

CHAPTER 15: THE DEVICE-INDEPENDENT BITMAP

The [Windows GDI bitmap object](#), also known as the device-dependent bitmap (DDB), is a versatile tool for graphics programming. However, as we saw in the previous chapter, its limitations become apparent when dealing with image persistence. Saving DDBs to disk and loading them back into memory is cumbersome and outdated due to their device-dependent nature.

Enter the [device-independent bitmap \(DIB\)](#), introduced in Windows 3.0 as a dedicated image file format for image interchange. While formats like GIF and JPEG dominate the internet due to their efficient compression, DIBs offer distinct advantages, especially for programmatic manipulation.

Device Dependence vs. Device Independence:

Imagine a [DDB](#) as a bitmap tailored to a specific display device. Its pixel format and color representation are intricately linked to that device's capabilities. Saving such a bitmap wouldn't translate well to other devices with different display characteristics. Colors might appear distorted, and the entire image could be unreadable.

A [DIB](#), in contrast, breaks free from these shackles. It encapsulates the image data along with a comprehensive color table. This table defines a precise mapping between pixel values and actual colors, independent of the display device. Think of it as a [universal translator for your image](#), ensuring consistent representation across different platforms.

Benefits of DIBs:

Direct Windows API Support: Unlike compressed formats like GIF and JPEG, DIBs are readily processed by the Windows API. You can directly pass a DIB in memory to various functions for displaying, manipulating, or converting it into a DDB for immediate rendering. This simplifies your programming tasks and eliminates the need for external decoders or converters.

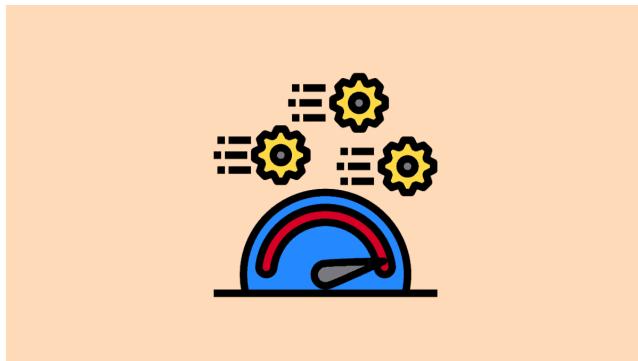


MICROSOFT®
WINDOWS®
COMPATIBLE

Lossless Image Quality: While DIBs offer optional compression, they often remain uncompressed. This might seem inefficient compared to compressed formats, but it holds a significant advantage: lossless image quality. Every pixel retains its original data, crucial for tasks like image editing or analysis where even minor distortions are undesirable.



Flexibility and Control: With direct access to the uncompressed pixel data, you have complete control over how you manipulate the image within your program. You can modify individual pixels, adjust color palettes, or perform complex image processing algorithms without the limitations imposed by compressed formats.



DIBs in the Modern Landscape:

While [DIBs may not be the internet's preferred image format](#) for casual sharing, their strengths shine in specific scenarios. Developers working with graphics-intensive applications, image editing tools, or scientific visualization software often rely on DIBs for their ease of use, direct API integration, and lossless image fidelity.

In conclusion, the [device-independent bitmap offers a valuable alternative](#) to compressed image formats when prioritizing programmatic manipulation and lossless image quality. While its uncompressed nature might seem bulky compared to its internet-savvy counterparts, DIBs remain a powerful tool for graphics professionals and developers seeking fine-grained control over their visual data.

Remember:

- DIBs are device-independent, meaning they retain their appearance across different devices due to their embedded color table.
- Unlike compressed formats like GIF and JPEG, DIBs are often uncompressed, offering lossless image quality but larger file sizes.
- DIBs are directly supported by the Windows API, simplifying image manipulation and integration within your programs.

DELVING DEEPER INTO THE DIB FILE FORMAT: A COMPREHENSIVE EXPLORATION

Origins and Evolution:

[Rooted in OS/2](#): Embarking on its journey in OS/2 1.1, the DIB format was initially known as the Presentation Manager (PM) bitmap format. It was later embraced by Windows 3.0 in 1990 and has undergone numerous refinements throughout subsequent Windows versions.

Key Characteristics:

[File Extensions](#): .BMP and .DIB serve as the common file extensions associated with DIB files.

[Device Independence](#): A hallmark feature of DIBs is their ability to maintain consistent visual integrity across a diverse spectrum of devices. This remarkable feat is achieved through the incorporation of color information directly within the file itself.

[Memory Representation](#): When loaded into memory, DIBs assume the form of a "packed-DIB" structure, a compact and efficient representation designed to facilitate seamless manipulation and exchange.

[Windows API Integration](#): Windows API offers a suite of functions specifically designed to interact with DIBs, enabling operations such as display, conversion, and printing.

[Versatile Customizability](#): DIBs extend their capabilities beyond the built-in API functions, empowering developers to craft custom code for more intricate image processing tasks.

Common Applications:

Application Resources: DIB files frequently serve as repositories for visual elements within applications, such as button images and icons.

Icons and Mouse Cursors: The structure of icons and mouse cursors shares a close kinship with DIBs, demonstrating their versatility in graphical user interface elements.

Clipboard Image Exchange: DIBs provide a robust foundation for transferring images seamlessly across applications via the Windows clipboard.

Brush Creation: The creation of custom brushes for painting and drawing operations often relies upon DIBs as a cornerstone.

Image Manipulation and Processing: DIBs provide a fertile ground for programmers to implement algorithms for image editing, analysis, filtering, and other advanced image-related tasks.

File Structure:

File Header: The first 14 bytes of a DIB file constitute the file header, which encapsulates general information such as file size and format identification.

DIB Header: Following the file header, the DIB header emerges, bearing variable size and disclosing crucial details about the image itself, including dimensions, color depth, compression methods, and the presence or absence of a color table.

Color Table (Optional): For DIBs utilizing indexed color palettes, a color table resides within the file, meticulously mapping pixel values to their corresponding colors.

Pixel Data: The heart of the DIB file resides in the pixel data, which meticulously stores the raw image information, often in an uncompressed format to preserve image fidelity.

In-Memory Representation:

Packed-DIB Format: Upon loading into memory, DIBs transform into the "packed-DIB" format, a streamlined structure optimized for memory efficiency and effortless manipulation.

Programmatic Creation: Developers wield the power to construct DIBs directly within memory, enabling subsequent saving to files or utilization for image processing tasks.

Windows API Support:

Display and Conversion: The Windows API furnishes a repertoire of functions designed to display DIBs gracefully on both screens and printers, as well as gracefully converting them to and from device-dependent bitmaps (DDBs).

Beyond Built-in Functions:

Custom Programming: To venture beyond the frontiers of the Windows API and achieve sophisticated image manipulation techniques such as color depth conversions, palette optimization, or the application of artistic filters and effects, custom programming often becomes indispensable.

Key Takeaways:

- DIBs stand as a potent tool for device-independent image storage and manipulation, offering a compelling blend of versatility and control.
- Windows API integration streamlines common DIB operations, providing a solid foundation for developers.
- The extensibility of DIBs empowers programmers to venture beyond the confines of built-in API functions, unlocking a realm of limitless possibilities in the realm of image processing and manipulation.

DELVING INTO THE OS/2-STYLE DIB FORMAT

File Structure:

File Header (14 bytes):

- **bfType:** Signature "BM" (0x4D42) to identify a bitmap file.
- **bfSize:** Total file size in bytes.
- **bfReserved1:** Always zero.
- **bfReserved2:** Always zero.
- **bfOffBits:** Offset in bytes to the pixel bits.

Information Header (12 bytes):

- **bcSize:** Size of the BITMAPCOREHEADER structure (12 bytes).
- **bcWidth:** Width of the bitmap in pixels.
- **bcHeight:** Height of the bitmap in pixels.
- **bcPlanes:** Always 1.
- **bcBitCount:** Number of bits per pixel (1, 4, 8, or 24).

Color Table (optional, for 1, 4, and 8 bits per pixel):

- Array of RGBTRIPLE structures representing colors.
- Size depends on bit count: 2 colors for 1 bit, 16 for 4 bits, 256 for 8 bits.

Pixel Bits:

- Raw image data, arranged sequentially row by row.
- Storage depends on bit count:
 - ✓ **1 bit:** Each byte represents 8 pixels.
 - ✓ **4 bits:** Each byte represents 2 pixels.
 - ✓ **8 bits:** Each byte represents 1 pixel.
 - ✓ **24 bits:** Each pixel uses 3 bytes (RGB).

Code Examples:

Allocating memory for an 8-bit DIB information structure:

```
PBITMAPCOREINFO pbmci = malloc(sizeof(BITMAPCOREINFO) + 255 * sizeof(RGBTRIPLE));
```

Accessing a color table entry:

```
RGBTRIPLE color = pbmci->bmciColors[i];
```

Key Points:

- OS/2-style DIBs support 1, 4, 8, or 24 bits per pixel.
- Color tables are only present for 1-, 4-, and 8-bit DIBs.
- Pixel data arrangement depends on bit count.
- Important colors should be placed first in the color table for optimal display.
- The pixel data block always starts at a WORD address boundary.

```

1  #include <stdlib.h>
2  // Define the file header structure
3  typedef struct tagBITMAPFILEHEADER {
4      uint16_t bfType;          // "BM" or 0x4D42
5      uint32_t bfSize;         // entire size of file
6      uint16_t bfReserved1;    // must be zero
7      uint16_t bfReserved2;    // must be zero
8      uint32_t bfOffsetBits;   // offset in file of DIB pixel bits
9  } BITMAPFILEHEADER, *PBITMAPFILEHEADER;
10
11 // Define the information header structure
12 typedef struct tagBITMAPCOREHEADER {
13     uint32_t bcSize;         // size of the structure = 12
14     uint16_t bcWidth;        // width of image in pixels
15     uint16_t bcHeight;       // height of image in pixels
16     uint16_t bcPlanes;       // = 1
17     uint16_t bcBitCount;     // bits per pixel (1, 4, 8, or 24)
18 } BITMAPCOREHEADER, *PBITMAPCOREHEADER;
19 // Define the RGBTRIPLE structure for the color table
20 typedef struct tagRGBTRIPLE {
21     uint8_t rgbtBlue;        // blue level
22     uint8_t rgbtGreen;       // green level
23     uint8_t rgbtRed;         // red level
24 } RGBTRIPLE;
25 // Define the combined structure for DIB with color table
26 typedef struct tagBITMAPCOREINFO {
27     BITMAPCOREHEADER bmciHeader;      // core-header structure
28     RGBTRIPLE bmciColors[1];         // color table array
29 } BITMAPCOREINFO, *PBITMAPCOREINFO;
30
31 int main() {
32     // Allocate memory for the combined structure including the color table
33     PBITMAPCOREINFO pbmci = malloc(sizeof(BITMAPCOREINFO) + 255 * sizeof(RGBTRIPLE));
34     // Access the RGBTRIPLE structure within the color table
35     RGBTRIPLE color = pbmci->bmciColors[i];
36     // Free the allocated memory when done
37     free(pbmci);
38     return 0;
39 }

```

The provided code is an example of C programming code that deals with bitmap image file headers and structures. Let's break down the code and explain its functionality in depth.

The code begins by including the standard library header file `<stdlib.h>`. This header file provides functions for memory allocation and deallocation, such as `malloc()` and `free()`.

Next, the code defines several structures that represent different parts of a bitmap image file.

