
  bmBitsPixel: Byte;
  bmBits: Pointer;
end;
```

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

Field	Description
bmType	Specifies the bitmap type. For logical bitmaps, the **bmType** field must be zero.
bmWidth	Specifies the width of the bitmap (in pixels). The width must be greater than zero.
bmHeight	Specifies the height of the bitmap (in raster lines). The height must be greater than zero.
bmWidthBytes	Specifies the number of bytes in each raster line. This value must be an even number since the graphics device interface (GDI) assumes that the bit values of a bitmap form an array of integer (two-byte) values. In other words, **bmWidthBytes** 8 must be the next multiple of 16 greater than or equal to the **bmWidth** field.
bmPlanes	Points to the number of color planes in the bitmap.
bmBitsPixel	Points to the number of adjacent color bits on each plane needed to define a pixel.
bmBits	Points to the location of the bit values for the bitmap. The **bmBits** field must be a long pointer to an array of character (one-byte) values.

Comments

The currently used bitmap formats are monochrome and color. The monochrome bitmap uses a one-bit, one-plane format. Each scan is a multiple of 16 bits.

Scans are organized as follows for a monochrome bitmap of height *n*:

```
Scan 0
Scan 1
⋮
```

```
Scan n-2
Scan n-1
```

The pixels on a monochrome device are either black or white. If the corresponding bit in the bitmap is 1, the pixel is turned on (white); if the corresponding bit in the bitmap is zero, the pixel is turned off (black).

All devices that have the RC_BITBLT bit set in the device capabilities support bitmaps.

Each device has its own unique color format. In order to transfer a bitmap from one device to another, use **GetDIBits** and **SetDIBits**.

See also The **CreateBitmapIndirect** and **GetObject** functions in Chapter 4, "Functions directory," in *Reference, Volume 1.*

BITMAPCOREHEADER 3.0

Device-independent bitmap format information

The **BITMAPCOREHEADER** structure contains information about the dimensions and color format of a device-independent bitmap that is compatible with Microsoft OS/2 Presentation Manager versions 1.1 and 1.2 bitmaps.

```
typedef struct tagBITMAPCOREHEADER {
        DWORD   bcSize;
        WORD    bcWidth;
        WORD    bcHeight;
        WORD    bcPlanes;
        WORD    bcBitCount;
} BITMAPCOREHEADER;
```

```
TBitmapCoreHeader = record
  bcSize: Longint;{ used to get to
                    color table }
  bcWidth: Word;
  bcHeight: Word;
  bcPlanes: Word;
  bcBitCount: Word;
end;
```

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

Field	Description
bcSize	Specifies the number of bytes required by the **BITMAPCOREHEADER** structure.
bcWidth	Specifies the width of the bitmap in pixels.
bcHeight	Specifies the height of the bitmap in pixels.
bcPlanes	Specifies the number of planes for the target device and must be set to 1.
bcBitCount	Specifies the number of bits per pixel. This value must be 1, 4, 8, or 24.

Comments The **BITMAPCOREINFO** data structure combines the **BITMAPCOREHEADER** structure and a color table to provide a complete definition of the dimensions and colors of a device-independent bitmap. See the description of the **BITMAPCOREINFO** data structure for more information about specifying a device-independent bitmap.

An application should use the information stored in the **bcSize** field to locate the color table in a **BITMAPCOREINFO** data structure with a method such as the following:

```
pColor = ((LPSTR) pBitmapCoreInfo + (WORD) (pBitmapCoreInfo
->> bcSize))
```

BITMAPCOREINFO 3.0

Device-independent bitmap information

The **BITMAPCOREINFO** structure fully defines the dimensions and color information for a device-independent bitmap that is compatible with Microsoft OS/2 Presentation Manager versions 1.1 and 1.2 bitmaps.

```
typedef struct _BITMAPCOREINFO {
        BITMAPCOREHEADER  bmciHeader;
        RGBTRIPLE
bmciColors[];
} BITMAPCOREINFO;
```

```
TBitmapCoreInfo = record
  bmciHeader: TBitmapCoreHeader;
  bmciColors: array[0..0] of
              TRGBTriple;
end;
```

The **BITMAPCOREINFO** structure contains the following fields:

Field	Description
bmciHeader	Specifies a **BITMAPCOREHEADER** data structure that contains information about the dimensions and color format of a device-independent bitmap.
bmciColors	Specifies an array of **RGBTRIPLE** data structures that define the colors in the bitmap.

Comments An OS/2 Presentation Manager device-independent bitmap consists of two distinct parts: a **BITMAPCOREINFO** data structure that describes the dimensions and colors of the bitmap, and an array of bytes which define the pixels of the bitmap. The bits in the array are packed together, but each scan line must be zero-padded to end on a **LONG** boundary. Segment boundaries can appear anywhere in the bitmap, however. The origin of the bitmap is the lower-left corner.

The **bcBitCount** field of the **BITMAPCOREHEADER** structure determines the number of bits which define each pixel and the maximum number of colors in the bitmap. This field may be set to any of the following values:

Value	Description
1	The bitmap is monochrome, and the **bmciColors** field must contain two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the **bmciColors** table; if the bit is set, the pixel has the color of the second entry in the table.
4	The bitmap has a maximum of 16 colors, and the **bmciColors** field contains 16 entries. Each pixel in the bitmap is represented by a four-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, then the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the 16th table entry.
8	The bitmap has a maximum of 256 colors, and the **bmciColors** field contains 256 entries. In this case, each byte in the array represents a single pixel.
24	The bitmap has a maximum of 2^{24} colors. The **bmciColors** field is NULL, and each three bytes in the bitmap array represents the relative intensities of red, green, and blue, respectively, of a pixel.

The colors in the **bmciColors** table should appear in order of importance.

Alternatively, for functions that use device-independent bitmaps, the **bmciColors** field can be an array of 16-bit unsigned integers that specify an index into the currently realized logical palette instead of explicit RGB values. In this case, an application using the bitmap must call device-independent bitmap functions with the *wUsage* parameter set to DIB_PAL_COLORS.

The **bmciColors** field should not contain palette indexes if the bitmap is to be stored in a file or transferred to another application. Unless the application uses the bitmap exclusively and under its complete control, the bitmap color table should contain explicit RGB values.

BITMAPFILEHEADER 3.0

Bitmap file information

The **BITMAPFILEHEADER** data structure contains information about the type, size, and layout of a device-independent bitmap (DIB) file.

```
typedef struct tagBITMAPFILEHEADER {
        WORD bfType;
        DWORD     bfSize;
        WORD     bfReserved1;
        WORD     bfReserved2;
        DWORD    bfOffBits;
} BITMAPFILEHEADER;
```

```
TBitmapFileHeader = record
  bfType: Word;
  bfSize: Longint;
  bfReserved1: Word;
  bfReserved2: Word;
  bfOffBits: Longint;
end;
```

The **BITMAPFILEHEADER** data structure contains the following fields:

Field	Description
bfType	Specifies the type of file. It must be BM.
bfSize	Specifies the size in **DWORD**s of the file.
bfReserved1	Is reserved and must be set to zero.
bfReserved2	Is reserved and must be set to zero.
bfOffBits	Specifies in bytes the offset from the **BITMAPFILEHEADER** of the actual bitmap in the file.

Comments A **BITMAPINFO** or **BITMAPCOREINFO** data structure immediately follows the **BITMAPFILEHEADER** structure in the DIB file.

BITMAPINFO 3.0

Device-independent bitmap information

The **BITMAPINFO** structure fully defines the dimensions and color information for a Windows 3.0 device-independent bitmap.

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

```
TBitmapInfo = record
  bmiHeader: TBitmapInfoHeader;
  bmiColors: array[0..0] of TRGBQuad;
end;
```

The **BITMAPINFO** structure contains the following fields:

Field	Description
bmiHeader	Specifies a **BITMAPINFOHEADER** data structure that contains information about the dimensions and color format of a device-independent bitmap.
bmiColors	Specifies an array of **RGBQUAD** data structures that define the colors in the bitmap.

Comments A Windows 3.0 device-independent bitmap consists of two distinct parts: a **BITMAPINFO** data structure that describes the dimensions and colors of the bitmap, and an array of bytes that define the pixels of the bitmap. The bits in the array are packed together, but each scan line must be zero-padded to end on a **LONG** boundary. Segment boundaries can appear anywhere in the bitmap, however. The origin of the bitmap is the lower-left corner.

The **biBitCount** field of the **BITMAPINFOHEADER** structure determines the number of bits which define each pixel and the maximum number of colors in the bitmap. This field may be set to any of the following values:

Value	Description
1	The bitmap is monochrome, and the **bmiColors** field must contain two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the **bmiColors** table; if the bit is set, the pixel has the color of the second entry in the table.
4	The bitmap has a maximum of 16 colors, and the **bmiColors** field contains up to 16 entries. Each pixel in the bitmap is represented by a four-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, then the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the 16th table entry.
8	The bitmap has a maximum of 256 colors, and the **bmiColors** field contains up to 256 entries. In this case, each byte in the array represents a single pixel.
24	The bitmap has a maximum of 2^{24} colors. The **bmiColors** field is NULL, and each three bytes in the bitmap array represents the relative intensities of red, green, and blue, respectively, of a pixel.

The **biClrUsed** field of the **BITMAPINFOHEADER** structure specifies the number of color indexes in the color table actually used by the bitmap. If the **biClrUsed** field is set to 0, the bitmap uses the maximum number of colors corresponding to the value of the **biBitCount** field.

The colors in the **bmiColors** table should appear in order of importance.

Alternatively, for functions that use device-independent bitmaps, the **bmiColors** field can be an array of 16-bit unsigned integers that specify an index into the currently realized logical palette instead of explicit RGB values. In this case, an application using the bitmap must call device-independent bitmap functions with the *wUsage* parameter set to DIB_PAL_COLORS.

The **bmiColors** field should not contain palette indices if the bitmap is to be stored in a file or transferred to another application. Unless the application uses the bitmap exclusively and under its complete control, the bitmap color table should contain explicit RGB values.

BITMAPINFOHEADER 3.0

Device-independent bitmap format information

The **BITMAPINFOHEADER** structure contains information about the dimensions and color format of a Windows 3.0 device-independent bitmap.

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

```
TBitmapInfoHeader = record
  biSize: Longint;
  biWidth: Longint;
  biHeight: Longint;
  biPlanes: Word;
  biBitCount: Word;
  biCompression: Longint;
  biSizeImage: Longint;
  biXPelsPerMeter: Longint;
  biYPelsPerMeter: Longint;
  biClrUsed: Longint;
  biClrImportant: Longint;
end;
```

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

Field	Description
biSize	Specifies the number of bytes required by the **BITMAPINFOHEADER** structure.
biWidth	Specifies the width of the bitmap in pixels.
biHeight	Specifies the height of the bitmap in pixels.
biPlanes	Specifies the number of planes for the target device and must be set to 1.
biBitCount	Specifies the number of bits per pixel. This value must be 1, 4, 8, or 24.

biCompression Specifies the type of compression for a compressed bitmap. It can be one of the following values:.

Value	**Description**
BI_RGB	Specifies that the bitmap is not compressed.
BI_RLE8	Specifies a run-length encoded format for bitmaps with 8 bits per pixel. The compression format is a two-byte format consisting of a count byte followed by a byte containing a color index. See the following "Comments" section for more information.
BI_RLE4	Specifies a run-length encoded format for bitmaps with 4 bits per pixel. The compression format is a two-byte format consisting of a count byte followed by two word-length color indexes. See the following "Comments" section for more information.

biSizeImage Specifies the size in bytes of the image.

biXPelsPerMeter Specifies the horizontal resolution in pixels per meter of the target device for the bitmap. An application can use this value to select a bitmap from a resource group that best matches the characteristics of the current device.

biYPelsPerMeter Specifies the vertical resolution in pixels per meter of the target device for the bitmap.

biClrUsed Specifies the number of color indexes in the color table actually used by the bitmap. If this value is 0, the bitmap uses the maximum number of colors corresponding to the value of the **biBitCount** field. See the description of the **BITMAPINFO** data structure earlier in this chapter for more information on the maximum sizes of the color table. If **biClrUsed** is nonzero, then the **biClrUsed** field specifies the actual number of colors which the graphics engine or device driver will access if the **biBitCount** field is less than 24. If the **biBitCount** field is set to 24, the **biClrUsed** field specifies the size of the reference color table used to optimize performance of Windows color palettes. If the bitmap is a "packed" bitmap (that is, a bitmap in which the bitmap array immediately follows the **BITMAPFINO** header and which is referenced by a single pointer), the **biClrUsed** field must be set to 0 or to the actual size of the color table.

biClrImportant Specifies the number of color indexes that are considered important for displaying the bitmap. If this value is 0, then all colors are important.

Comments

The **BITMAPINFO** data structure combines the **BITMAPINFOHEADER** structure and a color table to provide a complete definition of the dimensions and colors of a Windows 3.0 device-independent bitmap. See the description of the **BITMAPINFO** data structure for more information about specifying a Windows 3.0 device-independent bitmap.

An application should use the information stored in the **biSize** field to locate the color table in a **BITMAPINFO** data structure with a method such as the following:

```
pColor = ((LPSTR) pBitmapInfo + (WORD) (pBitmapInfo ->> biSize))
```

Bitmap compression formats

Windows supports formats for compressing bitmaps that define their colors with 8 bits per pixel and with 4 bits per pixel. Compression reduces the disk and memory storage required for the bitmap. The following paragraphs describe these formats.

When the **biCompression** field is set to BI_RLE8, the bitmap is compressed using a run-length encoding format for an 8-bit bitmap. This format may be compressed in either of two modes:

- Encoded
- Absolute

Both modes can occur anywhere throughout a single bitmap.

Encoded mode consists of two bytes: the first byte specifies the number of consecutive pixels to be drawn using the color index contained in the second byte. In addition, the first byte of the pair can be set to zero to indicate an escape that denotes an end of line, end of bitmap, or a delta. The interpretation of the escape depends on the value of the second byte of the pair. The following list shows the meaning of the second byte:

Second Byte Of Escape	Meaning
0	End of line.
1	End of bitmap.
2	Delta. The two bytes following the escape contain unsigned values indicating the horizontal and vertical offset of the next pixel from the current position.

Absolute mode is signalled by the first byte set to zero and the second byte set to a value between 03H and FFH. In absolute mode, the second byte represents the number of bytes which follow, each of which contains the color index of a single pixel. When the second byte is set to 2 or less, the escape has the same meaning as in encoded mode. In absolute mode, each run must be aligned on a 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
```

This bitmap would expand as follows (two-digit values represent a color index for a single pixel):

```
04 04 04
06 06 06 06 06
45 56 67
78 78
move current position 5 right and 1 down
78 78
end of line
1E 1E 1E 1E 1E 1E 1E 1E 1E
end of RLE bitmap
```

When the **biCompression** field is set to BI_RLE4, the bitmap is compressed using a run-length encoding format for a 4-bit bitmap, which also uses encoded and absolute modes. In encoded mode, the first byte of the pair contains the number of pixels to be drawn using the color indexes in the second byte. The second byte contains two color indexes, one in its high-order nibble (that is, its low-order four bits) and one in its low-order nibble. The first of the pixels is drawn using the color specified by the high-order nibble, the second is drawn using the color in the low-order nibble, the third is drawn with the color in the high-order nibble, and so on, until all the pixels specified by the first byte have been drawn.

In absolute mode, the first byte contains zero, the second byte contains the number of color indexes that follow, and subsequent bytes contain color indexes in their high- and low-order nibbles, one color index for each pixel. In absolute mode, each run must be aligned on a word boundary. The end-of-line, end-of-bitmap, and delta escapes also apply to BI_RLE4.

The following example shows the hexadecimal values of a 4-bit compressed bitmap:

```
03 04 05 06 00 06 45 56 67 00 04 78 00 02 05 01
04 78 00 00 09 1E 00 01
```

This bitmap would expand as follows (single-digit values represent a color index for a single pixel):

```
0 4 0
0 6 0 6 0
4 5 5 6 6 7
7 8 7 8
move current position 5 right and 1 down
7 8 7 8
end of line
1 E 1 E 1 E 1 E 1
end of RLE bitmap
```

CLIENTCREATESTRUCT 3.0

MDI client window creation structure

The **CLIENTCREATESTRUCT** data structure contains information about the menu and first multiple document interface (MDI) child window of an MDI client window. An application passes a long pointer to this structure as the *lpParam* parameter of the **CreateWindow** function when creating an MDI client window.

```
typedef struct tagCLIENTCREATESTRUCT
        {
          HMENU   hWindowMenu;
          WORD    idFirstChild;
        } CLIENTCREATESTRUCT;
```

```
TClientCreateStruct = record
  hWindowMenu: THandle;
  idFirstChild: Word;
end;
```

The **CLIENTCREATESTRUCT** structure contains the following fields:

Field	Description
hWindowMenu	Is the menu handle of the application's Window menu. An application can retrieve this handle from the MDI frame window's menu using the **GetSubMenu** function.
idFirstChild	Is the child window ID of the first MDI child window created. Windows increments the ID for each additional MDI child window that the application creates, and reassigns identifiers when the application destroys a window to keep the range of identifiers continuous. These identifiers are used in WM_COMMAND messages to the application's MDI frame window when a child window is selected from the Window menu, and should not conflict with any other command identifiers.

COLORREF

Color specification

A **COLORREF** color value is a long integer that specifies a color. GDI functions that require a color (such as **CreatePen** and **FloodFill**) accept a **COLORREF** value as a parameter. Depending on how an application uses the **COLORREF** value, the value has three distinct forms. It may specify any of the following:

- Explicit values for red, green, and blue (RGB)
- An index into a logical color palette
- A palette-relative RGB value

```
TColorRef = Longint;
```

Explicit RGB

When specifying an explicit RGB value, 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 FF (hexadecimal). The following list illustrates the hexadecimal values that produce the indicated colors.

Value	Color
0x000000FF	Pure red
0x0000FF00	Pure green
0x00FF0000	Pure blue
0x00000000	Black
0x00FFFFFF	White
0x00808080	Medium gray

The **RGB** macro accepts values for red, green, and blue, and returns an explicit RGB **COLORREF** value.

Palette index

When specifying an index into a logical color palette, the **COLORREF** value has the following hexadecimal form:

```
0x0100iiii
```

The two low-order bytes consist of a 16-bit integer specifying an index into a logical palette. The third byte is not used and must be zero. The fourth (high-order) byte must be set to 1.

For example, the hexadecimal value 0x01000000 specifies the color in the palette entry of index 0; 0x0100000C specifies the color in the entry of index 12, and so on.

The **PALETTEINDEX** macro accepts an integer representing an index into a logical palette and returns a palette-index **COLORREF** value.

Palette-relative rgb

When specifying a palette-relative RGB value, the **COLORREF** value has the following hexadecimal form:

```
0x02bbggrr
```

As with an explicit RGB, the three low-order bytes contain values for red, green, and blue; the high-order byte must be set to 2.

For output devices that support logical palettes, Windows matches a palette-relative RGB value to the nearest color in the logical palette of the device context, as though the application had specified an index to that palette entry. If an output device does not support a system palette, then Windows uses the palette-relative RGB as though it were an explict RGB **COLORREF** value.

The **PALETTERGB** macro accepts values for red, green, and blue, and returns a palette-relative RGB **COLORREF** value.

Comments Before passing a palette-index or palette-relative RGB **COLORREF** value to a function that also requires a device-context parameter, an application that uses its own palette must select its palette into the device context (by calling the **SelectPalette** function) and realize the palette (by calling **RealizePalette**). This ensures that the function will use the correct palette-entry color. For functions that create an object (such as **CreatePen**), the application must select and realize the palette before selecting the object for the device context.

COMPAREITEMSTRUCT 3.0

Owner-draw item-sorting information

The **COMPAREITEMSTRUCT** structure supplies the identifiers and application-supplied data for two items in a sorted owner-draw combo box or list box.

Whenever an application adds a new item to an owner-draw combo or list box created with the CBS_SORT or LBS_SORT style, Windows sends the owner a WM_COMPAREITEM message. The *lParam* parameter of the message contains a long pointer to a **COMPAREITEMSTRUCT** data structure. When the owner receives the message, the owner compares the two items and returns a value indicating which item sorts before the other. For more information, see the description of the WM_COMPAREITEM message in Chapter 6, "Messages directory," in *Reference, Volume 1.*

```
typedef struct tagCOMPAREITEMSTRUCT {
   WORD   CtlType;
   WORD   CtlID;
   HWND   hwndItem;
   WORD   itemID1;
   DWORD  itemData1;
   WORD   itemID2;
   DWORD  itemData2;
} COMPAREITEMSTRUCT;
```

```
TCompareItemStruct = record
  CtlType: Word;
  CtlID: Word;
  hwndItem: HWnd;
  itemID1: Word;
  itemData1: Longint;
  itemID2: Word;
  itemData2: Longint;
end;
```

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

Field	Description
CtlType	Is ODT_LISTBOX (which specifies an owner-draw list box) or ODT_COMBOBOX (which specifies an owner-draw combo box).
CtlID	Is the control ID for the list box or combo box.
hwndItem	Is the window handle of the control.
itemID1	Is the index of the first item in the list box or combo box being compared.
itemData1	Is application-supplied data for the first item being compared. This value was passed as the *lParam* parameter of the message that added the item to the combo or list box.
itemID2	Is the index of the second item in the list box or combo box being compared.
itemData2	Is application-supplied data for the second item being compared. This value was passed as the *lParam* parameter of the message that added the item to the combo or list box.

COMSTAT

Communication device status

The **COMSTAT** structure contains information about a communications device.

```
typedef struct tagCOMSTAT {
   BYTE fCtsHold: 1;
   BYTE fDsrHold: 1;
   BYTE fRlsdHold: 1;
   BYTE fXoffHold: 1;
   BYTE fXoffSent: 1;
   BYTE fEof: 1;
   BYTE fTxim: 1;
   WORD cbInQue;
   WORD cbOutQue;
} COMSTAT;
```

```
TComStat = record
  Flags: Byte;
  cbInQue: Word;
  cbOutQue: Word;
end;
```

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

Field	Description
fCtsHold: 1	Specifies whether transmission is waiting for the clear-to-send (CTS) signal to be sent.
fDsrHold: 1	Specifies whether transmission is waiting for the data-set-ready (DSR) signal to be sent.
fRlsdHold: 1	Specifies whether transmission is waiting for the receive-line-signal-detect (RLSD) signal to be sent.

fXoffHold: 1	Specifies whether transmission is waiting as a result of the **XoffChar** character being received.
fXoffSent: 1	Specifies whether transmission is waiting as a result of the **XoffChar** character being transmitted. Transmission halts when the **XoffChar** character is transmitted and used by systems that take the next character as XON, regardless of the actual character.
fEof: 1	Specifies whether the **EofChar** character has been received.
fTxim: 1	Specifies whether a character is waiting to be transmitted.
cbInQue	Specifies the number of characters in the receive queue.
cbOutQue	Specifies the number of characters in the transmit queue.

See also The **GetCommError** function in Chapter 4, "Functions directory," in *Reference, Volume 1.*

CREATESTRUCT

Window-creation structure

The **CREATESTRUCT** structure defines the initialization parameters passed to an application's window function.

```
typedef struct tagCREATESTRUCT {     TCreateStruct = record
   LPSTR   lpCreateParams;             lpCreateParams: PChar;
   HANDLE  hInstance;                  hInstance: THandle;
   HANDLE  hMenu;                      hMenu: THandle;
   HWND    hwndParent;                 hwndParent: HWnd;
   int     cy;                         cy: Integer;
   int     cx;                         cx: Integer;
   int     y;                          y: Integer;
   int     x;                          x: Integer;
   long    style;                      style: LongInt;
   LPSTR   lpszName;                   lpszName: PChar;
   LPSTR   lpszClass;                  lpszClass: PChar;
   long    ExStyle;                    dwExStyle: Longint;
} CREATESTRUCT;                      end;
```

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

Field	Description
lpCreateParams	Points to data to be used for creating the window.
hInstance	Identifies the module-instance handle of the module that owns the new window.
hMenu	Identifies the menu to be used by the new window.
hwndParent	Identifies the window that owns the new window. This field is NULL if the new window is a top-level window.
cy	Specifies the height of the new window.

cx Specifies the width of the new window.

y Specifies the *y*-coordinate of the upper-left corner of the new window. Coordinates are relative to the parent window if the new window is a child window. Otherwise, the coordinates are relative to the screen origin.

x Specifies the *x*-coordinate of the upper-left corner of the new window. Coordinates are relative to the parent window if the new window is a child window. Otherwise, the coordinates are relative to the screen origin.

style Specifies the new window's style.

lpszName Points to a null-terminated character string that specifies the new window's name.

lpszClass Points to a null-terminated character string that specifies the new window's class name.

ExStyle Specifies extended style for the new window.

DCB

Communications device control block

The **DCB** structure defines the control setting for a serial communications device.

```
typedef struct tagDCB {             TDCB = record
   BYTE Id;                           Id: Byte;
   WORD BaudRate;                     BaudRate: Word;
   BYTE ByteSize;                     ByteSize: Byte;
   BYTE Parity;                       Parity: Byte;
   BYTE StopBits;                     StopBits: Byte;
   WORD RlsTimeout;                   RlsTimeout: Word;
   WORD CtsTimeout;                   CtsTimeout: Word;
   WORD DsrTimeout;                   DsrTimeout: Word;
                                      Flags: Word;
   BYTE fBinary: 1;                   XonChar: Char;
   BYTE fRtsDisable: 1;               XoffChar: Char;
   BYTE fParity: 1;                   XonLim: Word;
   BYTE fOutxCtsFlow: 1;              XoffLim: Word;
   BYTE fOutxDsrFlow: 1;              PeChar: Char;
   BYTE fDummy: 2;                    EofChar: Char;
   BYTE fDtrDisable: 1;               EvtChar: Char;
                                      TxDelay: Word;
   BYTE fOutX: 1;                   end;
   BYTE fInX: 1;
   BYTE fPeChar: 1;
   BYTE fNull: 1;
   BYTE fChEvt: 1;
   BYTE fDtrflow: 1;
   BYTE fRtsflow: 1;
```

```
        BYTE fDummy2: 1;

        char XonChar;
        char XoffChar;
        WORD XonLim;
        WORD XoffLim;
        char PeChar;
        char EofChar;
        char EvtChar;
        WORD TxDelay;
    } DCB;
```

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

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><b>Id</b></td><td>Specifies the communication device. This value is set by the device driver. If the most significant bit is set, then the <b>DCB</b> structure is for a parallel device.</td></tr>
<tr><td><b>BaudRate</b></td><td>Specifies the baud rate at which the communications device operates.</td></tr>
<tr><td><b>ByteSize</b></td><td>Specifies the number of bits in the characters transmitted and received. The <b>ByteSize</b> field can be any number from 4 to 8.</td></tr>
<tr><td><b>Parity</b></td><td>Specifies the parity scheme to be used. The <b>Parity</b> field can be any one of the following values:
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>EVENPARITY</td><td>Even</td></tr>
<tr><td>MARKPARITY</td><td>Mark</td></tr>
<tr><td>NOPARITY</td><td>No parity</td></tr>
<tr><td>ODDPARITY</td><td>Odd</td></tr>
<tr><td>SPACEPARITY</td><td>Space</td></tr>
</table></td></tr>
<tr><td><b>StopBits</b></td><td>Specifies the number of stop bits to be used. The <b>StopBits</b> field can be any one of the following values:
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>ONESTOPBIT</td><td>1 stop bit</td></tr>
<tr><td>ONE5STOPBITS</td><td>1.5 stop bits</td></tr>
<tr><td>TWOSTOPBITS</td><td>2 stop bits</td></tr>
</table></td></tr>
<tr><td><b>RlsTimeout</b></td><td>Specifies the maximum amount of time (in milliseconds) the device should wait for the receive-line-signal-detect (RLSD) signal. (RLSD is also known as the carrier detect (CD) signal.)</td></tr>
<tr><td><b>CtsTimeout</b></td><td>Specifies the maximum amount of time (in milliseconds) the device should wait for the clear-to-send (CTS) signal.</td></tr>
<tr><td><b>DsrTimeout</b></td><td>Specifies the maximum amount of time (in milliseconds) the device should wait for the data-set-ready (DSR) signal.</td></tr>
<tr><td><b>fBinary: 1</b></td><td>Specifies binary mode. In nonbinary mode, the <b>EofChar</b> character is recognized on input and remembered as the end of data.</td></tr>
<tr><td><b>fRtsDisable: 1</b></td><td>Specifies whether or not the request-to-send (RTS) signal is disabled. If the <b>fRtsDisable</b> field is set, RTS is not used and</td></tr>
</table>

remains low. If **fRtsDisable** is clear, RTS is sent when the device is opened and turned off when the device is closed.

fParity: 1 Specifies whether parity checking is enabled. If the **fParity** field is set, parity checking is performed and errors are reported.

fOutxCtsFlow: 1 Specifies that clear-to-send (CTS) signal is to be monitored for output flow control. If the **fOutxCtsFlow** field is set and CTS is turned off, output is suspended until CTS is again sent.

fOutxDsrFlow: 1 Specifies that the data-set-ready (DSR) signal is to be monitored for output flow control. If the **fOutxDsrFlow** field is set and DSR is turned off, output is suspended until DSR is again sent.

fDummy: 2 Reserved.

fDtrDisable: 1 Specifies whether the data-terminal-ready (DTR) signal is disabled. If the **fDtrDisable** field is set, DTR is not used and remains low. If **fDtrDisable** is clear, DTR is sent when the device is opened and turned off when the device is closed.

fOutX: 1 Specifies that XON/XOFF flow control is used during transmission. If the **fOutX** field is set, transmission stops when the **XoffChar** character is received, and starts again when the **XonChar** character is received.

fInX: 1 Specifies that XON/XOFF flow control is used during reception. If the **fInX** field is set, the **XonChar** character is sent when the receive queue comes within **XoffLim** characters of being full, and the **XonChar** character is sent when the receive queue comes within **XonLim** characters of being empty.

fPeChar: 1 Specifies that characters received with parity errors are to be replaced with the character specified by the **fPeChar** field. The **fParity** field must be set for the replacement to occur.

fNull: 1 Specifies that received null characters are to be discarded.

fChEvt: 1 Specifies that reception of the **EvtChar** character is to be flagged as an event.

fDtrflow: 1 Specifies that the data-terminal-ready (DTR) signal is to be used for receive flow control. If the **fDtrflow** field is set, DTR is turned off when the receive queue comes within **XoffLim** characters of being full, and sent when the receive queue comes within **XonLim** characters of being empty.

fRtsflow: 1 Specifies that the ready-to-send (RTS) signal is to be used for receive flow control. If the **fRtsflow** field is set, RTS is turned off when the receive queue comes within **XoffLim** characters of being full, and sent when the receive queue comes within **XonLim** characters of being empty.

fdummy2: 1 Reserved.

XonChar Specifies the value of the XON character for both transmission and reception.

XoffChar Specifies the value of the XOFF character for both transmission and reception.

XonLim Specifies the minimum number of characters allowed in the receive queue before the XON character is sent.

XoffLim Specifies the maximum number of characters allowed in the receive queue before the XOFF character is sent. The **XoffLim**

	value is subtracted from the size of the receive queue (in bytes) to calculate the maximum number of characters allowed.
PeChar	Specifies the value of the character used to replace characters received with a parity error.
EofChar	Specifies the value of the character used to signal the end of data.
EvtChar	Specifies the value of the character used to signal an event.
TxDelay	Not currently used.

See also The **BuildCommDCB**, **GetCommState**, and **SetCommState** functions in Chapter 4, "Functions directory," in *Reference, Volume 1.*

DELETEITEMSTRUCT 3.0

Deleted owner-draw list-box item

The **DELETEITEMSTRUCT** structure describes a deleted owner-draw list-box or combo-box item. When an item is removed from the list box or combo box, or when the list box or combo box is destroyed, Windows sends the WM_DELETEITEM message to the owner for each deleted item; the *lParam* parameter of the message contains a pointer to this structure.

```
typedef struct tagDELETEITEMSTRUCT
  {
    WORD       CtlType
    WORD       CtlID;
    WORD       itemID;
    HWND       hwndItem;
    DWORD      itemData;
  } DELETEITEMSTRUCT;
```

```
TDeleteItemStruct = record
  CtlType: Word;
  CtlID: Word;
  itemID: Word;
  hwndItem: HWnd;
  itemData: Longint;
end;
```

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

Field	Description
CtlType	Is ODT_LISTBOX (which specifies an owner-draw list box) or ODT_COMBOBOX (which specifies an owner-draw combo box).
CtlID	Is the control ID for the list box or combo box.
itemID	Is the index of the item in the list box or combo box being removed.
hwndItem	Is the window handle of the control.
itemData	Contains the value passed to the control in the *lParam* parameter of the LB_INSERTSTRING, LB_ADDSTRING, CB_INSERTSTRING, or CB_ADDSTRING message when the item was added to the list box.

DEVMODE 3.0

Printer driver initialization information

The **DEVMODE** data structure contains information about the device initialization and environment of a printer driver. An application passes this structure to the **DeviceCapabilities** and **ExtDeviceMode** functions.

```
typedef struct _devicemode {
   char     dmDeviceName[32];
   WORD     dmSpecVersion;
   WORD     dmDriverVersion;
   WORD     dmSize;
   WORD     dmDriverExtra;
   DWORD    dmFields;
   short    dmOrientation;
   short    dmPaperSize;
   short    dmPaperLength;
   short    dmPaperWidth;
   short    dmScale;
   short    dmCopies;
   short    dmDefaultSource;
   short    dmPrintQuality;
   short    dmColor;
   short    dmDuplex;
   BYTE
dmDriverData[dmDriverExtra];
   } DEVMODE;
```

```
TDevMode = record
  dmDeviceName:
array[0..cchDeviceName-1] of Char;
  dmSpecVersion: Word;
  dmDriverVersion: Word;
  dmSize: Word;
  dmDriverExtra: Word;
  dmFields: LongInt;
  dmOrientation: Integer;
  dmPaperSize: Integer;
  dmPaperLength: Integer;
  dmPaperWidth: Integer;
  dmScale: Integer;
  dmCopies: Integer;
  dmDefaultSource: Integer;
  dmPrintQuality: Integer;
  dmColor: Integer;
  dmDuplex: Integer;
end;
```

The **DEVMODE** structure contains the following fields:

Field	Description
dmDeviceName	Specifies the name of the device the driver supports; for example, "PCL/HP LaserJet" in the case of PCL/HP® LaserJet®. This string is unique among device drivers.
dmSpecVersion	Specifies the version number of the initialization data specification upon which the structure is based. The version number follows the Windows version number and is currently 0x300.
dmDriverVersion	Specifies the printer driver version number assigned by the printer driver developer.
dmSize	Specifies the size in bytes of the **DEVMODE** structure *except* the **dmDriverData** (device-specific) field. If an application manipulates only the driver-independent portion of the data, it can use this field to determine the length of the structure without having to account for different versions.

dmDriverExtra Contains the size of the **dmDriverData** field and is the length of the device-specific data in the **DEVMODE** structure. If an application does not use device-specific information, it should set this field to zero.

dmFields Is a bitfield that specifies which of the remaining fields in the **DEVMODE** structure have been initialized. Bit 0 (defined as DM_ORIENTATION) corresponds to **dmOrientation**; bit 1 (defined as DM_PAPERSIZE) specifies **dmPaperSize**, and so on. A printer driver supports only those fields that are appropriate for the printer technology.

dmOrientation Selects the orientation of the paper. It can be either DMORIENT_PORTRAIT (1) or DMORIENT_LANDSCAPE (2).

dmPaperSize Selects the size of the paper to print on. This field may be set to zero if the length and width of the paper are both set by the **dmPaperLength** and **dmPaperWidth** fields. Otherwise, the **dmPaperSize** field can be set to one of the following predefined values:

Value	**Meaning**
DMPAPER_LETTER	8/2-by-11-inch paper
DMPAPER_LEGAL	8/2-by-14-inch paper
DMPAPER_A4	210-by-297-millimeter paper
DMPAPER_CSHEET	17-by-22-inch paper
DMPAPER_DSHEET	22-by-34-inch paper
DMPAPER_ESHEET	34-by-44-inch paper
DMPAPER_ENV_9	3/8-by-8/8-inch #9 envelope
DMPAPER_ENV_10	4/8-by-9/5-inch #10 envelope
DMPAPER_ENV_11	4/2-by-10/8-inch #11 envelope
DMPAPER_ENV_12	4/4-by-11-inch #12 envelope
DMPAPER_ENV_14	5-by-11/2-inch #14 envelope

dmPaperLength Overrides the length of the paper specified by the **dmPaperSize** field, either for custom paper sizes or for devices such as dot-matrix printers which can print on a page of arbitrary length. These values, along with all other values which specify a physical length, are in tenths of a millimeter.

dmPaperWidth Overrides the width of the paper specified by the **dmPaperSize** field.

dmScale Scales the printed output. The apparent page size is scaled by a factor of **dmScale**/100 from the physical page size. A letter-size paper with a **dmScale** value of 50 would appear to be 17 by 22 inches, and output text and graphics would be correspondingly half their normal height and width.

dmCopies Selects the number of copies printed if the device supports multiple-page copies.

dmDefaultSource Specifies the paper bin from which the paper is fed by default. The application can override this selection by using the GETSETPAPERBINS escape. Possible bins include the following:

- DMBIN_DEFAULT
- DMBIN_UPPER
- DMBIN_LOWER
- DMBIN_MANUAL
- DMBIN_TRACTOR
- DMBIN_ENVELOPE

There is also a range of values reserved for device-specific bins. The GETSETPAPERBINS and ENUMPAPERBINS escapes use these indexes to be consistent with initialization information.

dmPrintQuality Specifies the printer resolution. There are four predefined device-independent values:

- DMRES_HIGH (–4)
- DMRES_MEDIUM (–3)
- DMRES_LOW (–2)
- DMRES_DRAFT (–1)

If a positive value is given, it specifies the number of dots per inch (DPI) and is therefore device dependent.

dmColor Switches between color and monochrome on color printers. Possible values are:

- DMCOLOR_COLOR (1)
- DMCOLOR_MONOCHROME (2).

dmDuplex Selects duplex or double-sided printing for printers capable of duplex printing. Values for this field include:

- DMDUP_SIMPLEX (1)
- DMDUP_HORIZONTAL (2)
- DMDUP_VERTICAL (3).

dmDriverData[ ] Contains device-specific data defined by the device driver.

