DIBs and Their Use

Ron Gery
Microsoft Developer Network Technology Group

Created: March 20, 1992

Abstract

This article discusses the DIB (device-independent bitmap) concept from definition and structure to the API that uses it. Included is a small sample application that illustrates some of the most common methods of using DIBs to display and manipulate digital images. Functions discussed are **GetDIBits**, **SetDIBits**, **CreateDIBitmap**, **SetDIBitsToDevice**, **StretchDIBits**, and **CreateDIBPatternBrush**. This article does not discuss using palettes with DIBs.

Overview

A DIB (device-independent bitmap) is a format used to define device-independent bitmaps in various color resolutions. The main purpose of DIBs is to allow bitmaps to be moved from one device to another (hence, the device-independent part of the name). A DIB is an *external* format, in contrast to a device-dependent bitmap, which appears in the system as a bitmap object (created by an application using **CreateBitmap**, **CreateCompatibleBitmap**, **CreateBitmapIndirect**, or **CreateDIBitmap**). A DIB is normally transported in metafiles (usually using the **StretchDIBits** function), BMP files, and the Clipboard (CF_DIB data format).

A DIB consists of two parts: the bits themselves and a header that describes the format of the bits. The header contains the color format, a color table, and the size of the bitmap. The current DIB format supports four color resolutions: 1 bit, 4 bit, 8 bit, and 24 bit. In 1-bit, 4-bit, and 8-bit DIBs, the pixels are defined by indexes (of the appropriate bit resolution) into the color table; 24-bit pixels are described as 24-bit values, 1 byte each for red, green, and blue.

The DIB functions are:

GetDIBits	Translates a device-dependent bitmap into the DIB format
SetDIBits	Translates a DIB's information into device-dependent form
CreateDIBitmap	Creates a device-dependent bitmap initialized with DIB information
SetDIBitsToDevice	Sets a DIB directly to the output surface
StretchDIBits	Moves a rectangle from the DIB to a rectangle on the destination surface, stretching or compressing as necessary
CreateDIBPatternBrush	Creates a pattern brush using a DIB for the bitmap description

Device Independence--What's It Good For?

Transferring color bitmaps from one device to another was not possible in versions of the Microsoft® Windows™ graphical environment earlier than 3.0. With DIBs, each device displays the image to the ability of its color resolution. An application can store an image in the DIB format and then display it, regardless of the output device; an application need no longer create a version of each image for each type of device.

This image transfer ability can be used to print halftone images. For example, the **StretchDIBits** function can pass a DIB directly to an intelligent printer driver. Given the full color information of the image instead of simply a monochrome version (the traditional method), the driver can use halftones to print a realistic picture.

Because the DIB format is publicly defined, an application can manipulate it on the fly. In fact, an application can build an image without any interaction with Windows. If Windows lacks a drawing primitive, the application can simulate it directly into the DIB instead of using the existing graphics device interface (GDI) primitives. Unfortunately, under Windows versions 3.0 and 3.1, GDI cannot perform output operations directly to a DIB.

BMP File Formats

The file extension of a Windows DIB file is BMP. The file consists of a **BITMAPFILEHEADER** structure followed by the DIB itself. Unfortunately, because the **BITMAPFILEHEADER** structure is never actually passed to the API, not every application that generates BMP files fills out the data structure carefully. To add to this confusion, the "proper" definition of the structure is at odds with the documentation. Properly, the data structure contains the following fields:

bfType	A **WORD** that defines the type of file. It must be 'BM'.
bfSize	A **DWORD** that specifies the size of the file in bytes. The Microsoft Windows Software Development Kit (SDK) documentation claims otherwise. To be on the safe side, many applications calculate their own sizes for reading in a file.
bfReserved1, bfReserved2	**WORD**s that must be set to 0.
bfOffBits	A **DWORD** that specifies the offset from the beginning of the **BITMAPFILEHEADER** structure to the start of the actual bits. The DIB header immediately follows the file header, but the actual image bits need not be placed next to the headers in the file.

The DIB header immediately follows the **BITMAPFILEHEADER** structure.

For a code sample that reads a BMP file, see the sample program.