BITMAPFILEHEADER:

This structure represents the file header of a bitmap image. It contains the following fields:

- **bfType:** A 16-bit field that specifies the file type. In the case of bitmap files, it should have the value "BM" or 0x4D42 in hexadecimal.
- **bfSize:** A 32-bit field that represents the entire size of the file in bytes.
- **bfReserved1** and **bfReserved2:** Two 16-bit reserved fields that must be set to zero.
- **bfOffsetBits:** A 32-bit field that indicates the offset in the file where the pixel data starts.

BITMAPCOREHEADER:

This structure represents the information header of a bitmap image. It contains the following fields:

- **bcSize:** A 32-bit field that specifies the size of the structure in bytes. For the core header, this value is fixed at 12.
- **bcWidth:** A 16-bit field that represents the width of the image in pixels.
- **bcHeight:** A 16-bit field that represents the height of the image in pixels.
- **bcPlanes:** A 16-bit field that indicates the number of color planes in the image. For bitmap images, this value is always 1.
- **bcBitCount:** A 16-bit field that specifies the number of bits per pixel. Valid values are 1, 4, 8, or 24.

RGBTRIPLE:

This structure represents a single pixel in the color table of a bitmap image. It contains the following fields:

- **rgbtBlue:** An 8-bit field that represents the intensity of blue color.
- **rgbtGreen:** An 8-bit field that represents the intensity of green color.
- **rgbtRed:** An 8-bit field that represents the intensity of red color.

BITMAPCOREINFO:

This structure combines the BITMAPCOREHEADER and RGBTRIPLE structures. It represents the DIB (Device Independent Bitmap) with color table. It contains the following fields:

- **bmciHeader:** A BITMAPCOREHEADER structure that represents the core header.
- **bmciColors:** An array of RGBTRIPLE structures that represents the color table. In this code example, it is defined as an array with a single element, but it can accommodate up to 256 elements.

After defining the structures, the main() function begins.

Inside the main() function, memory is allocated for the combined structure BITMAPCOREINFO using the malloc() function.

The size of the allocated memory is calculated as the sum of the size of BITMAPCOREINFO and the size of the color table (255 * sizeof(RGBTRIPLE)).

The malloc() function returns a pointer to the allocated memory, which is assigned to the pbmci pointer of type PBITMAPCOREINFO.

Next, there is a line of code that accesses a specific element of the color table. However, the variable i is not defined in the provided code snippet, so it's unclear what the intention is.

The code should include a loop or a specific value assigned to i to access a valid element of the color table.

Finally, the allocated memory is freed using the free() function. This step is important to release the memory back to the system when it is no longer needed.

In summary, the code demonstrates the allocation and deallocation of dynamic memory for a bitmap image's DIB structure, including the color table. It provides the framework for accessing individual elements of the color table, but the specific usage is incomplete without more context or additional code.

BOTTOM'S UP! DEMYSTIFYING THE DIB'S COUNTERINTUITIVE PIXEL ORDER

The DIB file format throws a curveball at those familiar with conventional bitmap organization. Unlike most other formats, where pixels march from top to bottom, [DIBs embrace a bottom-up approach](#). The first row you encounter in the file actually corresponds to the bottom row of the image, while the top row resides at the file's tail.

Terminology:

Top and Bottom Rows: Visual representation of the image, with "top" being the hair in a portrait and "bottom" being the chin.

First and Last Rows: File-based perspective, with "first" following the color table and "last" marking the end of the file.

Why the Bottom-Up Order?

This seemingly bizarre arrangement stems from the legacy of OS/2's Presentation Manager (PM). IBM designers sought a unified coordinate system across windows, graphics, and bitmaps. A debate ensued:

Traditionalists: Most people, accustomed to text programming and windowing environments, favor increasing vertical coordinates downward.



Mathematicians: Hardcore graphics programmers, rooted in analytic geometry, prefer Cartesian coordinates with "up" represented by higher y-values.



The mathematicians won. This bottom-up philosophy permeated PM, including window coordinates, and consequently, DIBs inherited this quirk.

Consequences:

- Code dealing with DIBs needs to account for this reversed order.
- Image processing algorithms might require adjustments for bottom-up processing.
- However, the internal consistency within PM offered its own advantages in terms of coherence and development efficiency.

In conclusion, while the bottom-up approach might initially seem counterintuitive, understanding its historical roots and the underlying design decisions can shed light on this unique characteristic of the DIB format.

Remember, programmers working with DIBs need to be mindful of this order and adjust their routines accordingly. However, the internal consistency within the OS/2 system offered its own merits in terms of streamlined development and cohesive behavior.

DECODING THE LANGUAGE OF DIB PIXEL BITS: A CLOSER LOOK

Revealing the Pixel Grid:

At the core of a DIB file lies the pixel information, arranging tiny dots to craft the visual masterpiece. Grasping this arrangement is vital for interpreting and tweaking DIB images. Let's dive into this intriguing world:

Bottom-Up Climb:

Unlike regular bitmaps that start from the top, DIBs adopt a bottom-up style. The initial bytes of the file represent the bottom row of the image, and rows build upwards. This quirky approach comes from the OS/2 Presentation Manager's quest for a consistent coordinate system.

Left-to-Right March:

Within each row, pixels move from left to right in an orderly fashion. This maintains consistency in visual representation and data handling.

Padding for Efficiency:

To match memory architecture and boost processing, each row gets padded with zeros on the right until its length is a multiple of 4 bytes. This ensures smooth memory access and harmony with hardware processing units.

Bit Depth Guides Pixel Encoding:

1-bit DIBs (Simple Black and White):

Every byte oversees 8 pixels. The leftmost pixel takes the lead by claiming the top bit. Pixel values of 0 or 1 map to the 2-color palette, deciding between the first or second color.

Pixel:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

4-bit DIBs (16 Distinct Shades):

Each byte manages two pixels. The leftmost pixel controls the high 4 bits, and the second pixel sits in the lower 4 bits. Values from 0 to 15 guide color selection from the 16-color palette.

8-bit DIBs (256 Vibrant Tones):

Each byte represents a single pixel. Pixel values from 0 to 255 link to the 256-color palette, creating a canvas of 256 unique shades.

24-bit DIBs (True Color Bliss):

Each pixel enjoys 3 dedicated bytes for red, green, and blue. Rows turn into arrays of RGBTRIPLE structures, encapsulating color intensity. Padding remains key for optimal memory alignment.

A repeat of the above page for clarity:

For DIBs with 1 bit per pixel, each byte corresponds to 8 pixels. The leftmost pixel is the most-significant bit of the first byte:

Pixel: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0

Each pixel can be either a 0 or a 1. A 0 bit means that the color of that pixel is given by the first RGBTRIPLE entry in the color table. A 1 bit is a pixel whose color is the second entry of the color table.

For DIBs with 4 bits per pixel, each byte corresponds to 2 pixels. The leftmost pixel is the high 4 bits of the first byte, and so on:

Pixel: —0— —1— —2— —3— —4— —5—
7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0

The value of each 4-bit pixel ranges from 0 to 15. This value is an index into the 16 entries in the color table.

For a DIB with 8 bits per pixel, each byte is 1 pixel:

Pixel: —0— —1— —2—
7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0

The value of the byte is 0 through 255. Again, this is an index into the 256 entries in the color table.

For DIBs with 24 bits-per-pixel, each pixel requires 3 bytes for the red, green, and blue color values. Each row of pixel bits is basically an array of RGBTRIPLE structures, possibly padded with 0 bytes at the end of each row so that the row has a multiple of 4 bytes:

Pixel: —Blue— —Green— —Red—
7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0

Again, the 24-bit-per-pixel DIB has no color table.

Note that the explanation below is AI generated, and might not actually represent the notes in the image above, but atleast adds some good points.

1. File Structure:

The image depicts the various sections of a DIB file, starting with the file header at the top and working its way down to the pixel data.

It accurately reflects the presence and order of these sections: file header, information header (including bit count and color table for certain bit depths), and finally, the pixel bits themselves.

2. Pixel Encoding:

The image visually showcases the different pixel encoding schemes based on bit depth:

1-bit DIBs: Each byte controls 8 pixels, with the leftmost bit being the most significant.

4-bit DIBs: Two pixels per byte, with the high 4 bits representing the first pixel and the low 4 bits representing the second.

8-bit DIBs: One byte per pixel, directly corresponding to the color table index.

24-bit DIBs: Three bytes per pixel, dedicated to red, green, and blue color values.

3. Bottom-Up Order:

The image subtly hints at the bottom-up organization of DIBs by placing the "bottom row" at the top of the diagram and the "top row" at the bottom. This might seem counterintuitive compared to most bitmap formats, but it reflects the legacy of the OS/2 Presentation Manager's coordinate system.

Navigating the Pixel Landscape:

Unraveling DIBs: Decoding and Tweaking Pixels with Confidence

Now that we've decoded the pixel mysteries in various DIB bit depths, let's explore practical techniques for accessing and altering individual pixels.

Understanding Pixel Access:

Pixel Location Computation: To access a pixel, translate its row and column coordinates into a byte offset within the pixel data.

Formula: `ByteOffset = Row * RowLength + Column * (bcBitCount / 8).`

Extracting Pixel Value: Once the byte offset is known, extract pixel color information based on bit depth using specific techniques.

Pixel Manipulation Magic:

With pixel access techniques in hand, let's dive into exciting image manipulation:

Changing Pixel Colors: Modify the pixel value for desired color changes, like flipping the bit value for inverting colors.

Applying Image Filters: Iterate through pixels, applying mathematical operations or custom algorithms for effects like grayscale conversion or artistic filters.

Transparency Control: Extend techniques for DIBs with alpha channels to manipulate transparency alongside RGB components, enabling blending effects.

Remember:

- Byte padding adds complexity, so be careful when iterating through pixels or performing operations.
- Error checking is crucial to avoid accessing invalid memory locations.

By applying these techniques and understanding pixel encoding intricacies, you can unleash the power of manipulating DIBs for your creative ventures. So, explore and harness the pixel magic to paint your digital masterpieces!

UNVEILING THE EXPANDED WINDOWS DIB: A DEEPER DIVE INTO ITS STRUCTURE AND ENHANCEMENTS

While the OS/2-compatible DIB format provided a solid foundation for image representation, Windows 3.0 introduced an expanded version known as the Windows DIB, offering enhanced capabilities and addressing potential limitations. Let's delve into its captivating structure:

1. File Header:

Remains identical to the OS/2-compatible DIB, employing the BITMAPFILEHEADER structure to initiate the file.

2. Information Header:

Here's where Windows DIBs deviate, embracing the BITMAPINFOHEADER structure instead of BITMAPCOREHEADER. Key distinctions include:

- **Size:** It boasts a size of 40 bytes, larger than the 12-byte BITMAPCOREHEADER.
- **Width and Height:** Utilizes 32-bit LONG values for width (biWidth) and height (biHeight), enabling representation of larger images compared to the 16-bit WORD values in BITMAPCOREHEADER.
- **Additional Fields:** Unveils six new fields, expanding its capabilities:
 - ✓ **biCompression:** Specifies compression method applied to pixel data.
 - ✓ **biSizeImage:** Indicates total number of bytes occupied by pixel data.
 - ✓ **biXPelsPerMeter:** Horizontal resolution in pixels per meter.
 - ✓ **biYPelsPerMeter:** Vertical resolution in pixels per meter.
 - ✓ **biClrUsed:** Number of color table entries actually used.
 - ✓ **biClrImportant:** Number of colors deemed essential for accurate image display.

3. Color Table (1-, 4-, and 8-bit DIBs):

Shifts from RGBTRIPLE structures to RGBQUAD structures, the latter incorporating an additional unused byte (rgbReserved) set to 0. This padding aligns color table entries on 32-bit address boundaries, optimizing access for 32-bit processors.