Comments Only drivers fully updated for Windows version 3.0 and which export the **ExtDeviceMode** function use the **DEVMODE** data structure.

DLGTEMPLATE

Dialog template

The **DLGTEMPLATE** defines the contents of a dialog box. This structure is divided into three distinct parts:

Part	Description
Header Data Structure	Contains a general description of the dialog box.
Font-Information Data Structure	Defines the font with which text is drawn in the dialog box. This part is optional.
List of Items	Describes the parts that compose the dialog box.

The **CreateDialogIndirect, CreateDialogIndirectParam, DialogBoxIndirect,** and **DialogBoxIndirectParam** functions use this structure.

Header data structure

The **DLGTEMPLATE** header is shown here:

```
typedef struct {
    long  dtStyle;
    BYTE  dtItemCount;
    int   dtX;
    int   dtY;
    int   dtCX;
    int   dtCY;
    char  dtMenuName[];
    char  dtClassName[];
    char  dtCaptionText[];
} DLGTEMPLATE;
```

The **DLGTEMPLATE** header has the following fields:

Field	Description
dtStyle	Specifies the style of the dialog box. This field may be any or all of these values:

Value	Meaning
DS_LOCALEDIT	Specifies that text storage for edit controls will be allocated in the application's local data segment. This allows the use of the EM_GETHANDLE and EM_SETHANDLE messages. If

this style is not specified, edit-control data is located in a separate global data block.

DS_SYSMODAL Specifies a system-modal dialog box.

DS_MODALFRAME Specifies a dialog box with a modal dialog-box border. This style can be combined with the WS_CAPTION and WS_SYSMENU style flags to create a dialog box with a title bar and System menu.

DS_ABSALIGN Indicates that **dtX** and **dtY** are relative to the screen origin, not to the owner of the dialog box.

DS_SETFONT Specifies that a font other than the system font is to be used to draw text in the dialog box. If this flag is set, the **FONTINFO** data structure described in the following paragraphs must immediately follow the **DLGTEMPLATE** header. When Windows creates a dialog box with this attribute, Windows sends the WM_SETFONT message to the dialog-box window prior to creating the controls.

DS_NOIDLEMSG Specifies that Windows will not send the WM_ENTERIDLE message to the owner of the dialog box while the dialog box is displayed.

dtItemCount Specifies the number of items in the dialog box. A dialog box can contain up to 255 controls.

dtX Specifies the *x*-coordinate of the upper-left corner of the dialog box in units of /4 of the current dialog base width unit. The dialog base units are computed from the height and width of the current system font; the **GetDialogBaseUnits** function returns the current dialog base units in pixels. Unless DS_ABSALIGN is set in the **dtStyle** field, this value is relative to the origin of the parent window's client area.

dtY Specifies the *y*-coordinate of the upper-left corner of the dialog box in units of /8 of the current dialog base height unit. Unless DS_ABSALIGN is set in the **dtStyle** field, this value is relative to the origin of the parent window's client area.

dtCX Specifies the width of the dialog box in units of /4 of the dialog base width unit.

dtCY Specifies the height of the dialog box in units of /8 of the dialog base height unit.

dtMenuName[ ]	Specifies a null-terminated string that specifies the name of the dialog box's menu. If this field is NULL, the dialog-box window does not have a menu.
dtClassName[ ]	Specifies a null-terminated string that supplies the name of the dialog box's class. If **dtClassName[ ]** is zero, it creates a dialog box with the standard dialog-box style. If an application specifies a class name, it should provide a dialog procedure that processes each dialog-box message directly or calls the **DefDlgProc** function to process the message. Also, the application must register the class with the **cbWndExtra** field of the **WNDCLASS** data structure set to **DLGWINDOWEXTRA**.
dtCaptionText[ ]	Specifies a null-terminated string that supplies the caption for the dialog box.

Font-information data structure

The **FONTINFO** data structure contains information about the point size and face name of the font which Windows is to use to draw text in the dialog box.

```
typedef struct{
    short int   PointSize;
    char        szTypeFace[]; /* A null-terminated string */
} FONTINFO;
```

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

Field	Description
PointSize	Specifies the size of the typeface in points.
szTypeFace	Specifies the name of the typeface; for example, "Courier".

Comments

The font specified must have been previously loaded, either from WIN.INI or explicitly by calling the **LoadFont** function.

Item list

The item list consists of one or more **DLGITEMTEMPLATE** data structures, one for each control in the dialog box. The first such structure immediately follows the **FONTINFO** structure or the header at the first byte after the terminating null character in the **szTypeFace** field or the **dtCaptionText[ ]** field. The following shows the format of the **DLGITEMTEMPLATE** structure.

```
typedef struct {
    int   dtilX;
    int   dtilY;
    int   dtilCX;
    int   dtilCY;
    int   dtilID;
    long  dtilStyle;
    char  dtilClass[];
    char  dtilText[];
    BYTE  dtilInfo;
    PTR   dtilData;
} DLGITEMTEMPLATE
```

The **DLGITEMTEMPLATE** data structure has the following fields:

Field	Description
dtilX	Specifies the *x*-coordinate of the upper-left corner of the dialog-box item in units of /4 of the current dialog base width unit, relative to the origin of the dialog box. The dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the current dialog base units in pixels.
dtilY	Specifies the *y*-coordinate of the upper-left corner of the dialog-box item in units of /8 of the current dialog base height unit. This value is relative to the origin of the dialog box.
dtilCX	Specifies the width-extent of the dialog-box item in units of /4 of the current dialog base width unit. Dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the current dialog base units.
dtilCY	Specifies the height of the dialog-box item in units of /8 of the dialog base height unit.
dtilID	Specifies the dialog-box item identification number.
dtilStyle	Specifies the style of the dialog-box item.
dtilClass[ ]	A null-terminated string that specifies the control's class. It may be one of the following class names: □ BUTTON □ EDIT □ STATIC □ LISTBOX □ SCROLLBAR □ COMBOBOX
dtilText[ ]	Specifies the text for the item; it is a null-terminated string.
dtilInfo	Specifies the number of bytes of additional data that follows this item description and precedes the next item description.
dtilData	Specifies additional data which the **CreateWindow** function receives through the **lpCreateParams** field of the **CREATESTRUCT** data structure. This field is zero length if **dtilInfo** is zero.

DRAWITEMSTRUCT 3.0

Owner-draw control drawing information

The **DRAWITEMSTRUCT** structure provides information the owner needs to determine how to paint an owner-draw control. The owner of the owner-draw control receives a pointer to this structure as the *lParam* parameter of the WM_DRAWITEM message.

```
typedef struct tagDRAWITEMSTRUCT
  {
    WORD   CtlType;
    WORD   CtlID;
    WORD   itemID;
    WORD   itemAction;
    WORD   itemState;
    HWND   hwndItem;
    HDC    hDC;
    RECT   rcItem;
    DWORD  itemData;
  } DRAWITEMSTRUCT;
```

```
TDrawItemStruct = record
  CtlType: Word;
  CtlID: Word;
  itemID: Word;
  itemAction: Word;
  itemState: Word;
  hwndItem: HWnd;
  hDC: HDC;
  rcItem: TRect;
  itemData: Longint;
end;
```

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

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><b>CtlType</b></td><td>Is the control type. The values for control types are as follows:
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>ODT_BUTTON</td><td>Owner-draw button.</td></tr>
<tr><td>ODT_COMBOBOX</td><td>Owner-draw combo box.</td></tr>
<tr><td>ODT_LISTBOX</td><td>Owner-draw list box.</td></tr>
<tr><td>ODT_MENU</td><td>Owner-draw menu.</td></tr>
</table></td></tr>
<tr><td><b>CtlID</b></td><td>Is the control ID for a combo box, list box or button. This field is not used for a menu.</td></tr>
<tr><td><b>itemID</b></td><td>Is the menu-item ID for a menu or the index of the item in a list box or combo box. For an empty list box or combo box, this field can be –1. This allows the application to draw only the focus rectangle at the coordinates specified by the <b>rcItem</b> field even though there are no items in the control. This indicates to the user whether the list box or combo box has input focus. The setting of the bits in the <b>itemAction</b> field determines whether the rectangle is to be drawn as though the list box or combo box has input focus.</td></tr>
<tr><td><b>itemAction</b></td><td>Defines the drawing action required. This will be one or more of the following bits:</td></tr>
</table>

Value	**Description**
ODA_DRAWENTIRE	This bit is set when the entire control needs to be drawn.
ODA_FOCUS	This bit is set when the control gains or loses input focus. The **itemState** field should be checked to determine whether the control has focus.
ODA_SELECT	This bit is set when only the selection status has changed. The **itemState** field should be checked to determine the new selection state.

itemState Specifies the visual state of the item *after* the current drawing action takes place. That is, if a menu item is to be grayed, the state flag ODS_GRAYED will be set. The state flags are:

Value	**Description**
ODS_CHECKED	This bit is set if the menu item is to be checked. This bit is used only in a menu.
ODS_DISABLED	This bit is set if the item is to be drawn as disabled.
ODS_FOCUS	This bit is set if the item has input focus.
ODS_GRAYED	This bit is set if the item is to be grayed. This bit is used only in a menu.
ODS_SELECTED	This bit is set if the item's status is selected.

hwndItem For combo boxes, list boxes and buttons, this field specifies the window handle of the control; for menus, it contains the handle of the menu (**HMENU**) containing the item.

hDC Identifies a device context; this device context must be used when performing drawing operations on the control.

rcItem Is a rectangle in the device context specified by the **hDC** field that defines the boundaries of the control to be drawn. Windows automatically clips anything the owner draws in the device context for combo boxes, list boxes, and buttons, but does not clip menu items. When drawing menu items, the owner must ensure that the owner does not draw outside the boundaries of the rectangle defined by the **rcItem** field.

itemData For a combo box or list box, this field contains the value that was passed to the list box in the *lParam* parameter of one of the the following messages:

- CB_ADDSTRING
- CB_INSERTSTRING
- LB_ADDSTRING
- LB_INSERTSTRING

For a menu, this field contains the **DWORD** value passed as the *lpNewItem* parameter of the **InsertMenu** which inserted the menu item. Its contents are undefined for buttons.

HANDLETABLE

Window-handle table

The **HANDLETABLE** structure is an array of handles, each of which identifies a GDI object.

```
HANDLE objectHandle[1]
```

```
THandleTable = record
  objectHandle: array[0..0] of
THandle;
end;
```

The **HANDLETABLE** structure has the following field:

Field	Description
objectHandle[1]	Identifies an array of handles.

LOGBRUSH

Logical-brush attribute information

The **LOGBRUSH** structure defines the style, color, and pattern of a physical brush to be created by using the **CreateBrushIndirect** function.

```
typedef struct tagLOGBRUSH {
   WORD      lbStyle;
   COLORREF  lbColor;
   short int lbHatch;
} LOGBRUSH;
```

```
TLogBrush = record
  lbStyle: Word;
  lbColor: Longint;
  lbHatch: Integer;
end;
```

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

Field	Description
lbStyle	Specifies the brush style. The **lbStyle** field can be any one of the following styles:

Style	Meaning
BS_DIBPATTERN	Specifies a pattern brush defined by a device-independent bitmap (DIB) specification.
BS_HATCHED	Specifies a hatched brush.
BS_HOLLOW	Specifies a hollow brush.

BS_PATTERN	Specifies a pattern brush defined by a memory bitmap.
BS_SOLID	Specifies a solid brush.

lbColor Specifies the color in which the brush is to be drawn. If **lbStyle** is BS_HOLLOW or BS_PATTERN, **lbColor** is ignored. If **lpStyle** is BS_DIBPATTERN, the low-order word of **lbColor** specifies whether the **bmiColors** fields of the **BITMAPINFO** data structure contain explicit RGB values or indexes into the currently realized logical palette. The **lbColor** field must be one of the following values:

Value	**Meaning**
DIB_PAL_COLORS	The color table consists of an array of 16-bit indexes into the currently realized logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.

lbHatch Specifies a hatch style. The meaning depends on the brush style. If **lbStyle** is BS_DIBPATTERN, the **lbHatch** field contains a handle to a packed DIB. To obtain this handle, an application calls the **GlobalAlloc** function to allocate a block of global memory and then fills the memory with the packed DIB. A packed DIB consists of a **BITMAPINFO** data structure immediately followed by the array of bytes which define the pixels of the bitmap. If **lbStyle** is BS_HATCHED, the **lbHatch** field specifies the orientation of the lines used to create the hatch. It can be any one of the following values:

Value	**Meaning**
HS_BDIAGONAL	45-degree upward hatch (left to right)
HS_CROSS	Horizontal and vertical crosshatch
HS_DIAGCROSS	45-degree crosshatch
HS_FDIAGONAL	45-degree downward hatch (left to right)
HS_HORIZONTAL	Horizontal hatch
HS_VERTICAL	Vertical hatch

If **lbStyle** is BS_PATTERN, **lbHatch** must be a handle to the bitmap that defines the pattern.
If **lbStyle** is BS_SOLID or BS_HOLLOW, **lbHatch** is ignored.

See also The **CreateBrushIndirect** function in Chapter 4, "Functions directory," in *Reference, Volume 1.*

LOGFONT

Logical-font descriptor

The **LOGFONT** structure defines the attributes of a font, a drawing object used to write text on a display surface.

```
typedef struct tagLOGFONT {
   short int lfHeight;
   short int lfWidth;
   short int lfEscapement;
   short int lfOrientation;
   short int lfWeight;
   BYTE      lfItalic;
   BYTE      lfUnderline;
   BYTE      lfStrikeOut;
   BYTE      lfCharSet;
   BYTE      lfOutPrecision;
   BYTE      lfClipPrecision;
   BYTE      lfQuality;
   BYTE      lfPitchAndFamily;
   BYTE      lfFaceName[LF_FACESIZE];
} LOGFONT;
```

```
TLogFont = record
  lfHeight: Integer;
  lfWidth: Integer;
  lfEscapement: Integer;
  lfOrientation: Integer;
  lfWeight: Integer;
  lfItalic: Byte;
  lfUnderline: Byte;
  lfStrikeOut: Byte;
  lfCharSet: Byte;
  lfOutPrecision: Byte;
  lfClipPrecision: Byte;
  lfQuality: Byte;
  lfPitchAndFamily: Byte;
  lfFaceName: array[0..lf_FaceSize -
1] of Byte;
end;
```

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

Field	Description
lfHeight	Specifies the average height of the font (in user units). The height of a font can be specified in the following three ways. If the **lfHeight** field is greater than zero, it is transformed into device units and matched against the cell height of the available fonts. If **lfHeight** is zero, a reasonable default size is used. If **lfHeight** is less than zero, it is transformed into device units and the absolute value is matched against the character height of the available fonts. To ensure compatibility with the font-scaling engine of future versions of Windows, **lfHeight** should be less than zero. Setting the high-order bit indicates that the font height does not take internal leading into consideration. This corresponds to the standard typographical EM height.
lfWidth	Specifies the average width of characters in the font (in device units). If the **lfWidth** field is zero, the aspect ratio of the device is matched against the digitization aspect ratio of the available fonts for the closest match by absolute value of the difference.
lfEscapement	Specifies the angle (in tenths of degrees) between the escapement vector and the *x*-axis of the display surface. The escapement vector is the line through the origins of the first

and last characters on a line. The angle is measured counterclockwise from the *x*-axis.

lfOrientation Specifies the angle (in tenths of degrees) between the baseline of a character and the *x*-axis. The angle is measured counterclockwise from the *x*-axis.

lfWeight Specifies the font weight (in inked pixels per 1000). Although the **lfWeight** field can be any integer value from 0 to 1000, the common values are as follows:

- 400 Normal
- 700 Bold

These values are approximate; the actual appearance depends on the font face. If **lfWeight** is zero, a default weight is used.

lfItalic Specifies an italic font if set to nonzero.

lfUnderline Specifies an underlined font if set to nonzero.

lfStrikeOut Specifies a strikeout font if set to nonzero.

lfCharSet Specifies the font's character set. The three values are predefined:

- ANSI_CHARSET
- OEM_CHARSET
- SYMBOL_CHARSET

The OEM character set is system-dependent. Fonts with other character sets may exist in the system. If an application uses a font with an unknown character set, it should not attempt to translate or interpret strings that are to be rendered with that font. Instead, the strings should be passed directly to the output device driver.

lfOutPrecision Specifies the font's output precision, which defines how closely the output must match the requested font's height, width, character orientation, escapement, and pitch. The default setting is OUT_DEFAULT_PRECIS.

lfClipPrecision Specifies the font's clipping precision, which defines how to clip characters that are partially outside the clipping region. The default setting is CLIP_DEFAULT_PRECIS.

lfQuality Specifies the font's output quality, which defines how carefully GDI must attempt to match the logical-font attributes to those of an actual physical font. It can be any one of the following values:

Value	**Meaning**
DEFAULT_QUALITY	Appearance of the font does not matter.
DRAFT_QUALITY	Appearance of the font is less important than when PROOF_QUALITY is used. For GDI fonts, scaling is enabled, which means that more font sizes are available, but the quality may be lower. Bold, italic, underline, and strikeout fonts are synthesized if necessary.

PROOF_QUALITY	Character quality of the font is more important than exact matching of the logical-font attributes. For GDI fonts, scaling is disabled and the font closest in size is chosen. Although the chosen font size may not be mapped exactly when PROOF_QUALITY is used, the quality of the font is high and there is no distortion of appearance. Bold, italic, underline, and strikeout fonts are synthesized if necessary.

lfPitchAndFamily Specifies the font pitch and family. The two low-order bits specify the pitch of the font and can be any one of the following values:

- DEFAULT_PITCH
- FIXED_PITCH
- VARIABLE_PITCH

The four high-order bits of the field specify the font family and can be any one of the following values:

- FF_DECORATIVE
- FF_DONTCARE
- FF_MODERN
- FF_ROMAN
- FF_SCRIPT
- FF_SWISS

The proper value can be obtained by using the Boolean OR operator to join one pitch constant with one family constant. Font families describe the look of a font in a general way. They are intended for specifying fonts when the exact typeface desired is not available. The values for font families are as follows:

Value	**Meaning**
FF_DECORATIVE	Novelty fonts. Old English, for example.
FF_DONTCARE	Don't care or don't know.
FF_MODERN	Fonts with constant stroke width (fixed-pitch), with or without serifs. Fixed-pitch fonts are usually modern. Pica, Elite, and Courier, for example.
FF_ROMAN	Fonts with variable stroke width (proportionally spaced) and with serifs. Times Roman, Palatino, and Century Schoolbook, for example.
FF_SCRIPT	Fonts designed to look like handwriting. Script and Cursive, for example.
FF_SWISS	Fonts with variable stroke width (proportionally spaced) and

		without serifs. Helvetica and Swiss, for example.
lfFaceName	Specifies the font's typeface. It must be a null-terminated character string. If **lfFaceName** is NULL, GDI uses a default typeface.	

See also The **CreateFontIndirect** function in Chapter 4, "Functions directory," in *Reference, Volume 1.*

LOGPALETTE 3.0

Logical color palette information

The **LOGPALETTE** data structure defines a logical color palette.

```
typedef struct
        {
        WORD          palVersion;
        WORD          palNumEntries;
        PALETTEENTRY  palPalEntry[];
        } LOGPALETTE;
```

```
TLogPalette = record
  palVersion: Word;
  palNumEntries: Word;
  palPalEntry: array[0..0] of
TPaletteEntry;
end;
```

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

Field	Description
palVersion	Specifies the Windows version number for the structure (currently 0x300).
palNumEntries	Specifies the number of palette color entries.
palPalEntry [ ]	Specifies an array of **PALETTEENTRY** data structures that define the color and usage of each entry in the logical palette.

Comments The colors in the palette entry table should appear in order of importance. This is because entries earlier in the logical palette are most likely to be placed in the system palette.

This data structure is passed as a parameter to the **CreatePalette** function.

LOGPEN

Logical-pen attribute information

The **LOGPEN** structure defines the style, width, and color of a pen, a drawing object used to draw lines and borders. The **CreatePenIndirect** function uses the **LOGPEN** structure.

```
typedef struct tagLOGPEN {
   WORD      lopnStyle;
   POINT     lopnWidth;
   COLORREF  lopnColor;
} LOGPEN;
```

```
TLogPen = record
  lopnStyle: Word;
  lopnWidth: TPoint;
  lopnColor: Longint;
end;
```

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

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><b>lopnStyle</b></td><td>Specifies the pen type, which can be any one of the following values:
<table>
<tr><th>Constant Name</th><th>Value</th><th>Result</th></tr>
<tr><td>PS_SOLID</td><td>0</td><td>___</td></tr>
<tr><td>PS_DASH</td><td>1</td><td>– – –</td></tr>
<tr><td>PS_DOT</td><td>2</td><td>. . . .</td></tr>
<tr><td>PS_DASHDOT</td><td>3</td><td>–.–.</td></tr>
<tr><td>PS_DASHDOTDOT</td><td>4</td><td>–. . –. .</td></tr>
<tr><td>PS_NULL</td><td>5</td><td></td></tr>
<tr><td>PS_INSIDEFRAME</td><td>6</td><td>___</td></tr>
</table>
If the width of the pen is greater than 1 and the pen style is PS_INSIDEFRAME, the line is drawn inside the frame of all primitives except polygons and polylines; the pen is drawn with a logical (dithered) color if the pen color does not match an available RGB value. The PS_INSIDEFRAME style is identical to PS_SOLID if the pen width is less than or equal to 1.</td></tr>
<tr><td><b>lopnWidth</b></td><td>Specifies the pen width (in logical units). If the <b>lopnWidth</b> field is zero, the pen is one pixel wide on raster devices.</td></tr>
<tr><td><b>lopnColor</b></td><td>Specifies the pen color.</td></tr>
</table>

Comments The *y* value in the **POINT** structure for **lopnWidth** is not used.

See also The **CreatePenIndirect** function in Chapter 4, "Functions directory," in *Reference, Volume 1.*

MDICREATESTRUCT 3.0

Mdi child window creation structure

The **MDICREATESTRUCT** data structure contains information about the class, title, owner, location, and size of a multiple document interface (MDI) child window.

```
typedef struct tagMDICREATESTRUCT
  {
    LPSTR   szClass;
    LPSTR   szTitle;
    HANDLE  hOwner;
    int     x;
    int     y;
    int     cx;
    int     cy;
    LONG    style;
    LONG    lParam;
  } MDICREATESTRUCT;
```

```
TMDICreateStruct = record
  szClass: PChar;
  szTitle: PChar;
  hOwner: THandle;
  x, y: Integer;
  cx, cy: Integer;
  style: LongInt;
  lParam: LongInt;
end;
```

The **MDICREATESTRUCT** structure contains the following fields:

Field	Description
szClass	Contains a long pointer to the application-defined class of the MDI child window.
szTitle	Contains a long pointer to the window title of the MDI child window.
hOwner	Is the instance handle of the application creating the MDI child window.
x	Specifies the initial position of the left side of the MDI child window. If set to CW_USEDEFAULT, the MDI child window is assigned a default horizontal position.
y	Specifies the initial position of the top edge of the MDI child window. If set to CW_USEDEFAULT, the MDI child window is assigned a default vertical position.
cx	Specifies the initial width of the MDI child window. If set to CW_USEDEFAULT, the MDI child window is assigned a default width.
cy	Specifies the initial height of the MDI child window. If set to CW_USEDEFAULT, the MDI child window is assigned a default height.
style	Specifies additional styles for the MDI child window. The **style** field may be set to one or more of the following values:

Value	Meaning
WS_MINIMIZE	The MDI child window is created in a minimized state.
WS_MAXIMIZE	The MDI child window is created in a maximized state.
WS_HSCROLL	The MDI child window is created with a horizontal scroll bar.
WS_VSCROLL	The MDI child window is created with a vertical scroll bar.

lParam Is an application-defined 32-bit value.

Comments When the MDI child window is created, Windows sends the WM_CREATE message to the window. The *lParam* parameter of the WM_CREATE message contains a pointer to a **CREATESTRUCT** data structure. The **lpCreateParams** field of the **CREATESTRUCT** structure contains a pointer to the **MDICREATESTRUCT** data structure passed with the WM_MDICREATE message that created the MDI child window.

MEASUREITEMSTRUCT 3.0

Owner-draw control dimensions

The **MEASUREITEMSTRUCT** data structure informs Windows of the dimensions of an owner-draw control. This allows Windows to process user interaction with the control correctly. The owner of an owner-draw control receives a pointer to this structure as the *lParam* parameter of an WM_MEASUREITEM message. The owner-draw control sends this message to its owner window when the control is created; the owner then fills in the appropriate fields in the structure for the control and returns. This structure is common to all owner-draw controls.

The **MEASUREITEMSTRUCT** structure has the following format:

```
typedef struct tagMEASUREITEMSTRUCT
  {
    WORD   CtlType;
    WORD   CtlID;
    WORD   itemID;
    WORD   itemWidth;
    WORD   itemHeight;
    DWORD  itemData
  } MEASUREITEMSTRUCT;
```

```
TMeasureItemStruct = record
  CtlType: Word;
  CtlID: Word;
  itemID: Word;
  itemWidth: Word;
  itemHeight: Word;
  itemData: Longint;
end;
```

The **MEASUREITEMSTRUCT** structure contains the following fields:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><b>CtlType</b></td><td>Is the control type. The values for control types are as follows:
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>ODT_BUTTON</td><td>Owner-draw button.</td></tr>
<tr><td>ODT_COMBOBOX</td><td>Owner-draw combo box.</td></tr>
<tr><td>ODT_LISTBOX</td><td>Owner-draw list box.</td></tr>
<tr><td>ODT_MENU</td><td>Owner-draw menu.</td></tr>
</table></td></tr>
<tr><td><b>CtlID</b></td><td>Is the control ID for a combo box, list box, or button. This field is not used for a menu.</td></tr>
<tr><td><b>itemID</b></td><td>Is the menu-item ID for a menu or the list-box item ID for a variable-height combo box or list box. This field is not used for a fixed-height combo box or list box, or for a button.</td></tr>
<tr><td><b>itemWidth</b></td><td>Specifies the width of a menu item. The owner of the owner-draw menu item must fill this field before returning from the message.</td></tr>
<tr><td><b>itemHeight</b></td><td>Specifies the height of an individual item in a list box or a menu. Before returning from the message, the owner of the owner-draw combo box, list box, or menu item must fill out this field.</td></tr>
<tr><td><b>itemData</b></td><td>Contains the value that was passed to the combo box or list box in the <i>lParam</i> parameter of one of the following messages:
□ CB_ADDSTRING
□ CB_INSERTSTRING
□ LB_ADDSTRING
□ LB_INSERTSTRING
Contains the <b>DWORD</b> value passed as the <i>lpNewItem</i> parameter of the <b>AppendMenu, InsertMenu</b>, or <b>ModifyMenu</b> function that added or modified the menu item. Its contents are undefined for buttons.</td></tr>
</table>

Comments Failure to fill out the proper fields in the **MEASUREITEM** structure will cause improper operation of the control.

MENUITEMTEMPLATE

Menuitemtemplate

A complete menu template consists of a header and one or more menu-item lists. The following shows the structure of the menu-template header:

```
typedef struct {
   WORD   versionNumber;
   WORD   offset;
} MENUITEMTEMPLATEHEADER;
```

```
TMenuItemTemplateHeader = record
  versionNumber: Word;
  offset: Word;
end;
```

The menu-template header contains the following fields:

Field	Description
versionNumber	Specifies the version number. Should be zero.
offset	Specifies the offset from the header in bytes where the menu-item list begins.

One or more **MENUITEMTEMPLATE** structures are combined to form the menu-item list.

```
typedef struct {
   WORD mtOption;
   WORD mtID;
   char mtString;
} MENUITEMTEMPLATE;
```

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

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><b>mtOption</b></td><td>Specifies a mask of one or more predefined menu options that specify the appearance of the menu item. The menu options are as follows:
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>MF_CHECKED</td><td>Item has a checkmark next to it.</td></tr>
<tr><td>MF_END</td><td>Item must be specified for the last item in a pop-up menu or a static menu.</td></tr>
<tr><td>MF_GRAYED</td><td>Item is initially inactive and drawn with a gray effect.</td></tr>
<tr><td>MF_HELP</td><td>Item has a vertical separator to its left.</td></tr>
<tr><td>MF_MENUBARBREAK</td><td>Item is placed in a new column. The old and new columns are separated by a bar.</td></tr>
<tr><td>MF_MENUBREAK</td><td>Item is placed in a new column.</td></tr>
<tr><td>MF_OWNERDRAW</td><td>The owner of the menu is responsible for drawing all visual aspects of the menu item, including highlighted, checked and inactive states. This option is not valid for a top-level menu item.</td></tr>
<tr><td>MF_POPUP</td><td>Item displays a sublist of menu items when selected.</td></tr>
</table>
</td></tr>
<tr><td><b>mtID</b></td><td>Specifies an identification code for a nonpop-up menu item. The <b>MENUITEMTEMPLATE</b> data structure for a pop-up menu item does not contain the <b>mtID</b> field.</td></tr>
<tr><td><b>mtString</b></td><td>Specifies a null-terminated character string that contains the name of the menu item.</td></tr>
</table>

See also The **LoadMenuIndirect** function in Chapter 4, "Functions directory," in *Reference, Volume 1.*

METAFILEPICT

Metafile picture structure

The **METAFILEPICT** structure defines the metafile picture format used for exchanging metafile data through the clipboard.

```
typedef struct tagMETAFILEPICT {      TMetaFilePict = record
   int     mm;                          mm: Integer;
   int     xExt, yExt;                  xExt: Integer;
   HANDLE  hMF;                         yExt: Integer;
} METAFILEPICT;                         hMF: THandle;
                                      end;
```

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

Field	Description
mm	Specifies the mapping mode in which the picture is drawn.
xExt	Specifies the size of the metafile picture for all modes except the MM_ISOTROPIC and MM_ANISOTROPIC modes. The *x*-extent specifies the width of the rectangle within which the picture is drawn. The coordinates are in units that correspond to the mapping mode.
yExt	Specifies the size of the metafile picture for all modes except the MM_ISOTROPIC and MM_ANISOTROPIC modes. The *y*-extent specifies the height of the rectangle within which the picture is drawn. The coordinates are in units that correspond to the mapping mode. For MM_ISOTROPIC and MM_ANISOTROPIC modes, which can be scaled, the **xExt** and **yExt** fields contain an optional suggested size in MM_HIMETRIC units. For MM_ANISOTROPIC pictures, **xExt** and **yExt** can be zero when no suggested size is supplied. For MM_ISOTROPIC pictures, an aspect ratio must be supplied even when no suggested size is given. (If a suggested size is given, the aspect ratio is implied by the size.) To give an aspect ratio without implying a suggested size, set **xExt** and **yExt** to negative values whose ratio is the appropriate aspect ratio. The magnitude of the negative **xExt** and **yExt** values will be ignored; only the ratio will be used.
hMF	Identifies a memory metafile.

MSG

Message data structure

The **MSG** structure contains information from the Windows application queue.

```
typedef struct tagMSG {
   HWND    hwnd;
   WORD    message;
   WORD    wParam;
   LONG    lParam;
   DWORD   time;
   POINT   pt;
} MSG;
```

```
TMsg = record
  hwnd: HWnd;
  message: Word;
  wParam: Word;
  lParam: LongInt;
  time: Longint;
  pt: TPoint;
end;
```

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

Field	Description
hwnd	Identifies the window that receives the message.
message	Specifies the message number.
wParam	Specifies additional information about the message. The exact meaning depends on the **message** value.
lParam	Specifies additional information about the message. The exact meaning depends on the **message** value.
time	Specifies the time at which the message was posted.
pt	Specifies the position of the cursor (in screen coordinates) when the message was posted.

MULTIKEYHELP

Windows help key word table structure

The **MULTIKEYHELP** structure specifies a key-word table and an associated key word to be used by the Windows Help application.

```
typedef struct tagMULTIKEYHELP {
   WORD  mkSize;
   BYTE  mkKeylist;
   BYTE  szKeyphrase[];
} MULTIKEYHELP;
```

```
TMultiKeyHelp = record
  mkSize: Word;
  mkKeyList: Byte;
  szKeyPhrase: array[0..0] of Byte;
end;
```

The **MULTIKEYHELP** data structure contains the following fields:

Field	Description
mkSize	Specifies the length of the **MULTIKEYHELP** structure (in bytes).
mkKeylist	Contains a single character that identifies the key-word table to be searched.
szKeyphrase[ ]	Contains a null-terminated text string that specifies the key word to be located in the key-word table.

OFSTRUCT

Open-file structure

The **OFSTRUCT** structure contains file information which results from opening that file.

```
typedef struct tagOFSTRUCT {
   BYTE  cBytes;
   BYTE  fFixedDisk;
   WORD  nErrCode;
   BYTE  reserved[4];
   BYTE  szPathName[120];
} OFSTRUCT;
```

```
TOFStruct = record
  cBytes: Byte;
  fFixedDisk: Byte;
  nErrCode: Word;
  reserved: array[0..3] of Byte;
  szPathName: array[0..127] of Char;
end;
```

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

Field	Description
cBytes	Specifies the length of the **OFSTRUCT** structure (in bytes).
fFixedDisk	Specifies whether the file is on a fixed disk. The **fFixedDisk** field is nonzero if the file is on a fixed disk.
nErrCode	Specifies the DOS error code if the **OpenFile** function returns –1 (that is, **OpenFile** failed).
reserved[4]	Reserved field. Four bytes reserved for future use.
szPathName[120]	Specifies 120 bytes that contain the pathname of the file. This string consists of characters from the OEM character set.

PAINTSTRUCT

WINDOWS paint information

The **PAINTSTRUCT** structure contains information for an application. This information can be used to paint the client area of a window owned by that application.

```
typedef struct tagPAINTSTRUCT {   TPaintStruct = record
   HDC  hdc;                         hdc: HDC;
   BOOL fErase;                      fErase: Bool;
   RECT rcPaint;                     rcPaint: TRect;
   BOOL fRestore;                    fRestore: Bool;
   BOOL fIncUpdate;                  fIncUpdate: Bool;
   BYTE rgbReserved[16];             rgbReserved: array[0..15] of Byte;
} PAINTSTRUCT;                    end;
```

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

Field	Description
hdc	Identifies the display context to be used for painting.
fErase	Specifies whether the background has been redrawn. It has been redrawn if nonzero.
rcPaint	Specifies the upper-left and lower-right corners of the rectangle in which the painting is requested.
fRestore	Reserved field. It is used internally by Windows.
fIncUpdate	Reserved field. It is used internally by Windows.
rgbReserved[16]	Reserved field. A reserved block of memory used internally by Windows.

PALETTEENTRY 3.0

Logical palette color entry

The **PALETTEENTRY** data structure specifies the color and usage of an entry in a logical color palette. A logical palette is defined by a **LOGPALETTE** data structure.

```
typedef struct                         TPaletteEntry = record
        {                                peRed: Byte;
        BYTE  peRed;                     peGreen: Byte;
        BYTE  peGreen;                   peBlue: Byte;
        BYTE  peBlue;                    peFlags: Byte;
        BYTE  peFlags;                 end;
        } PALETTEENTRY;
```

The **PALETTEENTRY** structure contains the following fields:

Field	Description
peRed	Specifies the intensity of red for the palette entry color.
peGreen	Specifies the intensity of green for the palette entry color.
peBlue	Specifies the intensity of blue for the palette entry color.
peFlags	Specifies how the palette entry is to be used. The **peFlags** field may be set to NULL or one of these values:

Flag	Meaning
PC_EXPLICIT	Specifies that the low-order word of the logical palette entry designates a hardware palette index. This flag allows the application to show the contents of the display-device palette.
PC_NOCOLLAPSE	Specifies that the color will be placed in an unused entry in the system palette instead of being matched to an existing color in the system palette. If there are no unused entries in the system palette, the color is matched normally. Once this color is in the system palette, colors in other logical palettes can be matched to this color.
PC_RESERVED	Specifies that the logical palette entry will be used for palette animation; this prevents other windows from matching colors to this palette entry since the color will frequently change. If an unused system-palette entry is available, this color is placed in that entry. Otherwise, the color will not be available for animation.

POINT

Point data structure

The **POINT** structure defines the *x*- and *y*-coordinates of a point.

```
typedef struct tagPOINT {          TPoint = record
   int x;                            x: Integer;
   int y;                            y: Integer;
} POINT;                           end;
```

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

Field	Description
x	Specifies the *x*-coordinate of a point.
y	Specifies the *y*-coordinate of a point.

See also The **ChildWindowFromPoint**, **PtInRect**, and **WindowFromPoint** functions in Chapter 4, "Functions directory," in *Reference, Volume 1*.

RECT

Rectangle data structure

The **RECT** structure defines the coordinates of the upper-left and lower-right corners of a rectangle.

```
typedef struct tagRECT {           TRect = record
   int left;                         left: Integer;
   int top;                          top: Integer;
   int right;                        right: Integer;
   int bottom;                       bottom: Integer;
} RECT;                            end;
```

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

Field	Description
left	Specifies the *x*-coordinate of the upper-left corner of a rectangle.
top	Specifies the *y*-coordinate of the upper-left corner of a rectangle.
right	Specifies the *x*-coordinate of the lower-right corner of a rectangle.
bottom	Specifies the *y*-coordinate of the lower-right corner of a rectangle.