The DIB Header

The header actually consists of two adjoining parts: the header proper and the color table. Both are combined in the **BITMAPINFO** structure, which is what all DIB APIs expect.

Windows supports two varieties of headers: **BITMAPINFOHEADER** and **BITMAPCOREHEADER**. If at all possible, applications should use only **BITMAPINFOHEADER** s. The **BITMAPCOREHEADER** definition is based on the bitmap definition from Presentation Manager™ version 1.1 and is supported for compatibility.

During a DIB setting operation, most fields are already filled in by whoever generated the DIB. Doing a **GetDIBits** call, however, provides more control. The way the header is filled in for this operation defines the resulting DIB, particularly its color resolution.

BITMAPINFOHEADER contains the following fields:

biSize	Should be set to sizeof(**BITMAPINFOHEADER**). This field defines the size of the header (minus the color table). If a new DIB definition is added, it is identified by a new value for the size. This field is also convenient for calculating a pointer to the color table, which immediately follows the **BITMAPINFOHEADER**.
biWidth, **biHeight**	Define the width and the height of the bitmap in pixels. They are **DWORD** values for future expansion, and the code in Windows versions 3.0 and 3.1 ignores the high word (which should be set to 0).
biPlanes	Should always be 1. All DIB definitions rely on **biBitCount** for defining the color

resolution.

biBitCount

Defines the color resolution (in bits per pixel) of the DIB. Only four values are valid for this field: 1, 4, 8, and 24. New resolutions (16 bit, for example) may be added in the future, but for now only these four define a valid DIB. Choosing the appropriate value when doing a **GetDIBits** is discussed below. When performing a **Set** operation, the value should already be defined for the bits.

biCompression

Specifies the type of compression. Can be one of three values: BI_RGB, BI_RLE4, or BI_RLE8. The most common and useful choice, BI_RGB, defines a DIB in which all is as it seems. Each block of **biBitCount** bits defines an index (or RGB value for 24-bit versions) into the color table. The other two options specify that the DIB is stored (or will be stored) using either the 4-bit or the 8-bit run length encoding (RLE) scheme that Windows supports. The RLE formats are especially useful for animation applications and also usually compress the bitmap. BI_RGB format is recommended for almost all purposes. RLE versions, although possibly smaller, are slower to decode, not as widely supported, and extremely painful to band properly.

biSizeImage

Contains the size of the bitmap proper in bytes or the value 0. A value of 0 indicates that the DIB is of default size. Calculating the size of a bitmap is not hard:

```
biSizeImage = ((((biWidth * biBitCount) + 31) &
                ~31) >> 3) * biHeight:
```

The crazy roundoffs and shifts account for the bitmap being **DWORD**-aligned at the end of every scanline. When nonzero, this field tells an application how much storage space the DIB's bits need. The **biSizeImage** field really becomes useful when dealing with an RLE bitmap, the size of which depends on how well the bitmap was encoded. If an RLE bitmap is to be passed around, the **biSizeImage** field is mandatory.

biXPelsPerMeter, **biYPelsPerMeter**

Define application-specified values for the desirable dimensions of the bitmap. This information can be used to maintain the physical dimensions of an image across devices of different resolutions. GDI never touches these fields. When not filled in, they should both be set to 0.

biClrUsed

Provides a way for getting smaller color tables. When this field is set to 0, the number of colors in the color table is based on the **biBitCount** field (1 indicates 2 colors, 4 indicates 16, 8 indicates 256, and 24 indicates no color table). A nonzero value specifies the exact number of colors in the table. So, for example, if an 8-bit DIB uses only 17 colors, then only those 17 colors need to be defined in the table, and **biClrUsed** is set to 17. Of course, no pixel can have an index pointing past the end of the table.

Note:

This field cannot be used during a **GetDIBits** operation. GDI always fills a full-size color table. The field is therefore more useful for post-processing operations, when an application trims down the contents of the DIB. If nonzero for a 24-bit DIB, it indicates the existence of a color table that the application can use for color reference.

biClrImportant

Specifies that the first *x* colors of the color table are important to the DIB. If the rest of the colors are not available, the image still retains its meaning in an acceptable manner. **biClrImportant** is purely for application use; GDI does not touch this value. When this field is set to 0, all the colors are important, or, rather, their relative importance has not been computed.