4. BITMAPINFO Structure:

Unifies BITMAPINFOHEADER and color table into a single structure for convenience:

```
typedef struct tagBITMAPINFO {
    BITMAPINFOHEADER bmiHeader; // Information header
    RGBQUAD bmiColors[1];      // Color table array
} BITMAPINFO, *PBITMAPINFO;
```

Key Points:

- ✓ Distinguish Windows DIBs from OS/2-compatible DIBs by checking the size of the information header: 40 bytes for Windows DIBs, 12 bytes for OS/2-compatible DIBs.
- ✓ Windows DIBs support larger image dimensions and compression, offer resolution information, and enhance memory alignment for 32-bit processors.
- ✓ The BITMAPINFO structure streamlines access to both header and color table data.

DEMYSTIFYING THE EXPANDED WINDOWS DIB: A SIMPLIFIED EXPLORATION

Remember the Windows DIB format introduced in Windows 3.0? Well, Windows 95 and NT 4.0 brought some upgrades and quirks to it. Buckle up as we unravel these changes in plain English:

New Fields and Features:

biHeight: Can now be negative! This flips the image upside down, a feature some programs might not understand, so avoid creating these unless you want potential crashes.

biBitCount: Expanded to include 16 and 32 bits per pixel, offering more color options (we'll explore these formats later).

biXPelsPerMeter and **biYPelsPerMeter:** Tell the image's real-world size in pixels per meter, useful for accurate display but rarely used by Windows itself.

biClrUsed: A crucial field! For 4-bit and 8-bit DIBs, it can shrink the color table, saving space. For 16-bit, 24-bit, or 32-bit DIBs, it shows the size of a custom palette for 256-color displays.

biClrImportant: Less important than biClrUsed, usually 0 or equal to biClrUsed. If set between 0 and biClrUsed, it means the image can be displayed okay using only those colors.

Color Table Twists:

For 16-bit, 24-bit, and 32-bit DIBs, Windows itself ignores the color table. But, it shows the size of a custom palette an app could use for 256-color displays.

Warning: Older programs might get confused if you include a color table in a 24-bit DIB!

Pixel Bits:

For 1-bit, 4-bit, 8-bit, and 24-bit DIBs, pixel organization remains the same as the OS/2-compatible format. We'll tackle 16-bit and 32-bit formats later.

Remember:

- ✓ These changes, while offering more flexibility, can cause compatibility issues with older programs. Use them with caution!
- ✓ The new fields and color table options provide more control and customization for developers working with DIBs.

Pixels per Meter: Revealing Real-World Dimensions

The "biXPelsPerMeter" and "biYPelsPerMeter" fields in a DIB file provide clues about the intended physical size of the image, similar to how we measure a photograph in inches or centimeters.

To understand these fields, imagine a flexible ruler where each tiny mark represents a pixel. This virtual ruler extends for a whole meter, and the fields tell us how many pixels fit in that meter.

While Windows doesn't pay much attention to these fields by default, some specialized image programs can use this information to make sure the image is displayed or scaled accurately.

In simpler terms, these fields give us an idea of how many pixels should be in a meter of physical space in the image. Although Windows doesn't use this information much, other programs that work with images may find it helpful to show or resize the image correctly.

Common Values:

- 0: No suggested real-world size, leaving it open to interpretation.
- 2835: A common value, roughly equivalent to 72 dots per inch, often used for video displays.
- 11811: Represents a resolution of 300 dots per inch, frequently encountered in the realm of printing.

Tailoring the Color Palette: The Power of biClrUsed

Curating the Colors: This field wields considerable influence over the composition of the color table, playing a key role in both image size and visual possibilities.

Color Table Minimalism (4-bit and 8-bit DIBs): By setting biClrUsed to a non-zero value, one can craft a more compact color table, potentially trimming the overall image size.

Custom Palettes for Limited Displays (16-bit, 24-bit, 32-bit DIBs): In these high-color-depth formats, biClrUsed sheds its space-saving role and instead reveals the size of a custom color palette. This palette can be employed by programs to gracefully display the image on devices with a restricted color range, such as 256-color displays.

Essential Guidelines:

A value of 0 indicates that there is a complete color table, and its size is determined by the "biBitCount" field.

Non-zero values indicate a more selective color table, which can result in space savings or adaptability for displays with limited color capabilities.

Windows 95's Colorful Twist:

Unlocking Flexibility: With Windows 95, 24-bit DIBs gained the ability to embrace color tables, a feature previously forbidden in earlier DIB formats. This opened doors for greater customization and compatibility, especially when working with devices or programs that might have color limitations.

Understanding DIBs (Device-Independent Bitmaps):

Key Points:

- ▀ DIBs store image data in a way that works across different devices.
- ▀ They have a header and a color table (sometimes).
- ▀ The header tells you important details about the image.
- ▀ The color table translates pixel values into actual colors.

Important Fields in the Header:

- ▀ **biClrUsed:** This field tells you how many colors are in the color table.
- ▀ For 1-bit DIBs, it's always 0 or 2 (meaning 2 colors).
- ▀ For 4-bit DIBs, it's usually 0 or 16 (16 colors), but it can be less.
- ▀ For 8-bit DIBs, it's usually 0 or 256 (256 colors), but it can be less.
- ▀ For 16-bit, 24-bit, and 32-bit DIBs, it's usually 0 (no color table).
- ▀ **biClrImportant:** This field is less important than biClrUsed. It usually tells you which colors are most important for a good image, but it's not always used.

Color Table:

- ▀ The color table is like a dictionary for pixel values.
- ▀ Each entry in the table tells you the real color that a pixel value stands for.
- ▀ Not all DIBs have color tables (like 24-bit and 32-bit DIBs usually don't).

Pixel Data:

- ▀ The pixel data is the actual image information, stored as a bunch of numbers.
- ▀ The way the pixel data is organized depends on the DIB's color depth (how many bits per pixel).

Additional Notes:

- ▀ Windows 95 made a change: 24-bit DIBs can now have color tables (but older programs might not expect it).
- ▀ The biClrImportant field can be useful for displaying multiple DIBs on 256-color displays.
- ▀ The organization of pixel bits is mostly the same as in OS/2-compatible DIBs, except for 16-bit and 32-bit DIBs (which we'll talk about later).

1. Overview of DIBs

When encountering a Device-Independent Bitmap (DIB) created by another program or person, various characteristics can be expected, depending on the historical context.

2. Evolution of DIBs

Originally, OS/2-style DIBs were common during the release of Windows 3.0 but have become rare in recent years. Some programmers tend to overlook 4-bit DIBs, often created in the Windows Paint program using a 16-color video display, with a standard color table.

3. Common 8-bit DIBs

The most common 8-bit DIBs fall into two categories: gray-shade DIBs and palletized color DIBs. Unfortunately, the DIB header lacks information about the type of 8-bit DIB being dealt with.

4. Gray-shade DIBs

Gray-shade DIBs with a bit count of 8 may have a color table with 64 entries. These entries represent ascending levels of gray, and the pixel values can be directly interpreted as proportional levels of gray. The color table is often calculated using specific formulas.

```
// Example formula for color table calculation
rgb[i].rgbRed = rgb[i].rgbGreen = rgb[i].rgbBlue = i * 256 / 64;
```

5. Variations in Gray-shade DIBs

Some gray-shade DIBs may have 256 entries in the color table. The biClrUsed field indicates the number of entries, ranging from 0 to 256. Notably, when the color table consists entirely of gray shades, pixel values directly represent proportional levels of gray.

6. Palletized 8-bit Color DIBs

Palletized 8-bit color DIBs often use the entire color table. The biClrUsed field can be 0 or 256, but occasionally, a smaller number may be used (e.g., 236) to accommodate program limitations in changing entries in the Windows color palette.

7. Uncommon Fields

Encountering non-zero values for biXPelsPerMeter and biYPelsPerMeter is rare. Similarly, a biClrImportant field with a value other than 0 or biClrUsed is infrequent.

8. Potential Enhancement

Given the prevalence of gray-shade DIBs, a potential enhancement to the BITMAPINFOHEADER structure could be the addition of a flag indicating that the DIB image is gray-shaded, lacks a color table, and that pixel values directly indicate the gray level.

Key Points:

Common DIB Types:

- 8-bit DIBs: Most common, either gray-shade or palletized color.
- 4-bit DIBs: Less common, often from Windows Paint.
- OS/2-style DIBs: Rare.

8-Bit DIBs:

Gray-Shade DIBs:

- **biClrUsed:** Number of entries in color table (often 64 or 256).
- **Color table:** Entries in ascending levels of gray.
- **Pixel values:** Directly represent gray levels when color table has equal RGB levels and uniform gray shades.

Palletized Color DIBs:

- **biClrUsed:** Usually 0 (full 256-color table) or 236 (due to Windows color palette limitations).

Code Examples:

Generating Gray-Shade Color Tables:

```
// Using formula for 64 shades:
for (int i = 0; i < 64; i++) {
    rgb[i].rgbRed = rgb[i].rgbGreen = rgb[i].rgbBlue = i * 256 / 64;
}

// Using formula for other numbers of shades (e.g., 32):
for (int i = 0; i < 32; i++) {
    rgb[i].rgbRed = rgb[i].rgbGreen = rgb[i].rgbBlue = i * 255 / 31;
```

Accessing Pixel Values in Gray-Shade DIBs:

```
// Assuming biClrUsed is 64:
for (int y = 0; y < bitmapHeight; y++) {
    for (int x = 0; x < bitmapWidth; x++) {
        int pixelValue = GetPixel(x, y); // Get pixel value (0x00 to 0x3F)
        int grayLevel = pixelValue * 255 / 63; // Calculate gray level (0 to 255)
        // Use grayLevel for processing or display
    }
}
```

Additional Notes:

- **biXPelsPerMeter, biYPelsPerMeter:** Often 0, rarely used.
- **biClrImportant:** Usually 0 or equal to biClrUsed.
- **BITMAPINFOHEADER:** Consider adding a flag to indicate gray-shade DIBs directly.

COMPRESSION IN DEVICE-INDEPENDENT BITMAPS (DIBS):

1. Overview

The biCompression and biSizeImage fields in the BITMAPINFOHEADER play a crucial role in specifying compression and size information for DIBs.

2. Compression Constants

The biCompression field can take four constants: BI_RGB, BI_RLE8, BI_RLE4, or BI_BITFIELDS, represented by values 0 through 3 in the WINGDI.H header. For 4-bit and 8-bit DIBs, it indicates whether the pixel bits are compressed using run-length encoding. For 16-bit and 32-bit DIBs, it signifies the use of color masking introduced in Windows 95.

3. RLE Compression

For 1-bit DIBs: biCompression is always BI_RGB.

For 4-bit DIBs: biCompression can be either BI_RGB or BI_RLE4.

For 8-bit DIBs: biCompression can be either BI_RGB or BI_RLE8.

For 24-bit DIBs: biCompression is always BI_RGB.

4. Run-Length Encoding (RLE) Overview

RLE is a simple data compression method based on the repetition of identical pixels. It encodes the pixel value and the number of repetitions.

5. RLE Compression for 8-bit DIBs

The table below illustrates how pixel bits are encoded when biCompression is BI_RLE8.

Byte 1	Byte 2	Meaning
00	00	End of row
00	01	End of image
00	02	dx dy (Move to x+dx, y+dy)
00	n (03-FF)	Use next n pixels
n (01-FF)	pixel	Repeat pixel n times

When decoding, pairs of bytes are examined, and if the first byte is nonzero, it indicates a run-length repetition factor.