Comments The width of the rectangle defined by the **RECT** structure must not exceed 32,768 units.

RGBQUAD 3.0

Rgb color structure

The **RGBQUAD** data structure describes a color consisting of relative intensities of red, green, and blue. The **bmiColors** field of the **BITMAPINFO** data structure consists of an array of **RGBQUAD** data structures.

```
typedef struct tagRGBQUAD {
   BYTE    rgbBlue;
   BYTE    rgbGreen;
   BYTE    rgbRed;
   BYTE    rgbReserved;
} RGBQUAD;
```

```
TRGBQuad = record
  rgbBlue: Byte;
  rgbGreen: Byte;
  rgbRed: Byte;
  rgbReserved: Byte;
end;
```

The **RGBQUAD** structure contains the following fields:

Field	Description
rgbBlue	Specifies the intensity of blue in the color.
rgbGreen	Specifies the intensity of green in the color.
rgbRed	Specifies the intensity of red in the color.
rgbReserved	Is not used and must be set to zero.

RGBTRIPLE 3.0

Rgb color structure

The **RGBTRIPLE** data structure describes a color consisting of relative intensities of red, green, and blue. The **bmciColors** field of the **BITMAPCOREINFO** data structure consists of an array of **RGBTRIPLE** data structures.

```
typedef struct tagRGBTRIPLE {
        BYTE    rgbtBlue;
        BYTE    rgbtGreen;
        BYTE    rgbtRed;
} RGBTRIPLE;
```

```
TRGBTriple = record
  rgbtBlue: Byte;
  rgbtGreen: Byte;
  rgbtRed: Byte;
end;
```

The **RGBTRIPLE** structure contains the following fields:

Field	Description
rgbtBlue	Specifies the intensity of blue in the color.
rgbtGreen	Specifies the intensity of green in the color.
rgbtRed	Specifies the intensity of red in the color.

TEXTMETRIC

Basic font metrics

The **TEXTMETRIC** structure contains basic information about a physical font. All sizes are given in logical units; that is, they depend on the current mapping mode of the display context.

```
typedef struct tagTEXTMETRIC {
   short int tmHeight;
   short int tmAscent;
   short int tmDescent;
   short int tmInternalLeading;
   short int tmExternalLeading;
   short int tmAveCharWidth;
   short int tmMaxCharWidth;
   short int tmWeight;
   BYTE      tmItalic;
   BYTE      tmUnderlined;
   BYTE      tmStruckOut;
   BYTE      tmFirstChar;
   BYTE      tmLastChar;
   BYTE      tmDefaultChar;
   BYTE      tmBreakChar;
   BYTE      tmPitchAndFamily;
   BYTE      tmCharSet;
   short int tmOverhang;
   short int tmDigitizedAspectX;
   short int tmDigitizedAspectY;
} TEXTMETRIC;
```

```
TTextMetric = record
  tmHeight: Integer;
  tmAscent: Integer;
  tmDescent: Integer;
  tmInternalLeading: Integer;
  tmExternalLeading: Integer;
  tmAveCharWidth: Integer;
  tmMaxCharWidth: Integer;
  tmWeight: Integer;
  tmItalic: Byte;
  tmUnderlined: Byte;
  tmStruckOut: Byte;
  tmFirstChar: Byte;
  tmLastChar: Byte;
  tmDefaultChar: Byte;
  tmBreakChar: Byte;
  tmPitchAndFamily: Byte;
  tmCharSet: Byte;
  tmOverhang: Integer;
  tmDigitizedAspectX: Integer;
  tmDigitizedAspectY: Integer;
end;
```

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

Field	Description
tmHeight	Specifies the height (ascent + descent) of characters.
tmAscent	Specifies the ascent (units above the baseline) of characters.
tmDescent	Specifies the descent (units below the baseline) of characters.
tmInternalLeading	Specifies the amount of leading (space) inside the bounds set by the **tmHeight** field. Accent marks and other foreign characters may occur in this area. The designer may set this field to zero.
tmExternalLeading	Specifies the amount of extra leading (space) that the application adds between rows. Since this area is outside the font, it contains no marks and will not be altered by text output calls in either OPAQUE or TRANSPARENT mode. The designer may set this field to zero.

tmAveCharWidth Specifies the average width of characters in the font (loosely defined as the width of the letter *x*). This value does not include overhang required for bold or italic characters.

tmMaxCharWidth Specifies the width of the widest character in the font.

tmWeight Specifies the weight of the font.

tmItalic Specifies an italic font if it is nonzero.

tmUnderlined Specifies an underlined font if it is nonzero.

tmStruckOut Specifies a struckout font if it is nonzero.

tmFirstChar Specifies the value of the first character defined in the font.

tmLastChar Specifies the value of the last character defined in the font.

tmDefaultChar Specifies the value of the character that will be substituted for characters that are not in the font.

tmBreakChar Specifies the value of the character that will be used to define word breaks for text justification.

tmPitchAndFamily Specifies the pitch and family of the selected font. The low-order bit specifies the pitch of the font. If it is 1, the font is variable pitch. If it is 0, the font is fixed pitch. The four high-order bits designate the font family. The **tmPitchAndFamily** field can be combined with the hexadecimal value 0xF0 by using the bitwise AND operator, and then be compared with the font family names for an identical match. For a description of the font families, see the **LOGFONT** structure, earlier in this chapter.

tmCharSet Specifies the character set of the font.

tmOverhang Specifies the per-string extra width that may be added to some synthesized fonts. When synthesizing some attributes, such as bold or italic, GDI or a device may have to add width to a string on both a per-character and per-string basis. For example, GDI makes a string bold by expanding the intracharacter spacing and overstriking by an offset value; it italicizes a font by skewing the string. In either case, there is an overhang past the basic string. For bold strings, the overhang is the distance by which the overstrike is offset. For italic strings, the overhang is the amount the top of the font is skewed past the bottom of the font. The **tmOverhang** field allows the application to determine how much of the character width returned by a **GetTextExtent** function call on a single character is the actual character width and how much is the per-string extra width. The actual width is the extent minus the overhang.

tmDigitizedAspectX Specifies the horizontal aspect of the device for which the font was designed.

tmDigitizedAspectY Specifies the vertical aspect of the device for which the font was designed. The ratio of the **tmDigitizedAspectX** and **tmDigitizedAspectY** fields is the aspect ratio of the device for which the font was designed.

See also The **GetDeviceCaps** and **GetTextMetrics** functions in Chapter 4, "Functions directory," in *Reference, Volume 1.*

WNDCLASS

Window class data structure

THE **WNDCLass** structure contains the class attributes that are registered by the **RegisterClass** function.

```
typedef struct tagWNDCLASS {
   WORD    style;
   long    (FAR PASCAL
*lpfnWndProc)();
   int     cbClsExtra;
   int     cbWndExtra;
   HANDLE  hInstance;
   HICON   hIcon;
   HCURSOR hCursor;
   HBRUSH  hbrBackground;
   LPSTR   lpszMenuName;
   LPSTR   lpszClassName;
} WNDCLASS;
```

```
TWndClass = record
  style: Word;
  lpfnWndProc: TFarProc;
  cbClsExtra: Integer;
  cbWndExtra: Integer;
  hInstance: THandle;
  hIcon: HIcon;
  hCursor: HCursor;
  hbrBackground: HBrush;
  lpszMenuName: PChar;
  lpszClassName: PChar;
end;
```

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

Field	Description
style	Specifies the class style. These styles can be combined by using the bitwise OR operator. The **style** field can be any combination of the following values:

Value	Meaning
CS_BYTEALIGNCLIENT	Aligns a window's client area on the byte boundary (in the *x* direction).
CS_BYTEALIGNWINDOW	Aligns a window on the byte boundary (in the *x* direction).
CS_CLASSDC	Gives the window class its own display context (shared by instances).
CS_DBLCLKS	Sends double-click messages to a window.
CS_GLOBALCLASS	Specifies that the window class is an application global class. An application global class is created by an application or library and is available to all applications. The class is destroyed when the application or library that created the class terminates; it is essential,

therefore, that all windows created with the application global class be closed before this occurs.

CS_HREDRAW Redraws the entire window if the horizontal size changes.

CS_NOCLOSE Inhibits the close option on the System menu.

CS_OWNDC Gives each window instance its own display context. Note that although the CS_OWNDC style is convenient, it must be used with discretion because each display context occupies approximately 800 bytes of memory.

CS_PARENTDC Gives the parent window's display context to the window class.

CS_SAVEBITS Saves the portion of the screen image that is obscured by a window; Windows uses the saved bitmap to re-create a screen image when the window is removed. Windows displays the bitmap at its original location and does not send WM_PAINT messages to windows which had been obscured by the window if the memory used by the bitmap has not been discarded and if other screen actions have not invalidated the stored image. An application should set this bit only for small windows that are displayed briefly and then removed before much other screen activity takes place. Setting this bit for a window increases the amount of time required to display the window due to the time required to allocate memory to store the bitmap.

CS_VREDRAW Redraws the entire window if the vertical size changes.

lpfnWndProc Points to the window function.

cbClsExtra Specifies the number of bytes to allocate following the window-class structure.

cbWndExtra Specifies the number of bytes to allocate following the window instance. If an application is using the **WNDCLASS** structure to register a dialog box created with the **CLASS** directive in the .RC script file, it must set this field to DLGWINDOWEXTRA.

hInstance Identifies the class module. The **hInstance** field must be an instance handle and must *not* be NULL.

hIcon Identifies the class icon. The **hIcon** field must be a handle to an icon resource. If **hIcon** is NULL, the application must draw an icon whenever the user minimizes the application's window.

hCursor Identifies the class cursor. The **hCursor** field must be a handle to a cursor resource. If **hCursor** is NULL, the application must explicitly set the cursor shape whenever the mouse moves into the application's window.

hbrBackground Identifies the class background brush. The **hbrBackground** field can be either a handle to the physical brush that is to be used for painting the background, or it can be a color value. If a color value is given, it must be one of the standard system colors listed below, and the value 1 must be added to the chosen color (for example, COLOR_BACKGROUND + 1 specifies the system background color). If a color value is given, it must be converted to one of these HBRUSH types:

- COLOR_ACTIVEBORDER
- COLOR_ACTIVECAPTION
- COLOR_APPWORKSPACE
- COLOR_BACKGROUND
- COLOR_BTNFACE
- COLOR_BTNSHADOW
- COLOR_BTNTEXT
- COLOR_CAPTIONTEXT
- COLOR_GRAYTEXT
- COLOR_HIGHLIGHT
- COLOR_HIGHLIGHTTEXT
- COLOR_INACTIVEBORDER
- COLOR_INACTIVECAPTION
- COLOR_MENU
- COLOR_MENUTEXT
- COLOR_SCROLLBAR
- COLOR_WINDOW
- COLOR_WINDOWFRAME
- COLOR_WINDOWTEXT

When **hbrBackground** is NULL, the application must paint its own background whenever it is requested to paint in its client area. The application can determine when the background needs painting by processing the WM_ERASEBKGND message or by testing the **fErase** field of the **PAINTSTRUCT** structure filled by the **BeginPaint** function.

lpszMenuName Points to a null-terminated character string that specifies the resource name of the class menu (as the name appears in the resource file). If an integer is used to identify the menu, the **MAKEINTRESOURCE** macro can be used. If the **lpszMenuName** field is NULL, windows belonging to this class have no default menu.

lpszClassName Points to a null-terminated character string that specifies the name of the window class.

CHAPTER

8

Resource script statements

This chapter describes the statements that define resources that the Microsoft Windows Resource Compiler (**RC**) adds to an application's executable file. See *Tools* for information on running the Resource Compiler.

This chapter describes resource script statements in the following categories:

- Single-line statements
- User-defined resources
- RCDATA statement
- STRINGTABLE statement
- ACCELERATORS statement
- Menu statements
- Dialog statements
- Directives

Single-line statements

The single-line statements define resources that are contained in a single file, such as cursors, icons, and fonts. The statements associate the filename of the resource with an identifying name or number. The resource is added to the executable file when the application is created, and can be extracted during execution by referring to the name or number.

The following is the general form for all single-line statements:

```
nameID resource-type [[load-option]] [[mem-option]] filename
```

The *nameID* field specifies either a unique name or an integer value identifying the resource. For a font resource, *nameID* must be a number; it cannot be a name.

The *resource-type* field specifies one of the following key words, which identify the type of resource to be loaded:

Key word	Resource Type
CURSOR	Specifies a bitmap that defines the shape of the cursor on the display screen.
ICON	Specifies a bitmap that defines the shape of the icon to be used for a given application.
BITMAP	Specifies a custom bitmap that an application is going to use in its screen display or as an item in a menu.
FONT	Specifies a file that contains a font.

The optional *load-option* field takes a key word that specifies when the resource is to be loaded. The key word must be one of the following:

Option	Description
PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called. This is the default option.

Icon and cursor resources can contain more than one image. If the resource is marked as **PRELOAD**, Windows loads all images in the resource when the application executes.

The optional *mem-option* field takes the following key word or key words, which specify whether the resource is fixed or moveable and whether it is discardable:

Option	Description
FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary in order to compact memory.
DISCARDABLE	Resource can be discarded if no longer needed.

The default is **MOVEABLE** and **DISCARDABLE** for cursor, icon, and font resources. The default for bitmap resources is **MOVEABLE**.

The *filename* field is an ASCII string that specifies the DOS filename of the file that contains the resource. A full pathname must be given if the file is not in the current working directory.

The following example demonstrates the correct usage for a single-line statement:

```
cursor CURSOR point.cur
cursor CURSOR DISCARDABLE point.cur
10      CURSOR custom.cur

desk    ICON desk.ico
desk    ICON DISCARDABLE desk.ico
11      ICON custom.ico

disk    BITMAP disk.bmp
disk    BITMAP DISCARDABLE disk.bmp
12      BITMAP custom.bmp

5 FONT  CMROMAN.FNT
```

User-defined resources

An application can also define its own resource. The resource can be any data that the application intends to use. A user-defined resource statement has the following form:

nameID typeID [[*load-option*]] [[*mem-option*]] {[[*filename*]] |
[[BEGIN
raw-data
END]]}

The *nameID* field specifies either a unique name or an integer value that identifies the resource.

The *typeID* field specifies either a unique name or an integer value that identifies the resource type. If a number is given, it must be greater than 255. The numbers 1 through 255 are reserved for existing and future predefined resource types.

The optional *load-option* field takes a key word that specifies when the resource is to be loaded. The key word must be one of the following:

Option	Description
PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called. This is the default option.

The optional *mem-option* field takes the following key word or key words, which specify whether the resource is fixed or moveable and whether it is discardable:

Option	Description
FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary in order to compact memory. This is the default option.
DISCARDABLE	Resource can be discarded if it is no longer needed.

The optional *filename* field is an ASCII string that specifies the DOS filename of the file that contains the resource. A full pathname must be given if the file is not in the current working directory. Do not use the *filename* field if you supply raw data between the optional **BEGIN** and **END** statements.

The *raw-data* field specifies one or more integers and strings. Integers can be in decimal, octal, or hexadecimal format. Do not use *raw-data* field and the **BEGIN** and **END** statements if you specify a filename.

The following example demonstrates the correct usage for user-defined statements:

```
array   MYRES   data.res
14      300     custom.res
18 MYRES2
BEGIN
  "Here is a data string\0", /* A string. Note: explicitly
                                null-terminated */
   1024,                     /* int        */
   0x029a,                   /* hex int    */
   0o733,                    /* octal int  */
   "\07"                     /* octal byte */
END
```

Rcdata statement

Syntax *nameID* RCDATA [[*load-option*]] [[*mem-option*]]
BEGIN
raw-data
END

The **RCDATA** statement defines a raw data resource for an application. Raw data resources permit the inclusion of binary data directly in the executable file.

The *nameID* field specifies either a unique name or an integer value that identifies the resource.

The optional *load-option* field takes a key word that specifies when the resource is to be loaded. It must be one of the following:

Option	Description
PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called. This is the default option.

The optional *mem-option* field takes the following key word or key words, which specify whether the resource is fixed or moveable and whether it is discardable:

Option	Description
FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary in order to compact memory.
DISCARDABLE	Resource can be discarded if no longer needed.

The default is **MOVEABLE** and **DISCARDABLE**.

The *raw-data* field specifies one or more integers and strings. Integers can be in decimal, octal, or hexadecimal format.

The following example demonstrates the correct usage for the **RCDATA** statement:

```
resname RCDATA
BEGIN
   "Here is a data string\0", /* A string. Note: explicitly
                                 null-terminated */
   1024,                 /* int       */
   0x029a,               /* hex int   */
   0o733,                /* octal int  */
   "\07"                 /* octal byte */
END
```

Stringtable statement

Syntax stringtable [[*load-option*]] [[*mem-option*]]
BEGIN
stringID string
END

The **STRINGTABLE** statement defines one or more string resources for an application. String resources are simply null-terminated ASCII strings that can be loaded when needed from the executable file, using the **LoadString** function.

The optional *load-option* field takes a key word that specifies when the resource is to be loaded. It must be one of the following:

Option	Description
PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called. This is the default option.

The optional *mem-option* field takes the following key word or key words, which specify whether the resource is fixed or moveable and whether or not it is discardable:

Option	Description
FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved if necessary in order to compact memory.
DISCARDABLE	Resource can be discarded if no longer needed.

The default is **MOVEABLE** and **DISCARDABLE.**

The *stringID* field specifies an integer value that identifies the resource.

The *string* field specifies one or more ASCII strings, enclosed in double quotation marks. The string must be no longer than 255 characters and must occupy a single line in the source file. To add a carriage return to the string, use this character sequence: \012. For example, "Line one\012Line two" would define a string that would be displayed as follows:

Line one
Line two

Grouping strings in separate segments allows all related strings to be read in at one time and discarded together. When possible, an application should make the table moveable and discardable. The Resource Compiler allocates 16 strings per segment and uses the identifier value to determine which segment is to contain the string. Strings with the same upper 12 bits in their identifiers are placed in the same segment.

The following example demonstrates the correct usage of the STRINGTABLE statement:

```
#define IDS_HELLO    1
#define IDS_GOODBYE  2

STRINGTABLE
BEGIN
```

```
    IDS_HELLO,    "Hello"
    IDS_GOODBYE, "Goodbye"
END
```

Accelerators statement

Syntax *acctablename* ACCELERATORS
BEGIN
event, idvalue, [[*type*]] [[NOINVERT]] [[ALT]] [[SHIFT]] [[CONTROL]]
⋮
END

The **ACCELERATORS** statement defines one or more accelerators for an application. An accelerator is a key stroke defined by the application to give the user a quick way to perform a task. The **TranslateAccelerator** function is used to translate accelerator messages from the application queue into WM_COMMAND or WM_SYSCOMMAND messages.

The *acctablename* field specifies either a unique name or an integer value that identifies the resource.

The *event* field specifies the key stroke to be used as an accelerator. It can be any one of the following:

Character	Description
"char"	A single ASCII character enclosed in double quotes. The character can be preceded by a caret (^), meaning that the character is a control character.
ASCII character	An integer value representing an ASCII character. The *type* field must be **ASCII**.
Virtual key character	An integer value representing a virtual key. The virtual key for alphanumeric keys can be specified by placing the uppercase letter or number in double quotation marks (for example, "9" or "C"). The *type* field must be **VIRTKEY**.

The *idvalue* field specifies an integer value that identifies the accelerator.

The *type* field is required only when *event* is an ASCII character or a virtual key character. The *type* field specifies either **ASCII** or **VIRTKEY**; the integer value of *event* is interpreted accordingly. When **VIRTKEY** is specified and the *event* field contains a string, the *event* field must be uppercase.

The **NOINVERT** option, if given, means that no top-level menu item is highlighted when the accelerator is used. This is useful when defining accelerators for actions such as scrolling that do not correspond to a menu item. If **NOINVERT** is omitted, a top-level menu item will be highlighted (if possible) when the accelerator is used.

The **ALT** option, if given, causes the accelerator to be activated only if the ALT key is down.

The **SHIFT** option, if given, causes the accelerator to be activated only if the SHIFT key is down.

The **CONTROL** option, if given, defines the character as a control character (the accelerator is only activated if the CONTROL key is down). This has the same effect as using a caret (^) before the accelerator character in the *event* field.

The **ALT, SHIFT,** and **CONTROL** options apply only to virtual keys.

The following example demonstrates the correct usage of accelerator keys:

```
1 ACCELERATORS
BEGIN
  "^C",  IDDCLEAR        ; control C
  "K",   IDDCLEAR        ; shift K
  "k",   IDDELLIPSE, ALT ; alt K
  98,    IDDRECT, ASCII  ; b
  66,    IDDSTAR, ASCII  ; B (shift b)
  "g",   IDDRECT         ; g
  "G",   IDDSTAR         ; G (shift G)
  VK_F1, IDDCLEAR, VIRTKEY                  ; F1
  VK_F1, IDDSTAR,  CONTROL, VIRTKEY         ; control F1
  VK_F1, IDDELLIPSE,  SHIFT, VIRTKEY        ; shift F1
  VK_F1, IDDRECT,  ALT, VIRTKEY             ; alt F1
  VK_F2, IDDCLEAR, ALT, SHIFT,     VIRTKEY  ; alt shift F2
  VK_F2, IDDSTAR,  CONTROL, SHIFT, VIRTKEY  ; ctrl shift F2
  VK_F2, IDDRECT,  ALT, CONTROL,   VIRTKEY  ; alt control F2
END
```

Menu statement

Syntax *menuID* MENU [[*load-option*]] [[*mem-option*]]
BEGIN
item-definitions
END

The **MENU** statement defines the contents of a menu resource. A menu resource is a collection of information that defines the appearance and function of an application menu. A menu is a special input tool that lets a user select commands from a list of command names.

The *menuID* field specifies a name or number used to identify the menu resource.

The optional *load-option* field takes a key word that specifies when the resource is to be loaded. It must be one of the following:

Option	Description
PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called. This is the default option.

The optional *mem-option* field takes the following key word or key words, which specify whether the resource is fixed or moveable and whether it is discardable:

Option	Description
FIXED	Resource remains at a fixed memory location.
MOVEABLE	Resource can be moved to compact memory.
DISCARDABLE	Resource can be discarded if no longer needed.

The default is **MOVEABLE** and **DISCARDABLE**.

The *item-definition* field specifies special resource statements that define the items in the menu. These statements are defined in the following sections. The following is an example of a complete **MENU** statement:

```
sample MENU
BEGIN
     MENUITEM "&Soup", 100
     MENUITEM "S&alad", 101
     POPUP "&Entree"
     BEGIN
          MENUITEM "&Fish", 200
          MENUITEM "&Chicken", 201, CHECKED
          POPUP "&Beef"
          BEGIN
               MENUITEM "&Steak", 301
               MENUITEM "&Prime Rib", 302
          END
     END
     MENUITEM "&Dessert", 103
END
```

Item-definition statements

The **MENUITEM** and **POPUP** statements are used in the *item-definition* section of a **menu** statement to define the names and attributes of the actual menu items. Any number of statements can be given; each defines a unique item. The order of the statements defines the order of the menu items.

The **MENUITEM** and **POPUP** statements can be used only within an *item-definition* section of a **MENU** statement.

MENUITEM

Syntax MENUITEM *text, result,* [[*optionlist*]]

This optional statement defines a menu item.

The *text* field takes an ASCII string, enclosed in double quotation marks, that specifies the name of the menu item.

The string can contain the escape characters **\t** and **\a.** The **\t** character inserts a tab in the string and is used to align text in columns. Tab characters should be used only in pop-up menus, not in menu bars. (See the following section for information on pop-up menus.) The **\a** character aligns all text that follows it flush right to the menu bar or pop-up menu.

To insert a double quotation mark (") in the string, use two double quotation marks ("").

To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. This will cause the letter to appear underlined in the control and to function as the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *result* field takes an integer value that specifies the result generated when the user selects the menu item. Menu-item results are always integers; when the user clicks the menu-item name, the result is sent to the window that owns the menu.

The optional *optionlist* field takes one or more predefined menu options, separated by commas or spaces, that specify the appearance of the menu item. The menu options are as follows:

Option	Description
CHECKED	Item has a checkmark next to it.
GRAYED	Item name is initially inactive and appears on the menu in gray or a lightened shade of the menu-text color.
HELP	Item has a vertical separator to its left.
INACTIVE	Item name is displayed, but it cannot be selected.
MENUBARBREAK	Same as MF_**MENU**BREAK except that for pop-up menus, it separates the new column from the old column with a vertical line.
MENUBREAK	Places the menu item on a new line for static menu-bar items. For pop-up menus, places the menu item in a new column, with no dividing line between the columns.

The **INACTIVE** and **GRAYED** options cannot be used together.

The following example demonstrates the correct usage of the MENUITEM statement:

```
MENUITEM  "&Alpha", 1, CHECKED, GRAYED
MENUITEM  "&Beta", 2
```

POPUP

Syntax POPUP *text*, [[*optionlist*]]
BEGIN
item-definitions
END

This statement marks the beginning of the definition of a pop-up menu. A pop-up menu (which is also known as a drop-down menu) is a special menu item that displays a sublist of menu items when it is selected.

The *text* field takes an ASCII string, enclosed in double quotation marks, that specifies the name of the pop-up menu.

The optional *optionlist* field takes one or more predefined menu options that specify the appearance of the menu item. The menu options are as follows:

Option	Description
CHECKED	Item has a checkmark next to it. This option is not valid for a top-level pop-up menu.
GRAYED	Item name is initially inactive and appears on the menu in gray or a lightened shade of the menu-text color.
INACTIVE	Item name is displayed, but it cannot be selected.

MENUBARBREAK	Same as MF_**MENU**BREAK except that for pop-up menus, it separates the new column from the old column with a vertical line.
MENUBREAK	Places the menu item on a new line for static menu-bar items. For pop-up menus, places the menu item in a new column, with no dividing line between the columns.

The options can be combined using the bitwise OR operator. The **INACTIVE** and **GRAYED** options cannot be used together.

The *item-definitions* field can specify any number of **MENUITEM** or **POPUP** statements. As a result, any pop-up menu item can display another pop-up menu.

The following example demonstrates the correct usage of the **POPUP** statement:

```
chem MENU
BEGIN

POPUP "&Elements"
BEGIN
     MENUITEM "&Oxygen", 200
     MENUITEM "&Carbon", 201, CHECKED
     MENUITEM "&Hydrogen", 202
     MENUITEM "&Sulfur", 203
     MENUITEM "Ch&lorine", 204
END

POPUP "&Compounds"
BEGIN
     POPUP "&Sugars"
     BEGIN
        MENUITEM "&Glucose", 301
        MENUITEM "&Sucrose", 302, CHECKED
        MENUITEM "&Lactose", 303, MENUBREAK
        MENUITEM "&Fructose", 304
     END

     POPUP "&Acids"
     BEGIN
          "&Hydrochloric", 401
          "&Sulfuric", 402
     END

END

END
```

MENUITEM SEPARATOR

Syntax MENUITEM SEPARATOR

This special form of the **MENUITEM** statement creates an inactive menu item that serves as a dividing bar between two active menu items in a pop-up menu.

The following demonstrates the correct usage of the MENUITEM SEPARATOR statement:

```
MENUITEM "&Roman", 206
MENUITEM SEPARATOR
MENUITEM "&20 Point", 301
```

DIALOG statement

The **DIALOG** statement defines a template that can be used by an application to create dialog boxes.

Syntax *nameID* DIALOG [[*load-option*]] [[*mem-option*]] *x*, *y*, *width*, *height*
[[*option-statements*]]
BEGIN
control-statements
END

This statement marks the beginning of a **DIALOG** template. It defines the name of the dialog box, the memory and load options, the box's starting location on the display screen, and the box's width and height.

The *nameID* field specifies either a unique name or an integer value that identifies the resource.

The optional *load-option* field takes a key word that specifies when the resource is to be loaded. It must be one of the following:

Option	Description
PRELOAD	Resource is loaded immediately.
LOADONCALL	Resource is loaded when called. This is the default option.

The optional *mem-option* field takes the following key word or key words, which specify whether the resource is fixed or moveable and whether it is discardable:

FIXED	Resource remains at a fixed memory location.

MOVEABLE Resource can be moved if necessary in order to compact memory. This is the default option.

DISCARDABLE Resource can be discarded if no longer needed.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the dialog box. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The exact meaning of the coordinates depends on the style defined by the **STYLE** option statement. For child-style dialog boxes, the coordinates are relative to the origin of the parent window, unless the dialog box has the style DS_ABSALIGN; in that case, the coordinates are relative to the origin of the display screen.

The *width* and *height* fields take integer values that specify the width and height of the box. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The option and control statements are described in the following sections.

The following demonstrates the correct usage of the **DIALOG** statement:

```
#include "WINDOWS.H"
errmess DIALOG  10, 10, 300, 110
STYLE WS_POPUP|WS_BORDER
CAPTION "Error!"
BEGIN
    CTEXT "Select One:", 1, 10, 10, 280, 12
    RADIOBUTTON "&Retry", 2, 75, 30, 60, 12
    RADIOBUTTON "&Abort", 3, 75, 50, 60, 12
    RADIOBUTTON "&Ignore", 4, 75, 80, 60, 12
END
```

Comments

Do not use the WS_CHILD style with a modal dialog box. The **DialogBox** function always disables the parent/owner of the newly-created dialog box. When a parent window is disabled, its child windows are implicitly disabled. Since the parent window of the child-style dialog box is disabled, the child-style dialog box is too.

If a dialog box has the DS_ABSALIGN style, the dialog coordinates for its upper-left corner are relative to the screen origin instead of to the upper-left corner of the parent window. You would typically use this style when you wanted the dialog box to start in a specific part of the display no matter where the parent window may be on the screen.

The name **DIALOG** can also be used as the class-name parameter to the **CreateWindow** function in order to create a window with dialog-box attributes.

Dialog option statements

The dialog option statements, given in the *option-statements* section of the **DIALOG** statement, define special attributes of the dialog box, such as its style, caption, and menu. The option statements are optional. If the application does not supply a particular option statement, the dialog box is given default attributes for that option. Dialog option statements include the following:

- **STYLE**
- **CAPTION**
- **MENU**
- **CLASS**
- **FONT**

The option statements are discussed individually in the following sections.

STYLE

Syntax STYLE *style*

This optional statement defines the window style of the dialog box. The window style specifies whether the box is a pop-up or a child window. The default style has the following attributes:

WS_POPUP
WS_BORDER
WS_SYSMENU

The *style* field takes an integer value or predefined name that specifies the window style. It can be any of the window styles defined in Table 8.1, "Window styles."

Comments If the predefined names are used, the #**include** directive must be used so that the WINDOWS.H file will be included in the resource script.

Table 8.1
Window styles

Style	**Meaning**
DS_LOCALEDIT	Specifies that edit controls in the dialog box will use memory in the application's data segment. By default, all edit controls in dialog boxes use memory outside the

Table 8.1: Window styles (continued)

	application's data segment. This feature can be suppressed by adding the DS_LOCALEDIT flag to the STYLE command for the dialog box. If this flag is not used, EM_GETHANDLE and EM_SETHANDLE messages must not be used since the storage for the control is not in the application's data segment. This feature does not affect edit controls created outside of dialog boxes.
DS_MODALFRAME	Creates a dialog box with a modal dialog-box frame that can be combined with a title bar and system menu by specifying the WS_CAPTION and WS_SYSMENU styles.
DS_NOIDLEMSG	Suppresses WM_ENTERIDLE messages that Windows would otherwise send to the owner of the dialog box while the dialog box is displayed.
DS_SYSMODAL	Creates a system-modal dialog box.
WS_BORDER	Creates a window that has a border.
WS_CAPTION	Creates a window that has a title bar (implies WS_BORDER).
WS_CHILD	Creates a child window. It cannot be used with WS_POPUP.
WS_CHILDWINDOW	Creates a child window that has the style WS_CHILD.
WS_CLIPCHILDREN	Excludes the area occupied by child windows when drawing within the parent window. Used when creating the parent window.
WS_CLIPSIBLINGS	Clips child windows relative to each other; that is, when a particular child window receives a WP_PAINT message, this style clips all other top-level child windows out of the region of the child window to be updated. (If WS_CLIPSIBLINGS is not given and child windows overlap, it is possible, when drawing in the client area of a child window, to draw in the client area of a neighboring child window.) For use with WS_CHILD only.
WS_DISABLED	Creates a window that is initially disabled.
WS_DLGFRAME	Creates a window with a modal dialog-box frame but no title.
WS_GROUP	Specifies the first control of a group of controls in which the user can move from one control to the next by using the arrow keys. All controls defined with the WS_GROUP style after the first control belong to the same group. The next control with the WS_GROUP style ends the style group and starts the next group (i.e., one group ends where the next begins). This style is valid only for controls.
WS_HSCROLL	Creates a window that has a horizontal scroll bar.
WS_ICONIC	Creates a window that is initially iconic. For use with WS_OVERLAPPED only.
WS_MAXIMIZE	Creates a window of maximum size.
WS_MAXIMIZEBOX	Creates a window that has a Maximize box.
WS_MINIMIZE	Creates a window of minimum size.
WS_MINIMIZEBOX	Creates a window that has a Minimize box.

Table 8.1: Window styles (continued)

WS_OVERLAPPED	Creates an overlapped window. An overlapped window has a caption and a border.
WS_OVERLAPPEDWINDOW	Creates an overlapped window having the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles.
WS_POPUP	Creates a pop-up window. It cannot be used with WS_CHILD.
WS_POPUPWINDOW	Creates a pop-up window that has the styles WS_POPUP, WS_BORDER, and WS_SYSMENU. The WS_CAPTION style must be combined with the WS_POPUPWINDOW style to make the system menu visible.
WS_SIZEBOX	Creates a window that has a size box. Used only for windows with a title bar or with vertical and horizontal scroll bars.
WS_SYSMENU	Creates a window that has a System-menu box in its title bar. Used only for windows with title bars. If used with a child window, this style creates a Close box instead of a System-menu box.
WS_TABSTOP	Specifies one of any number of controls through which the user can move by using the TAB key. The TAB key moves the user to the next control specified by the WS_TABSTOP style. This style is valid only for controls.
WS_THICKFRAME	Creates a window with a thick frame that can be used to size the window.
WS_VISIBLE	Creates a window that is initially visible. This applies to overlapping and pop-up windows. For overlapping windows, the *y* parameter is used as a **ShowWindow** function parameter.
WS_VSCROLL	Creates a window that has a vertical scroll bar.

CAPTION

Syntax CAPTION *captiontext*

This optional statement defines the dialog box's title. The title appears in the box's caption bar (if it has one).

The default caption is empty.

The *captiontext* field specifies an ASCII character string enclosed in double quotation marks.

The following example demonstrates the correct usage of the **CAPTION** statement:

```
CAPTION "Error!"
```

MENU

Syntax MENU *menuname*

This optional statement defines the dialog box's menu. If no statement is given, the dialog box has no menu.

The *menuname* field specifies the resource name or number of the menu to be used.

The following example demonstrates the correct usage of the **MENU** statement:

```
MENU errmenu
```

CLASS

Syntax CLASS *class*

This optional statement defines the class of the dialog box. If no statement is given, the Windows standard dialog class will be used as the default.

The *class* field specifies an integer or a string, enclosed in double quotation marks, that identifies the class of the dialog box. If the window procedure for the class does not process a message sent to it, it must call the **DefDlgProc** function to ensure that all messages are handled properly for the dialog box. A private class can use **DefDlgProc** as the default window procedure. The class must be registered with the **cbWndExtra** field of the **WNDCLASS** data structure set to DLGWINDOWEXTRA.

The following example demonstrates the correct usage of the **CLASS** statement:

```
CLASS "myclass"
```

Comments The **CLASS** statement should be used with special cases, since it overrides the normal processing of a dialog box. The **CLASS** statement converts a dialog box to a window of the specified class; depending on the class, this could give undesirable results. Do not use the predefined control class names with this statement.

FONT

Syntax FONT *pointsize, typeface*

This optional statement defines the font with which Windows will draw text in the dialog box. The font must have been previously loaded, either from WIN.INI or by calling **LoadFont**.

The *pointsize* field is an integer that specifies the size in points of the font.

The *typeface* field specifies an ASCII character string enclosed in double quotation marks that specifies the name of the typeface. This name must be identical to the name defined in the [fonts] section of WIN.INI.

The following example demonstrates the correct usage of the **FONT** statement:

```
FONT 12, "Helv"
```

Dialog control statements

The dialog control statements, given in the *control-statements* section of the **DIALOG** statement, define the attributes of the control windows that appear in the dialog box. A dialog box is empty unless one or more control statements are given. Control statements include the following:

- **LTEXT**
- **RTEXT**
- **CTEXT**
- **CHECKBOX**
- **PUSHBUTTON**
- **LISTBOX**
- **GROUPBOX**
- **DEFPUSHBUTTON**
- **RADIOBUTTON**
- **EDITTEXT**
- **COMBOBOX**
- **ICON**
- **SCROLLBAR**
- **CONTROL**

The control statements are discussed individually in the following sections. For more information on control classes and styles, see Tables 8.2, "Control classes," and 8.3, "Control styles."

LTEXT

Syntax LTEXT *text, id, x, y, width, height,* [[*style*]]

This statement defines a flush-left text control. It creates a simple rectangle that displays the given text flush-left in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP

These styles are described in Table 8.1, "Window styles." Styles can be combined using the bitwise OR operator.

Comments The *x, y, width,* and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **LTEXT** is SS_LEFT and WS_GROUP.