The color table immediately follows the header information. No color table is defined for 24-bit DIBs. The table consists of an array of **RGBQUAD** data structures. (The table for the **BITMAPCOREINFO** format is built with the **RGBTRIPLE** data structure.) Red, green, and blue bytes are *in reverse order* (red swaps position with blue) from the Windows convention. This is another leftover from Presentation Manager compatibility.

The size of the color table depends on the **biBitCount** value (and can be overwritten using the **biClrUsed** field; see above):

```
if (!(nNumColors = biClrUsed))
{
   if (biBitCount != 24)
      nNumColors = 1 << biBitCount;
}
nTableSize = nNumColors * sizeof(RGBQUAD);
```

Most DIBs floating around currently have **biClrUsed** set to 0, but if any full-fledged DIB bashing is planned, it is a good idea to set it properly. If **biClrUsed** is nonzero, a color table with 24-bit DIBs is possible. GDI does not use this color table, but the application can use it to determine the important colors used in the DIB.

All DIB functions include a *wUsage* parameter, which can affect the definition of the color table. This article avoids using palettes with DIBs and thereby assumes that *wUsage* is always set to DIB_RGB_COLORS and that the color table is therefore always composed of RGB values. When DIB_PAL_COLORS is used, the color table consists of **WORD** values that are indexes into the currently selected logical palette. (This topic is discussed in detail in the "Using DIBs with Palettes" article.)

Bit Formats

The header defines the format of the bits, but all formats share the following rules:

- Every scanline is **DWORD**-aligned. The scanline is buffered to alignment; the buffering is not necessarily 0.
- The scanlines are stored upside down, with the first scan (scan 0) in memory being the bottommost scan in the image. (See Figure 1.) This is another artifact of Presentation Manager compatibility. GDI automatically inverts the image during the **Set** and **Get** operations.

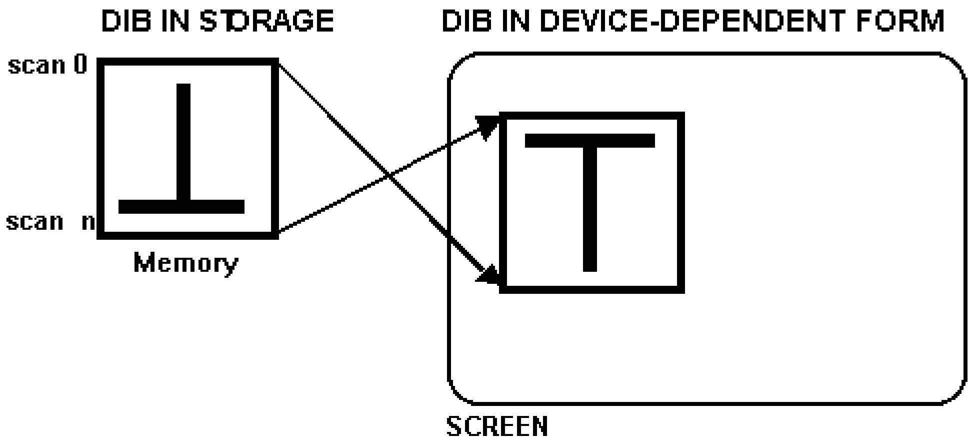

Figure 1.

- 64K segment boundaries are not respected; scanlines can cross such boundaries (unlike the device-dependent bitmap format that is buffered to 64K boundaries).

Each format has the following specifics:

- 1-bit DIBs are stored using each bit as an index into the color table. The most significant bit is the leftmost pixel.
- 4-bit DIBs are stored with each 4 bits representing an index into the color table. The most

significant nibble is the leftmost pixel.

- 8-bit DIBs are the easiest to store because each byte is an index.
- 24-bit DIBs have every 3 bytes representing a color, using the same ordering as the color table. This format is especially tricky during processing because a 64K boundary can exist in the middle of a color triple—an ugly condition that must be handled with care.