If the first byte is 00 followed by 02, the next two bytes are added as unsigned increments to the current x and y values.

If the first byte is 00 followed by 00, the row is finished, resetting x to 0 and incrementing y.

If the first byte is 00 followed by 01, decoding is complete.

6. Compression for 4-bit DIBs

For 4-bit DIBs, the encoding is similar but complicated due to a lack of one-to-one correspondence between bytes and pixels.

If the first byte is nonzero, it's a repetition factor n . The second byte contains 2 pixels, alternating for n pixels.

If the first byte is 00 and the second is 03 or greater, use the number of pixels indicated by the second byte.

7. Handling DIB Image Areas

The last three rows of the table allow DIBs to contain undefined areas, useful for encoding nonrectangular images or creating digital animations.

8. Size Information

When `biCompression` is `BI_RLE4` or `BI_RLE8`, `biSizeImage` indicates the size of DIB pixel data in bytes.

If `biCompression` is `BI_RGB`, `biSizeImage` is usually 0, but it could be set to `biHeight` times the byte length of the row.

9. Note on Compression

Top-down DIBs (those with negative `biHeight` fields) cannot be compressed, as per current documentation.

Understanding Compression Fields in-depth:

- 💻 **`biCompression`**: This field within the `BITMAPINFOHEADER` structure reveals the compression method applied to the DIB's pixel data. It can hold four possible values:
 - 💻 **`BI_RGB`**: No compression (standard storage for 1-bit, 4-bit, 8-bit, and 24-bit DIBs).
 - 💻 **`BI_RLE8`**: Run-length encoding for 8-bit DIBs.
 - 💻 **`BI_RLE4`**: Run-length encoding for 4-bit DIBs.
 - 💻 **`BI_BITFIELDS`**: Color masking for 16-bit and 32-bit DIBs (introduced in Windows 95).
- 💻 **`biSizeImage`**: Specifies the size of compressed pixel data in bytes (when compression is used). It's typically 0 for uncompressed DIBs (`BI_RGB`), but can be set to image height multiplied by row byte length.

Run-Length Encoding (RLE):

Principle: RLE leverages frequent occurrences of identical pixels within DIB images to reduce storage space. It encodes pixel data as repetition factors and pixel values.

Implementation:

8-bit RLE:

- Uses repetition codes (non-zero first byte followed by pixel value) to indicate repeated pixels.
- Uses literal codes (0x00 followed by number of pixels) for non-repeating blocks.
- Employs special codes for end-of-row, end-of-image, and position jumps.

4-bit RLE: Similar to 8-bit RLE, but encodes two pixels per byte, handling alternating pixel patterns.

Key Considerations:

- **Top-Down DIBs:** DIBs with negative biHeight values cannot be compressed.
- **Color Masking (BI_BITFIELDS):** A separate compression technique for 16-bit and 32-bit DIBs, involving masks for red, green, and blue color components.

Additional Insights:

RLE Effectiveness: RLE's compression efficiency depends on the image content. It's most effective with images containing large areas of uniform colors.

Choosing Compression Methods: The appropriate compression method depends on image characteristics and usage requirements. Consider factors such as compression ratio, decoding speed, and compatibility with target applications.

Alternative Compression Techniques: DIBs also support other compression methods like JPEG and PNG, offering different trade-offs between compression ratio and image quality.

Code Example (8-bit RLE Decoding):

```
60 #include <stdio.h>
61
62 typedef unsigned char BYTE; // Assuming BYTE is defined as unsigned char
63
64 void DecodeRLE8(BYTE *compressedData, BYTE *pixelData, int width, int height) {
65     int x = 0, y = 0; // Current position in the image
66
67     while (compressedData < end_of_compressed_data) { // Iterate through compressed data
68         BYTE code = *compressedData++; // Get the RLE code
69
70         if (code == 0) {
71             BYTE num_pixels = *compressedData++; // Get the number of pixels
72
73             if (num_pixels == 0) { // End of row
74                 x = 0;
75                 y++;
76             } else if (num_pixels == 1) { // End of image
77                 break;
78             } else { // Literal pixels
79                 for (int i = 0; i < num_pixels; i++) {
80                     *pixelData++ = *compressedData++;
81                     x++;
82                     if (x >= width) {
83                         x = 0;
84                         y++;
85                     }
86                 }
87             }
88         } else { // Repetition code
89             BYTE pixel = *compressedData++; // Get the pixel value
90             for (int i = 0; i < code; i++) {
91                 *pixelData++ = pixel;
92                 x++;
93                 if (x >= width) {
94                     x = 0;
95                     y++;
96                 }
97             }
98         }
99     }
100 }
101
102 int main() {
103     // Example usage
104     BYTE compressedData[] = { 0x03, 0x45, 0x32, 0x77, 0x02, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06 };
105     int width = 3;
106     int height = 3;
107     BYTE pixelData[width * height];
108
109     // Call the DecodeRLE8 function
110     DecodeRLE8(compressedData, pixelData, width, height);
111
112     // Print the decoded pixel data for demonstration
113     for (int i = 0; i < width * height; i++) {
114         printf("%02X ", pixelData[i]);
115         if ((i + 1) % width == 0) {
116             printf("\n");
117         }
118     }
119
120     return 0;
121 }
```

Pg 585 book.

UNDERSTANDING THE DIB STRUCTURE AND COLOR MASKING:

A Device-Independent Bitmap (DIB) stores image data in a format independent of specific devices or platforms. It allows consistent image representation across various Windows systems. A DIB consists of two main parts:

- **BITMAPINFOHEADER:** This structure contains information about the image, such as its dimensions, color depth, and compression format.
- **Pixel Data:** This section holds the actual image data, represented as pixels of different color depths (e.g., 16-bit, 24-bit, 32-bit).

The biCompression field within the BITMAPINFOHEADER plays a crucial role in determining how pixel data is stored and interpreted. This is where color masking comes into play for certain compression formats.

Color Masking Techniques:

Color masking refers to the process of extracting individual color components (red, green, and blue) from a pixel's encoded value. This technique is particularly relevant for compressed DIB formats like BI_RGB and BI_BITFIELDS.

BI_RGB Compression:

16-bit DIBs:

- Each pixel occupies 2 bytes.
- Color components are interwoven within these bytes, not stored in separate channels.
- Specific bit masks (0x7C00, 0x03E0, 0x001F) are used to isolate the red, green, and blue bit patterns within the pixel value.
- Bitwise AND operations with these masks extract the desired color component bit patterns.
- Right-shifting the extracted bit patterns by specific values (10 bits for red, 5 bits for green, 0 bits for blue) aligns them to occupy the full 8-bit range (0x00 to 0xFF).
- Finally, left-shifting by 3 bits scales the color values to the desired range (0x00 to 0xF8).

32-bit DIBs:

- Each pixel takes up 4 bytes.
- Color components are assigned dedicated bytes: blue in the first byte, green in the second, red in the third, and the fourth byte is always 0.
- Masks (0x00FF0000, 0x0000FF00, 0x000000FF) are applied to extract the respective color components from their designated bytes.
- Bit-shifting these extracted values by specific amounts (16 bits for red, 8 bits for green, 0 bits for blue) aligns them to occupy the full 8-bit range.

BI_BITFIELDS Compression:

This compression format allows for more flexibility in storing color components within the pixel data. Instead of relying on predefined byte allocation like BI_RGB, BI_BITFIELDS uses three 32-bit masks to explicitly define the bit positions for red, green, and blue within the 16-bit or 32-bit pixel value. These masks specify which bits represent each color component and their order within the pixel data.

To extract color values, you apply the corresponding mask to the pixel value using a bitwise AND operation.

This isolates the relevant bit pattern for each color component.

Similar to BI_RGB, specific right-shift values are applied based on the mask definition to align the extracted bit patterns to the full 8-bit range.

Left-shifting these values by 3 bits scales them to the desired range.

Visualizing Color Extraction:

16-bit DIB (BI_RGB):

- Imagine a pixel value of 0xABCD.
- Applying the red mask (0x7C00) and bitwise AND operation extracts the red bit pattern: $(0x7C00 \& 0xABCD) = 0x15$.
- Right-shifting this by 10 bits aligns it to 8 bits: $0x15 \gg 10 = 0x01$.
- Left-shifting by 3 bits scales it to the desired range: $0x01 \ll 3 = 0x08$ (actual red value).
- Similar processes extract green and blue values using their respective masks and shift values.

32-bit DIB (BI_RGB):

Extracting Color Components

1. Pixel Data Layout:

Each pixel is represented by 4 bytes, arranged as follows:

0x00BBGGRR

Blue occupies the first byte (BB).

Green occupies the second byte (GG).

Red occupies the third byte (RR).

The fourth byte is always 0.

2. Applying Masks and Shifting:

To extract the red value:

Apply the red mask: $(0x00FF0000 \& 0x00BBGGRR) = 0x000000RR$

Right-shift by 16 bits to align it to 8 bits: $(0x000000RR) >> 16 = 0xRR$

To extract the green value:

Apply the green mask: $(0x0000FF00 \& 0x00BBGGRR) = 0x0000GG00$

Right-shift by 8 bits to align it to 8 bits: $(0x0000GG00) >> 8 = 0xGG$

To extract the blue value:

Apply the blue mask: $(0x000000FF \& 0x00BBGGRR) = 0x000000BB$

No right-shifting is needed as it's already in the first byte.

```

// Assume 0x00BBGGRR is the 32-bit pixel value

// Extract the red value
BYTE red = (BYTE)((0x00FF0000 & 0x00BBGGRR) >> 16);

// Extract the green value
BYTE green = (BYTE)((0x0000FF00 & 0x00BBGGRR) >> 8);

// Extract the blue value
BYTE blue = (BYTE)(0x000000FF & 0x00BBGGRR);

```

3. Illustrative Example:

Pixel value: 0x0048E058

Red value extraction:

$$(0x00FF0000 \& 0x0048E058) = 0x00004800$$

$$(0x00004800) >> 16 = 0x48$$

Green value extraction:

$$(0x0000FF00 \& 0x0048E058) = 0x0000E000$$

$$(0x0000E000) >> 8 = 0xE0$$

Blue value extraction:

$$(0x000000FF \& 0x0048E058) = 0x00000058$$

```

// Given pixel value: 0x0048E058

// Extract the red value
BYTE red = (BYTE)((0x00FF0000 & 0x0048E058) >> 16);

// Extract the green value
BYTE green = (BYTE)((0x0000FF00 & 0x0048E058) >> 8);

// Extract the blue value
BYTE blue = (BYTE)(0x000000FF & 0x0048E058);

```

Visual Representation:

Key Points:

No left-shifting is required for 32-bit DIBs (BI_RGB) as the color values are already in the desired 0x00 to 0xFF range.

The order of colors in 32-bit DIBs differs from the COLORREF value used in Windows GDI functions, where red is the least significant byte.