The following example demonstrates the correct usage of the **LTEXT** statement:

```
LTEXT "Enter Name:", 3, 10, 10, 40, 10
```

RTEXT

Syntax RTEXT *text, id, x, y, width, height,* [[*style*]]

This statement defines a flush-right text control. It creates a simple rectangle that displays the given text flush-right in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP

These styles are described in Table 8.1, "Window styles." Styles can be combined using the bitwise OR operator.

Comments The *x, y, width,* and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **RTEXT** is SS_RIGHT and WS_GROUP.

The following example demonstrates the correct usage of the **RTEXT** statement:

```
RTEXT "Number of Messages", 4, 30, 50, 100, 10
```

CTEXT

Syntax CTEXT *text, id, x, y, width, height,* [[*style*]]

This statement defines a centered text control. It creates a simple rectangle that displays the given text centered in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP

These styles are described in Table 8.1, "Window styles." Styles can be combined using the bitwise OR operator.

Comments The *x, y, width,* and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **CTEXT** is SS_CENTER and WS_GROUP.

The following example demonstrates the correct usage of the **CTEXT** statement:

```
CTEXT "Title", 3, 10, 50, 40, 10
```

CHECKBOX

Syntax CHECKBOX *text, id, x, y, width, height,* [[*style*]]

This statement defines a check-box control belonging to the BUTTON class. It creates a small rectangle (check box) that is highlighted when clicked. The given text is displayed just to the right of the check box. The control highlights the rectangle when the user clicks the mouse in it, and removes the highlight on the next click.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

□ WS_TABSTOP
□ WS_GROUP

These styles are described in Table 8.1, "Window styles."

In addition to these styles, the *style* field may contain any combination (or none) of the BUTTON-class styles described in Table 8.3, "Control styles." Styles can be combined using the bitwise OR operator.

Comments The *x, y, width,* and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **CHECKBOX** is BS_CHECKBOX and WS_TABSTOP.

The following example demonstrates the correct usage of the **CHECKBOX** statement:

```
CHECKBOX "Arabic", 3, 10, 10, 40, 10
```

PUSHBUTTON

Syntax PUSHBUTTON *text, id, x, y, width, height,* [[*style*]]

This statement defines a push-button control belonging to the BUTTON class. It creates a rectangle containing the given text. The control sends a message to its parent whenever the user clicks the mouse inside the rectangle.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_DISABLED
- WS_GROUP

These styles are described in Table 8.1, "Window styles."

In addition to these styles, the *style* field may contain any combination (or none) of the BUTTON-class styles described in Table 8.3, "Control styles." Styles can be combined using the bitwise OR operator.

Comments The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **PUSHBUTTON** is BS_PUSHBUTTON and WS_TABSTOP.

The following example demonstrates the correct usage of the **PUSHBUTTON** statement:

```
PUSHBUTTON "ON", 7, 10, 10, 20, 10
```

LISTBOX

Syntax LISTBOX *id*, *x*, *y*, *width*, *height*, [[*style*]]

This statement defines a list box belonging to the LISTBOX class. It creates a rectangle that contains a list of strings (such as filenames) from which the user can make selections.

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_BORDER
- WS_VSCROLL

These styles are described in Table 8.1, "Window styles."

In addition to these styles, the *style* field may contain any combination (or none) of the LISTBOX-class styles described in Table 8.3, "Control styles." Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **LISTBOX** is LBS_NOTIFY, WS_VSCROLL, and WS_BORDER.

For information on the recommended keys for use in list-box controls, see the *System Application Architecture, Common User Access: Advanced Interface Design Guide.*

The following example demonstrates the correct usage of the **LISTBOX** statement:

```
LISTBOX 666, 10, 10, 50, 54
```

GROUPBOX

Syntax

GROUPBOX *text*, *id*, *x*, *y*, *width*, *height*, [[*style*]]

This statement defines a group box belonging to the BUTTON class. It creates a rectangle that groups other controls together. The controls are grouped by drawing a border around them and displaying the given text in the upper-left corner.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. Selecting the mnemonic moves the input focus to the next control in the group, in the order set in the resource file. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_DISABLED

These styles are described in Table 8.1, "Window styles."

In addition to these styles, the *style* field may contain any combination (or none) of the BUTTON-class styles described in Table 8.3, "Control styles." Styles can be combined using the bitwise OR operator.

Comments

The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **GROUPBOX** is BS_GROUPBOX and WS_TABSTOP.

The following example demonstrates the correct usage of the **GROUPBOX** statement:

```
GROUPBOX "Output", 42, 10, 10, 30, 50
```

DEFPUSHBUTTON

Syntax

DEFPUSHBUTTON *text*, *id*, *x*, *y*, *width*, *height*, [[*style*]]

This statement defines a default push-button control that belongs to the BUTTON class. It creates a small rectangle with a bold outline that represents the default response for the user. The given text is displayed inside the button. The control highlights the button in the usual way when the user clicks the mouse in it and sends a message to its parent window.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP
- WS_DISABLED

These styles are described in Table 8.1, "Window styles."

In addition to these styles, the *style* field may contain any combination (or none) of the BUTTON-class styles described in Table 8.3, "Control styles." Styles can be combined using the bitwise OR operator.

Comments The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **DEFPUSHBUTTON** is BS_DEFPUSHBUTTON and WS_TABSTOP.

The following example demonstrates the correct usage of the **DEFPUSHBUTTON** statement:

```
DEFPUSHBUTTON "ON", 7, 10, 10, 20, 10
```

RADIOBUTTON

Syntax **RADIOBUTTON** *text, id, x, y, width, height, [[style]]*

This statement defines a radio-button control belonging to the BUTTON class. It creates a small circle that has the given text displayed just to its right. The control highlights the button when the user clicks the mouse in it and sends a message to its parent window. The control removes the highlight and sends a message on the next click.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks. To add a mnemonic to the text string, place the ampersand (&) ahead of the letter that will be the mnemonic. To use the ampersand as a character in a string, insert two ampersands (&&).

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP
- WS_DISABLED

These styles are described in Table 8.1, "Window styles."

In addition to these styles, the *style* field may contain any combination (or none) of the BUTTON-class styles described in Table 8.3, "Control styles." Styles can be combined using the bitwise OR operator.

Comments The *x, y, width,* and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **RADIOBUTTON** is BS_RADIOBUTTON and WS_TABSTOP.

The following example demonstrates the correct usage of the **RADIOBUTTON** statement:

```
RADIOBUTTON "AM 101", 10, 10, 10, 40, 10
```

EDITTEXT

Syntax EDITTEXT *id, x, y, width, height,* [[*style*]]

This statement defines an EDIT control belonging to the EDIT class. It creates a rectangular region in which the user can enter and edit text. The control displays a cursor when the user clicks the mouse in it. The user can then use the keyboard to enter text or edit the existing text. Editing keys include the BACKSPACE and DELETE keys. The user can also use the mouse to select characters to be deleted, or to select the place to insert new characters.

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP
- WS_VSCROLL
- WS_HSCROLL
- WS_DISABLED

These styles are described in Table 8.1, "Window styles."

In addition to these styles, the *style* field may contain any combination (or none) of the EDIT-class styles described in Table 8.3, "Control styles." Styles can be combined using the bitwise OR operator. The EDIT-class styles must not conflict with each other.

Comments

The *x*, *y*, *width*, and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **EDITTEXT** is WS_TABSTOP, ES_LEFT, and WS_BORDER.

Keyboard use is predefined for edit controls. Predefined keys are listed in the *System Application Architecture, Common User Access: Advanced Interface Design Guide.*

The following example demonstrates the correct usage of the **EDITTEXT** statement:

```
EDITTEXT  3, 10, 10, 100, 10
```

COMBOBOX

Syntax COMBOBOX *id, x, y, width, height,* [[*style*]]

This statement defines a combo box belonging to the COMBOBOX class. A combo box consists of either a static text field or edit field combined with a list box. The list box can be displayed at all times or pulled down by the user. If the combo box contains a static text field, the text field always displays the selection (if any) in the list-box portion of the combo box. If it uses an edit field, the user can type in the desired selection; the list box highlights the first item (if any) which matches what the user has entered in the edit field. The user can then select the item highlighted in the list box to complete the choice. In addition, the combo box can be owner-draw and of fixed or variable height.

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- WS_TABSTOP
- WS_GROUP
- WS_VSCROLL
- WS_DISABLED

These styles are described in Table 8.1, "Window styles."

In addition to these styles, the *style* field may contain any combination (or none) of the combo-box styles described in Table 8.3, "Control styles." Styles can be combined using the bitwise OR operator.

Comments The *x, y, width,* and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **COMBOBOX** is WS_TABSTOP and CBS_SIMPLE.

The following example demonstrates the correct usage of the **COMBOBOX** statement:

```
COMBOBOX 777, 10, 10, 50, 54, CBS_SIMPLE | WS_VSCROLL | WS_TABSTOP
```

ICON

Syntax

ICON *text, id, x, y, width, height,* [[*style*]]

This statement defines an icon control belonging to the STATIC class. It creates an icon displayed in the dialog box.

The *text* field specifies the name of an icon (not a filename) defined elsewhere in the resource file.

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

For the **ICON** statement, the *width* and *height* fields are ignored; the icon automatically sizes itself.

The optional *style* field allows only the SS_ICON style.

Comments

The *x, y, width,* and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

The default style for **ICON** is SS_ICON.

The following example demonstrates the correct usage of the **ICON** statement:

```
ICON "myicon" 901, 30, 30
```

SCROLLBAR

Syntax SCROLLBAR *id, x, y, width, height,* [[*style*]]

This statement defines a scroll-bar control belonging to the SCROLLBAR class. It is a rectangle that contains a scroll thumb and has direction arrows at both ends. The scroll-bar control sends a notification message to its parent whenever the user clicks the mouse in the control. The parent is responsible for updating the thumb position. Scroll-bar controls can be positioned anywhere in a window and used whenever needed to provide scrolling input.

The *id* field takes a unique integer value that identifies the control.

The *x* and *y* fields take integer values that specify the location of the upper-left corner of the control in dialog units relative to the origin of the dialog box. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

The optional *style* field can contain any combination (or none) of the following styles:

- □ WS_TABSTOP
- □ WS_GROUP
- □ WS_DISABLED

These styles are described in Table 8.1, "Window styles."

In addition to these styles, the *style* field may contain any combination (or none) of the SCROLLBAR-class styles described in Table 8.3, "Control styles." Styles can be combined using the bitwise OR operator.

Comments The *x, y, width,* and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field. The default style for SCROLLBAR is SBS_HORZ.

The following example demonstrates the correct usage of the **SCROLLBAR** statement:

```
SCROLLBAR 999, 25, 30, 10, 100
```

CONTROL

Syntax CONTROL text, id, class, style, x, y, width, height

This statement defines a user-defined control window.

The *text* field takes an ASCII string that specifies the text to be displayed. The string must be enclosed in double quotation marks.

The *id* field takes a unique integer value that identifies the control.

The *class* field takes a predefined name, character string, or integer value that defines the class. This can be any one of the control classes; for a list of the control classes, see Table 8.2, "Control classes." If the value is a predefined name supplied by the application, it must be an ASCII string enclosed in double quotation marks.

The *style* field takes a predefined name or integer value that specifies the style of the given control. The exact meaning of *style* depends on the *class* value. Tables 8.2, "Control classes," and 8.3, "Control styles," list the control classes and corresponding styles.

The *x* and *y* fields take integer values that specify the *x* and *y* coordinates of the upper-left corner of the control. The horizontal units are 1/4 of the dialog base width unit; the vertical units are 1/8 of the dialog base height unit. The current dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the dialog base units in pixels. The coordinates are relative to the origin of the dialog box.

The *width* and *height* fields take integer values that specify the width and height of the control. The width units are 1/4 of the dialog base width unit; the height units are 1/8 of the dialog base height unit.

Comments The *x, y, width,* and *height* fields can use the addition operator (+) for relative positioning. For example, "15 + 6" can be used for the *x* field.

Table 8.2 describes the six control classes:

Table 8.2
Control classes

Class	Description
BUTTON	A button control is a small rectangular child window that represents a "button" that the user can turn on or off by clicking it with the mouse. Button controls can be used alone or in groups, and can either be labeled or appear without text.

Table 8.2: Control classes (continued)

	Button controls typically change appearance when the user clicks them.
COMBOBOX	Combo-box controls consist of a selection field similar to an edit control plus a list box. The list box may be displayed at all times or may be dropped down when the user selects a "pop box" next to the selection field. Depending on the style of the combo box, the user can or cannot edit the contents of the selection field. If the list box is visible, typing characters into the selection box will cause the first list box entry which matches the characters typed to be highlighted. Conversely, selecting an item in the list box displays the selected text in the selection field.
EDIT	An edit control is a rectangular child window in which the user can enter text from the keyboard. The user selects the control, and gives it the input focus, by clicking the mouse inside it or pressing the TAB key. The user can enter text when the control displays a flashing caret. The mouse can be used to move the cursor and select characters to be replaced, or to position the cursor for inserting characters. The BACKSPACE key can be used to delete characters. Edit controls use the fixed-pitch font and display ANSI characters. They expand tab characters into as many space characters as are required to move the cursor to the next tab stop. Tab stops are assumed to be at every eighth character position.
LISTBOX	List-box controls consist of a list of character strings. The control is used whenever an application needs to present a list of names, such as filenames, that the user can view and select. The user can select a string by pointing to the string with the mouse and clicking a mouse button. When a string is selected, it is highlighted, and a notification message is passed to the parent window. A scroll bar can be used with a list-box control to scroll lists that are too long or too wide for the control window.
SCROLLBAR	A scroll-bar control is a rectangle that contains a scroll thumb and has direction arrows at both ends. The scroll bar sends a notification message to its parent whenever the user clicks the mouse in the control. The parent is responsible for updating the thumb position, if necessary. Scroll-bar controls have the same appearance and function as the scroll bars used in ordinary windows. But unlike scroll bars, scroll-bar controls can be positioned anywhere within a window and used whenever needed to provide scrolling input for a window. The scroll-bar class also includes size-box controls. A size-box control is a small rectangle that the user can expand to change the size of the window.
STATIC	Static controls are simple text fields, boxes, and rectangles that can be used to label, box, or separate other controls. Static controls take no input and provide no output.

Table 8.3 describes the control styles for each of the control classes:

Table 8.3 Control styles

Style	Description
BUTTON class	
BS_PUSHBUTTON	A small elliptical button containing the given text. The control sends a message to its parent whenever the user clicks the mouse inside the rectangle.
BS_DEFPUSHBUTTON	A small elliptical button with a bold border. This button represents the default user response. Any text is displayed within the button. Windows sends a message to the parent window when the user clicks the mouse in this button.
BS_CHECKBOX	A small rectangular button that can be checked; its border becomes bold when the user clicks the mouse in it. Any text appears to the right of the button.
BS_AUTOCHECKBOX	Identical to BS_CHECKBOX except that the button automatically toggles its state whenever the user clicks it.
BS_RADIOBUTTON	A small circular button whose border becomes bold when the user clicks the mouse in it. In addition, to make the border bold, Windows sends a message to the button's parent notifying it that a click occurred. On the next click, Windows makes the border normal again and sends another message.
BS_AUTORADIOBUTTON	Identical to BS_RADIOBUTTON except that when the button is checked, the application is notified with BN_CLICKED, and all other radio buttons in the group are unchecked.
BS_LEFTTEXT	Text appears on the left side of the radio button or check-box button. Use this style with BS_CHECKBOX, BS_3STATE, or BS_RADIOBUTTON styles.
BS_3STATE	Identical to BS_CHECKBOX except that a button can be grayed as well as checked or unchecked. The grayed state is typically used to show that a check box has been disabled.
BS_AUTO3STATE	Identical to BS_3STATE except that the button automatically toggles its state when the user clicks it.
BS_GROUPBOX	A rectangle into which other buttons are grouped. Any text is displayed in the rectangle's upper-left corner.
BS_OWNERDRAW	An owner-draw button. The parent window is notified when the button is clicked. Notification includes a request to paint, invert, and disable the button.

Table 8.3: Control styles (continued)

COMBOBOX class	
CBS_SIMPLE	Displays the list box at all times. The current selection in the list box is displayed in the edit control.
CBS_DROPDOWN	Is similar to CBS_SIMPLE, except that the list box is not displayed unless the user selects an icon next to the selection field.
CBS_DROPDOWNLIST	Is similar to CBS_DROPDOWN, except that the edit control is replaced by a static text item which displays the current selection in the list box.
CBS_OWNERDRAWFIXED	Specifies a fixed-height owner-draw combo box. The owner of the list box is responsible for drawing its contents; the items in the list box are all the same height.
CBS_OWNERDRAWVARIABLE	Specifies a variable-height owner-draw combo box. The owner of the list box is responsible for drawing its contents; the items in the list box can have different heights.
CBS_AUTOHSCROLL	Scrolls the text in the edit control to the right when the user types a character at the end of the line. If this style is not set, only text which fits within the rectangular boundary is allowed.
CBS_SORT	Sorts strings entered into the list box.
CBS_HASSTRINGS	Specifies an owner-draw combo box that contains items consisting of strings. The combo box maintains the memory and pointers for the strings so that the application can use the LB_GETTEXT message to retrieve the text for a particular item.
CBS_OEMCONVERT	Text entered in the combo box edit control is converted from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the **AnsiToOem** function to convert an ANSI string in the combo box to OEM characters. This style is most useful for combo boxes that contain filenames and applies only to combo boxes created with the CBS_SIMPLE or CBS_DROPDOWN styles.
EDIT class	
ES_LEFT	Flush-left text.
ES_CENTER	Centered text. This style is valid in multiline edit controls only.
ES_RIGHT	Flush-right text. This style is valid in multiline edit controls only.

Table 8.3: Control styles (continued)

ES_LOWERCASE	Lowercase edit control. An edit control with this style converts all characters to lowercase as they are typed into the edit control.
ES_UPPERCASE	Uppercase edit control. An edit control with this style converts all characters to uppercase as they are typed into the edit control.
ES_PASSWORD	Password edit control. An edit control with this style displays all characters as an asterisk (*) as they are typed into the edit control. An application can use the EM_SETPASSWORDCHAR message to change the character that is displayed.
ES_MULTILINE	Multiple-line edit control. (The default is single-line.) If the ES_AUTOVSCROLL style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the ENTER key. (This is actually the carriage-return character, which the edit control expands to a carriagereturn/line-feed combination. A line feed is not treated the same as a carriage return.) If ES_AUTOVSCROLL is not given, the edit control shows as many lines as possible and beeps if the user presses ENTER when no more lines can be displayed. If the ES_AUTOHSCROLL style is specified, the multiple-line edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press the ENTER key. If ES_AUTO-HSCROLL is not given, the control automatically wraps words to the beginning of the next line when necessary; a new line is also started if the user presses ENTER. The position of the wordwrap is determined by the window size. If the window size changes, the wordwrap position changes, and the text is redisplayed. Multiple-line edit controls can have scroll bars. An edit control with scroll bars processes its own scroll-bar messages. Edit controls without scroll bars scroll as described above, and process any scroll messages sent by the parent window.
ES_AUTOVSCROLL	Text is automatically scrolled up one page when the user presses the ENTER key on the last line.
ES_AUTOHSCROLL	Text is automatically scrolled to the right by 10 characters when the user types a character at the end of the line. When the user presses

Table 8.3: Control styles (continued)

	the ENTER key, the control scrolls all text back to position 0.
ES_NOHIDESEL	Normally, an edit control hides the selection when the control loses the input focus, and inverts the selection when the control receives the input focus. Specifying ES_NOHIDESEL overrides this default action.
ES_OEMCONVERT	Text entered in the edit control is converted from the ANSI character set to the OEM character set and then back to ANSI. This ensures proper character conversion when the application calls the **AnsiToOem** function to convert an ANSI string in the edit control to OEM characters. This style is most useful for edit controls that contain filenames.
LISTBOX class	
LBS_STANDARD	Strings in the list box are sorted alphabetically and the parent window receives an input message whenever the user clicks or double-clicks a string. The list box contains borders on all sides.
LBS_EXTENDEDSEL	The user can select multiple items using the mouse with the SHIFT and/or the CONTROL key or special key combinations.
LBS_HASSTRINGS	An owner-draw list box contains items consisting of strings. The list box maintains the memory and pointers for the strings so the application can use the LB_GETTEXT message to retrieve the text for a particular item.
LBS_NOTIFY	The parent receives an input message whenever the user clicks or double-clicks a string.
LBS_MULTIPLESEL	The string selection is toggled each time the user clicks or double-clicks the string. Any number of strings can be selected.
LBS_MULTICOLUMN	The list box contains multiple columns. The list box can be scrolled horizontally. The LB_SETCOLUMNWIDTH message sets the width of the columns.
LBS_NOINTEGRALHEIGHT	The size of the list box is exactly the size specified by the application when it created the list box. Normally, Windows sizes a list box so that the list box does not display partial items.
LBS_SORT	The strings in the list box are sorted alphabetically.
LBS_NOREDRAW	The list-box display is not updated when changes are made. This style can be changed

	at any time by sending a WM_SETREDRAW message.
LBS_OWNERDRAWFIXED	The owner of the list box is responsible for drawing its contents; the items in the list box are all the same height.
LBS_OWNERDRAWVARIABLE	The owner of the list box is responsible for drawing its contents; the items in the list box are variable in height.
LBS_USETABSTOPS	The list box is able to recognize and expand tab characters when drawing its strings. The default tab positions are set at every 32 dialog units. (A dialog unit is a horizontal or vertical distance. One horizontal dialog unit is equal to 1/4 of the current dialog base width unit. The dialog base units are computed from the height and width of the current system font. The **GetDialogBaseUnits** function returns the size of the dialog base units in pixels.)
LBS_WANTKEYBOARDINPUT	The owner of the list box receives WM_VKEYTOITEM or WM_CHARTOITEM messages whenever the user presses a key when the list box has input focus. This allows an application to perform special processing on the keyboard input.
SCROLLBAR class	
SBS_VERT	Vertical scroll bar. If neither SBS_RIGHTALIGN nor SBS_LEFTALIGN is specified, the scroll bar has the height, width, and position given in the **CreateWindow** function.
SBS_RIGHTALIGN	Used with SBS_VERT. The right edge of the scroll bar is aligned with the right edge of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The scroll bar has the default width for system scroll bars.
SBS_LEFTALIGN	Used with SBS_VERT. The left edge of the scroll bar is aligned with the left edge of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The scroll bar has the default width for system scroll bars.
SBS_HORZ	Horizontal scroll bar. If neither SBS_BOTTOMALIGN nor SBS_TOPALIGN is specified, the scroll bar has the height, width, and position given in the **CreateWindow** function.
SBS_TOPALIGN	Used with SBS_HORZ. The top edge of the scroll bar is aligned with the top edge of the

Table 8.3: Control styles (continued)

	rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The scroll bar has the default height for system scroll bars.
SBS_BOTTOMALIGN	Used with SBS_HORZ. The bottom edge of the scroll bar is aligned with the bottom edge of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The scroll bar has the default height for system scroll bars.
SBS_SIZEBOX	Size box. If neither SBS_SIZEBOXBOTTOMRIGHTALIGN nor SBS_SIZEBOXTOPLEFTALIGN is specified, the size box has the height, width, and position given in the **CreateWindow** function.
SBS_SIZEBOXTOPLEFTALIGN	Used with SBS_SIZEBOX. The top-left corner of the size box is aligned with the top-left corner of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The size box has the default size for system size boxes.
SBS_SIZEBOXBOTTOMRIGHTALIGN	Used with SBS_SIZEBOX. The bottom-right corner of the size box is aligned with the bottom-right corner of the rectangle specified by the *x*, *y*, *width*, and *height* values given in the **CreateWindow** function. The size box has the default size for system size boxes.
STATIC class	
SS_LEFT	A simple rectangle displaying the given text flush left in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.
SS_CENTER	A simple rectangle displaying the given text centered in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.
SS_RIGHT	A simple rectangle displaying the given text flush right in the rectangle. The text is formatted before it is displayed. Words that would extend past the end of a line are automatically wrapped to the beginning of the next line.
SS_LEFTNOWORDWRAP	A simple rectangle displaying the given text flush left in the rectangle. Tabs are expanded, but words are not wrapped. Text that extends past the end of a line is clipped.

Table 8.3: Control styles (continued)

SS_SIMPLE	Designates a simple rectangle and displays a single line of text flush-left in the rectangle. The line of text cannot be shortened or altered in any way. (The control's parent window or dialog box must not process the WM_CTLCOLOR message.)
SS_NOPREFIX	Unless this style is specified, windows will interpret any "&" characters in the control's text to be accelerator prefix characters. In this case, the "&" is removed and the next character in the string is underlined. If a static control is to contain text where this feature is not wanted, SS_NOPREFIX may be added. This static-control style may be included with any of the defined static controls. You can combine SS_NOPREFIX with other styles by using the bitwise OR operator. This is most often used when filenames or other strings that may contain an "&" need to be displayed in a static control in a dialog box.
SS_ICON	An icon displayed in the dialog box. The given text is the name of an icon (not a filename) defined elsewhere in the resource file. For the **ICON** statement, the *width* and *height* parameters in the **CreateWindow** function are ignored; the icon automatically sizes itself.
SS_BLACKRECT	A rectangle filled with the color used to draw window frames. This color is black in the default Windows color scheme.
SS_GRAYRECT	A rectangle filled with the color used to fill the screen background. This color is gray in the default Windows color scheme.
SS_WHITERECT	A rectangle filled with the color used to fill window backgrounds. This color is white in the default Windows color scheme.
SS_BLACKFRAME	Box with a frame drawn with the same color as window frames. This color is black in the default Windows color scheme.
SS_GRAYFRAME	Box with a frame drawn with the same color as the screen background (desktop). This color is gray in the default Windows color scheme.
SS_WHITEFRAME	Box with a frame drawn with the same color as window backgrounds. This color is white in the default Windows color scheme.
SS_USERITEM	User-defined item.

Directives

The resource directives are special statements that define actions to be performed on the script file before it is compiled. The directives can assign values to names, include the contents of files, and control compilation of the script file.

The resource directives are identical to the directives used in the C programming language.

#include statement

Syntax #include *filename*

This directive copies the contents of the file specified by *filename* into your resource script before the Resource Compiler processes the script. It replaces the **rcinclude** directive of versions prior to Windows 3.0.

The *filename* field is an ASCII string that specifies the DOS filename of the file to be included, using the same syntax as the C-language preprocessor #**include** directive. A forward slash (/) can be used instead of a backslash (for example, "root/sub"). If the filename has the .H or .C extension, only the preprocessor directives in the file are processed. Otherwise, this directive processes the entire contents of the file.

The following example demonstrates the correct usage of the #**include** statement:

```
#include "WINDOWS.H"

PenSelect MENU

BEGIN
    Menuitem "&Black pen", BLACK_PEN
END
```

#define statement

Syntax #define *name value*

This directive assigns the given *value* to *name*. All subsequent occurrences of *name* are replaced by *value*.

The *value* field takes any integer value, character string, or line of text.

The following example demonstrates the correct usage of the **#define** statement:

```
#define     nonzero       1
#define     USERCLASS   "MyControlClass"
```

#undef statement

Syntax #undef *name*

This directive removes the current definition of *name*. All subsequent occurrences of *name* are processed without replacement.

The following example demonstrates the correct usage of the **#undef** statement:

```
#undef     nonzero
#undef     USERCLASS
```

#ifdef statement

Syntax #ifdef *name*

This directive carries out conditional compilation of the resource file by checking the specified *name*. If *name* has been defined using a **#define** directive, **#ifdef** directs the Resource Compiler to continue with the statement immediately after **#ifdef**. If *name* has not been defined, **#ifdef** directs the compiler to skip all statements up to the next **#endif** directive.

The following example demonstrates the correct usage of the**#ifdef** statement:

```
#ifdef Debug
errbox BITMAP errbox.bmp
#endif
```

#ifndef statement

Syntax #ifndef *name*

This directive carries out conditional compilation of the resource file by checking the specified *name*. If *name* has not been defined or if its definition has been removed using the **#undef** directive, **#ifndef** directs the Resource Compiler to continue processing statements up to the next **#endif**, **#else**, or **#elif** directive, and then to skip to the statement after **#endif**. If *name* is defined, **#ifndef** directs the compiler to skip to the next **#endif**, **#else**, or **#elif** directive.

The following example demonstrates the correct usage of the **#ifndef** statement:

```
#ifndef Optimize
errbox BITMAP errbox.bmp
#endif
```

#if statement

Syntax #if *constant-expression*

This directive carries out conditional compilation of the resource file by checking the specified *constant-expression*. If *constant-expression* is nonzero, **#if** directs the Resource Compiler to continue processing statements up to the next **#endif**, **#else**, or **#elif** directive, then skip to the statement after **#endif**. If *constant-expression* is zero, **#if** directs the compiler to skip to the next **#endif**, **#else**, or **#elif** directive.

The *constant-expression* field specifies a defined name, an integer constant, or an expression consisting of names, integers, and arithmetical and relational operators.

The following example demonstrates the correct usage of the **#if** statement:

```
#if Version<3
errbox BITMAP errbox.bmp
#endif
```

#elif statement

Syntax #elif *constant-expression*

This directive marks an optional clause of a conditional compilation block defined by an **#ifdef**, **#ifndef**, or **#if** directive. The **#elif** directive carries out conditional compilation of the resource file by checking the specified *constant-expression*. If *constant-expression* is nonzero, **#elif** directs the Resource Compiler to continue processing statements up to the next **#endif**, **#else**, or **#elif** directive, then skip to the statement after **#endif**. If *constant-expression* is zero, **#elif** directs the compiler to skip to the next **#endif**, **#else**, or **#elif** directive. Any number of **#elif** directives can be used in a conditional block.

The *constant-expression* field specifies a defined name, an integer constant, or an expression consisting of names, integers, and arithmetical and relational operators.

The following demonstrates the correct usage of the **#elif** statement:

```
#if Version<3
errbox BITMAP errbox.bmp
#elif Version<7
errbox BITMAP userbox.bmp
#endif
```

#else statement

Syntax #else

This directive marks an optional clause of a conditional compilation block defined by an **#ifdef**, **#ifndef**, or **#if** directive. The **#else** directive must be the last directive before **#endif.**

The following example demonstrates the correct usage of the **#else** statement:

```
#ifdef Debug
errbox BITMAP errbox.bmp
#else
errbox BITMAP userbox.bmp
#endif
```

#endif statement

Syntax #endif

This directive marks the end of a conditional compilation block defined by an #if or #ifdef directive. One #if or #endif is required for each **#ifdef** directive.

CHAPTER

9

File formats

This chapter describes the file formats used to create, execute, and supply data to Microsoft Windows applications. These files include the following:

- Bitmap files
- Icon resource files
- Cursor resource files
- Clipboard files
- Metafiles

Bitmap file formats

Windows version 3.0 bitmap files store a bitmap in a device-independent format which allows Windows to display the bitmap on any device. In this case, the term "device independent" means that the bitmap specifies pixel color in a form independent of the method used by any particular device to represent color. The assumed file extension of a Windows device-independent bitmap file is .BMP.

Each bitmap file contains a **BITMAPFILEHEADER** data structure immediately followed by a single, device-independent bitmap (DIB) consisting of a **BITMAPINFO** data structure and an array of bytes that defines the bitmap bits.

Windows version 3.0 also reads bitmap files in the format read by Microsoft OS/2 Presentation Manager version 1.2. These files consist of a **BITMAPFILEHEADER** data structure immediately followed by a

BITMAPCOREINFO data structure. Following this data structure is an array of bytes that defines the bitmap bits.

See Chapter 7, "Data types and structures," for information on the **BITMAPFILEHEADER, BITMAPCOREINFO** and **BITMAPINFO** data structures.

Icon resource file format

An icon resource file (with the .ICO file extension) can be device independent both for color and resolution.

Icon resource files can contain multiple device-independent bitmaps defining the icon image, one for each targeted display-device resolution. Windows detects the resolution of the current display and matches it against the *x* and *y* pixel-size values specified for each version of the image. If Windows determines that there is an exact match between an icon image and the current device, then it uses the matching image; otherwise, it selects the closest match and stretches the image to the proper size.

If an icon resource file contains more than one image for a particular resolution, Windows uses the icon image that most closely matches the color capabilities of the current display device. If no image exists which exactly matches the device capabilities, Windows selects the image which has the greatest number of colors without exceeding the number of display-device colors. If all images exceed the color capabilities of the current display device, then Windows uses the icon image with the least number of colors.

The icon resource file contains a header structure at the beginning of the file which identifies the type and number of icon images contained in the file. The following shows the format of this header:

Field	Type/Description
icoReserved	**WORD** Is reserved and must be set to 0.
icoResourceType	**WORD** Specifies the type of resource contained in the file. For an icon resource, this field must be 1.
icoResourceCount	**WORD** Specifies the number of images contained in the file.

The resource directory follows this header. The resource directory consists of one or more arrays of resource descriptors. The **icoResorceCount** specifies the number of arrays. This list shows the format of the array:

Field	Type/Description
Width	**BYTE** Specifies the width in pixels of this form of the icon image. Acceptable values are 16, 32, or 64.
Height	**BYTE** Specifies the height in pixels of this form of the icon image. Acceptable values are 16, 32, or 64.
ColorCount	**BYTE** Specifies the number of colors in this form of the icon image. Acceptable values are 2, 8, or 16.
Reserved	**BYTE** Reserved for future use.
Reserved	**WORD** Reserved for future use.
Reserved	**WORD** Reserved for future use.
icoDIBSize	**DWORD** Specifies in bytes the size of the pixel array for this form of the icon image.
icoDIBOffset	**DWORD** Specifies the offset in bytes from the beginning of the file to the device-independent bitmap for this form.

Icons can be in color. To achieve transparency, the DIB for each icon will consist of two parts:

1. A color bitmap which supplies the XOR mask for the icon.
2. A monochrome bitmap which provides the AND mask that defines the transparent portion of the icon.

The monochrome bitmap does not contain a DIB header, but instead immediately follows the color bitmap. It must have the same pixel height as the color bitmap.

Cursor resource file format

Like icon resource files, cursor resource files (with the .CUR file extension) may contain multiple images to match targeted display-device resolutions. In the case of cursors, Windows determines the best match for a particular display-device driver by examining the width and height of the cursor images.

The cursor resource file contains a header structure at the beginning of the file which identifies the type and number of resources in the file. The following shows the format of this header:

Field	Type/Description
curReserved	**WORD** Is reserved and must be set to 0.
curResourceType	**WORD** Specifies the type of resource contained in the file. For a cursor resource, this field must be 2.
curResourceCount	**WORD** Specifies the number of resources contained in the file.

The resource directory follows this header. The resource directory consists of one or more arrays of resource descriptors. The **curResorceCount** specifies the number of arrays. The following shows the format of the array:

Field	Type/Description
curWidth	**BYTE** Specifies the width in pixels of this form of the cursor image.
curHeight	**BYTE** Specifies the height in pixels of this form of the cursor image.
ColorCount	**BYTE** Specifies the number of colors in this form of the icon image. Acceptable values are 2, 8, or 16.
Reserved	**BYTE** Is reserved and must be set to 0.
curXHotspot	**WORD** Specifies in pixels the horizontal position of the hotspot.
curYHotspot	**WORD** Specifies in pixels the vertical position of the hotspot.
curDIBSize	**DWORD** Specifies in bytes the size of the pixel array for this form of the cursor image.
curDIBOffset	**DWORD** Specifies in bytes the offset to the device-independent bitmap for this form. The offset is from the beginning of the file.

Cursors are monochrome. The bitmap for a cursor consists of two parts; the first half is the XOR mask specifying the visible image, and the second half is the AND mask specifying the transparent portion of the cursor image. The two parts must be of equal width and height. By combining the values in corresponding mask bits, Windows determines whether a pixel is black, white, inverted, or transparent.

Table 9.1 shows what values are necessary to produce the corresponding colors, inversions, or transparencies:

Table 9.1
Bit mask results

	Bit Value	**Bit Value**	**Bit Value**	**Bit Value**
AND mask	0	0	1	1
XOR mask	0	1	0	1
Resultant Pixel	Black	White	Transparent	Inverted

Clipboard file format

The Windows clipboard saves and reads clipboard data in files with the .CLP extension. A clipboard-data file contains a value that identifies it as a clipboard-data file, one or more data structures defining the format, size, and location of the clipboard data, and one or more blocks of the actual data.

The clipboard-data file begins with a header consisting of two fields. The following describes these fields:

Field	Type/Description
FileIdentifier	**WORD** Identifies the file as a clipboard-data file. This field must be set to CLP_ID.
FormatCount	**WORD** Specifies the number of clipboard formats contained in the file.

This header is followed by one or more data structures, each of which identifies the format, size, and offset of a block of clipboard data. The following shows the fields of this data structure:

Field	Type/Description
FormatID	**WORD** Specifies the clipboard-format ID of the clipboard data. See the description of the **SetClipboardData** function in Chapter 4, "Functions directory," in *Reference, Volume 1*, for information on clipboard formats.
LenData	**DWORD** Specifies in bytes the length of the clipboard data.
OffData	**DWORD** Specifies in bytes the offset of the clipboard-data block. The offset is from the beginning of the file.
Name	Is a 79-character array that specifies the format name for a private clipboard format.

The first block of clipboard data follows the last of these structures. For bitmaps and metafiles, the bits follow immediately after the bitmap header and the **METAFILEPICT** data structures.

Metafile format

A metafile consists of a collection of graphics device interface (GDI) function calls that create specific images on a device. Metafiles provide convenient storage for images that appear repeatedly in applications, and also allow you to use the clipboard to cut and paste images from one application to another.

Metafiles store images as a series of GDI function calls. After storing the function calls, applications play a metafile to generate an image on a device.