Using the DIB API

GetDeviceCaps (*hDC*, RASTERCAPS) returns a **WORD** value with flags set indicating which DIB functions the driver supports. RC_DI_BITMAP indicates support of **GetDIBits** and **SetDIBits** , RC_DIBTODEV indicates support of **SetDIBitsToDevice**, and RC_STRETCHDIB indicates support of **StretchDIBits**. Any function not supported can be simulated, although the simulations are often not as useful as the real thing (mainly because color information is lost). A device may be unable to support the full functionality even if a bit is set. For example, a device could support **StretchDIBits** but only for integral stretches. Unfortunately, an application has no way to determine the completeness of the implementation. In these cases, GDI simulates the function.

GetDIBits and SetDIBits

These two functions are used to convert device-independent bitmaps into device-dependent bitmaps and vice versa. **SetDIBits** converts a DIB to a device-dependent bitmap, and **GetDIBits** generates a DIB from a device-dependent bitmap.

The device driver referenced by the *hDC* passed into both calls performs the actual translation. Some device drivers may not have this functionality (for example, a Windows version 2.0 driver or a primitive Windows version 3.0 driver). In this case, GDI simulates the translation, but only in *monochrome*—color information is converted to black and white. For the most part, though, this is not a concern. All self-respecting display drivers support this functionality, and only a few printer drivers do not provide the translation, usually monochrome drivers for which the GDI simulations suffice.

The parameters are the same for both **GetDIBits** and **SetDIBits**:

GetDIBits(*hDC*, *hBitmap*, *nStartScan*, *nNumScans*, *lpBits*, *lpBitmapInfo*, *wUsage*)

SetDIBits(*hDC*, *hBitmap*, *nStartScan*, *nNumScans*, *lpBits*, *lpBitmapInfo*, *wUsage*)

where:

hDC	The device context (DC) responsible for the translation operation. *hDC* must be compatible with the *hBitmap* parameter.
hBitmap	The device-dependent bitmap from which (**Get**) or to which (**Set**) the DIB will be translated. Because of how the simulation code operates, this bitmap should not be currently selected into any DC.
nStartScan, nNumScans	Define the contents of *lpBits*. For example, a *StartScan* of 5 indicates that *lpBits* points to the fifth scan of the DIB. A *NumScans* of 14 indicates that *lpBits* points to 14 scans of the DIB. Normally, *nStartScan* is set to 0 and *nNumScans* is set to **biHeight** to denote that the whole DIB is pointed to by *lpBits*.
lpBits	The actual bitmap of the DIB. The pixel information is pointed to by this parameter.
lpBitmapInfo	The header (with color table) defining the DIB. The height and width in this header *must* match the height and width of the *hBitmap* parameter (the translation is always one-to-one). The color resolution of the DIB need not match that of *hBitmap*.
wUsage	For the purposes of this article, assume this to be DIB_RGB_COLORS, indicating RGB colors in the color table.

Using **SetDIBits** is reasonably straightforward. A DIB is taken from somewhere (for example, from the Clipboard or from a disk file) and is converted to a bitmap object, which can then be selected into a DC and blted to the screen for display. This is the simplest way to display a DIB.

Note:

For many printers that can do halftones, this method is not preferred; **StretchDIBits** (discussed below) is far more useful.

The following is a simple display of a DIB to a DC (with no error handling):

```
HBITMAP hBitmap;
HDC hMemDC;

hBitmap = CreateCompatibleBitmap(hDC, (WORD)lpInfo->biWidth,
          lpInfo->(WORD)biHeight);
hMemDC = CreateCompatibleDC(hDC);
SetDIBits(hDC, hBitmap, 0, (WORD)lpInfo->biHeight, lpBits,
         lpInfo, DIB_RGB_COLORS);
hBitmap = SelectObject(hMemDC, hBitmap);
BitBlt(hDC, 0, 0, (WORD)lpInfo->biWidth, (WORD)lpInfo->biHeight,
      hMemDC, 0, 0, SRCCOPY);
DeleteObject(SelectObject(hMemDC, hBitmap));
DeleteDC(hMemDC);
```

Using **GetDIBits** is more complex because the application can choose *what kind* of DIB to generate. The size of the source bitmap regulates the DIB's dimensions (a piece can be extracted by blting into a smaller bitmap), but the application's need can dictate the color resolution.