Color masking techniques:

```
155 #include <stdio.h>
156
157 typedef unsigned int DWORD;
158
159 // Function to calculate right-shift value from a color mask
160 int MaskToRShift(DWORD dwMask) {
161     int iShift = 0;
162     if (dwMask == 0)
163         return 0;
164     while (!(dwMask & 1)) {
165         iShift++;
166         dwMask >= 1;
167     }
168     return iShift;
169 }
170
171 // Function to calculate left-shift value from a color mask
172 int MaskToLShift(DWORD dwMask) {
173     int iShift = 0;
174     if (dwMask == 0)
175         return 0;
176     while (!(dwMask & 1))
177         dwMask >= 1;
178     while (dwMask & 1) {
179         iShift++;
180         dwMask >= 1;
181     }
182     return 8 - iShift;
183 }
184
185 int main() {
186     // Given color masks for a 16-bit DIB
187     DWORD dwMask[3] = {0x00007C00, 0x000003E0, 0x0000001F};
188
189     // Calculate right-shift values
190     int iRShift[3];
191     iRShift[0] = MaskToRShift(dwMask[0]);
192     iRShift[1] = MaskToRShift(dwMask[1]);
193     iRShift[2] = MaskToRShift(dwMask[2]);
194
195     // Calculate left-shift values
196     int iLShift[3];
197     iLShift[0] = MaskToLShift(dwMask[0]);
198     iLShift[1] = MaskToLShift(dwMask[1]);
199     iLShift[2] = MaskToLShift(dwMask[2]);
200
201     // Example usage with a 16-bit pixel value
202     unsigned short wPixel = 0x0048E058;
203
204     // Extract color values
205     unsigned char Red = (unsigned char)((dwMask[0] & wPixel) >> iRShift[0]) << iLShift[0];
206     unsigned char Green = (unsigned char)((dwMask[1] & wPixel) >> iRShift[1]) << iLShift[1];
207     unsigned char Blue = (unsigned char)((dwMask[2] & wPixel) >> iRShift[2]) << iLShift[2];
208
209     // Display the extracted values
210     printf("Red: %u\nGreen: %u\nBlue: %u\n", Red, Green, Blue);
211
212     return 0;
213 }
```

Header and Typedef:

Importing Standard Library: The code initiates by including stdio.h, a fundamental C library that offers essential input/output functions like printf for formatted console output.

Defining a Convenient Alias: It introduces DWORD as a type alias for unsigned int, a common choice for representing pixel data due to its ability to store a wide range of integer values without a sign bit.

MaskToRShift Function:

Purpose: This function's primary role is to determine the appropriate right-shift value needed to accurately extract a specific color component (red, green, or blue) from a given pixel value.

Handling Zero Masks: It first checks if the input mask is 0, in which case it returns 0, indicating no shifting is required.

Iteratively Locating Significant Bits: If the mask is not 0, it enters a loop that repeatedly shifts the mask to the right using the right-shift operator ($>>$). With each shift, it also increments a counter (iShift). This process continues until the first 1 bit emerges within the mask, signifying the starting position of the relevant color component.

Returning the Shift Count: The function ultimately returns the calculated iShift value, which represents the number of positions by which the pixel value needs to be right-shifted to align the desired color component correctly.

MaskToLShift Function:

Purpose: Analogous to MaskToRShift, this function focuses on calculating the necessary left-shift value for proper color component placement.

Zero Mask Handling: It similarly begins by checking for a 0 mask and returning 0 if found.

Two-Phase Shifting and Counting: It employs a two-step approach:

Locating the First 1 Bit: It shifts the mask to the right until the first 1 bit appears, keeping track of the shifts.

Counting Remaining 1 Bits: It continues shifting and counting until all 1 bits within the mask are cleared.

Calculating Final Shift Value: The final value returned is 8 minus the total number of shifts counted, representing the amount of left-shifting required to position the color component accurately.

Main Function and Pixel Value Extraction:

Storing Color Masks: The code defines an array named dwMask containing three DWORD elements. These elements represent the bit masks for extracting red, green, and blue components from a 16-bit pixel value. Each mask specifies the relevant bit positions for the corresponding color within the overall pixel data.

Precalculated Shift Arrays: Two arrays, iRShift and iLShift, are declared to hold the pre-computed right-shift and left-shift values for each color mask. These values, obtained from calling the respective masking functions earlier, facilitate efficient extraction without repetitive calculations.

Extracting Color Components:

Sample Pixel: An example 16-bit pixel value, wPixel, is assigned. This variable represents the raw data containing the encoded color information for a specific pixel.

Masking and Shifting for Individual Colors: Each color component is extracted using a similar approach:

Masking: The pixel value is bitwise ANDed with the corresponding color mask from dwMask. This isolates the relevant bits for the specific color within the pixel data.

Right-Shifting: The masked value is then right-shifted by the corresponding iRShift value from the precomputed array. This aligns the extracted color bits to the rightmost positions.

Left-Shifting: Finally, the shifted value is left-shifted by the corresponding iLShift value. This places the extracted color data within the appropriate range for an 8-bit unsigned character (unsigned char).

Displaying Extracted Values:

The extracted red, green, and blue components are stored in separate unsigned char variables (Red, Green, and Blue). These variables represent the individual color intensities scaled to the 0-255 range commonly used for image representation.

Finally, the printf function displays the extracted color values as decimals.

Overall Functionality:

This C code serves as a tool for extracting individual color components (red, green, and blue) from a 16-bit pixel value within a Digital Image Bitmap (DIB) format. It utilizes pre-defined bit masks and calculated shift values to efficiently isolate and manipulate the relevant color data, enabling further processing or analysis of individual pixel colors.

The Role of Color Masks:

In Device-Independent Bitmaps (DIBs), color masks play a crucial role in determining how color information is stored and extracted within pixel data, especially for compressed formats like BI_BITFIELDS.

These masks act as blueprints, defining the precise arrangement of red, green, and blue color components within each pixel value.

Understanding BI_BITFIELDS Compression:

When the biCompression field in the DIB header is set to BI_BITFIELDS, it grants flexibility in color component storage.

Instead of relying on fixed patterns like in BI_RGB, BI_BITFIELDS utilizes three 32-bit masks to explicitly specify the bit positions for red, green, and blue within each pixel.

This allows for customized color layouts, potentially optimizing color representation for specific content or scenarios.

Decoding Colors Using Masks:

Retrieving the Masks:

The first step involves extracting the red, green, and blue masks from the DIB header, as they hold the key to deciphering pixel color values.

Calculating Shift Values:

To accurately extract color components, precise bit shifting is necessary. The MaskToRShift and MaskToLShift functions (provided previously) play a crucial role in determining the appropriate right-shift and left-shift values based on each mask.

Applying Masks and Shifting:

The actual extraction involves applying the masks to the pixel value using bitwise AND operations. This isolates the relevant bit patterns for each color component.

The masked values are then shifted right and left according to the calculated shift values, aligning them to their correct positions within the 8-bit color range.

Illustrative Example (16-bit DIB with BI_BITFIELDS):

```
// Assuming dwMask[0] = 0x0000F800 (Red), dwMask[1] = 0x000007E0 (Green), dwMask[2] = 0x0000001F (Blue)

// Calculate shift values:
int iRShift[3], iLShift[3];
// ... (using MaskToRShift and MaskToLShift functions)

// Extract colors from a pixel value wPixel:
Red = ((dwMask[0] & wPixel) >> iRShift[0]) << iLShift[0];
Green = ((dwMask[1] & wPixel) >> iRShift[1]) << iLShift[1];
Blue = ((dwMask[2] & wPixel) >> iRShift[2]) << iLShift[2];
```

Key Considerations:

When working with 32-bit DIBs (Device-Independent Bitmaps) using the BI_BITFIELDS compression format, there are some similarities and differences compared to 16-bit DIBs. The overall process follows a similar pattern, but there are a few key points to consider.

Larger Masks: In 32-bit DIBs with BI_BITFIELDS, the masks used for extracting color channels can be larger than 0x0000FFFF. This allows for a wider range of color values and a more extensive color gamut when needed.



Extended Color Range: Both 16-bit and 32-bit DIBs with BI_BITFIELDS can have color values that exceed 255. This means that they offer a broader range of colors and can represent more vivid and nuanced shades when required.

Windows 95/98 Mask Restrictions:

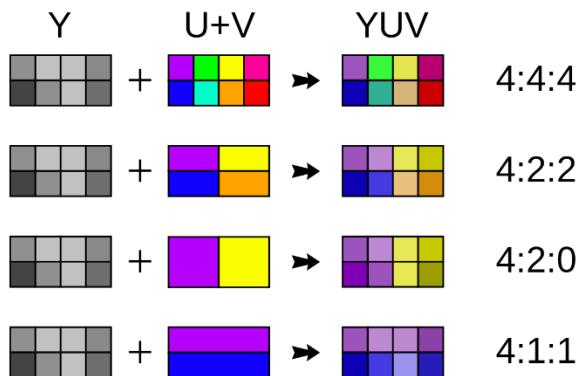
It's important to be aware of the mask limitations imposed by Windows 95 and Windows 98 for compatibility purposes. The allowable mask values in these operating systems are specified in a table that should be referenced for accurate implementation.

In more [recent versions of Windows](#), such as Windows 10 and Windows 11, the mask restrictions for DIBs with the BI_BITFIELDS format are not as strict as they were in Windows 95 and Windows 98. Developers have more flexibility in choosing the mask values to suit their needs. However, it is still important to consider [backward compatibility](#) with older systems if required.

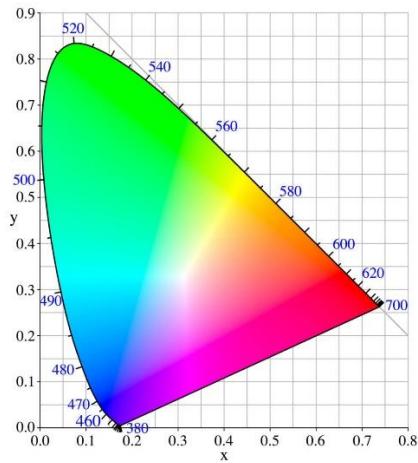
To determine the appropriate mask values for Windows 95 and Windows 98 compatibility, developers should [refer to the documentation provided by Microsoft](#). The documentation specifies the allowable mask values for each color channel (red, green, and blue) in the BI_BITFIELDS format. By adhering to these restrictions, developers can ensure that their DIBs will be compatible with Windows 95 and Windows 98 systems.

Custom color layouts in DIBs with BI_BITFIELDS can optimize color representation in various ways. Here are a few examples:

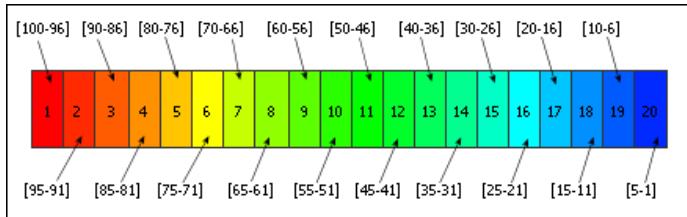
Subsampling: By assigning higher bit values to the color channels that contribute more to the overall appearance of the image, developers can achieve a more visually accurate representation while reducing the bit depth or file size for less significant color channels.



Color Space Optimization: Custom mask values can be chosen to match specific color spaces, such as sRGB or Adobe RGB. This allows for more accurate color reproduction and ensures compatibility with color-managed systems.



Channel Priority: In certain applications, certain color channels may be more important than others. By assigning larger masks to the critical channels, developers can prioritize their preservation during color transformations or manipulations.



Advanced color manipulation and image processing tasks can be achieved using color masking techniques. Here are a few examples:

Color Correction: By applying different masks and modifying the color channel values, developers can perform color correction operations, such as white balance adjustments, color cast removal, or gamma correction.



Color Effects: Custom masks can be used to selectively apply color effects to specific regions of an image while leaving other areas untouched. This allows for targeted color grading, sepia or grayscale conversions, or creative color manipulations.



Image Segmentation: Color masks can be utilized to segment an image based on specific color ranges or color patterns. This enables tasks like object detection, background removal, or region-based processing.