When an object is created during playback, GDI adds the handle of the object to the first available entry in the metafile handle table. GDI clears the table entry corresponding to the object when it is deleted during playback, allowing the table entry to be reused when another object is created.

Functions described in this section are discussed in greater detail in Chapter 4, "Functions directory," in *Reference, Volume 1.*

The metafile itself consists of two parts: a header and a list of records. The header contains a description of the size (in words) of the metafile and the number of drawing objects it uses. The list of records contains the GDI functions. The drawing objects can be pens, brushes, or bitmaps.

Metafile header

The following structured list shows the format of the metafile header:

```
struct{
 WORD  mtType;
 WORD  mtHeaderSize;
 WORD  mtVersion;
 DWORD mtSize;
 WORD  mtNoObjects;
 DWORD mtMaxRecord;
 WORD  mtNoParameters;
}
```

The metafile header contains the following fields:

Field	Description
mtType	Specifies whether the metafile is in memory or recorded in a disk file. It is one of these two values: **Value** **Meaning** 1 Metafile is in memory 2 Metafile is in a disk file
mtHeaderSize	Specifies the size in words of the metafile header.
mtVersion	Specifies the Windows version number. The version number for Windows version 3.0 is 0x300.
mtSize	Specifies the size in words of the file.
mtNoObjects	Specifies the maximum number of objects that exist in the metafile at the same time.

mtMaxRecord Specifies the size in words of the largest record in the metafile.

mtNoParameters Is not used.

Metafile records

A series of records follows the metafile header. Metafile records describe GDI functions. GDI stores most of the GDI functions that an application can use to create metafiles in similar, typical records. "Typical metafile record," later in this section, shows the format of the typical metafile record. Table 9.2, "GDI functions and values," lists the functions which GDI records in typical records, along with their respective function numbers.

The remainder of the functions contain more complex structures in their records. "Function-specific records," later in this section, describes the records for these functions.

In some cases, there are two versions of a metafile record. One version represents the record created by versions of Windows prior to version 3.0, while the second version represents the record created by Windows versions 3.0 and later. Windows 3.0 plays all metafile versions, but stores only 3.0 versions. Windows versions prior to 3.0 will not play metafiles recorded by Windows 3.0.

Table 9.2
GDI functions and values

Function	Value
Arc	0x0817
Chord	0x0830
Ellipse	0x0418
ExcludeClipRect	0x0415
FloodFill	0x0419
IntersectClipRect	0x0416
LineTo	0x0213
MoveTo	0x0214
OffsetClipRgn	0x0220
OffsetViewportOrg	0x0211
OffsetWindowOrg	0x020F
PatBlt	0x061D
Pie	0x081A
RealizePalette (3.0 and later)	0x0035
Rectangle	0x041B
ResizePalette (3.0 and later)	0x0139
RestoreDC	0x0127
RoundRect	0x061C
SaveDC	0x001E
ScaleViewportExt	0x0412

Table 9.2: GDI functions and values (continued)

ScaleWindowExt	0x0400
SetBkColor	0x0201
SetBkMode	0x0102
SetMapMode	0x0103
SetMapperFlags	0x0231
SetPixel	0x041F
SetPolyFillMode	0x0106
SetROP2	0x0104
SetStretchBltMode	0x0107
SetTextAlign	0x012E
SetTextCharExtra	0x0108
SetTextColor	0x0209
SetTextJustification	0x020A
SetViewportExt	0x020E
SetViewportOrg	0x020D
SetWindowExt	0x020C
SetWindowOrg	0x020B

Typical metafile record

The following structured list shows the format of a typical metafile record:

```
struct{
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

A typical metafile record contains the following fields:

Field	Description
rdSize	Specifies the size in words of the record.
rdFunction	Specifies the function number.
rdParm[ ]	Is an array of words containing the function parameters, in the reverse order in which they are passed to the function.

Function-specific records

Some metafile records contain data structures in the parameter field. This section contains definitions for these records.

AnimatePalette record 3.0

The **AnimatePalette** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0436.
rdParm[ ]	Contains the following elements:

Element	**Description**
start	First entry to be animated.
numentries	Number of entries to be animated.
entries	PALETTEENTRY blocks.

BitBlt record (prior to 3.0)

The **BitBlt** record stored by Windows versions prior to 3.0 contains a device-dependent bitmap which may not be suitable for playback on all devices. The following is the format of this record:

```
struct {
DWORD rdSize;
WORD rdFunction;
WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0922 .
rdParm[ ]	Contains the following elements:

Element	**Description**
raster op	High word of the raster operation.
SY	The *y*-coordinate of the source origin.
SX	The *x*-coordinate of the source origin.
DYE	Destination *y*-extent.
DXE	Destination *x*-extent.
DY	The *y*-coordinate of destination origin.
DX	The *x*-coordinate of destination origin.

bmWidth	Width of bitmap (in pixels)
bmHeight	Height of bitmap (in raster lines)
bmWidthBytes	Number of bytes in each raster line.
bmPlanes	Number of color planes in the bitmap.
bmBitsPixel	Number of adjacent color bits.
bits	Actual device-dependent bitmap bits.

BitBlt record 3.0

The **BitBlt** record stored by Windows versions 3.0 and later contains a device-independent bitmap suitable for playback on any device. The following is the format of this record:

```
struct {
DWORD rdSize;
WORD rdFunction;
WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0940.
rdParm[ ]	Contains the following elements:

Element	Description
raster op	High word of the raster operation.
SY	The *y*-coordinate of the source origin.
SX	The *x*-coordinate of the source origin.
DYE	The *y*-extent of the destination.
DXE	The *x*-extent of the destination.
DY	The *y*-coordinate of destination origin.
DX	The *x*-coordinate of destination origin.
BitmapInfo	**BITMAPINFO** data structure.
bits	Actual device-independent bitmap bits.

CreateBrushIndirect record

The **CreateBrushIndirect** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 LOGBRUSH rdParm;
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x02FC.
rdParm	Specifies the logical brush.

CreateFontIndirect record

The **CreateFontIndirect** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 LOGFONT rdParm;
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x02FB.
rdParm	Specifies the logical font.

CreatePalette record 3.0

The **CreatePalette** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 LOGPALETTE rdParm;
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x00F7.
rdParm	Specifies the logical palette.

CreatePatternBrush record (prior to 3.0)

The **CreatePatternBrush** record stored by Windows versions prior to 3.0 contains a device-dependent bitmap which may not be suitable for playback on all devices. The following is the format of this record:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x01F9.
rdParm[ ]	Contains the following elements:

Element	Description
bmWidth	Bitmap width.
bmHeight	Bitmap height.
bmWidthBytes	Bytes per raster line.
bmPlanes	Number of color planes.
bmBitsPixel	Number of adjacent color bits that define a pixel.
bmBits	Pointer to bit values.
bits	Actual bits of pattern.

CreatePatternBrush record 3.0

The **CreatePatternBrush** record stored by Windows versions 3.0 and later contains a device-independent bitmap suitable for playback on all devices. The following is the format of this record:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0142.
rdParm[ ]	Contains the following elements:

Element	Description
type	Bitmap type. This field may be either of these two values: ■ BS_PATTERN—Brush is defined by a device-dependent bitmap through a call to the **CreatePatternBrush** function.

	■ BS_DIBPATTERN—Brush is defined by a device-independent bitmap through a call to the **CreateDIBPatternBrush** function.
Usage	Specifies whether the **bmiColors[ ]** field of the **BITMAPINFO** data structure contains explicit RGB values or indexes into the currently realized logical palette. This field must be one of the following values: ■ DIB_RGB_COLORS—The color table contains literal RGB values. ■ DIB_PAL_COLORS—The color table consists of an array of indexes into the currently realized logical palette.
BitmapInfo	**BITMAPINFO** data structure.
bits	Actual device-independent bitmap bits.

CreatePenIndirect record

The format and field descriptions of the **CreatePenIndirect** record follow:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 LOGPEN rdParm;
}
```

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x02FA.
rdParm	Specifies the logical pen.

Create region record

The format and field descriptions of the **Create Region** record follow:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x06FF.
rdParm[ ]	Specifies the region to be created.

DeleteObject 3.0

The **DeleteObject** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm;
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x01F0.
rdParm	Specifies the handle-table index of the object to be deleted.

DrawText record

The **DrawText** record has the following format:

```
struct{
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description	
rdSize	Specifies the record size in words.	
rdFunction	Specifies the function number 0x062F.	
rdParm[ ]	Contains the following elements:	
	Element	**Description**
	format	Method of formatting.
	count	Number of bytes in the string.
	rectangle	Rectangular structure defining area where text is to be defined.
	string	Byte array containing the string. The array is ((count + 1) >>>> 1) words long.

Escape record

The format and field descriptions of the **Escape** record follow:

```
struct {
 DWORD rdSize;
```

```
  WORD rdFunction;
  WORD rdParm[];
}
```

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0626.
rdParm[ ]	Contains the following elements:

Element	**Description**
escape number	Number identifying individual escape.
count	Number of bytes of information.
input data	Variable length field. The field is ((count+1)/ >>>> 1) words long.

ExtTextOut record

The **ExtTextOut** record has the following format:

```
struct{
  DWORD rdSize;
  WORD rdFunction;
  WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0A32.
rdParm[ ]	Contains the following elements:

Element	**Description**
y	Logical *y*-value of string’s starting point.
x	Logical *x*-value of string’s starting point.
count	Length of the string.
options	Rectangle type.
rectangle	**RECT** structure defining the **ExtTextOut** rectangle if options element is nonzero; nonexistent if options element equals zero
string	Byte array containing the string. The array is ((count + 1) >>>> 1) words long.
dxarray	Optional word array of intercharacter distances.

Polygon record

The **Polygon** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0324.
rdParm[ ]	Contains the following elements:

Element	**Description**
count	Number of points.
list of points	List of individual points.

PolyPolygon record

The **PolyPolygon** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0538.
rdParm[ ]	Contains the following elements:

Element	**Description**
count	Total number of points.
list of polygon counts	List of number of points for each polygon.
list of points	List of individual points.

Polyline record

The **Polyline** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description	
rdSize	Specifies the record size in words.	
rdFunction	Specifies the function number 0x0325.	
rdParm[ ]	Contains the following elements:	
	Element	**Description**
	count	Number of points.
	list of points	List of individual points.

SelectClipRegion

The **SelectClipRegion** record has the following format:

```
struct{
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm;
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x012C.
rdParm	Specifies the handle-table index of the region being selected.

SelectObject

The **SelectObject** record has the following format:

```
struct{
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm;
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x012D.
rdParm	Specifies the handle-table index of the object being selected.

SelectPalette record 3.0

The **SelectPalette** record has the following format:

```
struct{
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm;
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0234.
rdParm	Specifies the handle-table index of the logical palette being selected.

SetDIBitsToDevice record 3.0

The **SetDIBitsToDevice** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><b>rdSize</b></td><td>Specifies the record size in words.</td></tr>
<tr><td><b>rdFunction</b></td><td>Specifies the function number 0x0D33.</td></tr>
<tr><td><b>rdParm[ ]</b></td><td>Contains the following elements:
<table>
<tr><th>Element</th><th>Description</th></tr>
<tr><td>wUsage</td><td>Flag indicating whether the bitmap color table contains RGB values or indexes into the currently realized logical palette</td></tr>
<tr><td>numscans</td><td>Number of scan lines in the bitmap.</td></tr>
</table>
</td></tr>
</table>

	startscan	First scan line in the bitmap.
	srcY	The *y*-coordinate of the origin of the source in the bitmap.
	srcX	The *x*-coordinate of the origin of the source in the bitmap.
	extY	Height of the source in the bitmap.
	extX	Width of the source in the bitmap.
	destY	The *y*-coordinate of the origin of the destination rectangle.
	destX	The *x*-coordinate of the origin of the destination rectangle.
	BitmapInfo	**BITMAPINFO** data structure.
	bits	Actual device-independent bitmap bits.

SetPaletteEntries record 3.0

The **SetPaletteEntries** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description	
rdSize	Specifies the record size in words.	
rdFunction	Specifies the function number 0x0037.	
rdParm[ ]	Contains the following elements:	
	Element	**Description**
	start	First entry to be set in the palette.
	numentries	Number of entries to be set in the palette.
	entries	PALETTEENTRY blocks.

StretchBlt record (prior to 3.0)

The **StretchBlt** record stored by Windows versions prior to 3.0 contains a device-dependent bitmap which may not be suitable for playback on all devices. The following is the format of this record:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0B23.
rdParm[ ]	Contains the following elements:

Element	Description
raster op	Low word of the raster operation.
raster op	High word of the raster operation.
SYE	The *y*-extent of the source.
SXE	The *x*-extent of the source.
SY	The *y*-coordinate of the source origin.
SX	The *x*-coordinate of the source origin.
DYE	The *y*-extent of the destination.
DXE	The *x*-extent of the destination.
DY	The *y*-coordinate of destination origin.
DX	The *x*-coordinate of destination origin.
bmWidth	Width of the bitmap in pixels.
bmHeight	Height of the bitmap in raster lines.
bmWidthBytes	Number of bytes in each raster line.
bmPlanes	Number of color planes in the bitmap.
bmBitsPixel	Number of adjacent color bits.
bits	Actual bitmap bits.

StretchBlt record 3.0

The **StretchBlt** record stored by Windows versions 3.0 and later contains a device-independent bitmap suitable for playback on all devices. The following is the format of this record:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0B41.
rdParm[ ]	Contains the following elements:

Element	Description
raster op	Low word of the raster operation.
raster op	High word of the raster operation.
SYE	The *y*-extent of the source.
SXE	The *x*-extent of the source.
SY	The *y*-coordinate of the source origin.
SX	The *x*-coordinate of the source origin.

DYE	The *y*-extent of the destination.
DXE	The *x*-extent of the destination.
DY	The *y*-coordinate of destination origin.
DX	The *x*-coordinate of destination origin.
BitmapInfo	**BITMAPINFO** data structure.
bits	Actual device-independent bitmap bits.

StretchDIBits record 3.0

The **StretchDIBits** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0F43.
rdParm[ ]	Contains the following elements:

Element	Description
dwRop	Raster operation to be performed.
wUsage	Flag indicating whether the bitmap color table contains RGB values or indexes into the currently realized logical palette
srcYExt	Height of the source in the bitmap.
srcXExt	Width of the source in the bitmap.
srcY	The *y*-coordinate of the origin of the source in the bitmap.
srcX	The *x*-coordinate of the origin of the source in the bitmap.
dstYExt	Height of the destination rectangle.
dstXExt	Width of the destination rectangle.
dstY	The *y*-coordinate of the origin of the destination rectangle.
dstX	The *x*-coordinate of the origin of the destination rectangle.
BitmapInfo	**BITMAPINFO** data structure.
bits	Actual device-independent bitmap bits.

TextOut record

The **TextOut** record has the following format:

```
struct {
 DWORD rdSize;
 WORD rdFunction;
 WORD rdParm[];
}
```

This record contains the following fields:

Field	Description
rdSize	Specifies the record size in words.
rdFunction	Specifies the function number 0x0521.
rdParm[ ]	Contains the following elements:

Element	Description
count	The string's length.
string	The actual string.
y-value	Logical *y*-coordinate of string's starting point.
x-value	Logical *x*-coordinate of string's starting point.

Sample metafile program output

This section shows the metafile created by a sample program.

The following sample program creates a small metafile in which a purple rectangle with a green border is drawn, and the words "Hello People" are written in the rectangle.

```
MakeAMetaFile(hDC)
HDC hDC;
{
HPEN     hMetaGreenPen;
HBRUSH   hMetaVioletBrush;
HDC      hDCMeta;
HANDLE   hMeta;

/* create the metafile with output going to the disk
*/
hDCMeta = CreateMetaFile( (LPSTR) "sample.met");

hMetaGreenPen = CreatePen(0, 0, (DWORD) 0x0000FF00);
SelectObject(hDCMeta, hMetaGreenPen);

hMetaVioletBrush = CreateSolidBrush( (DWORD)
0x00FF00FF);
SelectObject(hDCMeta, hMetaVioletBrush);

Rectangle(hDCMeta, 0, 0, 150, 70);
TextOut(hDCMeta, 10, 10, (LPSTR) "Hello People", 12);

/* we are done with the metafile */
hMeta = CloseMetaFile(hDCMeta);

/* play the metafile that we just created */
PlayMetaFile(hDC, hMeta);
}
```

The resulting binary file SAMPLE.MET looks like this:

```
0001         mtType... disk metafile
0009         mtSize...
0100         mtVersion
0000 0036    mtSize
0002         mtNoObjects
0000 000C    mtMaxrecord
0000         mtNoParameters

0000 0008    rdSize
02FA         rdFunction (CreatePen function call)
0000 0000 0000 0000 FF00  rdParm (LOGPEN structure defining pen)

0000 0004    rdSize
012D         rdFunction (SelectObject)
0000         rdParm (index to object #0... the above pen)

0000 0007    rdSize
02FC         rdFunction (CreateBrush)
0000 00FF 00FF 0000 rdParm (LOGBRUSH structure defining the brush)

0000 0004    rdSize
012D         rdFunction (SelectObject)
0001         rdParm (index to object #1... the brush)

0000 0007    rdSize
041B         rdFunction (Rectangle)
0046 0096 0000 0000 rdParm (parameters sent to Rectangle...in reverse order)

0000 000C    rdSize
0521         rdFunction (TextOut)
rdParm
000C         count
string
48 65 6C 6C 6F 20 50 65 6F 70 6C 65   "Hello People"
000A             y-value
000A             x-value
```

Summary

Windows files store information required to create Windows applications as well as data needed by the Windows system and Windows applications during execution. For more information on topics related to Windows files, see the following:

Topic	Reference
Metafile functions	*Reference, Volume 1*: Chapter 1, "Window manager interface functions," and Chapter 4, "Functions directory"

CHAPTER

10

Module-definition statements

This chapter describes the statements contained in the module-definition file that defines the application's contents and system requirements for the **LINK** program. **LINK** links compiled source files with Microsoft Windows and other libraries to create an executable Windows application. For information on running **LINK**, see *Tools.*

The module-definition file contains one or more of the following module statements:

Statement	Description
CODE	Code-segment attributes
DATA	Data-segment attributes
DESCRIPTION	One-line description of the module
EXETYPE	.EXE header type (Windows or OS/2)
EXPORTS	Exported functions
HEAPSIZE	Size of local heap in bytes
IMPORTS	Imported functions
LIBRARY	Dynamic-link library name
NAME	Module name
SEGMENTS	Additional code segment
STACKSIZE	Size of local stack in bytes
STUB	Old-style executable

This chapter describes these statements, their syntax, required and optional parameters, and usage.

CODE

Syntax CODE [[FIXED | MOVEABLE]] [[DISCARDABLE]] [[\PRELOAD | LOADONCALL]]

This statement defines the attributes of the standard code segment. The standard code segment is the application segment having the name _TEXT and belonging to the class CODE. In C applications, the standard data segment is created automatically if no specific segment name is included in the C-Compiler command line.

The **FIXED** option, if included, means that the segment remains at a fixed memory location; the **MOVEABLE** option means that the segment can be moved, if necessary, in order to compact memory.

The **DISCARDABLE** option, if included, means that the segment can be discarded if it is no longer needed.

The **PRELOAD** option, if included, means that the segment is loaded when the module is first loaded; the **LOADONCALL** option means that the segment is loaded when it is called. The Resource Compiler may override this option. See *Tools* for more information.

Comments There are no default attributes for code segments. The .DEF file should always explicitly define code-segment attributes.

If conflicting options are included in the same statement, **LINK** uses the overriding option to determine the segment attributes. The following list shows which options override which:

MOVEABLE overrides **FIXED**.

PRELOAD overrides **LOADONCALL**.

Example

```
CODE MOVEABLE LOADONCALL
```

In this example, the loader forces all fixed and moveable (but not discardable) code segments to be loaded. Libraries cannot have code that is moveable but not discardable.

DATA

Syntax Data [[NONE | SINGLE | MULTIPLE]] [[FIXED | MOVEABLE]]

This statement defines the attributes of the standard data segment. The standard data segment is all application segments belonging to the group

DGROUP and the class DATA. In C applications, the standard data segment is created automatically. The data is always preloaded.

The **NONE** option, if included, means that there is no data segment. To be effective, this option should be the only attribute of the segment. This option is available only for libraries.

The **SINGLE** option, if included, means that a single segment is shared by all instances of the module, and is valid only for libraries.
The **MULTIPLE** option means that one segment exists for each instance, and is only valid for applications.

NONE, **SINGLE**, and **MULTIPLE** are mutually exclusive.

The **FIXED** option, if included, means that the segment remains at a fixed memory location. The **MOVEABLE** option means that the segment can be moved if necessary, in order to compact memory.

Comments There are no default attributes for data segments. The .DEF file should always explicitly define data-segment attributes.

Data segments are always preloaded.

If conflicting options are included in the same statement, **LINK** uses the overriding option to determine the segment attributes. The following list shows which options override which:

MULTIPLE overrides **NONE**.

SINGLE overrides **NONE**.

MOVEABLE overrides **FIXED**.

Example

```
DATA MOVEABLE SINGLE
```

This example tells **LINK** that this module has a single, moveable data segment.

DESCRIPTION

Syntax DESCRIPTION 'text'

This statement inserts text into the application's module. It is useful for embedding source-control or copyright information

Parameters *text* Specifies one or more ASCII characters. The string must be enclosed in single quotation marks.

Example

```
DESCRIPTION 'Microsoft Windows Template Application'
```

This example embeds the text "Microsoft Windows Template Application" in the application module.

EXETYPE

Syntax EXETYPE headertype

This statement specifies the default executable-file (.EXE) header type (Windows or OS/2). It is required for every Windows application.

Parameters

headertype Determines the header type. When linking an application intended for the Windows environment, you must set this parameter to the value "WINDOWS". For an MS OS/2 application, set this parameter to the value "OS/2".

Example

```
EXETYPE WINDOWS
```

EXPORTS

Syntax EXPORTS exportname [[ordinal-option]] [[\res-option]] [[data-option]] [[parameter-option]]

This statement defines the names and attributes of the functions to be exported to other applications. The **EXPORTS** key word marks the beginning of the definitions. It can be followed by any number of export definitions, each on a separate line.

Parameters

exportname Specifies one or more ASCII characters that define the function name. It has the following form:

<*entryname*>=[[*internalname*]]

where the *entryname* parameter specifies the name to be used by other applications to access the exported function, and *internalname* is an optional parameter that defines the actual name of the function if *entryname* is not the actual name.

ordinal-option Defines the function's ordinal value. It has the following form:

@ordinal

where *ordinal* takes an integer value that specifies the function's ordinal value. The ordinal value defines the location of the function's name in the application's string table. (When exporting functions from libraries, it is better to use an ordinal rather than a name; using ordinals conserves space.)

res-option Is the optional key word **RESIDENTNAME**, which specifies that the function's name must be resident at all times.

data-option Is the optional key word **NODATA**, which specifies that the function is not bound to a specific data segment. When invoked, the function uses the current data segment.

parameter-option
Is an optional integer value that specifies the number of words the function expects to be passed as parameters.

Example

```
EXPORTS
        SampleRead=read2bin @1 8
        StringIn=str1 @2 4
        CharTest NODATA
```

This example exports the functions SampleRead, StringIn and CharTest so that other applications, or Windows itself, can call them.

HEAPSIZE

Syntax HEAPSIZE bytes

This statement defines the number of bytes needed by the application for its local heap. An application uses the local heap whenever it allocates local memory

The default heap size is zero. The minimum size is 256 bytes. For an application, the size of the local heap must be at least large enough to hold the current environment.

Parameters *bytes* Is an integer value that specifies the heap size in bytes. It must not exceed 65,536 (the size of a single physical segment).

Example

```
HEAPSIZE 4096
```

This example sets the size of the application's local heap to 4096 bytes.

IMPORTS

Syntax IMPORTS [[internal-option]] modulename [[entry-option]]

This statement defines the names and attributes of the functions to be imported from dynamic-link libraries. The **IMPORTS** key word marks the beginning of the definitions. It can be followed by any number of import definitions, each on a separate line.

Parameters

internal-option
Specifies the name that the application will use to call the function. It has the following form:

internal-name=

where *internal-name* is one or more ASCII characters. This name must be unique.

modulename Specifies one or more uppercase ASCII characters that define the name of the executable module that contains the function. The module name must match the name of the executable file. For example, an application with the executable file SAMPLE.DLL has the module name "SAMPLE". The executable file must be named with the .DLL extension.

entry-option Specifies the function to be imported. It can be one of the following:

.entryname

.entryordinal

where *entryname* is the actual name of the function, and *entryordinal* is the ordinal value of the function.

Example

```
IMPORTS
        Sample.SampleRead
        write2hex=Sample.SampleWrite
        Read.1
```

➡ Instead of listing imported DLL functions in the **IMPORTS** statement, you can specify an "import library" for the DLL in your application's **LINK** command line. It also saves space to import by ordinal.

LIBRARY

Syntax LIBRARY libraryname

This statement defines the name of a library module. Library modules are resource modules that contain code, data, and other resources but are not intended to be executed as an independent program. Like an application's module name, a library's module name must match the name of the executable file. For example, the library USER.EXE has the module name "USER".

Parameters *libraryname* Specifies one or more ASCII characters that define the name of the library module.

Comments The start address of the library module is determined by the library's object files; it is an internally defined function.

The *libraryname* parameter is optional. If the parameter is not included, **LINK** uses the filename part of the executable file (that is, the name with the extension removed).

If the .DEF file includes neither a **NAME** nor a **LIBRARY** statement, **LINK** assumes a **NAME** statement without a *modulename* parameter is desired.

Example

```
LIBRARY Utilities
```

This example gives a library the module name "Utilities."

NAME

Syntax NAME modulename

This statement defines the name of the application's executable module. The module name identifies the module when exporting functions.

Parameters *modulename* Specifies one or more uppercase ASCII characters that define the name of the executable module. The module name must match the name of the executable file. For example, an application with the executable file SAMPLE.EXE has the module name "SAMPLE". Do not use OS/2 system library names. Examples of these names are DOSCALLS, VIOCALLS, and MOUCALLS.

Comments The *modulename* parameter is optional. If the parameter is not included, **LINK** assumes that the module name matches the the filename of the executable file. For example, if you do not specify a module name and the executable file is named MYAPP.EXE, **LINK** assumes that the module name is "MYAPP".

If the .DEF file includes neither a **NAME** nor a **LIBRARY** statement, **LINK** assumes a **NAME** statement without a *modulename* parameter is desired.

Example

```
NAME Calendar
```

This example gives an application the module name "Calendar".

SEGMENTS

Syntax

SEGMENTS segmentname [[CLASS 'class-name']] [[minalloc]]\
[[FIXED I MOVEABLE]]
[[DISCARDABLE]] [[SHARED I NONSHARED]] [[PRELOAD I
LOADONCALL]]

This statement defines the segment attributes of additional code and data segments.

The **FIXED** option, if included, means that the segment remains at a fixed memory location. The **MOVEABLE** option means that the segment can be moved if necessary, in order to compact memory.

The **DISCARDABLE** option, if included, means that the segment can be discarded if it is no longer needed.

The **PRELOAD** option, if included, means that the segment is loaded immediately The **LOADONCALL** option means that the segment is loaded when it is accessed or called. The Resource Compiler may override this option. See *Tools* for more information.

Parameters

segmentname Specifies a character string that names the new segment. It can be any name, including the standard segment names _TEXT and _DATA, which represent the standard code and data segments.

class-name Is an optional key word that specifies the class name of the specified segment. If no class name is specified, **LINK** uses the class name CODE by default.

minalloc Is an optional integer value that specifies the minimum allocation size for the segment.

Comments

There are no default attributes for additional segments. The .DEF file should always explicitly define the attributes of additional segments.

If conflicting options are included in the same statement, **LINK** uses the overriding option to determine the segment attributes. The following list shows which options override which:

MOVEABLE overrides **FIXED**.

PRELOAD overrides **LOADONCALL**.

Example

```
SEGMENTS
      _TEXT FIXED
      _INIT PRELOAD DISCARDABLE
      _RES  CLASS 'DATA' PRELOAD DISCARDABLE
```

STACKSIZE

Syntax STACKSIZE bytes

This statement defines the number of bytes needed by the application for its local stack. An application uses the local stack whenever it makes function calls.

The default stack size is zero if the application makes no function calls. If your application does make function calls and you specify a stack size smaller than 5K bytes, Windows automatically sets the stack size to 5K bytes.

Parameters *bytes* Is an integer value that specifies the stack size in bytes.

Comments Do not use the **STACKSIZE** statement for dynamic-link libraries.

Example

```
STACKSIZE 6144
```

This example sets the size of an application's stack to 6144 bytes.

STUB

Syntax STUB "filename"

This statement appends the old-style executable file specified by filename to the beginning of the module. The executable stub should display a warning message and terminate if the user attempts to execute the module without having loaded Windows. The default file WINSTUB.EXE can be used if no other actions are required.

Parameters *filename* Specifies the name of the old-style executable file that will be appended to the module. The name must have the DOS filename format.

Comments If the file named by *filename* is not in the current directory, **LINK** searches for the file in the directories specified by the user's PATH environment variable.

Example

```
STUB 'WINSTUB.EXE'
```

This example specifies the executable file WINSTUB.EXE as the application's stub. If a user tries to run this application in the DOS environment, rather than with Windows, the program WINSTUB.EXE starts instead.

CHAPTER

11

Binary and ternary raster-operation codes

This chapter lists and describes the binary and ternary raster operations used by the graphics device interface (GDI). A binary raster operation uses two operands: a pen and a destination bitmap. A ternary raster operation uses three operands: a source bitmap, a brush, and a destination bitmap. Both binary and ternary raster operations use Boolean operators.

Binary raster operations

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 selected pen and the destination bitmap are combined. There are two operands used in these operations:

- D Destination bitmap
- P Selected pen

The Boolean operators used in these operations are as follows:

- 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 destination with a combination of the pen and the selected brush:

```
DPo
```

Each raster-operation code is a 32-bit integer value 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 pen and destination values. For example, the operation indexes for the DPo and DPan operations are shown in Table 11.1:

Table 11.1
Operation indexes for DPo and DPan

P	D	PSo	DPSoo
0	0	0	1
0	1	1	1
1	0	1	1
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

When a monochrome device is used, GDI maps the value 0 to black and the value 1 to white. Given an application that attempts to draw with a black pen on a white destination by using the available binary raster operations, the following results will occur:

Raster operation	Result
R2_BLACK	Visible black line
R2_COPYPEN	Visible black line
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

When a color device is used, 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 given color. Intensities range from 0 to 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. Given an application that attempts to draw with a white pen on a blue destination by using the available binary raster operations, the following results will 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
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

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 source, the selected brush, and the destination bitmap are combined. There are three operands used in these operations:

- D Destination bitmap
- P Selected brush (also called pattern)
- S Source bitmap

The Boolean operators used in these operations are as follows:

- 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 destination with a combination of the source and brush:

```
PSo
```

The following operation combines the source and brush with the destination (there are alternate spellings of the same function, so although a particular spelling may not be in the list, an equivalent form will be):

```
DPSoo
```

Each raster-operation code is a 32-bit integer value 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 Table 11.2:

Table 11.2 Operation Indexes for PSo and DPSoo

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

Table 11.2: Operation Indexes for PSo and DPSoo (continued)

1	0	1	1	1
1	1	0	1	1
1	1	1	1	1
Operation index:			00FC	00FE

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 11.1, "Operation indexes for DPo and DPan." The PSo operation is in line 252 (FCh) of the table; DPSoo is in line 254 (FEh).

The most commonly used raster operations have been given special names in the Windows include file, windows.h. You should use these names whenever possible in your applications.

*For more information about RGB values, see the **RGB** structure in Chapter 7, "Data types and structures."*

When the source and destination 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 are color, those colors are represented with RGB values.

Table 11.3 lists the raster-operation codes:

Table 11.3 Raster-operation codes

Boolean Function in Hex, Hex	**Hex ROP**	**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	–
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	–

Table 11.3: Raster-operation codes (continued)

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	–
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	–
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	–

Table 11.3: Raster-operation codes (continued)

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	–
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	–
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	–

Table 11.3: Raster-operation codes (continued)

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	–
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	–
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	–

Table 11.3: Raster-operation codes (continued)

AF	00AF0229	DPno	–
B0	00B00E05	PDSnoa	–
B1	00B10665	PDSPxoxn	–
B2	00B21974	SSPxDSxox	–
B3	00B30CE8	SDPanan	–
B4	00B4070A	PSDnax	–
B5	00B507A9	DPSDoaxn	–
B6	00B616E9	DPSDPaoxx	–
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	–
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	–

Table 11.3: Raster-operation codes (continued)

E2	00E20746	DSPDxax	–
E3	00E306EA	PSDPaoxn	–
E4	00E40748	SDPSxax	–
E5	00E506E5	PDSPaoxn	–
E6	00E61CE8	SDPSanax	–
E7	00E70D79	SPxPDxan	–
E8	00E81D74	SSPxDSxax	–
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

For more information on topics related to raster-operation codes, see the following:

Topic	Reference
Using raster-operation codes with GDI functions	*Reference, Volume 1*: Chapter 2, "Graphics device interface functions," and Chapter 4, "Functions directory"
Setting the current drawing mode with **SetROP2**	*Reference, Volume 1*: Chapter 4, "Functions directory"

CHAPTER

12

Printer escapes

This chapter contains an alphabetical list of the individual Microsoft Windows printer escapes. The printer escapes allow applications to access facilities of a particular output device that are not available directly through the graphics device interface (GDI). The escape calls are made by an application, translated by Windows, and then sent to the printer device driver.

ABORTDOC

Syntax short Escape(hDC, ABORTDOC, NULL, NULL, NULL)

This escape terminates the current job, erasing everything the application has written to the device since the last **ENDDOC** escape.

The **ABORTDOC** escape should be used to terminate:

- Printing operations that do not specify an abort function using the **SETABORTPROC** escape
- Printing operations that have not yet reached their first **NEWFRAME** or **NEXTBAND** escape call

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments If an application encounters a printing error or a canceled print operation, it must not attempt to terminate the operation by using the **Escape** function with either the **ENDDOC** or **ABORTDOC** escape. GDI automatically terminates the operation before returning the error value.

If the application displays a dialog box to allow the user to cancel the print operation, it must send the **ABORTDOC** escape before destroying the dialog box.

The application must send the **ABORTDOC** escape before freeing the procedure-instance address of the abort function, if any.

BANDINFO

Syntax short Escape(hDC, BANDINFO, sizeof(BANDINFOSTRUCT), lpInData, lpOutData)

This escape copies information about a device with banding capabilities to a structure pointed to by the *lpOutData* parameter. It is implemented only for devices that use banding.

Banding is a property of an output device that allows a page of output to be stored in a metafile and divided into bands, each of which is sent to the device to create a complete page.

The information copied to the structure pointed to by *lpOutData* includes:

- A value that indicates whether there are graphics in the next band
- A value that indicates whether there is text on the page
- A **RECT** data structure that contains a bounding rectangle for all graphics on the page

The *lpOutData* parameter is NULL if no data are returned.

The *lpInData* parameter specifies information sent by the application to the device driver. This information is read by the device driver only on the first **BANDINFO** escape call on a page.

Parameters

hDC	**HDC** Identifies the device context.
lpInData	**BANDINFOSTRUCT FAR** * Points to a **BANDINFOSTRUCT** data structure that contains information to be passed to the driver. See the following "Comments" section for more information on the **BANDINFOSTRUCT** data structure.
lpOutData	**BANDINFOSTRUCT FAR** * Points to a **BANDINFOSTRUCT** data structure that contains information returned by the

driver. See the following "Comments" section for more information on the **BANDINFOSTRUCT** data structure.

Return value The return value specifies the outcome of the escape. It is 1 if the escape is successful. It is zero if the function fails or is not implemented by the driver.

Comments The **BANDINFOSTRUCT** data structure contains information about the contents of a page and supplies a bounding rectangle for graphics on the page. The following shows the format of **BANDINFOSTRUCT**:

```
typedef struct {
    BOOL   fGraphicsFlag;
    BOOL   fTextFlag;
    RECT   GraphicsRect;
} BANDINFOSTRUCT;
```

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

Field	Description
fGraphicsFlag	Is TRUE if graphics are or are expected to be on the page or in the band; otherwise, it is FALSE.
fTextFlag	Is TRUE if text is or is expected to be on the page or in the band; otherwise, it is FALSE.
GraphicsRect	Contains a **RECT** data structure that supplies a bounding region for all graphics on the page.

Table 12.1 shows the meaning of these fields, depending on which parameter contains the structure.

Table 12.1 Meaning of BANDINFOSTRUCT fields

Field	When used in *lpInData*	When used in *lpOutData*
fGraphicsFlag	TRUE if the application is informing the driver that graphics are on the page.	TRUE if the driver is informing the application that it expects graphics in this band.
fTextFlag	TRUE if the application is informing the driver that text is on the page.	TRUE if the driver is informing the application that it expects text in this band.
GraphicsRect	Supplies the bounding rectangle for all graphics on the page.	No valid return data.

An application should call this escape immediately after each call to the **NEXTBAND** escape. It is in reference to the band the driver returned to that escape.

An application should use this escape in the following manner:

On the first band, the driver may give the application a full-page band and ask for text only (**fGraphicsFlag** is set to FALSE and **fTextFlag** is set to TRUE). The application sends only text to the driver.

If in the first band the application indicated that it had graphics (**fGraphicsFlag** is set to TRUE), or that the driver encountered vector fonts, then the driver will band the rest of the page. If there are no graphics or vector fonts, then the next **NEXTBAND** will return an empty rectangle to indicate that the application should move on to the next page.

If there are graphics but no vector fonts (the application set **fGraphicsFlag** to TRUE, but there were no graphics in the first full-page text band), then for subsequent bands the driver may optionally band only into the rectangle the application passed. This rectangle bounds all graphics on the page. If there are vector fonts, then the driver will band the entire width and depth of the page with **fTextFlag** set to TRUE. It will also set **fGraphicsFlag** to true if the application set it.

The driver assumes that an application using **BANDINFO** will only send text in the first full-page text band since that is all the driver requested. Therefore, if the driver encounters a vector font or graphics in the band, it assumes they were generated by a text primitive and sets **fTextFlag** to TRUE for all subsequent graphics bands so they can be output as graphics. If the application does not satisfy this expectation, the image will still be generated properly, but the driver will spend time sending spurious text primitives to graphics bands.

Older drivers written before the **BANDINFO** escape was designed used full-page banding for text. If a particular driver does not support the **BANDINFO** escape but sets RC_BANDING, the application can detect full-page banding for text by determining if the first band on the page covers the entire page.

BEGIN_PATH

Syntax short Escape(hDC, BEGIN_PATH, NULL, NULL, NULL)

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

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

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

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the current path nesting level. If the escape is successful, the return value is the number of **BEGIN_PATH** escape calls without a corresponding **END_PATH** escape call. Otherwise, the return value is zero.

Comments An application may begin a subpath within another path. If the subpath is closed, it is treated exactly like a polygon. If it is open, it is treated exactly like a polyline.

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

CLIP_TO_PATH

Syntax short Escape(hDC, CLIP_TO_PATH, sizeof(int), lpClipMode, NULL)

This escape defines a clipping area bounded by the currently open path. It enables the application to save and restore the current clipping area and to set up an inclusive or exclusive clipping area bounded by the currently open path. If the path defines an inclusive clipping area, portions of primitives falling outside the interior bounded by the path are clipped. If the path defines an exclusive clipping area, portions of primitives falling inside the interior are clipped.

Parameters *hDC* **HDC** Identifies the device context.

lpClipMode **LPINT** Points to a short integer specifying the clipping mode. It can be one of the following values:

- CLIP_SAVE (0) Saves the current clipping area.
- CLIP_RESTORE (1) Restores the previous clipping area.
- CLIP_INCLUSIVE (2) Sets an inclusive clipping area.
- CLIP_EXCLUSIVE (3) Sets an exclusive clipping area.

Return value The return value specifies the outcome of the escape. It is nonzero if the escape was successful. Otherwise, it is zero.

Comments To clip a set of primitives against a path, an application should follow these steps:

1. Save the current clipping area using the CLIP_TO_PATH escape.

2. Begin a path using the BEGIN_PATH escape.
3. Draw the primitives bounding the clipping area.
4. Close the path using the END_PATH escape.
5. Set the clipping area using the CLIP_TO_PATH escape.
6. Draw the primitives to be clipped.
7. Restore the original clipping area using the CLIP_TO_PATH escape.

DEVICEDATA

Syntax short Escape(hDC, DEVICEDATA, nCount, lpInData, lpOutData)

This escape is identical to the **PASSTHROUGH** escape. See the description of **PASSTHROUGH** for further information.

DRAFTMODE

Syntax short Escape(hDC, DRAFTMODE, sizeof(int), lpDraftMode, NULL)

This escape turns draft mode off or on. Turning draft mode on instructs the device driver to print faster and with lower quality (if necessary). The draft mode can be changed only at page boundaries, for example, after a **NEWFRAME** escape directing the driver to advance to a new page.

Parameters *hDC* **HDC** Identifies the device context.

lpDraftMode **LPINT** Points to a short-integer value that specifies the draft mode. It may be one of the following values:

- 0 Specifies draft mode off.
- 1 Specifies draft mode on.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments The default draft mode is off.

DRAWPATTERNRECT

Syntax short Escape(hDC, DRAWPATTERNRECT, sizeof(PRECTSTRUCT), lpInData, NULL)

This escape creates a pattern, gray scale, or solid black rectangle by using the pattern/rule capabilities of Page Control Language (PCL) on

Hewlett-Packard® LaserJet® or LaserJet-compatible printers. A gray scale is a gray pattern that contains a specific mixture of black and white pixels.

Parameters

hDC	**HDC** Identifies the device context.
lpInData	**PRECT_STRUCT FAR** * Points to a **PRECT_STRUCT** data structure that describes the rectangle. See the following "Comments" section for more information on the **PRECT_STRUCT** data structure.

Return value

The return value specifies the outcome of the escape. It is 1 if the escape is successful. Otherwise, it is zero.

Comments

The *lpInData* parameter points to a **PRECT_STRUCT** data structure that defines the rectangle to be created. The **PRECT_STRUCT** structure has the following format:

```
typedef struct {
    POINT  prPosition;
    POINT  prSize;
    WORD   prStyle;
    WORD   prPattern;
} PRECT_STRUCT;
```

This structure has the following fields:

Field	Description
prPosition	Specifies the upper-left corner of the rectangle.
prSize	Specifies the lower-right corner of the rectangle.
prStyle	Specifies the type of pattern. It may be one of the following values: **Value** **Meaning** 0 Black rule 1 White rule that erases bitmap data previously written to same area; this pattern is available on the HP LaserJet IIP only. 2 Gray scale 3 HP-defined
prPattern	Specifies the pattern. It is ignored for a black rule. It specifies the percentage of gray for a gray-scale pattern. It represents one of six Hewlett-Packard-defined patterns.

An application should use the **QUERYESCSUPPORT** escape to determine whether a device is capable of drawing patterns and rules before using the **DRAWPATTERNRECT** escape. If an application uses the **BANDINFO** escape, all patterns and rectangles sent by using **DRAWPATTERNRECT** should be treated as text and sent on a text band.

Do not try to erase patterns and rules created with the **DRAWPATTERNRECT** escape by placing opaque objects over them. To erase such patterns and rules, use the function calls provided by GDI.

ENABLEDUPLEX

Syntax short Escape(hDC, ENABLEDUPLEX, sizeof(WORD), lpInData, NULL)

This escape enables the duplex printing capabilities of a printer. A device that possesses duplex printing capabilities is able to print on both sides of the output medium.

Parameters

hDC **HDC** Identifies the device context.

lpInData **WORD FAR *** Points to an unsigned 16-bit integer that specifies whether duplex or simplex printing is used. It may be one of the following values:

- 0 Simplex
- 1 Duplex with vertical binding
- 2 Duplex with horizontal binding

Return value The return value specifies the outcome of the escape. It is 1 if the escape is successful. Otherwise, it is zero.

Comments An application should use the **QUERYESCSUPPORT** escape to determine whether an output device is capable of creating duplex output. If **QUERYESCSUPPORT** returns a nonzero value, the application should send the **ENABLEDUPLEX** escape even if simplex printing is desired. This guarantees replacement of any values set in the driver-specific dialog box. If duplex printing is enabled and an uneven number of **NEXTFRAME** escapes are sent to the driver prior to the **ENDDOC** escape, the driver will eject an additional page before ending the print job.

ENABLEPAIRKERNING

Syntax short Escape(hDC, ENABLEPAIRKERNING, sizeof(int), lpNewKernFlag, lpOldKernFlag)

This escape enables or disables the driver's ability to kern character pairs automatically. Kerning is the process of adding or subtracting space between characters in a string of text.

When pair kerning is enabled, the driver automatically kerns those pairs of characters that are listed in the font's character-pair kerning table. The driver reflects this kerning both on the printer and in **GetTextExtent** function calls.

Parameters

hDC **HDC** Identifies the device context.

lpNewKernFlag **LPINT** Points to a short-integer value that specifies whether automatic pair kerning is to be enabled (1) or disabled (0).

lpOldKernFlag **LPINT** Points to a short-integer value that will receive the previous automatic pair-kerning value.

Return value The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful or not implemented.

Comments The default state of this escape is zero; automatic character-pair kerning is disabled.

A driver does not have to support the **ENABLEPAIRKERNING** escape just because it supplies the character-pair kerning table to the application via the **GETPAIRKERNTABLE** escape. In the case where the **GETPAIRKERNTABLE** escape is supported but the **ENABLEPAIRKERNING** escape is not, the application must properly space the kerned characters on the output device using the **ExtTextOut** function.

ENABLERELATIVEWIDTHS

Syntax short Escape(hDC, ENABLERELATIVEWIDTHS, sizeof(int), lpNewWidthFlag, lpOldWidthFlag)

This escape enables or disables relative character widths. When relative widths are disabled (the default), each character's width can be expressed as a number of device units. This guarantees that the extent of a string will equal the sum of the extents of the characters in the string. This allows applications to build an extent table by using one-character **GetTextExtent** function calls.

When relative widths are enabled, the sum of a string may not equal the sum of the widths of the characters. Applications that enable this feature are expected to retrieve the font's extent table and compute relatively scaled string widths.

Parameters

hDC **HDC** Identifies the device context.

lpNewWidthFlag **LPINT** Points to a short-integer value that specifies whether relative widths are to be enabled (1) or disabled (0).

lpOldWidthFlag **LPINT** Points to a short-integer value that will receive the previous relative character width value.

Return value The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful or not implemented.

Comments The default state of this escape is zero; relative character widths are disabled.

The values specified as font units and accepted and returned by the escapes described in this chapter are returned in the relative units of the font when the **ENABLERELATIVEWIDTHS** escape is enabled.

It is assumed that only linear-scaling devices will be dealt with in a relative mode. Nonlinear-scaling devices do not implement this escape.

ENDDOC

Syntax short Escape(hDC, ENDDOC, NULL, NULL, NULL)

This escape ends a print job started by a **STARTDOC** escape.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments If an application encounters a printing error or a canceled print operation, it must not attempt to terminate the operation by using the **Escape** function with either the **ENDDOC** or **ABORTDOC** escape. GDI automatically terminates the operation before returning the error value.

If the application displays a dialog box to allow the user to cancel the print operation, it must send the **ENDDOC** escape before destroying the dialog box.

The application must send the **ENDDOC** escape before freeing the procedure-instance address of the abort function, if any.

END_PATH

Syntax short Escape(hDC, END_PATH, sizeof(PATH_INFO), lpInData, NULL)

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

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

To draw a path, an application first issues the **BEGIN_PATH** escape. It then draws the primitives defining the border of the desired shape and issues an **END_PATH** escape.

The **END_PATH** escape takes as a parameter a pointer to a structure specifying the manner in which the path is to be rendered. The structure specifies whether or not the path is to be drawn and whether it is open or closed. Open paths define polylines, and closed paths define fillable polygons.

Parameters

hDC **HDC** Identifies the device context.

lpInData **PATH_INFO FAR *** Points to a **PATH_INFO** data structure that defines how the path is to be rendered. See the following "Comments" section for more information on this data structure.

Return value

The return value specifies the current path nesting level. If the escape is successful, the return value is the number of **BEGIN_PATH** escape calls without a corresponding **END_PATH** call. Otherwise, the return value is –1.

Comments

An application may begin a subpath within another path. If the subpath is closed, it is treated exactly like a polygon. If it is open, it is treated exactly like a polyline.

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

The *lpInData* parameter points to a **PATH_INFO** data structure that specifies how to render the path. This data structure has the following form:

```
typedef struct {
       short    RenderMode;
       BYTE     FillMode;
       BYTE     BkMode;
       LOGPEN   Pen;
       LOGBRUSH Brush;
       DWORD    BkColor;
}PATH_INFO;
```

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

Field	Description
RenderMode	Specifies how the path is to be rendered. It may be one of the following values: **Value** — **Meaning** NO_DISPLAY (0) — The path is not drawn. OPEN (1) — The path is drawn as an open polygon. CLOSED (2) — The path is drawn as a closed polygon.
FillMode	Specifies how the path is to be filled. It can be one of the following values: **Value** — **Meaning** ALTERNATE (1) — The fill is done using the alternate fill algorithm. WINDING (2) — The fill is done using the winding fill algorithm.
BkMode	Specifies the background mode for filling the path. It can be one of the following values: **Value** — **Meaning** OPAQUE — The background is filled with the background color before the brush is drawn. TRANSPARENT — The background is not changed.
Pen	Specifies the pen with which the path is to be drawn. If **RenderMode** is set to NO_DISPLAY, the pen is ignored.
Brush	Specifies the brush with which the path is to be filled. If **RenderMode** is set to NO_DISPLAY or OPEN, the brush is ignored.
BkColor	Specifies the color with which the path is filled if **BkMode** is set to OPAQUE.

ENUMPAPERBINS

Syntax short Escape(hDC, ENUMPAPERBINS, sizeof(int), lpNumBins, lpOutData)

This escape retrieves attribute information about a specified number of paper bins. The **GETSETPAPERBINS** escape retrieves the number of bins available on a printer. This escape is provided only for backward compatibility. An application should call the **ExtDeviceMode** function instead.

Parameters

hDC **HDC** Identifies the device context.

lpNumBins **LPINT** Points to an integer that specifies the number of bins for which information is to be retrieved.

lpOutData **LPSTR** Points to a data structure to which information about the paper bins is copied. The size of the structure depends on the number of bins for which information was requested. See the following "Comments" section for a description of this data structure.

Return value The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful or not implemented.

Comments The data structure to which the *lpOutData* parameter points consists of two arrays. The first is an array of short integers containing the paper-bin identifier numbers in the following format:

```
short      BinList[cBinMax]
```

The number of integers in the array (*cBinMax*) is equal to the value pointed to by the *lpNumBins* parameter.

The second array in the data structure to which *lpOutData* points is an array of characters in the following format:

```
char PaperNames[cBinMax][cchBinName]
```

The *cBinMax* value is equal to the value pointed to by the *lpNumBins* parameter; the *cchBinName* value is the length of each string (currently 24).

ENUMPAPERMETRICS

Syntax short Escape(hDC, ENUMPAPERMETRICS, sizeof(int), lpMode, lpOutData)

This escape performs one of two functions according to the mode:

- It determines the number of paper types supported and returns this value, which can then be used to allocate an array of **RECT** data structures.
- It returns one or more **RECT** data structures that define the areas on the page that can receive an image.

This escape is provided only for backward compatibility. An application should call the **ExtDeviceMode** function instead.

Parameters

hDC **HDC** Identifies the device context.

lpMode **LPINT** Points to an integer that specifies the mode for the escape. It can be one of the following values:

- 0 The return value indicates how many **RECT** data structures are required to contain the information about the available paper types.
- 1 The array of **RECT** structures to which *lpOutData* points is filled with the information.

lpOutData **LPRECT** Points to an array of **RECT** data structures that return all the areas that can receive an image.

Return value The return value is positive if successful, zero if the escape is not implemented, and negative if an error occurred.

EPSPRINTING

Syntax short Escape(hDC, EPSPRINTING, sizeof(BOOL), lpBool, NULL)

This escape suppresses the output of the Windows PostScript header control section, which is about 7K. If an application uses this escape, no GDI calls are allowed.

Parameters *hDC* **HDC** Identifies the device context.

lpBool **BOOL FAR** * Points to a Boolean value indicating that downloading should be enabled (TRUE) or disabled (FALSE).

Return value The return value is positive if successful, zero if the escape is not implemented, and negative if an error occurred.

EXT_DEVICE_CAPS

Syntax short Escape(hDC, EXT_DEVICE_CAPS, sizeof(int), lpIndex, lpCaps)

This escape retrieves information about device-specific capabilities. It supplements the **GetDeviceCaps** function.

Parameters *hDC* **HDC** Identifies the device context.

lpIndex **LPINT** Points to a short integer specifying the index of the capability to be retrieved. It can be any one of the following values:

- R2_CAPS (1) The *lpCaps* parameter indicates which of the 16 binary raster operations the device driver supports. A bit will be set for each supported raster operation. For further information, see the description of the **SetROP2** function in Chapter 4, "Functions directory," in *Reference, Volume 1.*

- PATTERN_CAPS (2) The *lpCaps* parameter returns the maximum dimensions of a pattern brush bitmap. The low-order word of the capability value contains the maximum width of a pattern brush bitmap, and the high-order word contains the maximum height.
- PATH_CAPS (3) The *lpCaps* parameter indicates whether the device is capable of creating paths using alternate and winding interiors, and whether the device can do exclusive or inclusive clipping to path interiors. The path capabilities are obtained using the logical OR operation on the following values:
  - PATH_ALTERNATE (1)
  - PATH_WINDING (2)
  - PATH_INCLUSIVE (4)
  - PATH_EXCLUSIVE (8)
- POLYGON_CAPS (4) The *lpCaps* parameter returns the maximum number of polygon points supported by the device. The capability value is an unsigned value specifying the maximum number of points.
- PATTERN_COLOR_CAPS (5) The *lpCaps* parameter indicates whether the device can convert monochrome pattern bitmaps to color. The capability value is 1 if the device can do pattern bitmap color conversions, and zero if it cannot.
- R2_TEXT_CAPS (6) The *lpCaps* parameter indicates whether the device is capable of performing binary raster operations on text. The low-order word of the capability value specifies which raster operations are supported for text. A bit is set for each supported raster operation, as in the R2_CAPS escape. The high-order word specifies the type of text to which the raster capabilities apply. It is obtained by applying the logical OR operation to the following values together:
  - RASTER_TEXT (1)
  - DEVICE_TEXT (2)
  - VECTOR_TEXT (4)
- POLYMODE_CAPS (7) Specifies which poly modes are supported by the printer driver. The capability value is obtained by using the bitwise OR operator to combine a bit in the corresponding position for each supported poly

mode. For example, if the printer supports the PM_POLYSCANLINE and PM_BEZIER poly modes, the capability value would be:

```
(1 PM_POLYSCANLINE)(PM_BEZIER)
```

See the description of the **SET_POLY_MODE** escape for information on the poly modes.

lpCaps **DWORD FAR *** Points to a 32-bit integer to which the capabilities will be copied.

Return value The return value is nonzero if the specified extended capability is supported, and zero if it is not.

EXTTEXTOUT

Syntax short Escape(hDC, EXTTEXTOUT, sizeof(EXTTEXT_STRUCT), lpInData, NULL)

This escape provides an efficient way for the application to call the GDI **TextOut** function when justification, letter spacing, and/or kerning are involved.

This function is provided only for backward compatibility. New applications should use the GDI **ExtTextOut** function instead.

Parameters *hDC* **HDC** Identifies the device context.

lpInData **EXTTEXT_STRUCT FAR *** Points to an **EXTTEXT_STRUCT** data structure that specifies the initial position, characters, and character widths of the string. See the following "Comments" section for more information on the **EXTTEXT_STRUCT** data structure.

Return value The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful or not implemented.

Comments The **EXTEXT_STRUCT** data structure has the following format:

```
typedef struct {
    WORD      X;
    WORD      Y;
    WORD FAR *lpText;
    WORD FAR *lpWidths;
} EXTTEXT_STRUCT;
```

This structure has the following fields.

Field	Description
X	Specifies the *x*-coordinate of the upper-left corner of the string's starting point.
Y	Specifies the *y*-coordinate of the upper-left corner of the string's starting point.
lpText	Points to an array of *cch* character codes, where *cch* is the number of bytes in the string (*cch* is also the number of words in the width array).
lpWidths	Points to an array of *cch* character widths to use when printing the string. The first character appears at (**X**,**Y**), the second at (**X** + **lpWidths**[0],**Y**), the third at (**X** + **lpWidths**[0] + **lpWidths**[1],**Y**), and so on. These character widths are specified in the font units of the currently selected font. (The character widths will always be equal to device units unless the application has enabled relative character widths.) The units contained in the width array are specified as font units of the device.

FLUSHOUTPUT

Syntax short Escape(hDC, FLUSHOUTPUT, NULL, NULL, NULL)

This escape clears all output from the device's buffer.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

GETCOLORTABLE

Syntax short Escape(hDC, GETCOLORTABLE, sizeof(int), lpIndex, lpColor)

This escape retrieves an RGB color-table entry and copies it to the location specified by the *lpColor* parameter.

Parameters

hDC **HDC** Identifies the device context.

lpIndex **LPINT** Points to a short-integer value that specifies the index of a color-table entry. Color-table indexes start at zero for the first table entry.

lpColor **DWORD FAR** * Points to the long-integer value that will receive the RGB color value for the given entry.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

GETEXTENDEDTEXTMETRICS

Syntax short Escape(hDC, GETEXTENDEDTEXTMETRICS, sizeof(WORD), lpInData, lpOutData)

This escape fills the buffer pointed to by the *lpOutData* parameter with the extended text metrics for the selected font.

Parameters

hDC	**HDC** Identifies the device context.
lpInData	**WORD FAR** * Points to an unsigned 16-bit integer that specifies the number of bytes pointed to by the *lpOutData* parameter.
lpOutData	**EXTTEXTMETRIC FAR** * Points to an **EXTTEXTMETRIC** data structure. See the following "Comments" section for a description of this data structure.

Return value The return value specifies the number of bytes copied to the buffer pointed to by the *lpOutData* parameter. This value will never exceed that specified in the *nSize* field pointed to by the *lpInData* parameter. The return value is zero if the escape fails or is not implemented.

Comments The *lpOutData* parameter points to an **EXTTEXTMETRIC** data structure which has the following format:

```
typedef struc{
   short etmSize;
   short etmPointSize;
   short etmOrientation;
   short etmMasterHeight;
   short etmMinScale;
   short etmMaxScale;
   short etmMasterUnits;
   short etmCapHeight;
   short etmXHeight;
   short etmLowerCaseAscent;
   short etmLowerCaseDescent;
   short etmSlant;
   short etmSuperScript;
   short etmSubScript;
   short etmSuperScriptSize;
   short etmSubScriptSize;
   short etmUnderlineOffset;
short etmUnderlineWidth;
short etmDoubleUpperUnderlineOffset;
short etmDoubleLowerUnderlineOffset;
short etmDoubleUpperUnderlineWidth;
short etmDoubleLowerUnderlineWidth;
short etmStrikeOutOffset;
short etmStrikeOutWidth;
WORD  etmKernPairs;
WORD  etmKernTracks;
}EXTTEXTMETRIC;
```

The **EXTTEXTMETRIC** data structure has the following fields:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><b>etmSize</b></td><td>Specifies the size of the structure in bytes.</td></tr>
<tr><td><b>etmPointSize</b></td><td>Specifies the nominal point size of this font in twips (twentieths of a point, or 1/1440 inch). This is the intended size of the font; the actual size may differ slightly depending on the resolution of the device.</td></tr>
<tr><td><b>etmOrientation</b></td><td>Specifies the orientation of the font. The <b>etmOrientation</b> field may be any of the following values:
<table>
<tr><th>Value</th><th>Meaning</th></tr>
<tr><td>0</td><td>Either orientation</td></tr>
<tr><td>1</td><td>Portrait</td></tr>
<tr><td>2</td><td>Landscape</td></tr>
</table>
These values refer to the ability of this font to be placed on a page with the given orientation. A portrait page has a height that is greater than its width. A landscape page has a width that is greater than its height.</td></tr>
<tr><td><b>etmMasterHeight</b></td><td>Specifies the font size in device units for which the values in this font's extent table are exact.</td></tr>
<tr><td><b>etmMinScale</b></td><td>Specifies the minimum valid size for this font. The following equation illustrates how the minimum point size is determined:
smallest point size
= etmMinScale * 72/dfVertRes
The value 72 represents the number of points per inch. The <i>dfVertRes</i> value is the number of dots per inch.</td></tr>
<tr><td><b>etmMaxScale</b></td><td>Specifies the maximum valid size for this font. The following equation illustrates how the maximum point size is determined:
largest point size
= etmMaxScale * 72/dfVertRes</td></tr>
</table>

The value 72 represents the number of points per inch. The *dfVertRes* value is the number of dots per inch.

etmMasterUnits Specifies the integer number of units per em where an em equals **etmMasterHeight**. That is, **etmMasterUnits** is **emtMasterHeight** expressed in font units rather than device units.

etmCapHeight Specifies the height in font units of uppercase characters in the font. Typically, this is the height of the capital H.

etmXHeight Specifies the height in font units of lowercase characters in the font. Typically, this is the height of the lowercase x.

etmLowerCaseAscent Specifies the distance in font units that the ascender of lowercase letters extends above the baseline. Typically, this is the height of the lowercase d.

etmLowerCaseDescent Specifies the distance in font units that the descender of lowercase letters extends below the baseline. Typically, this is specified for the descender of the lowercase p.

etmSlant Specifies for an italicized or slanted font the angle of the slant measured in tenths of a degree clockwise from the upright version of the font.

etmSuperScript Specifies in font units the recommended amount to offset superscript characters from the baseline. This is typically a negative value.

etmSubScript Specifies in font units the recommended amount to offset subscript characters from the baseline. This is typically a positive value.

etmSuperScriptSize Specifies in font units the recommended size of superscript characters for this font.

etmSubScriptSize Specifies in font units the recommended size of subscript characters for this font.

etmUnderlineOffset Specifies in font units the offset downward from the baseline where the top of a single underline bar should appear.

etmUnderlineWidth Specifies in font units the thickness of the underline bar.

etmDoubleUpperUnderlineOffset Specifies the offset in font units downward from the baseline where the top of the upper double underline bar should appear.

etmDoubleLowerUnderlineOffset Specifies the offset in font units downward from the baseline where the top of the lower double underline bar should appear.

etmDoubleUpperUnderlineWidth Specifies in font units the thickness of the upper underline bar.

etmDoubleLowerUnderlineWidth	Specifies in font units the thickness of the lower underline bar.
etmStrikeOutOffset	Specifies in font units the offset upward from the baseline where the top of a strike-out bar should appear.
etmStrikeOutWidth	Specifies the thickness in font units of the strike-out bar.
etmKernPairs	Specifies the number of character kerning pairs defined for this font. An application can use this value to calculate the size of the pair-kern table returned by the **GETPAIRKERNTABLE** escape. It will not be greater than 512 kern pairs.
etmKernTracks	Specifies the number of kerning tracks defined for this font. An application can use this value to calculate the size of the track-kern table returned by the **GETTRACKKERNTABLE** escape. It will not be greater than 16 kern tracks.

The values returned in many of the fields of the **EXTTEXTMETRIC** structure are affected by whether relative character widths are enabled or disabled. For more information, see the description of **ENABLERELATIVEWIDTHS** escape earlier in this chapter.

GETEXTENTTABLE

Syntax short Escape(hDC, GETEXTENTTABLE, sizeof(CHAR_RANGE_STRUCT), lpInData, lpOutData)

This escape retrieves the width (extent) of individual characters from a group of consecutive characters in the selected font's character set.

Parameters

hDC **HDC** Identifies the device context.

lpInData **LPSTR** Points to a **CHAR_RANGE_STRUCT** data structure that defines the range of characters for which the width is to be retrieved. See the following "Comments" section for more information on the **CHAR_RANGE_STRUCT** data structure.

lpOutData **LPINT** Points to an array of short integers that receives the character widths. The size of the array must be at least (**chLast** – **chFirst** + 1).

Return value The return value specifies the outcome of the escape. It is 1 if the escape is successful, and zero if the escape is not successful. If the escape is not implemented, the return value is zero.

Comments The *lpInData* parameter points to a **CHAR_RANGE_STRUCT** data structure that defines the range of characters for which the width is to be retrieved. The **CHAR_RANGE_STRUCT** structure has the following format:

```
typedef struct {
    BYTE chFirst;
    BYTE chLast;
} CHAR_RANGE_STRUCT
```

This structure has the following fields:

Field	Description
chFirst	Specifies the character code of the first character whose width is to be retrieved.
chLast	Specifies the character code of the last character whose width is to be retrieved.

The values retrieved are affected by whether relative character widths are enabled or disabled. For more information, see the **ENABLERELATIVEWIDTHS** escape, earlier in this chapter.

GETFACENAME

Syntax short Escape(hDC, GETFACENAME, NULL, NULL, lpFaceName)

This escape retrieves the face name of the current physical font.

Parameters *hDC* **HDC** Identifies the device context.

lpFaceName **LPSTR** Points to a buffer of characters to receive the face name. This buffer must be at least 60 bytes in length.

Return value The return value is positive if the escape was successful, zero if the escape is not implemented, or negative if an error occurred.

GETPAIRKERNTABLE

Syntax short Escape(hDC, GETPAIRKERNTABLE, NULL, NULL, lpOutData)

This escape fills the buffer pointed to by the *lpOutData* parameter with the character-pair kerning table for the selected font.

Parameters *hDC* **HDC** Identifies the device context.

lpOutData **KERNPAIR FAR** * Points to an array of **KERNPAIR** data structures. This array must be large enough to accommodate the font's entire character-pair kerning table. The number of character-kerning pairs in the font can be obtained from the **EXTTEXTMETRIC** data structure returned by the **GETEXTENDEDTEXTMETRICS** escape. See the following "Comments" section for the format of the **KERNPAIR** data structure.

Return value The return value specifies the number of **KERNPAIR** structures copied to the buffer. This value is zero if the font does not have kerning pairs defined, the escape fails, or is not implemented.

Comments The **KERNPAIR** data structure has the following format:

```
typedef struc {
   union {
           BYTE each  [2];  /* UNION: 'each'  and 'both'
                               share the same memory */
           WORD both;
           } kpPair;
   short    kpKernAmount;
   } KERNPAIR;
```

The **KERNPAIR** structure contains the following fields:

Field	Description
kpPair.each[0]	Specifies the character code for the first character in the kerning pair.
kpPair.each[1]	Specifies the character code for the second character in the kerning pair.
kpPair.both	Specifies a **WORD** in which the first character in the kerning pair is in the low-order byte and the second character is in the high-order byte.
kpKernAmount	Specifies the signed amount that this pair will be kerned if they appear side by side in the same font and size. This value is typically negative since pair-kerning usually results in two characters being set more tightly than normal.

The array of **KERNPAIR** structures is sorted in increasing order by the **kpPair.both** field.

The values returned in the **KERNPAIR** structures are affected by whether relative character widths are enabled or disabled. For more information, see the description of the **ENABLERELATIVEWIDTHS** escape earlier in this chapter.

GETPHYSPAGESIZE

Syntax short Escape(hDC, GETPHYSPAGESIZE, NULL, NULL, lpDimensions)

This escape retrieves the physical page size and copies it to the location pointed to by the *lpDimensions* parameter.

Parameters

hDC	**HDC** Identifies the device context.
lpDimensions	**LPPOINT** Points to a **POINT** data structure that will receive the physical page dimensions. The **x** field of the **POINT** data structure receives the horizontal size in device units, and the **y** field receives the vertical size in device units.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

GETPRINTINGOFFSET

Syntax short Escape(hDC, GETPRINTINGOFFSET, NULL, NULL, lpOffset)

This escape retrieves the offset from the upper-left corner of the physical page where the actual printing or drawing begins. This escape is generally not useful for devices that allow the user to set the origin of the printable area directly.

Parameters

hDC	**HDC** Identifies the device context.
lpOffset	**LPPOINT** Points to a **POINT** structure that will receive the printing offset. The **x** field of the **POINT** structure receives the horizontal coordinate of the printing offset in device units, and the **y** field receives the vertical coordinate of the printing offset in device units.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

GETSCALINGFACTOR

Syntax short Escape(hDC, GETSCALINGFACTOR, NULL, NULL, lpFactors)

This escape retrieves the scaling factors for the *x*- and *y*-axes of a printing device. For each scaling factor, the escape copies an exponent of 2 to the location pointed to by the *lpFactors* parameter. For example, the value 3 is copied to *lpFactors* if the scaling factor is 8.

Scaling factors are used by printing devices that support graphics at a smaller resolution than text.

Parameters

hDC **HDC** Identifies the device context.

lpFactors **LPPOINT** Points to the **POINT** data structure that will receive the scaling factor. The **x** field of the **POINT** structure receives the scaling factor for the *x*-axis, and the **y** field receives the scaling factor for the *y*-axis.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

GETSETPAPERBINS

Syntax short Escape(hDC, GETSETPAPERBINS, nCount, lpInData, lpOutData)

This escape retrieves the number of paper bins available on a printer and sets the current paper bin. See the following "Comments" section for more information on the actions performed by this escape.

Parameters

hDC **HDC** Identifies the device context.

nCount **int** Specifies the number of bytes pointed to by the *lpInData* parameter.

lpInData **BinInfo FAR** * Points to a **BinInfo** data structure that specifies the new paper bin. It may be set to NULL.

lpOutData **BinInfo FAR** * Points to a **BinInfo** data structure that contains information about the current or previous paper bin and the number of bins available.

Comments There are three possible actions for this escape, depending on the values passed in the *lpInData* and *lpOutData* parameters:

lpInData	**lpOutData**	**Action**
NULL	**BinInfo**	Retrieves the number of bins and the number of the current bin.
BinInfo	**BinInfo**	Sets the current bin to the number specified in the **BinNumber** field of the data structure to which *lpInData* points and retrieves the number of the previous bin.
BinInfo	NULL	Sets the current bin to the number specified in the **BinNumber** field of the data structure to which *lpInData* points.

The **BinInfo** data structure has the following format:

```
typedef struct{
    DWORD  BinNumber;
    DWORD  NbrofBins;
    DWORD         Reserved;
    DWORD         Reserved;
    DWORD         Reserved;
    DWORD         Reserved;
} BinInfo;
```

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

Field	Description
BinNumber	Identifies the current or previous paper bin.
NbrofBins	Specifies the number of paper bins available.

When setting a new bin, the setting does not take effect until a new device context is created (without initialization data). The setting will take immediate effect if the high bit of the bin number is set, so that the next page printed will come from the new bin. For example, 0x8001 uses the second bin immediately whenever 0x0001 sets the same bin as the default for later print jobs.

In general, only the immediate-selection form should be used by applications. Setting the bin for future print jobs is supported for backward compatibility to an earlier form of this escape which appeared in some versions of HP's Page Control Language (PCL) and PostScript.

GETSETPAPERMETRICS

Syntax short Escape(hDC, GETSETPAPERMETRICS, sizeof(RECT), lpNewPaper, lpPrevPaper)

This escape sets the paper type according to the given paper metrics information. It also retrieves the current printer's paper metrics information. This escape is provided only for backward compatibility. An application should call the **ExtDeviceMode** function instead.

This escape expects a **RECT** data structure representing the imageable area of the physical page and assumes the origin is in the upper-left corner.

Parameters *hDC* **HDC** Identifies the device context.

lpNewPaper **LPRECT** Points to a **RECT** data structure that defines the new imageable area.

lpPrevPaper **LPRECT** Points to a **RECT** data structure that receives the previous imageable area.

Return value The return value is positive if successful, zero if the escape is not implemented, and negative if an error occurs.

Comments This escape is provided only for backward compatibility. New applications should use the GDI **DeviceCapabilities** and **ExtDeviceMode** functions instead.

GETSETPAPERORIENT

Syntax short Escape(hDC, GETSETPAPERORIENT, nCount, lpInData, NULL)

This escape returns or sets the current paper orientation. This escape is provided only for backward compatibility. An application should call the **ExtDeviceMode** function instead.

Parameters

hDC **HDC** Identifies the device context.

nCount Specifies the number of bytes pointed to by the *lpInData* parameter.

lpInData **ORIENT FAR *** Points to an **ORIENT** data structure that specifies the new paper orientation. See the following "Comments" section for a description of this data structure. It may be set to NULL, in which case the **GETSETPAPERORIENT** escape returns the current paper orientation.

Return value The return value specifies the current orientation if *lpInData* is NULL; otherwise, it is the previous orientation. The return value is –1 if the escape failed.

Comments This escape is provided only for backward compatibility. New applications should use the GDI **DeviceCapabilities** and **ExtDeviceMode** functions instead.

The **ORIENT** data structure has the following format:

```
typedef struct{
    DWORD  Orientation;
    DWORD  Reserved;
    DWORD  Reserved;
    DWORD  Reserved;
    DWORD  Reserved;
} ORIENT;
```

The **Orientation** field can be either of these values:

Value	Meaning
1	The new orientation is portrait.
2	The new orientation is landscape.

This escape is also known as **GETSETPAPERORIENTATION.**

GETSETSCREENPARAMS

Syntax short Escape(hDC, GETSETSCREENPARAMS, sizeof(SCREENPARAMS), lpInData, lpOutData)

This escape retrieves or sets the current screen information for rendering halftones.

Parameters

hDC **HDC** Identifies the device context.

lpInData **SCREENPARAMS FAR** * Points to a **SCREENPARAMS** data structure that contains the new screen information. This parameter may be NULL.

lpOutData **SCREENPARAMS FAR** * Points to a **SCREENPARAMS** data structure that retrieves the previous screen information. This parameter may be NULL.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments This escape affects how device-independent bitmaps (DIBs) are rendered and how color objects are filled.

The **SCREENPARAMS** data structure has the following format:

```
typedef struct {
  int   angle;
  int   frequency;
  DWORD types;
} SCREENPARAMS;
```

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

Field	Description
angle	Specifies in degrees the angle of the halftone screen.
frequency	Specifies in dots per inch of the screen frequency.
types	Is a mask containing bits which indicate the type of screen cell. If a pointer to this structure is passed as the *lpInData* parameter, only one bit may be set. If the *lpOutData* parameter contains a pointer to

this structure, when the escape returns, the *types* field will have a bit set for each type supported by the printer driver. Acceptable bit values are:

- DIAMOND
- DOT
- ELLIPSE
- LINE

GETTECHNOLOGY

Syntax short Escape(hDC, GETTECHNOLOGY, NULL, NULL, lpTechnology)

This escape retrieves the general technology type for a printer, thereby allowing an application to perform technology-specific actions.

Parameters *hDC* **HDC** Identifies the device context.

lpTechnology **LPSTR** Points to a buffer to which the driver copies a null-terminated string containing the printer technology type, such as "PostScript."

Return value The return value specifies the outcome of the escape. It is 1 if the escape is successful, and is zero if the escape is not successful or is not implemented.

GETTRACKKERNTABLE

Syntax short Escape(hDC, GETTRACKKERNTABLE, NULL, NULL, lpOutData)

This escape fills the buffer pointed to by the *lpOutData* parameter with the track-kerning table for the currently selected font.

Parameters *hDC* **HDC** Identifies the device context.

lpOutdata **KERNTRACK FAR** * Points to an array of **KERNTRACK** structures. This array must be large enough to accommodate all the font's kerning tracks. The number of tracks in the font can be obtained from the **EXTTEXTMETRIC** structure returned by the **GETEXTENDEDTEXTMETRICS** escape. See the following "Comments" section for the format of the **KERNTRACK** data structure.

Return value The return value specifies the number of **KERNTRACK** structures copied to the buffer. This value is zero if the font does not have kerning tracks defined, or if the escape fails or is not implemented.

Comments The **KERNTRACK** data structure has the following format:

```
typedef struct {
   short    ktDegree;
   short    ktMinSize;
   short    ktMinAmount;
   short    ktMaxSize;
   short    ktMaxAmount;
   } KERNTRACK;
```

The **KERNTRACK** structure contains the following fields:

Field	Description
ktDegree	Specifies the amount of track kerning. Increasingly negative values represent tighter track kerning, and increasingly positive values represent looser track kerning.
ktMinSize	Specifies in device units the minimum font size for which linear track kerning applies.
ktMinAmount	Specifies in font units the amount of track kerning to apply to font sizes less than or equal to the size specified by the **ktMinSize** field.
ktMaxSize	Specifies in device units the maximum font size for which linear track kerning applies.
ktMaxAmount	Specifies in font units the amount of track kerning to apply to font sizes greater than or equal to the size specified by the **ktMaxSize** field.

Between the **ktMinSize** and **ktMaxSize** font sizes, track kerning is a linear function from **ktMinAmount** to **ktMaxAmount**. The values returned in the **KERNTRACK** structures are affected by whether relative character widths are enabled or disabled. For more information, see the description of the **ENABLERELATIVEWIDTHS** escape earlier in this chapter.

GETVECTORBRUSHSIZE

Syntax short Escape(hDC, GETVECTORBRUSHSIZE, sizeof(LOGBRUSH), lpInData, lpOutData)

This escape retrieves in device units the size of a plotter pen used to fill closed figures. GDI uses this information to prevent the plotter pen from writing over the borders of the figure when filling closed figures.

Parameters

hDC	**HDC** Identifies the device context.
lpInData	**LOGBRUSH FAR** * Points to a **LOGBRUSH** data structure that specifies the brush for which data are to be returned.

lpOutData **LPPOINT** Points to a **POINT** data structure that contains in its second word the width of the pen in device units.

Return value The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful or is not implemented.

GETVECTORPENSIZE

Syntax short Escape(hDC, GETVECTORPENSIZE, sizeof(LOGPEN), lpInData, lpOutData)

This escape retrieves the size in device units of a plotter pen. GDI uses this information to prevent hatched brush patterns from overwriting the border of a closed figure.

Parameters *hDC* **HDC** Identifies the device context.

lpInData **LOGPEN FAR *** Points to a **LOGPEN** data structure that specifies the pen for which the width is to be retrieved.

lpOutData **LPPOINT** Points to a **POINT** data structure that contains in its second word the width of the pen in device units.

Return value The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful or if it is not implemented.

MFCOMMENT

Syntax BOOL Escape(hDC, MFCOMMENT, nCount, lpComment, NULL)

This escape adds a comment to a metafile.

Parameters *hDC* **HDC** Identifies the device context for the device on which the metafile appears.

nCount **short** Specifies the number of characters in the string pointed to by the *lpComment* parameter.

lpComment **LPSTR** Points to a string that contains the comment that will appear in the metafile.

Return value The return value specifies the outcome of the escape. It is –1 if an error such as insufficient memory or an invalid port specification occurs. Otherwise, it is positive.

NEWFRAME

Syntax short Escape(hDC, NEWFRAME, NULL, NULL, NULL)

This escape informs the device that the application has finished writing to a page. This escape is typically used with a printer to direct the device driver to advance to a new page.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is one of the following values:

Value	Meaning
SP_APPABORT	Job was terminated because the application's abort function returned zero.
SP_ERROR	General error.
SP_OUTOFDISK	Not enough disk space is currently available for spooling, and no more space will become available.
SP_OUTOFMEMORY	Not enough memory is available for spooling.
SP_USERABORT	User terminated the job through the Print Manager.

Comments Do not use the **NEXTBAND** escape with **NEWFRAME**. For banding drivers, GDI replays a metafile to the printer, simulating a sequence of **NEXTBAND** escapes.

The **NEWFRAME** escape restores the default values of the device context. Consequently, if a font other than the default font is selected when the application calls the **NEWFRAME** escape, the application must select the font again following the **NEWFRAME** escape.

NEXTBAND

Syntax short Escape(hDC, NEXTBAND, NULL, NULL, lpBandRect)

This escape informs the device driver that the application has finished writing to a band, causing the device driver to send the band to the Print Manager and return the coordinates of the next band. Applications that process banding themselves use this escape.

Parameters *hDC* **HDC** Identifies the device context.

lpBandRect **LPRECT** Points to the **RECT** data structure that will receive the next band coordinates. The device driver copies the device coordinates of the next band into this structure.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is one of the following values:

Value	Meaning
SP_APPABORT	Job was terminated because the application's abort function returned zero.
SP_ERROR	General error.
SP_OUTOFDISK	Not enough disk space is currently available for spooling, and no more space will become available.
SP_OUTOFMEMORY	Not enough memory is available for spooling.
SP_USERABORT	User terminated the job through the Print Manager.

Comments The **NEXTBAND** escape sets the band rectangle to the empty rectangle when printing reaches the end of a page.

Do not use the **NEWFRAME** escape with **NEXTBAND**.

PASSTHROUGH

Syntax short Escape(hDC, PASSTHROUGH, nCount, lpInData, NULL)

This escape allows the application to send data directly to the printer, bypassing the standard print-driver code.

⇨ To use this escape, an application must have thorough knowledge of how the particular printer operates.

Parameters

hDC **HDC** Identifies the device context.

nCount **short** Specifies the number of bytes to which the *lpInData* parameter points.

lpInData **LPSTR** Points to a structure whose first word (16 bits) contains the number of bytes of input data. The remaining bytes of the structure contain the data itself.

Return value The return value specifies the number of bytes transferred to the printer if the escape is successful. It is less than zero if the escape is not implemented, and less than or equal to zero if the escape is not successful.

Comments There may be restrictions on the kinds of device data an application can send to the device without interfering with the operation of the driver. In general, applications must avoid resetting the printer or causing the page to be printed.

It is strongly recommended that applications not perform functions that consume printer memory, such as downloading a font or a macro.

An application can avoid corrupting its data stream when issuing multiple, consecutive **PASSTHROUGH** escapes if it does not access the printer any other way during the sequence.

QUERYESCSUPPORT

Syntax short Escape(hDC, QUERYESCSUPPORT, sizeof(int), lpEscNum, NULL)

This escape determines whether a particular escape is implemented by the device driver.

Parameters *hDC* **HDC** Identifies the device context.

lpEscNum **LPINT** Points to a short-integer value that specifies the escape function to be checked.

Return value The return value specifies whether a particular escape is implemented. It is nonzero for implemented escape functions. Otherwise, it is zero.

If the *lpEscNum* parameter is set to DRAWPATTERNRECT, the return value is one of the following:

Value	Meaning
0	DRAWPATTERNRECT is not implemented.
1	DRAWPATTERNRECT is implemented for a printer other than the HP LaserJet IIP; this printer supports white rules.
2	DRAWPATTERNRECT is implemented for the HP LaserJet IIP.

RESTORE_CTM

Syntax short Escape(hDC, RESTORE_CTM, NULL, NULL, NULL)

This escape restores the previously saved current transformation matrix.

The current transformation matrix controls the manner in which coordinates are translated, rotated, and scaled by the device. By using matrices, an application can combine these operations in any order to produce the desired mapping for a particular picture.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the number of **SAVE_CTM** escape calls without a corresponding **RESTORE_CTM** call. If the escape is unsuccessful, the return value is –1.

Comments Applications should not make any assumptions about the initial contents of the current transformation matrix.

This escape uses a matrix specification based on the Microsoft OS/2 Presentation Manager graphics programming interface (GPI) model, which is an integer-coordinate system similar to the system which GDI uses.

SAVE_CTM

Syntax short Escape(hDC, SAVE_CTM, NULL, NULL, NULL)

This escape saves the current transformation matrix.
The current transformation matrix controls the manner in which coordinates are translated, rotated, and scaled by the device. By using matrices, an application can combine these operations in any order to produce the desired mapping for a particular picture.

An application can restore the matrix by using the **RESTORE_CTM** escape.

An application typically saves the current transformation matrix before changing it. This allows the application to restore the previous state upon completion of a particular operation.

Parameters *hDC* **HDC** Identifies the device context.

Return value The return value specifies the number of **SAVE_CTM** escape calls without a corresponding **RESTORE_CTM** call. The return value is zero if the escape was unsuccessful.

Comments Applications should not make any assumptions about the initial contents of the current transformation matrix.

Applications are expected to restore the contents of the current transformation matrix.

This escape uses a matrix specification based on the OS/2 Presentation Manager graphics programming interface (GPI) model, which is an integer-coordinate system similar to the system that GDI uses.

SELECTPAPERSOURCE

This escape has been superseded by the **GETSETPAPERBINS** escape and is provided only for backward compatibility. New applications should use the **GETSETPAPERBINS** escape instead.

SETABORTPROC

Syntax short Escape(hDC, SETABORTPROC, NULL, lpAbortFunc, NULL)

This escape sets the abort function for the print job.
If an application is to allow the print job to be canceled during spooling, it must set the abort function before the print job is started with the **STARTDOC** escape. Print Manager calls the abort function during spooling to allow the application to cancel the print job or to process out-of-disk-space conditions. If no abort function is set, the print job will fail if there is not enough disk space for spooling.

Parameters

hDC	**HDC** Identifies the device context.
lpAbortFunc	**FARPROC** Points to the application-supplied abort function. See the following "Comments" section for details.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments The address passed as the *lpAbortFunc* parameter must be created by using the **MakeProcInstance** function.

The callback function must use the Pascal calling convention and must be declared **FAR**. The abort function must have the following form:

Callback Function

```
short  FAR PASCAL AbortFunc(hPr, code)
HDC hPr;
short code;
```

AbortFunc is a placeholder for the application-supplied function name. The actual name must be exported by including it in an **EXPORTS** statement in the application's module-definition file.

Parameters

hPr	Identifies the device context.
code	Specifies whether an error has occurred. It is zero if no error has occurred. It is SP_OUTOFDISK if Print Manager is currently out of disk space and more disk space will become available if the application waits.

If *code* is SP_OUTOFDISK, the application does not have to abort the print job. If it does not, it must yield to Print Manager by calling the **PeekMessage** or **GetMessage** function.

Return value The return value should be nonzero if the print job is to continue, and zero if it is canceled.

SETALLJUSTVALUES

Syntax short Escape(hDC, SETALLJUSTVALUES, sizeof(JUST_VALUE_STRUCT), lpInData, NULL)

This escape sets all of the text-justification values that are used for text output.

Text justification is the process of inserting extra pixels among break characters in a line of text. The blank character is normally used as a break character.

Parameters *hDC* **HDC** Identifies the device context.

lpInData **JUST_VALUE_STRUCT FAR *** Points to a **JUST_VALUE_STRUCT** data structure that defines the text-justification values. See the following "Comments" section for more information on the **JUST_VALUE_STRUCT** data structure.

Return value The return value specifies the outcome of the escape. It is 1 if the escape is successful. Otherwise, it is zero.

Comments The *lpInData* parameter points to a **JUST_VALUE_STRUCT** data structure that describes the text-justification values used for text output. The **JUST_VALUE_STRUCT** structure has the following format:

```
typedef struct {
    short   nCharExtra;
    WORD    nCharCount;
    short   nBreakExtra;
    WORD    nBreakCount;
} JUST_VALUE_STRUCT;
```

This structure has the following fields:

Field	Description
nCharExtra	Specifies the total extra space (in font units) that must be distributed over **nCharCount** characters.

nCharCount Specifies the number of characters over which **nCharExtra** is distributed.

nBreakExtra Specifies the total extra space (in font units) that is distributed over **nBreakCount** characters.

nBreakCount Specifies the number of break characters over which **nBreakExtra** is distributed.

The units used for **nCharExtra** and **nBreakExtra** are the font units of the device and are dependent on whether relative character widths were enabled with the **ENABLERELATIVEWIDTHS** escape.

The values set with this escape apply to subsequent calls to the **TextOut** function. The driver stops distributing the extra space specified in the **nCharExtra** field when it has output the number of characters specified in the **nCharCount** field. Likewise, it stops distributing the space specified by the **nBreakExtra** field when it has output the number of characters specified by the **nBreakCount** field. A call on the same string to the **GetTextExtent** function made immediately after the call to the **TextOut** function will be processed in the same manner.

To re-enable justification with the **SetTextJustification** and **SetTextCharacterExtra** functions, an application should call the **SETALLJUSTVALUES** escape and set the **nCharExtra** and **nBreakExtra** fields to zero.

SET_ARC_DIRECTION

Syntax short Escape(hDC, SET_ARC_DIRECTION, sizeof(int), lpDirection, NULL)

This escape specifies the direction in which elliptical arcs are drawn using the GDI **Arc** function.

By convention, elliptical arcs are drawn counterclockwise by GDI. This escape lets an application draw paths containing arcs drawn clockwise.

Parameters *hDC* **HDC** Identifies the device context.

lpDirection **LPINT** Points to a short integer specifying the arc direction. It can be either of the following values:

- COUNTERCLOCKWISE (0)
- CLOCKWISE (1)

Return value The return value is the previous arc direction.

Comments This escape maps to PostScript language elements and is intended for PostScript line devices.

SET_BACKGROUND_COLOR

Syntax short Escape(hDC, SET_BACKGROUND_COLOR, nCount, lpNewColor, lpOldColor)

This escape sets and retrieves the current background color for the device.

The background color is the color of the display surface before an application draws anything on the device. This escape is particularly useful for color printers and film recorders.

This escape should be sent before the application draws anything on the current page.

Parameters

hDC **HDC** Identifies the device context.

nCount **int** Specifies the number of bytes pointed to by the *lpNewColor* parameter.

lpNewColor **DWORD FAR *** Points to a 32-bit integer specifying the desired background color. This parameter can be NULL if the application is merely retrieving the current background color.

lpOldColor **DWORD FAR *** Points to a 32-bit integer which receives the previous background color. This parameter can be NULL if the application does not use the previous background color.

Return value The return value is TRUE if the escape was successful and FALSE if it was unsuccessful.

Comments The default background color is white.

The background color is reset to the default when the device driver receives an **ENDDOC** or **ABORTDOC** escape.

SET_BOUNDS

Syntax short Escape(hDC, SET_BOUNDS, sizeof(RECT), lpInData, NULL)

This escape sets the bounding rectangle for the picture being produced by the device driver supporting the given device context. It is used when creating images in a file format such as Encapsulated PostScript (EPS) and Hewlett-Packard Graphics Language (HPGL) for which there is a device driver.

Parameters

hDC **HDC** Identifies the device context.

lpInData **LPRECT** Points to a **RECT** data structure that specifies in device coordinates a rectangle that bounds the image to be output.

Return value The return value is TRUE if the escape was successful; otherwise, the return value is FALSE.

Comments An application should issue this escape before each page in the image. For single-page images, this escape should be issued immediately before the **STARTDOC** escape.

When an application uses coordinate-transformation escapes, device drivers may not perform bounding box calculations correctly. When an application uses the **SET_BOUNDS** escape, the driver does not have to calculate the bounding box.

Applications should always use this escape to ensure support for the Encapsulated PostScript (EPS) printing capabilities that will be built into future PostScript drivers.

SET_CLIP_BOX

Syntax

```
short Escape(hDC, SET_CLIP_BOX, sizeof(RECT), lpInData,
(LPSTR)NULL)
```

This escape sets the clipping rectangle or restores the previous clipping rectangle. This escape is implemented by printer drivers that implement the coordinate-transformation escapes **TRANSFORM_CTM, SAVE_CTM,** and **RESTORE_CTM**.

When an application calls a GDI output function, GDI calculates a clipping rectangle bounding the primitive and passes both the primitive and the clipping rectangle to the printer driver. The printer driver is expected to clip the primitive to the specified bounding rectangle. However, when an application uses the coordinate-transformation escapes, the clipping rectangle calculated by GDI is usually invalid. An application can use the **SET_CLIP_BOX** escape to specify the correct clipping rectangle when coordinate transformations are used.

Parameters *hDC* **HDC** Identifies the device context.

lpClipBox **LPRECT** Points to a **RECT** data structure containing the bounding rectangle of the clipping area. If *lpClipBox* is not NULL, the previous clipping rectangle is saved and the current clipping rectangle is set to the specified bounds. If *lpClipBox* is NULL, the previous clipping rectangle is restored.

Return value The return value is TRUE if the clipping rectangle was properly set. Otherwise, it is FALSE.

SETCOLORTABLE

Syntax short Escape(hDC, SETCOLORTABLE, sizeof(COLORTABLE_STRUCT), lpInData, lpColor)

This escape sets an RGB color-table entry. If the device cannot supply the exact color, the function sets the entry to the closest possible approximation of the color.

Parameters *hDC* **HDC** Identifies the device context.

lpInData **COLORTABLE_STRUCT FAR *** Points to a **COLORTABLE_STRUCT** data structure that contains the index and RGB value of the color-table entry. See the following "Comments" section for more information on the **COLORTABLE_STRUCT** data structure.

lpColor **DWORD FAR *** Points to the long-integer value that is to receive the RGB color value selected by the device driver to represent the requested color value.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments The **COLORTABLE_STRUCT** data structure has the following format:

```
typedef struct {
    WORD   Index;
    DWORD  rgb;
} COLORTABLE_STRUCT;
```

This structure has the following fields:

Field	Description
Index	Specifies the color-table index. Color-table entries start at zero for the first entry.
rgb	Specifies the desired RGB color value.

A device's color table is a shared resource; changing the system display color for one window changes it for all windows. Only applications developers who have a thorough knowledge of the display driver should use this escape.

The **SETCOLORTABLE** escape has no effect on devices with fixed color tables.

This escape is intended for use by both printer and display drivers. However, the EGA and VGA color drivers do not support it.

This escape changes the palette used by the display driver. However, since the driver's color-mapping algorithms will probably no longer work with a different palette, an extension has been added to this function.

If the color index pointed to by the *lpInData* parameter is 0XFFFF, the driver is to leave all color-mapping functionality to the calling application. The application must use the proper color-mapping algorithm and take responsibility for passing the correctly mapped physical color to the driver (instead of the logical RGB color) in such device-driver functions as **RealizeObject** and **ColorInfo**.

For example, if the device supports 256 colors with palette indexes of 0 through 255, the application would determine which index contains the color that it wants to use in a certain brush. It would then pass this index in the low-order byte of the **DWORD** logical color passed to the **RealizeObject** device-driver function. The driver would then use this color exactly as passed instead of performing its usual color-mapping algorithm. If the application wants to reactivate the driver's color-mapping algorithm (that is, if it restores the original palette when switching from its window context), then the color index pointed to by *lpInData* should be 0xFFFE.

SETCOPYCOUNT

Syntax short Escape(hDC, SETCOPYCOUNT, sizeof(int), lpNumCopies, lpActualCopies)

This escape specifies the number of uncollated copies of each page that the printer is to print.

Parameters

hDC **HDC** Identifies the device context.

lpNumCopies **LPINT** Points to a short-integer value that contains the number of uncollated copies to be printed.

lpActualCopies **LPINT** Points to a short-integer value that will receive the number of copies to be printed. This may be less than the number requested if the requested number is greater than the device's maximum copy count.

Return value The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful. If the escape is not implemented, the return value is zero.

SETKERNTRACK

Syntax short Escape(hDC, SETKERNTRACK, sizeof(int), lpNewTrack, lpOldTrack)

This escape specifies which kerning track to use for drivers that support automatic track kerning. A kerning track of zero disables automatic track kerning.

When track kerning is enabled, the driver will automatically kern all characters according to the specified track. The driver will reflect this kerning both on the printer and in **GetTextExtent** function calls.

Parameters

hDC **HDC** Identifies the device context.

lpNewTrack **LPINT** Points to a short-integer value that specifies the kerning track to use. A value of zero disables this feature. Values in the range 1 to *nKernTracks* correspond to positions in the track-kerning table (using 1 as the first item in the table). For more information, see the description of the **EXTTEXTMETRIC** structure provided under the description of the **GETEXTENDEDTEXTMETRICS** escape.

lpOldTrack **LPINT** Points to a short-integer value that will receive the previous kerning track.

Return value The return value specifies the outcome of the escape. It is 1 if the escape is successful; it is zero if the escape is not successful or not implemented.

Comments Automatic track kerning is disabled by default.

A driver does not have to support the **ENABLEPAIRKERNING** escape just because it supplies the track-kerning table to the application by using the **GETTRACKKERNTABLE** escape. In the case where **GETTRACKKERNTABLE** is supported but the **SETKERNTRACK** escape is not, the application must properly space the characters on the output device.

SETLINECAP

Syntax short Escape(hDC, SETLINECAP, sizeof(int), lpNewCap, lpOldCap)

This escape sets the line end cap.

A line end cap is that portion of a line segment that appears on either end of the segment. The cap may be square or circular. It can extend past, or remain flush with the specified segment end points.

Parameters *hDC* **HDC** Identifies the device context.

lpNewCap **LPINT** Points to a short-integer value that specifies the end-cap type. The possible values and their meanings are given in the following list:

- –1 Line segments are drawn by using the default GDI end cap.
- 0 Line segments are drawn with a squared end point that does not project past the specified segment length.
- 1 Line segments are drawn with a rounded end point; the diameter of this semicircular arc is equal to the line width.
- 2 Line segments are drawn with a squared end point that projects past the specified segment length. The projection is equal to half the line width.

lpOldCap **LPINT** Points to a short-integer value that specifies the previous end-cap setting.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments The interpretation of this escape varies with page-description languages (PDLs). Consult the PDL documentation for its exact meaning.

This escape is also known as **SETENDCAP**.

SETLINEJOIN

Syntax short Escape(hDC, SETLINEJOIN, sizeof(int), lpNewJoin, lpOldJoin)

This escape specifies how a device driver will join two intersecting line segments. The intersection can form a rounded, squared, or mitered corner.

Parameters *hDC* **HDC** Identifies the device context.

lpNewJoin **LPINT** Points to a short-integer value that specifies the type of intersection. The possible values and their meanings are given in the following list:

- –1 Line segments are joined by using the default GDI setting.
- 0 Line segments are joined with a mitered corner; the outer edges of the lines extend until they meet at an angle. This is referred to as a miter join.
- 1 Line segments are joined with a rounded corner; a semicircular arc with a diameter equal to the line width is drawn around the point where the lines meet. This is referred to as a round join.
- 2 Line segments are joined with a squared end point; the outer edges of the lines are not extended. This is referred to as a bevel join.

lpOldJoin **LPINT** Points to a short-integer value that specifies the previous line join setting.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments The interpretation of this escape varies with page-description languages (PDLs). Consult the PDL documentation for its exact meaning.

If an application specifies a miter join but the angle of intersection is too small, the device driver ignores the miter setting and uses a bevel join instead.

SET_MIRROR_MODE

Syntax short Escape(hDC, SET_MIRROR_MODE, sizeof(WORD), lpInData, (LPSTR)NULL)

This escape sets the current mirror mode. The mirror mode produces mirror images along the horizontal axis, the vertical axis, or both axes.

To produce a mirror image of a given page, the application issues the **SET_MIRROR_MODE** escape before drawing the first primitive to be mirrored. When the last mirrored primitive has been drawn, the application issues a second **SET_MIRROR_MODE** escape to turn off mirroring.

Parameters *hDC* **HDC** Identifies the device context.

lpMirrorMode **LPINT** Points to a short integer that specifies the mirror mode. It must be one of the following values:

- MIRROR_NONE (0) Disable mirroring.
- MIRROR_HORIZONTAL (1) Mirror along the horizontal axis.
- MIRROR_VERTICAL (2) Mirror along the vertical axis.
- MIRROR_BOTH (3) Mirror along both axes.

Return value The return value is the previous mirror mode.

Comments The default mirror mode is MIRROR_NONE.

Mirrored and unmirrored output can be mixed on a page. This allows the application to produce mirrored output with unmirrored page labels, crop marks, and so on.

SETMITERLIMIT

Syntax short Escape(hDC, SETMITERLIMIT, sizeof(int), lpNewMiter, lpOldMiter)

This escape sets the miter limit for a device. The miter limit controls the angle at which a device driver replaces a miter join with a bevel join.

Parameters

hDC **HDC** Identifies the device context.

nCount **short** Specifies the number of bytes to which the *lpNewMiter* parameter points.

lpNewMiter **LPINT** Points to a short-integer value that specifies the desired miter limit. Only values greater than or equal to –1 are valid. If this value is –1, the driver will use the default GDI miter limit.

lpOldMiter **LPINT** Points to a short-integer value that specifies the previous miter-limit setting.

Return value The return value specifies the outcome of the escape. It is positive if the escape is successful. Otherwise, it is negative.

Comments The miter limit is defined as follows:

$$\frac{\text{miter length}}{\text{line width}} = \frac{1}{\sin(x/2)}$$

X is the angle of the line join in radians.

The interpretation of this escape varies with page-description languages (PDLs). Consult the PDL documentation for its exact meaning.

SET_POLY_MODE

Syntax short Escape(hDC, SET_POLY_MODE, sizeof(int), lpMode, NULL)

This escape sets the poly mode for the device driver. The poly mode is a state variable indicating how to interpret calls to the GDI **Polygon** and **Polyline** functions.

The **SET_POLY_MODE** escape enables a device driver to draw shapes (such as Bezier curves) not supported directly by GDI. This permits applications that draw complex curves to send the curve description directly to a device without having to simulate the curve as a polygon with a large number of points.

Parameters *hDC* **HDC** Identifies the device context.

lpMode **LPINT** Points to a short integer specifying the desired poly mode. The poly mode is a state variable indicating how points in **Polygon** or **Polyline** function calls should be interpreted. All device drivers are not required to support all possible modes. A device driver returns zero if it does not support the specified mode. The *lpMode* parameter may be one of the following values:

- PM_POLYLINE (1) The points define a conventional polygon or polyline.
- PM_BEZIER (2) The points define a sequence of 4-point Bezier spline curves. The first curve passes through the first four points, with the first and fourth points as end points, and the second and third points as control points. Each subsequent curve in the sequence has the end point of the previous curve as its start point, the next two points as control points, and the third as its end point.
  The last curve in the sequence is permitted to have fewer than four points. If the curve has only one point, it is considered a point. If it has two points, it is a line segment. If it has three points, it is a parabola defined by drawing a Bezier curve with the first and third points as end points and the two control points equal to the second point.
- PM_POLYLINESEGMENT (3) The points specify a list of coordinate pairs. Line segments are drawn connecting each successive pair of points.
- PM_POLYSCANLINE (4) The points specify a list of coordinate pairs. Line segments are drawn connecting each successive pair of points. Each line segment is a

nominal-width line drawn using the current brush. Each line segment must be strictly vertical or horizontal, and scan lines must be passed in strictly increasing or decreasing order. This mode is only used for polygon calls.

Return value The return value is the previous poly mode. If the return value is zero, the device driver did not handle the request.

Comments An application should issue the **SET_POLY_MODE** escape before it draws a complex curve. It should then call the **Polyline** or **Polygon** function with the desired control points defining the curve. After drawing the curve, the application should reset the driver to its previous state by issuing the **SET_POLY_MODE** escape.

Polyline calls draw using the currently selected pen.

Polygon calls draw using the currently selected pen and brush. If the start and end points are not equal, a line is drawn from the start point to the end point before filling the polygon (or curve).

GDI treats **Polygon** calls using PM_POLYLINESEGMENT mode exactly the same as **Polyline** calls.

Four points define a Bezier curve. GDI generates the curve by connecting the first and second, second and third, and third and fourth points. GDI then connects the midpoints of these consecutive line segments. Finally, GDI connects the midpoints of the lines connecting the midpoints, and so forth.

The line segments drawn in this way converge to a curve defined by the following parametric equations, expressed as a function of the independent variable *t*.

$$X(t) = (1\text{-}t)^3x_1 + 3(1\text{-}t)^2tx_2 + 3(1\text{-}t)t^2x_3 + t^3x_4$$

$$Y(t) = (1\text{-}t)^3y_1 + 3(1\text{-}t)^2ty_2 + 3(1\text{-}t)t^2y_3 + t^3y_4$$

The points (x_1,y_1), (x_2,y_2), (x_3,y_3) and (x_4,y_4) are the control points defining the curve. The independent variable *t* varies from 0 to 1.

The points (Cx_1,Cy_1) and (Cx_2,Cy_2) are third-degree control points of a second-degree Bezier curve specified by the points (X_1,Y_1), (X_2,Y_2), and (X_3,Y_3).

Primitive types other than PM_BEZIER and PM_POLYLINESEGMENT may be added to this escape in the future. Applications should check the return value from this escape to determine whether or not the driver supports the specified poly mode.

SET_SCREEN_ANGLE

Syntax short Escape(hDC, SET_SCREEN_ANGLE, sizeof(int), lpAngle, NULL)

This escape sets the current screen angle to the desired angle and enables an application to simulate the tilting of a photographic mask in producing a color separation for a particular primary.

Parameters

hDC	**HDC** Identifies the device context.
lpAngle	**LPINT** Points to a short-integer value specifying the desired screen angle in tenths of a degree. The angle is measured counterclockwise.

Return value The return value is the previous screen angle.

Comments Four-color process separation is the process of separating the colors comprising an image into four component primaries: cyan, magenta, yellow, and black. The image is then reproduced by overprinting each primary.

In traditional four-color process printing, half-tone images for each of the four primaries are photographed against a mask tilted to a particular angle. Tilting the mask in this manner minimizes unwanted moire patterns caused by overprinting two or more colors.

The device driver defines the default screen angle.

SET_SPREAD

Syntax short Escape(hDC, SET_SPREAD, sizeof(int), lpSpread, NULL)

This function sets the amount that nonwhite primitives are expanded for a given device to provide a slight overlap between primitives to compensate for imperfections in the reproduction process.

Spot color separation is the process of separating an image into each distinct color used in the image. The image is reproduced by overprinting each successive color in the image.

When reproducing a spot-separated image, the printing equipment must be calibrated to align each page exactly on each pass. However, differences in temperature, humidity, and so forth, between passes often cause images to align imperfectly on subsequent passes. For this reason, lines in spot separations are often widened (spread) slightly to make up for problems in registering subsequent passes through the printer. This process is called trapping. The **SET_SPREAD** escape implements this process.

Parameters

hDC **HDC** Identifies the device context.

lpSpread **LPINT** Points to a short-integer value that specifies the amount, in pixels, by which all nonwhite primitives are to be expanded.

Return value The return value is the previous spread value.

Comments The default spread is zero.

The current spread applies to all bordered primitives (whether or not the border is visible) and text.

STARTDOC

Syntax short Escape(hDC, STARTDOC, nCount, lpDocName, NULL)

This escape informs the device driver that a new print job is starting and that all subsequent **NEWFRAME** escape calls should be spooled under the same job until an **ENDDOC** escape call occurs. This ensures that documents longer than one page will not be interspersed with other jobs.

Parameters

hDC **HDC** Identifies the device context.

nCount **short** Specifies the number of characters in the string pointed to by the *lpDocName* parameter.

lpDocName **LPSTR** Points to a null-terminated string that specifies the name of the document. The document name is displayed in the Print Manager window. The maximum length of this string is 31 characters plus the terminating null character.

Return value The return value specifies the outcome of the escape. It is –1 if an error such as insufficient memory or an invalid port specification occurs. Otherwise, it is positive.

Comments The correct sequence of events in a printing operation is as follows:

1. Create the device context.
2. Set the abort function to keep out-of-disk-space errors from terminating a printing operation.

   An abort procedure that handles these errors must be set by using the **SETABORTPROC** escape.
3. Begin the printing operation with the **STARTDOC** escape.

4. Begin each new page with the **NEWFRAME** escape, or each new band with the **NEXTBAND** escape.
5. End the printing operation with the **ENDDOC** escape.
6. Destroy the cancel dialog box, if any.
7. Free the procedure-instance address of the abort function.

If an application encounters a printing error or a canceled print operation, it must not attempt to terminate the operation by using the **Escape** function with either the **ENDDOC** or **ABORTDOC** escape. GDI automatically terminates the operation before returning the error value.

TRANSFORM_CTM

Syntax short Escape(hDC, TRANSFORM_CTM, 36, lpMatrix, NULL)

This escape modifies the current transformation matrix. The current transformation matrix controls the manner in which coordinates are translated, rotated, and scaled by the device. By using matrices, you can combine these operations in any order to produce the desired mapping for a particular picture.

The new current transformation matrix will contain the product of the matrix referenced by the lpMatrix parameter and the previous current transformation matrix (CTM = M " CTM).

Parameters

hDC **HDC** Identifies the device context.

lpMatrix **LPSTR** Points to a 3-by-3 array of 32-bit integer values specifying the new transformation matrix. Entries in the matrix are scaled to represent fixed-point real numbers. Each matrix entry is scaled by 65,536. The high-order word of the entry contains the whole integer portion, and the low-order word contains the fractional portion.

Return value The return value is TRUE if the escape was successful and FALSE if it was unsuccessful.

Comments When an application modifies the current transformation matrix, it must specify the clipping rectangle by issuing the **SET_CLIP_BOX** escape.

Applications should not make any assumptions about the initial value of the current transformation matrix.

The matrix specification used for this escape is based on the Microsoft OS/2 Presentation Manager graphics programming interface (GPI) model, which is an integer-coordinate system similar to the one used by GDI.

CHAPTER

13

Windows DDE protocol definition

The Microsoft Windows Dynamic Data Exchange (DDE) protocol defines the method for communicating among applications. This communication takes place as applications send messages to each other to initiate conversations, to request and share data, and to terminate conversations. This chapter describes these messages and the rules associated with their use. It also briefly describes several clipboard formats which a DDE application can register for use in a DDE conversation.

Guide to Programming provides an overview of DDE programming, including such concepts as client, server, application, topic and item. It also introduces the modes of DDE communication, including permanent data links, one-time transfers, and remote command execution, and it explains the flow of DDE messages.

Message-specific argument names bear prefixes indicating their type, as follows:

Prefix	Description
a	An atom of word length (16 bits); for example, *aName*.
cf	A registered clipboard format number (word length); for example, *cfFormat*.
f	A flag bit; for example, *fName*.
h	A handle (word length) to a global memory bject; for example, *hName*.
w	Any other word-length argument; for example, *wName*.

Using the DDE message set

Each DDE message has two parameters. The first parameter, *wParam* (word length), carries the handle of the sender's window; it is the same in all cases and so is not shown in Table 13.1. The second parameter, *lParam* (a long word, 32 bits), is composed of a low-order word and a high-order word containing message-specific arguments, as follows:

Table 13.1
DDE messages

Message	Arguments in *lParam* Low-order word	High-order word
WM_DDE_ACK		
In reply to INITIATE	*aApplication*	*aTopic*
In reply to EXECUTE	*wStatus*	*hCommands*
All other messages	*wStatus*	*aItem*
WM_DDE_ADVISE	*hOptions*	*aItem*
WM_DDE_DATA	*hData*	*aItem*
WM_DDE_EXECUTE	(Reserved)	*hCommands*
WM_DDE_INITIATE	*aApplication*	*aTopic*
WM_DDE_POKE	*hData*	*aItem*
WM_DDE_REQUEST	*cfFormat*	*aItem*
WM_DDE_TERMINATE	(Reserved)	(Reserved)
WM_DDE_UNADVISE	(Reserved)	*aItem*

An application calls the **SendMessage** function to issue the WM_DDE_INITIATE message or a WM_DDE_ACK message sent in response to WM_DDE_INITIATE. All other messages are sent using the **PostMessage** function. The window handle of the receiving window appears as the first parameter of these calls. The second parameter contains the message to be sent, the third parameter identifies the sending window, and the fourth parameter contains the message-specific arguments. For example:

```
PostMessage(hwndRecipient, WM_DDE_MESSAGE, hwndSender,
    MAKELONG(low_word, high_word))
```

The MAKELONG macro combines low_word and high_word into a long word.

Synchronizing the DDE conversation

An application window that processes DDE requests from the window of a DDE partner must process them strictly in the order in which they are received from that partner. However, when handling messages from multiple DDE partners, the window does not have to follow this "first in, first out" rule. In other words, only the conversations themselves must be synchronous; the window can shift from one conversation to another asynchronously.

For example, suppose the following messages are in a window's queue:

```
Message from window X
Message from window Y
Message from window X
```

The window must process message 1 before message 3, but it need not process message 2 before message 3. If window Y is a lower-priority DDE-conversation partner than window X, the window can defer processing the messages from window Y until it has finished dealing with the messages sent by window X. The following table shows acceptable processing orders for these messages and the relative priority implied by each order:

Order	Relative Priority
1 2 3	Window X = window Y
1 3 2	Window X > window Y
2 1 3	Window X < window Y

If an application is unable to process an incoming request because it is waiting for a DDE response, it must post a WM_DDE_ACK message with the **fBusy** flag set to 1 to prevent deadlock. An application can also send a busy WM_DDE_ACK message if for any reason the application cannot process an incoming request within a reasonable amount of time.

An application should be able to deal with the situation in which its DDE partner fails to respond with a message within a certain time-out interval. Since the length of this interval may vary depending on the nature of the application and the configuration of the user's system (including whether it is on a network), the application should provide a way for the user to specify the time-out interval.

Using atoms

The section "DDE message directory" describes the rules for allocating and deleting atoms used by each message.

Certain arguments of DDE messages (*aItem*, *aTopic*, and *aApplication*) are global atoms. Applications using these atoms must explicitly delete them to purge them from the atom list.

In all cases, the sender of a message must delete any atom which the intended receiver will not receive due to an error condition, such as failure of the **PostMessage** function.

Using shared memory objects

DDE uses shared memory objects for three purposes:

- To carry a data item value to be exchanged. This is an item referenced by the *hData* argument in the WM_DDE_DATA and WM_DDE_POKE messages.
- To carry options in a message. This is an item referenced by the *hOptions* argument in a WM_DDE_ADVISE message.
- To carry an execution-command string. This is an item referenced by the *hCommands* argument in the WM_DDE_EXECUTE message and its corresponding WM_DDE_ACK message.

Applications that receive a DDE shared memory object must treat it as read only. It must not be used as a mutual read/write area for the free exchange of data.

"DDE message directory" on page 40 describes the rules for allocating and deleting shared memory objects used by each message.

As with a DDE atom, a shared memory object should be freed properly to provide for effective memory management. Shared memory objects should be properly locked and unlocked.

In all cases, the sender of a message must delete any shared memory object which the intended receiver will not receive due to an error condition, such as failure of the **PostMessage** function.

Using clipboard formats

You can pass data by means of any of the standard clipboard formats or with a registered clipboard formats. See the description of the **SetClipboardData** function in Chapter 4, "Functions directory," in *Reference, Volume 1,* for more information on standard clipboards. See the description of the **RegisterClipboardFormat** function for information on registering clipboard formats.

A special, registered format named Link is used to identify an item in a DDE conversation. For more information, see *Guide to Programming.*

Using the System topic

Applications are encouraged to support at all times a special topic with the name System. This topic provides a context for items of information that may be of general interest to another application.

The following list contains suggested items for the System topic. This list is not exclusive. The data item values should be rendered in the CF_TEXT format. Individual elements of a System topic item value should be delimited by tab characters.

Item	Description
SysItems	A list of the System-topic items supported by the application.
Topics	A list of the topics supported by the application at the current time; this list can vary from moment to moment.
ReturnMessage	Supporting detail for the most recently used WM_DDE_ACK message. This is useful when more than eight bits of application-specific return data are required.
Status	An indication of the current status of the application. When a server receives a WM_DDE_REQUEST message for this System-topic item, it should respond by posting a WM_DDE_DATA message with a string containing either "Busy" or "Ready," as appropriate.
Formats	A list of clipboard format numbers that the application can render.

DDE message directory

This section describes the nine DDE messages. Included in each description is a list of the message-specific arguments and the rules for posting and receiving each message. The SDK contains the DDE.H header file, which defines the DDE messages and data structures described in this section.

WM_DDE_ACK

This message notifies an application of the receipt and processing of a WM_DDE_INITIATE, WM_DDE_EXECUTE, WM_DDE_DATA, WM_DDE_ADVISE, WM_DDE_UNADVISE, or WM_DDE_POKE message, and in some cases, of a WM_DDE_REQUEST message.

Parameter	Description
wParam	Identifies the sending window.
lParam	The meaning of the low-order and high-order words depends on the message to which the WM_DDE_ACK message is responding. When responding to WM_DDE_INITIATE:

Argument	Description
aApplication	Low-order word of *lParam*. An atom that contains the name of the replying application.
aTopic	High-order word of *lParam*. An atom that contains the topic with which the replying server window is associated.

When responding to WM_DDE_EXECUTE:

Argument	Description
wStatus	Low-order word of *lParam*. A series of flags that indicate the status of the response.
hCommands	High-order word of *lParam*. A handle that identifies the data item containing the command string.

When replying to all other messages:

Argument	Description
wStatus	Low-order word of *lParam*. A series of flags that indicate the status of the response.
aItem	High-order word of *lParam*. An atom that specifies the data item for which the response is sent.

Comments The *wStatus* word consists of a **DDEACK** data structure that contains the following information:

Bit	Name	Meaning
15	**fAck**	1 = Request accepted. 0 = Request not accepted.
14	**fBusy**	1 = Busy. An application is expected to set **fBusy** if it is unable to respond to the request at the time it is received. The **fBusy** flag is defined only when **fAck** is zero. 0 = Not busy.
13–8		Reserved for Microsoft use.
7–0	**bAppReturnCode**	Reserved for application-specific return codes.

Posting Except in response to the WM_DDE_INITIATE message, post the WM_DDE_ACK message by calling the **PostMessage** function, not **SendMessage**. When responding to WM_DDE_INITIATE, send the WM_DDE_ACK message with **SendMessage**.

When acknowledging any message with an accompanying *aItem* atom, the application that sends WM_DDE_ACK can reuse the *aItem* atom that accompanied the original message, or it may delete it and create a new one.

When acknowledging WM_DDE_EXECUTE, the application that sends WM_DDE_ACK should reuse the *hCommands* object that accompanied the original WM_DDE_EXECUTE message.

If an application has initiated the termination of a conversation by sending WM_DDE_TERMINATE and is awaiting confirmation, the waiting application should not acknowledge (positively or negatively) any subsequent message sent by the other application. The waiting application should delete any atoms or shared memory objects received in these intervening messages.

Receiving The application that receives WM_DDE_ACK should delete all atoms accompanying the message.

If the application receives WM_DDE_ACK in response to a message with an accompanying *hData* object, the application should delete the *hData* object.

If the application receives a negative WM_DDE_ACK message sent in reply to a WM_DDE_ADVISE message, the application should delete the *hOptions* object sent with the original WM_DDE_ADVISE message.

If the application receives a negative WM_DDE_ACK message sent in reply to a WM_DDE_EXECUTE message, the application should delete the *hCommands* object sent with the original WM_DDE_EXECUTE message.

WM_DDE_ADVISE

This message, posted by a client application, requests the receiving (server) application to supply an update for a data item whenever it changes.

Parameter	Description	
wParam	Identifies the sending window.	
lParam	Identifies the requested data and specifies how the data is to be sent.	
	Argument	**Description**
	hOptions	Low-order word of *lParam*. A handle to a global memory object that specifies how the data is to be sent.
	aItem	High-order word of *lParam*. An atom that specifies the data item being requested.

Comments The global memory object identified by *hOptions* consists of a **DDEADVISE** data structure that contains the following:

Word	Name	Content
1	**fAckReq**	If bit 15 is 1, the receiving (server) application is requested to send its WM_DDE_DATA messages with the ACK-requested bit (**fAckReq**) set. This offers a flow-control technique whereby the client application can avoid overload from incoming DATA messages.
	fDeferUpd	If bit 14 is 1, the server is requested to send its WM_DDE_DATA messages with a null *hData* handle. These messages are alarms telling the client that the source data has changed. Upon receiving one of these alarms, the client can choose to call for the latest version of the data by issuing a WM_DDE_REQUEST message, or it can choose to ignore the alarm altogether. This would typically be used when there is a substantial resource cost associated with rendering and/or assimilating the data.
	reserved	Bits 13–0 are reserved.
2	**cfFormat**	The client's preferred type of data. Must be a standard or registered clipboard data format number.

If an application supports more than one clipboard format for a single topic and item, it can post multiple WM_DDE_ADVISE messages for the topic and item, specifying a different clipboard format with each message.

Posting Post the WM_DDE_ADVISE message by calling the **PostMessage** function, not **SendMessage**.

Allocate *hOptions* by calling the **GlobalAlloc** function with the GEMEM_DDE_SHARE option.

Allocate *aItem* by calling the **GlobalAddAtom** function.

If the receiving (server) application responds with a negative WM_DDE_ACK message, the sending (client) application must delete the *hOptions* object.

Receiving Post the WM_DDE_ACK message to respond positively or negatively. When posting WM_DDE_ACK, reuse the *aItem* atom or delete it and create a new one. If the WM_DDE_ACK message is positive, delete the *hOptions* object; otherwise, do not delete the object.

WM_DDE_DATA

This message, posted by a server application, sends a data item value to the receiving (client) application, or notifies it of the availability of data.

Parameter	Description	
wParam	Identifies the sending window.	
lParam	Identifies the available data and specifies how it is sent.	
	Argument	**Description**
	hData	Low-order word of *lParam*. A handle that identifies the global memory object containing the data and additional information. The handle should be set to NULL if the server is notifying the client that the data item value has changed during a "warm link." A warm link is established by the client sending a WM_DDE_ADVISE message with the *fDeferUpd* bit set.
	aItem	High-order word of *lParam*. An atom that identifies the data item for which data or notification is sent.

Comments The global memory object identified by *hData* consists of a **DDEDATA** data structure that contains the following:

Word	Name	Content
1	**fAckReq**	If bit 15 is 1, the receiving (client) application is expected to send a WM_DDE_ACK message after the WM_DDE_DATA message has been processed. If bit 15 is zero, the client application should not send a WM_DDE_ACK message.
	reserved	Bit 14 is reserved.
	fRelease	If bit 13 is 1, the client application is expected to free the *hData* memory object after processing it. If bit 13 is zero, the client application should not free the object. See the "Posting" and "Receiving" sections for exceptions.
	fRequested	If bit 12 is 1, this data is offered in response to a WM_DDE_REQUEST message. If bit 12 is zero, this data is offered in response to a WM_DDE_ADVISE message.
	reserved	Bits 11–0 are reserved.
2	**cfFormat**	This specifies the format in which the data are sent or offered to the client application. It must be a standard or registered clipboard data format.
3–*n*	**Value[ ]**	This is the data. It is in the format specified by **cfFormat**.

Posting

Post the WM_DDE_DATA message by calling the **PostMessage** function, not **SendMessage**.

Allocate *hData* by calling the **GlobalAlloc** function with the GMEM_DDESHARE option.

Allocate *aItem* by calling the **GlobalAddAtom** function.

If the receiving (client) application responds with a negative WM_DDE_ACK message, the sending (server) application must delete the *hData* object.

If the sending (server) application sets the **fRelease** flag to zero, the sender is responsible for deleting *hData* upon receipt of either a positive or negative acknowledgement.

Do not set both the **fAckReq** and **fRelease** flags to zero. If both flags are set to zero, it is difficult for the sending (server) application to determine when to delete *hData*.

Receiving

If **fAckReq** is 1, post the WM_DDE_ACK message to respond positively or negatively. When posting WM_DDE_ACK, reuse the *aItem* atom or delete it and create a new one.

If **fAckReq** is zero, delete the *aItem* atom.

If the sending (server) application specified *hData* as NULL, the receiving (client) application can request the server to send the actual data by posting a WM_DDE_REQUEST message.

After processing the WM_DDE_DATA message in which *hData* is not NULL, delete *hData* unless either of the following conditions is true:

- The **fRelease** flag is zero.
- The **fRelease** flag is 1, but the receiving (client) application responds with a negative WM_DDE_ACK message.

WM_DDE_EXECUTE

This message, posted by a client application, sends a string to a server application to be processed as a series of commands. The server application is expected to post a WM_DDE_ACK message in response.

Parameter	Description
wParam	Identifies the sending window.
lParam	Specifies the commands to be executed.

Argument	Description
reserved	The low-order word of *lParam* is reserved.
hCommands	High-order word of *lParam*. A handle that identifies a global memory object containing the command(s) to be executed.

Comments

The command string is null-terminated. The command string should adhere to the syntax shown below. Optional syntax elements are enclosed in double brackets ([[ ]]); single brackets ([ ]) are a syntax element.

[*opcodestring*] [[[*opcodestring*]]] ...

The *opcodestring* uses the following syntax:

opcode[[(*parameter* [[,*parameter*]] ...)]]

The *opcode* is any application-defined single token. It may not include spaces, commas, parentheses, or quotation marks.

The *parameter* is any application-defined value. Multiple parameters are separated by commas, and the entire parameter list is enclosed in parentheses. The parameter may not include commas or parentheses except inside a quoted string. If a bracket or parenthesis character is to appear in a quoted string, it must be doubled: ((.

The following examples show valid command strings:

```
[connect][download(query1,results.txt)][disconnect]
[query("sales per employee for each district")]
[open("sample.xlm")][run("r1c1")]
```

Posting

Post the WM_DDE_EXECUTE message by calling the **PostMessage** function, not **SendMessage**.

Allocate *hCommands* by calling the **GlobalAlloc** function with the GMEM_DDE_SHARE option.

When processing WM_DDE_ACK sent in reply to WM_DDE_EXECUTE, the sender of the original WM_DDE_EXECUTE message must delete the *hCommands* object sent back in the WM_DDE_ACK message.

Receiving

Post the WM_DDE_ACK message to respond positively or negatively, reusing the *hCommands* object.

WM_DDE_INITIATE

This message, sent by either a client or server application, initiates a conversation with applications responding to the specified application and topic names.

Upon receiving this message, all applications with names that match the *aApplication* application and that support the *aTopic* topic are expected to acknowledge it (see the WM_DDE_ACK message).

Parameter	Description	
wParam	Identifies the sending window.	
lParam	Specifies the target application and the topic.	
	Argument	**Description**
	aApplication	Low-order word of *lParam*. An atom that specifies the name of the application with which a conversation is requested. The application name may not contain slashes or backslashes. These characters are reserved for future use in network implementations. If the application name is NULL, a conversation with all applications is requested.
	aTopic	High-order word of *lParam*. An atom that specifies the topic for which a conversation is requested. If the topic is NULL, a conversation for all available topics is requested.

Comments If the *aApplication* argument is NULL, any application may respond. If the *aTopic* argument is NULL, any topic is valid. Upon receiving a WM_DDE_INITIATE request with a null topic, an application is expected to send a WM_DDE_ACK message for each of the topics it supports.

Sending Send the WM_DDE_INITIATE message by calling the **SendMessage** function, not the **PostMessage** function. Broadcast the message to all windows by setting the first parameter of **SendMessage** to –1, as shown:

```
SendMessage(-1,WM_DDE_INITIATE,hwndClient,MAKELONG(aApp,aTopic));
```

If the application has already obtained the window handle of the desired server, it can send WM_DDE_INITIATE directly to the server window by passing the server's window handle as the first parameter of **SendMessage**.

Allocate *aApplication* and *aTopic* by calling **GlobalAddAtom**.

When **SendMessage** returns, delete the *aApplication* and *aTopic* atoms.

Receiving To complete the initiation of a conversation, respond with one or more WM_DDE_ACK messages, where each message is for a separate topic. When sending WM_DDE_ACK message, create new *aApplication* and *aTopic* atoms; do not reuse the atoms sent with the WM_DDE_INITIATE message.

WM_DDE_POKE

This message, posted by a client application, requests the receiving (server) application to accept an unsolicited data item value.

The receiving application is expected to reply with a positive WM_DDE_ACK message if it accepts the data, or with a negative WM_DDE_ACK message if it does not.

Parameter	Description
wParam	Identifies the sending window.
lParam	Identifies the data and specifies how it is sent.

Argument	Description
hData	Low-order word of *lParam*. A handle that specifies the global memory object containing the data and other information.
aItem	High-order word of *lParam*. An atom that identifies the data item offered to the server application.

Comments The global memory object identified by *hData* consists of a **DDEPOKE** data structure that contains the following:

Word	Name	Content
1	reserved	Bits 15–14 are reserved.
	fRelease	If bit 13 is 1, the receiving (server) application is expected to free the memory object after processing it. If bit 13 is zero, the receiving application should not free the object. See the following "Posting" and "Receiving" sections for exceptions.
	reserved	Bits 12–0 are reserved.
2	**cfFormat**	This specifies the client's preferred type of data. It must be a standard or registered clipboard data format.
3–*n*	**Value[ ]**	This is the data. It is in the format specified by **cfFormat**.

Posting Post the WM_DDE_POKE message by calling the **PostMessage** function, not **SendMessage**.

Allocate *hData* by calling the **GlobalAlloc** function with the GMEM_DDESHARE option.

Allocate *aItem* by calling the **GlobalAddAtom** function.

If the receiving (server) application responds with a negative WM_DDE_ACK message, the sending (client) application must delete the *hData* object.

If the sending (client) application sets the **fRelease** flag to zero, the sending application must delete *hData* upon receiving either a positive or negative WM_DDE_ACK message.

Receiving Post the WM_DDE_ACK message to respond positively or negatively. When posting WM_DDE_ACK, reuse the *aItem* atom or delete it and create a new one.

After processing the WM_DDE_POKE message, delete *hData* unless either of the following conditions is true:

- The **fRelease** flag is zero.
- The **fRelease** flag is 1, but the receiving (server) application responds with a negative WM_DDE_ACK message.

WM_DDE_REQUEST

This message, posted by a client application, requests the receiving (server) application to provide the value of a data item.

Parameter	Description
wParam	Identifies the sending window.
lParam	Specifies the requested data and the clipboard format number for the data

Argument	Description
cfFormat	Low-order word of *lParam*. A standard or registered clipboard format number.
aItem	High-order word of *lParam*. An atom that specifies which data item is being requested from the server.

Posting Post the WM_DDE_REQUEST message by calling the **PostMessage** function, not **SendMessage**.

Allocate **aItem** by calling the **GlobalAddAtom** function.

Receiving If the receiving (server) application can satisfy the request, it responds with a WM_DDE_DATA message containing the requested data. Otherwise, it responds with a negative WM_DDE_ACK message.

When responding with either a WM_DDE_DATA or WM_DDE_ACK message, reuse the *aItem* atom or delete it and create a new one.

WM_DDE_TERMINATE

This message, posted by either a client or server application, terminates a conversation.

Parameter	Description
wParam	Identifies the sending window.
lParam	Is reserved.

Posting Post the WM_DDE_TERMINATE message by calling the **PostMessage** function, not **SendMessage**.

While waiting for confirmation of the termination, the sending application should not acknowledge any other messages sent by the receiving application. If the sending application receives messages (other than WM_DDE_TERMINATE) from the receiving application, it should delete any atoms or shared memory objects accompanying the messages.

Receiving Respond by posting a WM_DDE_TERMINATE message.

WM_DDE_UNADVISE

This message, sent by a client application, informs a server application that the specified item, or a particular clipboard format for the item, should no longer be updated. This terminates the warm or hot link for the specified item.

Parameter	Description
wParam	Identifies the sending window.
lParam	Specifies the data-request item to be canceled.

Argument	Description
aItem	High-order word of *lParam*. An atom that specifies the data for which the update request is being retracted. When *aItem* is NULL, all active

WM_DDE_ADVISE conversations associated with the client are to be terminated.

cfFormat Low-order word of *lParam*. The clipboard format of the item that specifies the clipboard format for which the update request is being retracted. When *cfFormat* is NULL, all active WM_DDE_ADVISE conversations for the item are to be terminated.

Posting Post the WM_DDE_UNADVISE message by calling the **PostMessage** function, not **SendMessage**.

Allocate *aItem* by calling the **GlobalAddAtom** function.

Receiving Post the WM_DDE_ACK message to respond positively or negatively. When posting WM_DDE_ACK, reuse the *aItem* atom or delete it and create a new one.

APPENDIX A

Virtual-key codes

The following table shows the symbolic constant names, hexadecimal values, and descriptive information for Microsoft Windows virtual-key codes. The codes are listed in numeric order.

Name	Value	Description
VK_LBUTTON	01H	Left mouse button
VK_RBUTTON	02H	Right mouse button
VK_CANCEL	03H	Used for control-break processing
VK_MBUTTON	04H	Middle mouse button (3-button mouse)
	05H–07H	Undefined
VK_BACK	08H	BACKSPACE key
VK_TAB	09H	TAB key
	0AH–0BH	Undefined
VK_CLEAR	0CH	CLEAR key
VK_RETURN	0DH	RETURN key
VK_SHIFT	10H	SHIFT key
VK_CONTROL	11H	CONTROL key
VK_MENU	12H	MENU key
VK_PAUSE	13H	PAUSE key
VK_CAPITAL	14H	CAPITAL key
	15H–19H	Reserved for Kanji systems
	1AH	Undefined
VK_ESCAPE	1BH	ESCAPE key
	1CH–1FH	Reserved for Kanji systems
VK_SPACE	20H	SPACEBAR
VK_PRIOR	21H	PAGE UP key
VK_NEXT	22H	PAGE DOWN key
VK_END	23H	END key
VK_HOME	24H	HOME key
VK_LEFT	25H	LEFT ARROW key

VK_UP	26H	UP ARROW key
VK_RIGHT	27H	RIGHT ARROW key
VK_DOWN	28H	DOWN ARROW key
VK_SELECT	29H	SELECT key
	2AH	OEM specific
VK_EXECUTE	2BH	EXECUTE key
VK_SNAPSHOT	2CH	PRINTSCREEN key for Windows version 3.0 and later
VK_INSERT	2DH	INSERT key
VK_DELETE	2EH	DELETE key
VK_HELP	2FH	HELP key
VK_0	30H	0 key
VK_1	31H	1 key
VK_2	32H	2 key
VK_3	33H	3 key
VK_4	34H	4 key
VK_5	35H	5 key
VK_6	36H	6 key
VK_7	37H	7 key
VK_8	38H	8 key
VK_9	39H	9 key
	3AH–40H	Undefined
VK_A	41H	A key
VK_B	42H	B key
VK_C	43H	C key
VK_D	44H	D key
VK_E	45H	E key
VK_F	46H	F key
VK_G	47H	G key
VK_H	48H	H key
VK_I	49H	I key
VK_J	4AH	J key
VK_K	4BH	K key
VK_L	4CH	L key
VK_M	4DH	M key
VK_N	4EH	N key
VK_O	4FH	O key
VK_P	50H	P key
VK_Q	51H	Q key
VK_R	52H	R key
VK_S	53H	S key
VK_T	54H	T key
VK_U	55H	U key
VK_V	56H	V key
VK_W	57H	W key
VK_X	58H	X key
VK_Y	59H	Y key
VK_Z	5AH	Z key
	5BH–5FH	Undefined
VK_NUMPAD0	60H	Numeric key pad 0 key
VK_NUMPAD1	61H	Numeric key pad 1 key
VK_NUMPAD2	62H	Numeric key pad2 key

VK_NUMPAD3	63H	Numeric key pad 3 key
VK_NUMPAD4	64H	Numeric key pad 4 key
VK_NUMPAD5	65H	Numeric key pad 5 key
VK_NUMPAD6	66H	Numeric key pad 6 key
VK_NUMPAD7	67H	Numeric key pad 7 key
VK_NUMPAD8	68H	Numeric key pad 8 key
VK_NUMPAD9	69H	Numeric key pad 9 key
VK_MULTIPLY	6AH	Multiply key
VK_ADD	6BH	Add key
VK_SEPARATER	6CH	Separater key
VK_SUBTRACT	6DH	Subtract key
VK_DECIMAL	6EH	Decimal key
VK_DIVIDE	6FH	Divide key
VK_F1	70H	F1 key
VK_F2	71H	F2 key
VK_F3	72H	F3 key
VK_F4	73H	F4 key
VK_F5	74H	F5 key
VK_F6	75H	F6 key
VK_F7	76H	F7 key
VK_F8	77H	F8 key
VK_F9	78H	F9 key
VK_F10	79H	F10 key
VK_F11	7AH	F11 key
VK_F12	7BH	F12 key
VK_F13	7CH	F13 key
VK_F14	7DH	F14 key
VK_F15	7EH	F15 key
VK_F16	7FH	F16 key
	80H–87H	OEM specific
	88H–8FH	Unassigned
VK_NUMLOCK	90H	NUM LOCK key
	91H	OEM specific
	92H–B9H	Unassigned
	BAH–C0H	OEM specific
	C1H–DAH	Unassigned
	DBH–E4H	OEM specific
	E5H	Unassigned
	E6H	OEM specific
	E7H–E8H	Unassigned
	E9H–F5H	OEM specific
	F6H–FEH	Unassigned

APPENDIX

B

RC diagnostic messages

This appendix contains descriptions of diagnostic messages produced by the Resource Compiler (**RC**). Many of these messages appear when the Resource Compiler is not able to compile your resources. The descriptions in this appendix can help you determine the problem.

See Chapter 8, "Resource script statements," for information on the keywords and fields specified in this appendix.

A (*V*) symbol at the beginning of a message description indicates that the message is displayed only if **RC** is run with the **–V** (verbose) option. These messages are generally informational and do not necessarily indicate errors.

The messages are listed in alphabetical order.

Accelerator Type required (ASCII or VIRTKEY)
The *type* field in the **ACCELERATORS** statement must contain either the **ASCII** or **VIRTKEY** value.

BEGIN expected in Accelerator Table
The **BEGIN** keyword must immediately follow the **ACCELERATORS** keyword.

BEGIN expected in Dialog
The **BEGIN** keyword must immediately follow the **DIALOG** keyword.

BEGIN expected in menu
The **BEGIN** keyword must immediately follow the **MENU** keyword.

BEGIN expected in RCData
The **BEGIN** keyword must immediately follow the **RCDATA** keyword.

BEGIN keyword expected in String or Error Table

The **BEGIN** keyword must immediately follow the **STRINGTABLE** or **ERRTABLE** keyword.

Cannot Reuse String Constants

You are using the same value twice in a **STRINGTABLE** or **ERRTABLE** statement. Make sure you are not mixing overlapping decimal and hexadecimal values.

Control Character out of range [^A – ^Z]

A control character in the **ACCELERATORS** statement is invalid. The character following the caret (^) must be between A and Z, inclusive.

copy of *temp-file-2* to *exe-file* failed

The temporary file was not able to create the new .EXE file. Make sure that the TEMP environment variable is pointing to a drive that is not write-protected.

Copying segment *id* (*size* bytes)

(V) **RC** is copying the specified segment to the .EXE file.

Could not find RCPP.EXE

RCPP.ERR must be in the current directory or a directory in the PATH environment.

Could not open *in-file-name*

RC could not open the specified file. Make sure the file exists and that you typed the filename correctly.

Couldn't open *resource-name*

RC could not open the specified file. Make sure the file exists and that you typed the filename correctly.

Couldn't write executable

The .EXE file could not be copied to the temporary file. Make sure that the TEMP environment variable is pointing to a drive that is not write-protected and that the .EXE file from the linker is correct. You can check the .EXE file with the EXEHDR program.

Creating *resource-name*

(V) **RC** is creating a new .RES file.

Empty menus not allowed

An **END** keyword appears before any menu items are defined in the **MENU** statement. Empty menus are not permitted by the Resource Compiler. Make sure you do not have any open quotation marks within the **MENU** statement.

END expected in Dialog

The **END** keyword must occur at the end of a **DIALOG** statement. Make sure there are no open quotes left from the preceding statement.

END expected in menu

The **END** keyword must come at the end of a **MENU** statement. Make sure you do not have any open quotation marks or a mismatched pair of **BEGIN** and **END** statements.

Error: Bitmap file *resource-file* is not in 3.00 format.

Use SDKPaint to convert version 2.x resource files to the 3.0 format.

Error Creating *resource-name*

Could not create specified .RES file. Make sure it is not being created on a read-only drive. Use the **–V** option to find out whether the file is being created.

Error: I/O error reading file.

Read failed. Since this is a generic routine, no specific filename is supplied.

Error: I/O error seeking in file

Seeking in file failed.

Error: I/O error writing file.

Write failed. Since this is a generic routine, no specific filename is supplied.

Error: Old DIB in *resource-name.* Pass it through SDKPAINT.

The resource file specified is not compatible with Windows 3.0. Make sure you have read and saved this file using the latest version of SDKPaint.

Error: Out of memory. Try not using resources with string identifiers.

There is not enough memory to allocate for a table of string names. You can view these names are when you use the **–V** option. Try to replace the string names with numbers. For example, you can change

```
MYICON  ICON myicon.ico
```

to

```
1       ICON myicon.ico
```

or provide the following statement in your header file:

```
#define MYICON 1
```

Error: Resource file *resouce-name* is not in 3.00 format.

Make sure your icons and cursors have been read and saved using the latest version of SDKPaint.

Errors in .EXE file

LINK failed. See the *CodeView and Utilities* manual in the Microsoft C 5.1 Optimizing Compiler documentation set for more information.

.EXE file too large; relink with higher /ALIGN value

The EXE file is too large. Relink the .EXE file with a larger **/ALIGN** value. If the .EXE file is larger than 800K, you should use the **/ALIGN:32** value on your **LINK** line.

.EXE not created by LINK

You must create the .EXE file with a version of **LINK** that is from C version 5.1 or later.

Expected Comma in Accelerator Table

RC requires a comma between the *event* and *idvalue* fields in the **ACCELERATORS** statement.

Expected control class name

The *class* field of a **CONTROL** statement in the **DIALOG** statement must be one of the following types: BUTTON, COMBOBOX, EDIT, LISTBOX, SCROLLBAR, STATIC, or user-defined. Make sure the class is spelled correctly.

Expected font face name

The *typeface* field of the **FONT** option in the **DIALOG** statement must be an ASCII character string enclosed in double quotation marks. This field specifies the name of a font.

Expected ID value for Menuitem

The **MENU** statement must contain a *menuID* field, which specifies the name or number that identifies the menu resource.

Expected Menu String

Each **MENUITEM** and **POPUP** statement must contain a *text* field, which is a string enclosed in double quotation marks that specifies the name of the menu item or pop-up menu. A **MENUITEM SEPARATOR** statement requires no quoted string.

Expected numeric command value

RC was expecting a numeric *idvalue* field in the **ACCELERATORS** statement. Make sure you have used a **#define** constant to specify the value and that the constant is spelled correctly.

Expected numeric constant in string table

A numeric constant, defined in a #**define** statement, must immediately follow the **BEGIN** keyword in a **STRINGTABLE** or **ERRTABLE** statement.

Expected numeric point size

The *pointsize* field of the **FONT** option in the **DIALOG** statement must be an integer point size value.

Expected Numerical Dialog constant

A **DIALOG** statement requires integer values for the *x*, *y*, *width*, and *height* fields. Make sure these values are included after the **DIALOG** keyword and that they are not negative.

Expected String in STRINGTABLE/ERRTABLE

A string is expected after each *stringid* value in a **STRINGTABLE** or **ERRTABLE** statement.

Expected String or Constant Accelerator command

RC was not able to determine what kind of key is being set up for the accelerator. The *event* field in the **ACCELERATORS** statement might be invalid.

Expecting number for ID

Expecting a number for the *id* field of a control statement in the **DIALOG** statement. Make sure you have a number or #**define** statement for the control ID.

Expecting quoted string in dialog class

The *class* field of the **CLASS** option in the **DIALOG** statement must be an integer or a string, enclosed in double quotation marks.

Expecting quoted string in dialog title

The *captiontext* field of the **CAPTION** option in the **DIALOG** statement must be an ASCII character string enclosed in double quotation marks.

File not found: *filename*

The file specified in the **RC** command line was not found. Check to see whether the file has been moved to another directory and whether the filename or pathname is typed correctly.

Font names must be ordinals

The *pointsize* field in the **FONT** statement must be an integer, not a string.

Gangload area is [size] bytes at offset 0x[address]

(V) This is the size (in bytes) of all the segments that have one of the following attributes:

- **PRELOAD**
- **DISCARDABLE**
- Code segments that contain the entry point, WinMain
- Data segments (which should not be discardable)

The segments are placed in a continguous area in the .EXE file for fast loading. The offset value is from the the beginning of the file. To disable gangloading, use the **–k** option.

Insufficient memory to spawn RCPP.EXE

There wasn't enough memory to run the preprocessor (RCPP). You can try not running any memory-resident software that might be taking up too much memory. Use the CHKDSK program to verify the amount of memory you have.

Invalid Accelerator

An *event* field in the **ACCELERATORS** statement was not recognized or was more than two characters in length.

Invalid Accelerator Type (ASCII or VIRTKEY)

The *type* field in the **ACCELERATORS** statement must contain either the **ASCII** or **VIRTKEY** value.

Invalid control character

A control character in the **ACCELERATORS** statement is invalid. A valid control character consists of one letter (only) following a caret (^).

Invalid Control type

Each control statement in a **DIALOG** statement must be one of the following: **CHECKBOX, COMBOBOX, CONTROL, CTEXT, DEFPUSHBUTTON, EDITTEXT, GROUPBOX, ICON, LISTBOX, LTEXT, PUSHBUTTON, RADIOBUTTON, RTEXT, SCROLLBAR**.

Make sure these control statements are spelled correctly.

Invalid .EXE file

The .EXE file is invalid. Make sure that the linker created it correctly and that the file exists. You can check the .EXE file with the EXEHDR program.

Invalid switch, *option*

You used an option that was not valid. Use **RC –?** for a list of the command-line options.

Invalid type

The resource type was not among the types defined in the windows.h file.

Invalid usage. Use rc –? for Help

Make sure you have at least one filename to work with. Use **RC –?** for a list of the command-line options.

No executable filename specified.

The **–FE** option was used, but no .EXE filename specified.

No resource binary filename specified.

The **–FO** option was used, but no .RES filename specified.

Not a Microsoft Windows format .EXE file

Make sure that the linker created the .EXE file correctly and that the file exists. You can check the .EXE file with the EXEHDR program.

Out of far heap memory

There wasn't enough memory. Try not running any memory-resident software that might be taking up too much space. Use the CHKDSK program to find out how much memory you have.

Out of memory, needed *n* bytes

RC was not able to allocate the specified amount of memory.

RC: Invalid swap area size: –S *string*

Invalid swap area size. Check your syntax for the **–S** option on the **RC** command line. The following are acceptable command lines:

```
RC S123
RC S123K  ;where K is kilobytes
RC S123p  ;where p is paragraphs
```

RC: Invalid switch: *option*

You used an option that was not valid. Use **RC –?** for a list of the command-line options.

RC: RCPP *preprocessor-command-string*

(V) **RC** is passing the specified string to the preprocessor.

RC: RCPP.ERR not found

RCPP.ERR must be in the current directory or a directory in the PATH environment.

RC terminated by user

A CONTROL+C key combination was pressed, terminating **RC**.

RC terminating after preprocessor errors

See the Microsoft C 5.1 Optimizing Compiler documentation for information about preprocessor errors.

RCPP.EXE command line greater than 128 bytes

The command line was too long.

RCPP.EXE is not a valid executable

RCPP.EXE is not valid. The file might have been altered. Try copying the file from the SDK disks.

Reading *resource-name*

(V) **RC** is reading the .RES file.

Resources will be aligned on *number* byte boundaries

(V) The alignment is determined by the **ALIGN:*number* option on the LINK line.**

Sorting preload segments and resources into gangload section

(V) **RC** is sorting the preloaded segments so that they can be loaded quickly.

Text string or ordinal expected in Control

The *text* field of a **CONTROL** statement in the **DIALOG** statement must be either a text string or an ordinal reference to the type of control is expected. If using an ordinal, make sure that you have a #**define** statement for the control.

The EXETYPE of this program is not Windows

The **EXETYPE WINDOWS** statement did not appear in the .DEF file. Since the linker might make optimizations for OS/2 (the default **EXETYPE**) that are not appropriate for Windows, the **EXETYPE WINDOWS** statement must be specified.

Unable to create *destination*

RC was not able to create the destination file. Make sure there is enough disk space.

Unable to open *exe-file*

RC could not open this .EXE file. Make sure that the linker created it correctly and that the file exists.

Unbalanced Parentheses

Make sure you have closed every open parenthesis in the **DIALOG** statement.

Unexpected value in RCData

The *raw-data* values in the **RCDATA** statement must be integers or strings, each separated by a comma. Make sure you did not leave out a comma or leave out a quotation mark around a string.

Unknown DIB header format

The bitmap header is not a **BITMAPCOREHEADER** or **BITMAPINFOHEADER** structure.

Unknown error spawning RCPP.EXE

For an unknown reason, RCPP was not started. Try copying the file from the SDK disks, and use the CHKDSK program to verify the amount of available memory.

Unknown Menu SubType

The *item-definition* field of the **MENU** statement can contain only **MENUITEM** and **POPUP** statements.

Warning: ASCII character not equivalent to virtual key code

There is an invalid virtual-key code in the **ACCELERATORS** statement. The ASCII value for some characters (such as *, ^, &,) is not equivalent to the virtual-key code for the corresponding key. (In the case of the asterisk (*), the virtual-key code is equivalent to the ASCII value for 8, the numeric character on the same key. Therefore the statement

```
VIRTKEY '* '
```

is invalid.) See Appendix A, "Virtual-key codes," and Appendix D, "Character tables," for these values.

Warning: Discardable segment *id* (*hex-size* bytes) is excessively large.

The segment is greater than 27FFh in size. **RC** displays this warning because very large segments can adversely affect memory usage. Check your map file listing for the exact size of your segments.

Warning: SHIFT or CONTROL used without VIRTKEY

The **ALT**, **SHIFT**, and **CONTROL** options apply only to virtual keys in the **ACCELERATORS** statement. Make sure you have used the **VIRTKEY** option with one of these other options.

Writing resource *resource-name or ordinal-id resource type* (*resource size*)

(V) **RC** is writing the resource name or ordinal ID, followed by a period and the resource type and size (in bytes).

Warning: *string* segment *number* set to PRELOAD

RC displays this warning when it copies a segment that must be preloaded but that is not marked **PRELOAD** in the linker .DEF file.

All nondiscardable segments should be preloaded, including automatic data segments, fixed segments and the entry point of the code (WinMain). The attributes of your code segments are set by the .DEF file. Check your map file listing for more information.

INDEX

WINDOWS API

VOLUME II

BORLAND

CORPORATE HEADQUARTERS: 1800 GREEN HILLS ROAD, P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001, (408) 438-5300. OFFICES IN: AUSTRALIA,
DENMARK, FRANCE, GERMANY, ITALY, JAPAN, NEW ZEALAND, SINGAPORE, SWEDEN AND THE UNITED KINGDOM • PART #14MN-API02-10 • BOR 3188