For **GetDIBits** to work properly, the application needs to set the following fields in the header:

biSize = sizeof(**BITMAPINFOHEADER**)

biWidth = {width of the bitmap}

biHeight = {height of the bitmap}

biPlanes = 1

biBitCount = {desired color resolution (1, 4, 8, or 24)}

biCompression = BI_RGB (For RLE information, see below.)

Also, the space allocated for the color table must be sufficient to hold a full-size table:

```
if (biBitCount != 24)
   nSizeTable = (1 << biBitCount) * sizeof(RGBQUAD)
else
   nSizeTable = 0;
```

The space allocated for *lpBits* also needs to be large enough to hold *nNumScans* of data.

The call fills in the following fields of the structure:

- **biSizeImage** = size in bytes of the DIB data
- color table (for non–24-bit case) is filled with appropriate colors
- *lpBits* is filled with the DIB data

If **GetDIBits** is called with *lpBits* set to NULL, no bits are returned; only **biSizeImage** and the color table are filled in. This option is useful for DIBs with RLE and is not worthwhile for non-encoded DIBs.

The application's goals for the DIB determine what color resolution to choose. The usual approach is to generate a DIB that preserves the color information of the source device-dependent bitmap. Choosing a lesser resolution results in a loss of color information, which is usually undesirable. Always using 24-bit resolution is unnecessary, however, because doing so adds no more color resolution if the source has 8-bit or less resolution.

```
BITMAP bm;
GetObject(hBitmap, sizeof(BITMAP), (LPVOID)&bm);  // get information
                                                  //  on bitmap
BitmapRes = bm.bmPlanes * bm.bmBitsPixel;
if (BitmapRes == 1)
   biBitCount = 1;
else if (BitmapRes <= 4)
   biBitCount = 4;
else if (BitmapRes <= 8)
   biBitCount = 8;
else
   biBitCount = 24;
```

The bitmap's resolution calculation must take into account that some device-dependent bitmaps are planar (notably EGA and VGA). DIBs, on the other hand, are always "packed pixel," with only one plane per pixel (**biPlanes** = 1).

The *nStartScan* and *nNumScans* parameters (a residue of Presentation Manager compatibility) are designed to be used for banding. If not enough memory is available to load the entire DIB into memory in one piece, *lpBits* can be made to point to only a portion of the bits. Consider the following example:

```
#define MAXREAD 5
WORD ReadXScans(LPSTR, WORD);      // read up to X scans; return
                                   //  NumRead
LPSTR lpBits;      // points to a block of memory for MAXREAD scans
LPBITMAPINFOHEADER lpInfo;
WORD nStart, nNumRead;

for (nStart = 0; nStart >= (WORD)lpInfo->biHeight; )
{
   nNumRead = ReadXScans(lpBits, MAXREAD);
   SetDIBits(hDC, hBitmap, nStart, nNumRead,
      lpBits,lpInfo,DIB_RGB_COLORS);
   nStart += nNumRead;
}
```

The **Set** code takes the given band, translates it, and puts the translated band in its proper location, accounting at all times for the upside-down nature of DIBs. Notice how **biHeight** does not change at any time because the band is placed in the bitmap based on the height of the full bitmap. *nStart* is based on the height of the full image (defined by **biHeight**).

CreateDIBitmap

The following code demonstrates calling **CreateDIBitmap** with the usual case:

```
hBitmap = CreateDIBitmap(hDC, lpInfo, CBM_INIT, lpBits, lpInfo,
          wUsage);
```

This is equivalent to:

```
   hBitmap = CreateCompatibleBitmap(hDC, (WORD)lpInfo->biWidth,
             (WORD)lpInfo->biHeight);
   SetDIBits(hDC, hBitmap, 0, (WORD)lpInfo->biHeight, lpBits, lpInfo,
             wUsage);
```

GDI's implementation skips the **SetDIBits** part if the third parameter is not set with the CBM_INIT flag. This function makes for nice shortcut coding of the conversion from DIB to device-dependent bitmap.

SetDIBitsToDevice

SetDIBitsToDevice allows an application to set a DIB directly to a device surface. Because this function is a holdout from early development, its interface is not as polished as it could be. **StretchDIBits** is a far more powerful function than **SetDIBitsToDevice**. **StretchDIBits** does all that **SetDIBitsToDevice** does and has a nicer interface. **SetDIBitsToDevice** is limited in the way it handles metafiles because it does not scale, and banding with the *nStartScan* and *nNumScans* parameters is nontrivial at best. **StretchDIBits** does not allow the banding.