Color Quantization: By manipulating the masks and reducing the number of available color values in specific channels, developers can perform color quantization to reduce the bit depth or create artistic effects, such as posterization or indexed color rendering.



Beyond the Basics:

In addition to the fundamental concepts, there are further possibilities and considerations when working with masks and BI_BITFIELDS in DIBs.

Mask Customization: The BI_BITFIELDS format allows developers to create custom color layouts tailored to specific image data or application requirements. This customization can optimize color representation and potentially reduce file sizes by efficiently encoding the color information.



Advanced Color Handling: Understanding color masking techniques opens the door to more sophisticated color manipulation and image processing tasks. It provides the ability to exert greater control over the color information within DIBs, enabling advanced color adjustments and transformations.



Bit Depth	Red Mask	Green Mask	Blue Mask	Shorthand
16-Bit DIB (5-5-5)	0x00007C00	0x000003E0	0x0000001F	5-5-5
16-Bit DIB (5-6-5)	0x0000F800	0x000007E0	0x0000001F	5-6-5
32-Bit DIB (8-8-8)	0x00FF0000	0x0000FF00	0x000000FF	8-8-8

Explanation of Table Elements:

Bit Depth: Refers to the number of bits used to represent each pixel in a DIB image.

Color Masks: These hexadecimal values act as filters to isolate specific color components (red, green, or blue) within a pixel value. Each mask contains bits set to 1 in the positions corresponding to the relevant color's bits within the pixel data.

Shorthand: A concise way to express the number of bits allocated to each color component in the pixel format (e.g., 5-6-5 indicates 5 bits for red, 6 bits for green, and 5 bits for blue).

Key Points:

The first two rows represent common 16-bit DIB pixel formats, while the last row denotes the standard 32-bit DIB format.

The specific color masks employed depend on the DIB's compression type and pixel format.

The shorthand notation offers a quick way to grasp the color depth distribution within a pixel format.

BITMAPV4HEADER: EXPANDING COLOR HORIZONS

The BITMAPV4HEADER structure is an extended header introduced in Windows 95 and also supported by Windows NT 4.0. It extends the standard BITMAPINFOHEADER structure to support additional features, particularly related to image color-matching technology. Below is a summary of the fields in the BITMAPV4HEADER structure:

Structure Breakdown:

```
typedef struct {
    DWORD bV4Size;           // size of the structure = 120
    LONG bV4Width;           // width of the image in pixels
    LONG bV4Height;          // height of the image in pixels
    WORD bV4Planes;          // = 1
    WORD bV4BitCount;         // bits per pixel (1, 4, 8, 16, 24, or 32)
    DWORD bV4Compression;    // compression code
    DWORD bV4SizeImage;      // number of bytes in the image
    LONG bV4XPelsPerMeter;   // horizontal resolution
    LONG bV4YPelsPerMeter;   // vertical resolution
    DWORD bV4ClrUsed;        // number of colors used
    DWORD bV4ClrImportant;   // number of important colors
    DWORD bV4RedMask;         // Red color mask
    DWORD bV4GreenMask;       // Green color mask
    DWORD bV4BlueMask;        // Blue color mask
    DWORD bV4AlphaMask;       // Alpha mask
    DWORD bV4CSType;          // color space type
    CIEXYZTRIPLE bV4Endpoints; // XYZ values
    DWORD bV4GammaRed;        // Red gamma value
    DWORD bV4GammaGreen;      // Green gamma value
    DWORD bV4GammaBlue;        // Blue gamma value
} BITMAPV4HEADER, * PBITMAPV4HEADER;
```

Structure Size: The bV4Size field must be set to 120 to ensure correct structure size.

Color Masks: The bV4RedMask, bV4GreenMask, and bV4BlueMask fields specify the arrangement of color components within pixels for 16-bit and 32-bit DIBs with BI_BITFIELDS compression.

Image Color Management: The bV4CSType, bV4Endpoints, and gamma fields provide information for color management systems.

CIEXYZTRIPLE: The bV4Endpoints field holds three CIE XYZ values, representing color in a device-independent color space.

Gamma Values: The bV4GammaRed, bV4GammaGreen, and bV4GammaBlue fields influence image brightness and contrast.

Familiar Foundation: The first 11 fields mirror those found in BITMAPINFOHEADER, detailing image dimensions, compression method, color depth, resolution, and color palette information.

Unveiling New Dimensions: The structure's true prowess emerges in its subsequent fields, each carefully crafted to orchestrate a symphony of color precision:

Color Masks: These fields are like filters that let us work with specific colors within each pixel. They're especially helpful for 16-bit and 32-bit images that use a special kind of compression called BI_BITFIELDS.

Alpha Mask: This field is like a placeholder for future features related to transparency. It's not active yet, but it hints at exciting visual effects that might be possible in the future.

Color Space Type: This field tells us exactly which color system the image uses. It's like a passport that ensures the image's colors look the same no matter what device you view it on. One common color system it can identify is called CIE XYZ.

XYZ Values: These are like coordinates that map out the image's colors within a universal color space called CIE XYZ. This space is like a common language that different devices can understand, making sure colors look consistent.

Gamma Values: These are like fine-tuning knobs for brightness. They make sure that red, green, and blue colors look balanced and accurate, even on different types of screens.

Navigating Color Consistency: The ICM Approach

The RGB Challenge: Despite its widespread use, the RGB color model has a downside. It relies on the unique traits of specific devices, causing color differences between monitors, printers, and other gadgets.

ICM Steps In: Meet Image Color Management (ICM) – a superhero in the color world. Its main goal? Creating a shared understanding of color. ICM strives to make colors look consistent across all kinds of devices, making sure our digital images stay true to their visual charm.

CIE Colorimetry, The Science Behind It: Imagine a color system that doesn't depend on devices. That's what the International Commission on Illumination gave us in 1931. Their colorimetry system uses three special functions (x, y, and z) to measure color based on its unique light distribution. This forms a solid base for keeping colors consistent, no matter the device.

In simpler terms, ICM is like a guardian making sure colors play nicely across our screens and prints, thanks to a timeless color science foundation.

Key Takeaways:

Unlocking Color Power with BITMAPV4HEADER:

Empowering Developers: The enhanced structure of BITMAPV4HEADER hands developers more control over color management. It becomes a tool for crafting digital experiences that are not just visually consistent but also vibrant.

Tackling Color Challenges: Dealing with colors that change with each device? That's where ICM (Image Color Management) steps in. It becomes crucial in preserving the true visual essence across various mediums, overcoming the hurdles of device-dependent color representation.

CIE Colorimetry, The Color Scientist's Friend: In the world of colors, CIE colorimetry stands out as a scientific hero. It provides a structured framework to understand and measure color independently of devices. Essentially, it's the common language ensuring colors communicate consistently.

Beyond the Basics:

ICM in Action: The last four fields in BITMAPV4HEADER work together like a conductor directing a complex dance of ICM. This dance ensures that colors look right, no matter what kind of device they're on.

XYZ Color Space: Think of the XYZ color space as a carefully designed area where colors match how we see them. It's like a solid foundation for colors to exist independently of devices, making sure they always look the same.

Gamma Values at Play: Gamma values are like the conductors of visual harmony. They adjust how colors behave, adapting to the unique traits of different displays. The result? A visual experience where colors work together seamlessly.

Discovering ICM's Secrets: The details of ICM are like a hidden treasure waiting to be found. Dive deeper, and you'll uncover a future where colors always stay true to how they're meant to look, creating a digital landscape that's faithful and vibrant.

$$X\!=\!\sum_{\lambda=380}^{780} S(\lambda) \overline{x}(\lambda)$$

$$Y\!=\!\sum_{\lambda=380}^{780} S(\lambda) \overline{y}(\lambda)$$

$$Z\!=\!\sum_{\lambda=380}^{780} S(\lambda) \overline{z}(\lambda)$$

Breakdown of X Equation:

This equation calculates the X value, which is one of the three coordinates used to describe colors in the CIE XYZ color space (the other two are Y and Z). It works like this:

- Σ means "sum up" or "add together".
- λ represents different wavelengths of light, like the colors of a rainbow.
- $S(\lambda)$ is a special function that measures how much of each wavelength is present in a particular color.
- $\bar{x}(\lambda)$ is one of the CIE color-matching functions, which acts like a filter that tells us how much humans perceive each wavelength.
- $d\lambda$ is a tiny slice of the visible light spectrum, like a super-thin piece of rainbow.

So, the equation is basically adding up all the different wavelengths of light in a color, weighted by how much humans perceive them, to get the X value.

The equations for Y and Z follow a similar pattern, but they use different color-matching functions ($\bar{y}(\lambda)$ and $\bar{z}(\lambda)$).

These equations work together to create a complete, device-independent description of a color within the CIE XYZ color space.

Think of it like baking a cake:

If RGB colors are like the ingredients (red, green, and blue), ICM is like the recipe that ensures the cake comes out perfectly no matter what oven you use.

The CIE colorimetry equations are like the measuring cups and spoons that help you get the proportions just right.

Breakdown of Y equation:

- Σ : Represents a sum, indicating that we'll be adding up values.
- λ : Stands for wavelength, ranging from 380 nanometers (violet) to 780 nanometers (red) to encompass the visible light spectrum.
- $S(\lambda)$: Represents the spectral power distribution of the color, quantifying the intensity of each wavelength present.
- $\bar{y}(\lambda)$: Is the CIE color-matching function specifically for the Y value, capturing how sensitive humans are to different wavelengths in terms of perceived brightness.
- $d\lambda$: Denotes a small interval of wavelength, like a tiny slice of the spectrum.

In simpler terms:

The equation adds up the products of the spectral power distribution (how much of each wavelength is present) and the Y color-matching function (how sensitive humans are to those wavelengths) for all wavelengths in the visible spectrum. It essentially measures the overall brightness of a color as perceived by humans.

Relationship to X and Z:

The equations for X and Z follow the same pattern, but each uses its respective color-matching function ($\bar{x}(\lambda)$ for X and $\bar{z}(\lambda)$ for Z).

Together, X, Y, and Z values create a comprehensive, device-independent representation of color within the CIE XYZ color space.

Breakdown of Z equation:

- Σ : Symbol for summation, indicating the addition of values across a range.
- λ : Represents wavelength, varying from 380 nanometers (violet) to 780 nanometers (red) to cover the visible light spectrum.
- $S(\lambda)$: The spectral power distribution of the color, specifying the intensity of each wavelength present.
- $\bar{z}(\lambda)$: The CIE color-matching function specifically for the Z value, modeling human visual sensitivity to different wavelengths in a distinct way from X and Y.
- $d\lambda$: A small interval of wavelength, like a tiny slice of the spectrum.

Interpretation:

The equation calculates the Z value by summing up the products of the spectral power distribution (how much of each wavelength is present) and the Z color-matching function (how sensitive humans are to those wavelengths) for all wavelengths in the visible spectrum.

While X and Y relate primarily to perceived brightness, Z captures a different dimension of color perception, often associated with blue-yellow color differences.

Significance:

Together with X and Y, the Z value forms a complete, three-dimensional representation of color within the CIE XYZ color space.

This device-independent color model enables accurate color communication and management across various devices and mediums.

IEXYZTRIPLE AND CIEXYZ: BUILDING BLOCKS FOR DEVICE-INDEPENDENT COLOR

CIEXYZTRIPLE: A structure designed to hold three CIEXYZ structures, each representing a primary color (red, green, blue) within the CIE XYZ color space. Acts as a bridge between the digital world of image data and the realm of human color perception.

CIEXYZ: Encapsulates three FXPT2DOT30 values, meticulously defining a color's position within the CIE XYZ coordinate system.

FXPT2DOT30: A fixed-point numerical format providing a balance between precision and efficient storage.

BITMAPV4HEADER: Embracing Color Management

bV4CSType: Essential for specifying the color space model used in the image.

Setting it to `LCS_CALIBRATED_RGB` (0) signals a commitment to accurate color representation, independent of device-specific variations.

bV4Endpoints: A trio of X, Y, Z values firmly anchoring the image's primary colors (red, green, blue) within the CIE XYZ color space.

Crucial for establishing a common ground for color interpretation across diverse devices.

Gamma Correction: Taming Nonlinearity for Accurate Visual Harmony

Gamma (γ): A numerical factor that addresses the nonlinear relationship between digital color levels (0-255) and the actual intensity of light emitted by display devices.

Arises due to the intricate physics of electron guns within cathode ray tubes (CRTs) and characteristics of other display technologies.

$I = (V + e)^\gamma$: The mathematical equation governing this relationship, where I represents pixel intensity, V is voltage, e is black level, and γ is the gamma value.

Video cameras often incorporate gamma correction (typically with a γ of 0.45) to counteract the inherent nonlinearity of display devices and ensure a visually consistent experience.

```

// Definition of CIEXYZTRIPLE structure
typedef struct tagCIEXYZTRIPLE {
    CIEXYZ ciexyzRed; // Holds color info for red
    CIEXYZ ciexyzGreen; // Holds color info for green
    CIEXYZ ciexyzBlue; // Holds color info for blue
} CIEXYZTRIPLE, * LPCIEXYZTRIPLE;

// Definition of CIEXYZ structure
typedef struct tagCIEXYZ {
    FXPT2DOT30 ciexyzX; // X value for color (like a coordinate)
    FXPT2DOT30 ciexyzY; // Y value for color (like another coordinate)
    FXPT2DOT30 ciexyzZ; // Z value for color (yet another coordinate)
} CIEXYZ, * LPCIEXYZ;

```

Explanation of Fields and Values:

- Big X, Big Y, and Big Z represent color-matching functions, resembling the human eye's response to visible light (bell curve from 380 nm to 780 nm).
- Y is known as CIE Luminance, indicating overall light intensity.
- For BITMAPV5HEADER, set bV4CSType to LCS_CALIBRATED_RGB (0) and provide valid values for the next four fields.
- CIEXYZTRIPLE holds three sets of CIEXYZ values for primary colors.
- FXPT2DOT30 values are fixed-point with a 2-bit integer part and a 30-bit fractional part.
- The bV4Endpoints field specifies X, Y, Z values for RGB colors (255, 0, 0), (0, 255, 0), and (0, 0, 255).
- These values denote the device-independent meaning of RGB colors in a DIB.
- Remaining three fields in BITMAPV4HEADER deal with "gamma," addressing nonlinearity in color level specifications.
- Gamma correction compensates for nonlinearity, with the equation $I = (V + e)^g$.
- Video cameras traditionally use a gamma correction of 0.45, implying a video display gamma of about 2.2.
- This correction ensures a visually consistent experience despite nonlinearity in display devices.

Summary:

The CIE XYZ color space provides a device-independent foundation for color representation, enabling accurate color communication across diverse devices.

The BITMAPV4HEADER structure offers a mechanism for incorporating color space information within image files, promoting consistent color reproduction.

Gamma correction plays a vital role in harmonizing the nonlinearities of display devices and image capture systems, ensuring accurate and visually pleasing color experiences.

The bV4Endpoints field provides X, Y, Z values indicating the device-independent meaning of RGB colors.

The remaining three BITMAPV4HEADER fields relate to "gamma," addressing nonlinearity in color level specifications and compensating for it.

Video cameras traditionally include gamma correction to align with display characteristics, ensuring a balanced visual experience.

NONLINEAR RESPONSE AND HUMAN PERCEPTION:

The nonlinear response of video monitors aligns favorably with human visual perception, which is also nonlinear. The CIE defines two measures: CIE Luminance (Y) and Lightness (L*). Lightness, calculated from Y, incorporates a linear segment and a cube root formula:

$$L^* = (Y / Y_n)^{(1/3)} * 100$$

Here, Yn is the white level. L* ranges from 0 to 100, with each integral increment representing the smallest perceivable change in lightness.

Coding Light Intensities Based on Perceptual Lightness:

To enhance coding efficiency and reduce noise, it's preferable to represent light intensities using perceptual lightness (L^*) rather than linear luminance. The process involves linearly converting a pixel value (P) to a normalized voltage level:

$$\text{Intensity} = (P / 255)^g$$

Assuming the monitor's black level is 0, the cube root of this intensity gives human perceptual lightness (L^*).

Gamma Correction in BITMAPV4HEADER:

The last three fields of BITMAPV4HEADER allow programs to indicate a gamma value assumed for pixel values. These values are interpreted as 16-bit integer and 16-bit fractional values.

For instance, 0x10000 represents 1.0. If the DIB is captured from a real-world image, the implied gamma value is likely 2.2 (encoded as 0x23333).

If generated algorithmically, the program would convert linear luminances to CIE lightness using a power function, with the inverse of the exponent being the encoded gamma in the DIB.

This process ensures that the gamma correction aligns with human perceptual lightness, enhancing the visual experience.

Let's continue in depth....

The nonlinear response of video monitors aligns well with human perception because human response to light is also nonlinear. As mentioned earlier, the CIE defines a Lightness value (L^*) that approximates human perception. The calculation of L^* from linear luminance (Y) involves two formulas.

The first formula represents a small linear segment, while the second formula indicates that human perception of lightness is related to the cube root of the linear luminance. L^* values range from 0 to 100, with each integral increment representing the smallest perceivable change in lightness.

$$L^* = \begin{cases} 903.3 \frac{Y}{Y_s} & \frac{Y}{Y_s} \leq 0.008856 \\ 116 \left(\frac{Y}{Y_s} \right)^{\frac{1}{3}} - 16 & 0.008856 < \frac{Y}{Y_s} \end{cases}$$

It is advantageous to code light intensities based on perceptual lightness rather than linear luminance. This approach keeps the number of bits required at a reasonable level and reduces noise in analog circuitry.

Let's walk through the entire process. The pixel value (P) ranges from 0 to 255. It is linearly converted to a voltage level normalized between 0.0 and 1.0. Assuming the monitor's black level is set to 0, the intensity of the pixel is determined.

$$I = V = \left(\frac{P}{255} \right)^g$$

where g is typically around 2.5. Human perception of lightness (L^*) is derived from the cube root of the intensity and ranges from 0 to 100. This approximation allows for a perceptually uniform representation of lightness, ensuring that subtle changes in intensity are perceived accurately by humans.

$$L^* = \frac{100}{100} \left(\frac{P}{255} \right)^{\frac{1}{3}}$$

The exponent for the cube root relationship between pixel values and CIE lightness is typically around 0.85. If the exponent were 1, the CIE lightness would perfectly match the pixel values. While we don't have an exact match, the cube root approximation brings us much closer than if the pixel values indicated linear luminance.

The last three fields of the BITMAPV4HEADER provide a way for programs creating a Device-Independent Bitmap (DIB) to specify a gamma value associated with the pixel values. These gamma values are represented as 16-bit integer and 16-bit fractional values. For example, 0x10000 represents a gamma value of 1.0.

If the DIB is generated by capturing a real-world image, the implied gamma value is likely set by the capture hardware and is commonly 2.2 (encoded as 0x23333). However, if the DIB is algorithmically generated by a program, the program would convert linear luminance values to CIE lightness values using a power function, where the inverse of the exponent corresponds to the gamma value encoded in the DIB.

Let's explain the formulas...

$$L^* = \begin{cases} 903.3 \frac{Y}{Y_s} & \frac{Y}{Y_s} \leq 0.008856 \\ 116 \left(\frac{Y}{Y_s} \right)^{\frac{1}{3}} - 16 & 0.008856 < \frac{Y}{Y_s} \end{cases}$$

L* Represents CIE Lightness, a nonlinear measure of perceived lightness that aligns more closely with human visual perception than linear luminance.

Y: The linear luminance value of a particular color.

Y_n: A reference white level, used as a normalization factor to compare lightness across different lighting conditions.

(Y / Y_n): Divides the luminance value by the reference white level, essentially expressing it as a proportion of the maximum perceived lightness.

(^{1/3}): The cube root function, which accounts for the nonlinear relationship between luminance and perceived lightness.

1/3: A scaling factor to adjust the range of L* values, typically from 0 to 100.

Key Points:

The formula calculates lightness based on luminance and a reference white level.

The cube root function reflects the nonlinear nature of human lightness perception.

The formula is designed to produce values that correspond to perceived lightness, rather than simply measuring raw light intensity.

BITMAPV4HEADER: The image likely mentions this structure as a way to store gamma information within image files, which can help ensure accurate color reproduction across different displays.

$$I = V = \left(\frac{P}{255} \right)^\gamma$$

Components:

I: Intensity of the pixel, representing how bright it appears on the screen.

V: Voltage level applied to the monitor, controlling the intensity of the corresponding pixel.

P: Pixel value in the image data, ranging from 0 to 255, with 0 being black and 255 being white.

255: Normalization factor, ensuring the equation works on a common scale.

NUMBER(power/exponent): Unknown exponent value, determining the nonlinear relationship between pixel value and voltage.

Possible Values for the Exponent:

Gamma (γ): The most likely candidate for the exponent is gamma, a value typically around 2.2 that accounts for the nonlinear response of CRT monitors. This means that increasing the pixel value by a small amount won't produce a proportionally small increase in brightness, but rather a more significant jump due to the nonlinearity.

Other Exponents: Depending on the specific context and technology involved, the exponent could also take other values. For example, some image processing algorithms might use different exponents for specific transformations.

Understanding the Formula:

This formula essentially describes how the digital information stored in the image (pixel value) is translated into an analog signal (voltage) that controls the intensity of the displayed pixel. The exponent plays a crucial role in shaping this relationship, introducing a non-linearity that better reflects how humans perceive brightness changes.

$$L^* = \frac{100}{255} \left(\frac{P}{255} \right)^{\frac{1}{3}}$$

L*: Represents CIE Lightness, a nonlinear measure of perceived lightness that more closely aligns with human visual perception than linear luminance.

P: Pixel value in the image data, ranging from 0 to 255.

255: Normalization factor to scale the pixel value to a 0-1 range.

EXP: The exponent, which is either 1/3 or $\gamma/3$ (gamma divided by 3), remains uncertain due to the image's ambiguity.

Possible Exponent Values:

1/3: This exponent suggests a direct cube root relationship between pixel value and lightness, often used in basic lightness calculations.

$\gamma/3$: Incorporating gamma (typically around 2.2) into the exponent would account for the nonlinear response of displays and better align with human perception. The specific value of $\gamma/3$ would depend on the display's gamma characteristic.

Key Points:

The formula calculates perceived lightness (L^*) from pixel values (P).

The exponent determines the specific nature of the relationship, either a simple cube root or a gamma-adjusted one.

Determining the exact exponent requires either clarifying the image content or consulting the relevant context in the book.