The following code performs the **SetDIBitsToDevice** functionality on the full bitmap (no banding) using **StretchDIBits**:

```
   StretchDIBits(hDC, x, y, (WORD)lpInfo->biWidth,
        (WORD)lpInfo->biHeight, 0, 0, (WORD)lpInfo->biWidth,
        (WORD)lpInfo->biHeight, lpBits, lpInfo, DIB_RGB_COLORS,
        SRCCOPY)
```

Assuming that *nStartScan* is set to 0 and that *nNumScans* is set to *lpInfo*->**biHeight** (that is, no banding), the function is basically a **BitBlt** with SRCCOPY as the ROP and with a DIB as the source. *SrcX* and *SrcY* are in the DIB's space and are therefore upside down in relation to the DC (Y = 0 is at the bottom of the image).

Dealing with the upside-down DIB is tricky when doing a partial setting. For example, if an application wants to get the bottom third of a DIB that is *w* by *h* pixels to the device at (*x*,*y*), the call would look something like the following:

```
   SetDIBitsToDevice(hDC, x, y, w, h/3, 0, h/3, 0,
         (WORD)lpInfo->biHeight, lpBits, lpInfo, DIB_RGB_COLORS);
```

A device-dependent bitmap would have a *SrcY* of 2*h*/3 for the bottom third, but with the upside-down system of the DIB, a *SrcY* of *h*/3 points to the proper place relative to Windows coordinates.

StretchDIBits

This function is the do-all darling for displaying a DIB on the surface of a device. It is especially nice for metafiling and for printing, for which the ability to stretch is important.

The one critical hole in the current implementation of **StretchDIBits** is that **StretchDIBits** is supported by printer drivers and not by many display drivers. Therefore, using this function repeatedly to stretch a DIB to the screen is significantly slower than using **SetDIBits** (to get a device-dependent bitmap) followed by repeated **StretchBlt** calls.

The implementation of this function in GDI is very straightforward. If the device driver can handle the call itself, it does. If not, and the call is one-to-one and the device supports **SetDIBitsToDevice**, the call is converted to a **SetDIBitsToDevice** call to the driver. (This works only with SRCCOPY as the ROP.) If neither of these methods is possible, **CreateDIBitmap** is used to make a device-dependent version of the bitmap, and **StretchBlt** is called to do the actual work.

The parameters for **StretchDIBits** are basically the same as for **StretchBlt** (with the source *hDC* replaced by *lpBits* and *lpInfo*). This function does not have the *nStartScan* and the *nNumScans* parameters of the other DIB functions, so *lpBits* always points to the first scan of the DIB.

When using this function for anything other than full bitmap stretches, remember that all of the source coordinates (the ones relating to the DIB) are in an upside-down system. The function will appropriately flip the image, but the source rectangle is defined with Y=0 at the bottom and extents going up. Fortunately, the x-coordinates use the same conventions as Windows.

Printer drivers that do support this functionality (for example, PSCRIPT and HPPCL) usually use a halftone algorithm to output good color images. Therefore, maintaining DIBs at the highest meaningful color resolution possible (usually 8 bit) is desirable even if the output device is monochrome, because the color information is still useful for good output. Unfortunately, most printer drivers do not support any ROP other than SRCCOPY.

CreateDIBPatternBrush

This function allows an application to create a pattern brush by specifying a DIB instead of a device-dependent bitmap, as used in the **CreatePatternBrush** function. A brush created using this function is used like any other brush. The DIB is turned into a device-dependent bitmap at **SelectObject** time for use by the device. This brush looks like a standard pattern brush to the device.

DIBS in the Clipboard

Two basic mechanisms for placing DIBs in the Clipboard are using the CF_DIB data format or placing the DIB into a metafile and using the CF_METAFILEPICT data format.

The CF_DIB format uses a packed DIB, in which the bits follow immediately after the header and the color table. When reading or creating a packed DIB, an application must properly calculate the size of the color table to ensure that the bits are in the proper place. Because all DIB functions expect the DIB as two pointers, one to the header and one to the bits, the bits pointer must be calculated before use. (For color table size computations, see the code sample in the color table description above.)

The simplest way to place a DIB into a metafile is to use **StretchDIBits**:

```
hMetaDC = CreateMetaFile((LPSTR) NULL));
StretchDIBits(hMetaDC, 0, 0, biWidth, biHeight, 0, 0, biWidth,
     biHeight, lpBits, lpInfo, DIB_RGB_COLORS, SRCCOPY);
hMetafile = CloseMetaFile(hMF);
```

This approach generates a metafile that when played back displays the DIB to the destination. This method also scales the image to fit the current mapping scheme if needed. Using metafiles for transfer enables even applications that are not DIB-aware to paste the contents of the Clipboard without losing the DIB information.

RLE Formats

When the **biCompression** field in a DIB's header is set to either BI_RLE4 (for **biBitCount** = 4) or BI_RLE8 (for **biBitCount** = 8), the image has been run length encoded. A description of the encoding schemes can be found in the SDK *Reference—Volume 2* manual, in the "BITMAPINFOHEADER" section of the "Datatypes and Structures" chapter. The basic scheme involves compressing multiple, horizontally adjacent, identical pixels into a run encoding. For example, 10 pixels of color index 17 are encoded as a run of length 10 and of index 17. Codes for end-of-scan and for delta moves are also provided, in which an *X* and a *Y* offset are provided for the next pixel.

This type of encoding usually compresses the bitmap and is also useful for creating sprite-type animations, in which only a small part of an image changes in each frame. The animation capabilities are accomplished by using delta codes to limit the number of pixels actually being set. Pixels skipped by a delta move are left untouched.

The main limitations of RLE DIBs are that an application can neither easily determine the size of the bitmap in bytes nor point to a certain scanline without decoding the bitmap from the first scan. The **biSizeImage** field is useful in solving the first problem. Decoding, encoding, and generally manipulating the RLE format is slower and more complicated than the noncompressed (BI_RGB) format. Some applications—for example, Paintbrush—refuse to read RLE DIBs. Although all APIs accept them, RLE DIBs will probably not become a universally supported format. Also, because of the relative rarity of these formats, some device drivers might not have fully tested support for the encoding and decoding processes.

To generate an RLE DIB, **GetDIBits** is called with **biCompression** set to the desired type of encoding. The amount of memory needed to store the bits is not easily computed. If **GetDIBits** is called with *lpBits* set to NULL, the amount of memory needed for the bits is returned in **biSizeImage**. A subsequent call with *lpBits* pointing to a properly sized block of memory returns an encoded bitmap.

Translating an RLE DIB into a device-dependent form requires no special processing. Any of the **Set** functions can be used normally with a header containing the proper **biSizeImage** and **biCompression** values to match the bits.

Shortcomings of DIBS

Probably the biggest limitation of DIBs is that they are slower than device-dependent bitmaps. Translating DIBs into a device-dependent form before they can actually be displayed requires extra processing, resulting in additional overhead. In an ideal world, a one-to-one **StretchDIBits** would be as fast as a **BitBlt**. This speed would allow an application to operate effectively in the realm of the logical bitmap, with full color and full access to each and every pixel, regardless of the physical device's limitations.

DIBs are based in a coordinate system that is upside down relative to Windows, making coding a bit frustrating and not intuitive. Always remembering this quirkiness should help limit the number of iterations needed to get bitmaps properly lined up.

You can get full color using 24-bit DIBs, but they are very slow to decode, read, and write. This is especially true on 8-bit palette devices, in which translation literally can take minutes. Also, the sheer size of 24-bit DIBs makes them a bit unwieldy for general use.

DIB-Related Problems in Windows Version 3.0

Metafile recording of **StretchDIBits** calls that use **BITMAPCOREHEADER** causes a UAE. Convert all headers to the **BITMAPINFO** style to avoid this problem. This workaround is recommended for general DIB processing.

The **SetDIBits** simulation code for >64K monochrome DIBs causes crashes or erroneous output when using **SetDIBits**, **SetDIBitsToDevice**, or **StretchDIBits** to a driver that does not support **SetDIBits**.