Additional Considerations:

Base 100: While the formula itself doesn't explicitly involve a base 100, it's worth noting that CIE Lightness values often range from 0 to 100, representing the perceived lightness scale.

```

typedef struct
{
    DWORD bV5Size;          // size of the structure = 120
    LONG bV5Width;         // width of the image in pixels
    LONG bV5Height;        // height of the image in pixels
    WORD bV5Planes;        // = 1
    WORD bV5BitCount;      // bits per pixel (1, 4, 8, 16, 24, or 32)
    DWORD bV5Compression;  // compression code
    DWORD bV5SizeImage;    // number of bytes in image
    LONG bV5XPelsPerMeter; // horizontal resolution
    LONG bV5YPelsPerMeter; // vertical resolution
    DWORD bV5ClrUsed;      // number of colors used
    DWORD bV5ClrImportant; // number of important colors
    DWORD bV5RedMask;      // Red color mask
    DWORD bV5GreenMask;    // Green color mask
    DWORD bV5BlueMask;     // Blue color mask
    DWORD bV5AlphaMask;    // Alpha mask
    DWORD bV5CSType;       // color space type
    CIEXYZTRIPLE bV5Endpoints; // XYZ values
    DWORD bV5GammaRed;     // Red gamma value
    DWORD bV5GammaGreen;   // Green gamma value
    DWORD bV5GammaBlue;    // Blue gamma value
    DWORD bV5Intent;       // rendering intent
    DWORD bV5ProfileData;  // profile data or filename
    DWORD bV5ProfileSize;  // size of embedded data or filename
    DWORD bV5Reserved;
} BITMAPV5HEADER, *PBITMAPV5HEADER;

```

Structure Overview:

Purpose: Expands on previous DIB headers to enhance color management capabilities.

Size: 120 bytes.

Fields:

- Core fields inherited from earlier headers (width, height, bit depth, compression, etc.).
- Four new fields specifically for color management:
- bV5CSType: Identifies the color space used in the DIB.
- bV5Endpoints: Holds XYZ values defining color space endpoints (if applicable).
- bV5GammaRed, bV5GammaGreen, bV5GammaBlue: Gamma values for red, green, and blue channels (if applicable).
- bV5ProfileData, bV5ProfileSize: Handle ICC profile information.

Color Space Handling:

bV5CSType Values:

- **LCS_CALIBRATED_RGB:** Compatible with BITMAPV4HEADER; requires valid bV5Endpoints and gamma fields.
- **LCS_sRGB:** Standard RGB color space for relative device independence, often used on the Internet.
- **LCS_WINDOWS_COLOR_SPACE:** Relies on default Windows color space determined by API calls.
- **PROFILE_EMBEDDED:** DIB contains an embedded ICC profile.
- **PROFILE_LINKED:** DIB links to an external ICC profile file.

ICC Profiles:

- Define device-independent color specifications based on CIE XYZ values.
- Used to manage color consistency across different devices.
- Can be embedded directly within the DIB or linked to an external file.

Key Points:

BITMAPV5HEADER offers advanced color management features.

The bV5CSType field dictates how color information is interpreted.

ICC profiles play a crucial role in ensuring accurate color representation across systems.

Development of BITMAPV5HEADER reflects growing importance of device-independent color management.

ICC profiles are essential for achieving consistent color across diverse devices and platforms.

Understanding these concepts is vital for working effectively with digital images in modern graphics applications.

Code illustration (Dibheads.c)

The provided code is a Windows program written in C that displays information about the header structures of a Device Independent Bitmap (DIB) file. This program, named DIBHEADS, allows users to open a DIB file, and it then extracts and presents various details from its header structures.

Program Structure:

The program is structured using the WinMain function as the entry point. It registers a window class, creates a main window, and enters the message loop to handle user interactions. The WndProc function serves as the window procedure, responding to various messages.

DisplayDibHeaders Function:

The DisplayDibHeaders function is responsible for extracting and displaying information about the DIB file. It takes the window handle (hwnd) and the filename of the DIB as parameters.

File Opening:

The function begins by attempting to open the specified DIB file using the CreateFile function. If successful, it proceeds to read the file into memory.

File Size Check:

It checks the size of the file to ensure it's within a manageable range, and allocates memory accordingly.

Reading File:

The function reads the contents of the file using ReadFile and subsequently closes the file handle.

BITMAPFILEHEADER:

It displays information from the BITMAPFILEHEADER structure, such as the file type, size, reserved fields, and the offset to the bitmap data.

Determining Header Type:

The function then determines the type of header structure (CORE, INFO, V4, or V5) by inspecting the bV5Size field.

Displaying Header Information:

Depending on the header type, it displays relevant information. For BITMAPCOREHEADER and BITMAPINFOHEADER, it shows width, height, planes, bit count, compression, image size, resolution, and color-related information. For BITMAPV4HEADER and BITMAPV5HEADER, additional fields like color masks, color space information, gamma values, rendering intent, and profile data are displayed.

WndProc Function:

The WndProc function handles messages related to window creation, resizing, and menu commands. It includes menu options like "File -> Open" for users to open DIB files.

Resource Files:

The program includes resource files (DIBHEADS.RC and RESOURCE.H) for defining accelerators and menus.

Accelerators and Menu:

The accelerators and menu are designed to allow users to open DIB files through the "File -> Open" menu or by pressing Ctrl+O.

User Interface and File Handling:

The program establishes a basic Windows interface featuring a menu with a "File | Open" option and an editable text field to display header information.

When a user selects "File | Open," a standard file dialog box guides them to choose a DIB file.

The program then reads the entire DIB file into memory for analysis.

DIB Header Analysis:

It begins by examining the `BITMAPFILEHEADER`, a fixed-size structure providing fundamental file information, including size and offset to the image data.

It then focuses on the information header, which can be one of several types: `BITMAPCOREHEADER`, `BITMAPINFOHEADER`, `BITMAPV4HEADER`, or `BITMAPV5HEADER`. This header reveals details about the image itself, such as width, height, color depth, compression method, and more.

The specific fields displayed depend on the encountered header type. For example, `BITMAPV5HEADER` offers additional features like color space information and gamma values.

The program extracts and presents each relevant field within the header structures, providing a thorough overview of the DIB's organization and characteristics.

Error Handling:

Any errors encountered during file opening, reading, or analysis are gracefully handled with informative messages displayed to the user.

Displaying and Printing DIBs:

While this program focuses on displaying header information, Windows offers functions for displaying and printing DIBs themselves:

`SetDIBitsToDevice`: Displays a DIB on a device (screen or printer) with the same pixel size as the original DIB.

`StretchDIBits`: Allows for stretching or shrinking a DIB's dimensions to fit a specific display size.

These functions can be used to render the actual image content of a DIB, going beyond header analysis.

Additional Insights:

For performance optimization, alternative methods for displaying DIBs, not directly covered in the provided code, might be preferred in certain scenarios.

When working with DIB (Device-Independent Bitmap) files and displaying them using appropriate functions, you typically need several pieces of information about the image. As mentioned earlier, DIB files consist of the following sections:

- **BITMAPFILEHEADER:** This section contains information about the file format, such as the type of file (BM), the file size, and the offset to the pixel data.
- **BITMAPINFOHEADER** or **BITMAPV5HEADER:** These sections define the properties of the bitmap image, including the image's width and height in pixels, the number of bits per pixel, compression type, color masks, color space information, gamma values, and rendering intent.
- **Color Table (for indexed color images):** If the image uses indexed colors, a color table follows the header sections. It contains the RGB color values used in the image, enabling the mapping of pixel values to specific colors.
- **Pixel Data:** This section stores the actual pixel information, represented by the specified number of bits per pixel. The pixel data can be organized in various formats, such as uncompressed RGB, compressed formats, or even encoded representations for specific purposes.



When displaying a DIB using appropriate functions, you typically provide the necessary information from these sections to ensure the correct interpretation and rendering of the image. This includes details such as image dimensions, color depth, compression type, color masks (if applicable), and the pixel data itself.

By utilizing the information from these sections, the displaying functions can properly decode and render the image, ensuring accurate representation on the screen or output device.

A [DIB file can be loaded into memory](#), and if the entire file excluding the file header is stored in a contiguous block of memory, it is referred to as a packed DIB. In a packed DIB, the pointer to the beginning of the memory block points to the start of the information header.

The [packed DIB format](#) allows for efficient memory storage and access, as all the relevant information required to interpret and display the image is stored in a single continuous block of memory. This includes the information header, color table (if applicable), and the pixel data.

By [using a pointer to the beginning of the memory block](#), applications can easily access and manipulate the various sections of the packed DIB, retrieve image properties, iterate over the pixel data, and perform any necessary operations on the image.

The [packed DIB format](#) is commonly used when working with DIB files in memory, as it provides a convenient representation that allows for efficient processing and manipulation of the image data, that is:



The packed DIB format is commonly used when transferring a DIB through the clipboard or creating a brush from a DIB. It provides a [convenient way to store the DIB in memory](#) because the entire DIB is referenced by a single pointer. This pointer, such as pPackedDib, can be defined as a pointer to a BYTE.

By using the structure definitions mentioned earlier, you can access all the information stored in the DIB using the [pPackedDib pointer](#). This includes accessing the color table (if applicable) and individual pixel bits.

With the packed DIB format, you can easily extract various properties of the image, such as width, height, color depth, compression type, and color masks. Additionally, by navigating the memory block using the pointer, you can access and manipulate the pixel data, allowing for operations such as modifying pixel values, applying filters, or performing other image processing tasks.

Advantages of Using Packed DIB Format:

The packed DIB format offers several advantages when storing DIBs in memory:

- **Compact Storage:** Packed DIBs store image data in a compressed format, resulting in more efficient memory usage compared to other formats.
- **Easy File Storage:** Packed DIBs are commonly used in file formats like BMP, making it convenient for direct storage and retrieval from files without additional transformations.
- **Simple Data Structure:** Packed DIBs use a straightforward data structure, allowing for easy interpretation and manipulation of pixel data.

Creating a Brush from a Packed DIB:

Creating a brush from a packed DIB involves using the `CreateDIBPatternBrushPt` function. Below is an example in C:

```
HBRUSH hBrush = CreateDIBPatternBrushPt(pPackedDib, DIB_RGB_COLORS);
```

Here, `pPackedDib` is a pointer to the packed DIB data. The `DIB_RGB_COLORS` flag indicates that the color table should be in RGB format.

Limitations and Considerations:

While the packed DIB format is widely used, there are considerations to keep in mind:

Format Variability: Different DIB formats, such as OS/2-compatible and newer versions, may require conditional checks to extract information, leading to more complex code.

Accessing Pixel Data: Accessing certain information, like pixel width, may require additional lines of code due to format differences. For example, handling OS/2-compatible formats requires checking the structure type.

Compatibility Checks: It's essential to check the DIB format before accessing specific fields to ensure compatibility and prevent potential errors.

Accessing Pixel Width in Packed DIB:

Accessing information from a packed DIB may require multiple lines of code due to potential variations in DIB formats. For example, directly accessing the pixel width with a statement like `iWidth = ((PBITMAPINFOHEADER) pPackedDib)->biWidth;` may not be straightforward. The presence of an OS/2-compatible format requires additional checks and conditional handling:

```
// Check if the DIB is in the OS/2-compatible format
if (((PBITMAPCOREHEADER) pPackedDib)->bcSize == sizeof(BITMAPCOREHEADER))
    iWidth = ((PBITMAPCOREHEADER) pPackedDib)->bcWidth;
else
    iWidth = ((PBITMAPINFOHEADER) pPackedDib)->biWidth;
```

In this code snippet, we are determining the pixel width (iWidth) of a DIB (pPackedDib). The check involves examining the bcSize field of the BITMAPCOREHEADER structure.

If bcSize matches the size of the BITMAPCOREHEADER, it indicates an OS/2-compatible format, and we access the pixel width using bcWidth.

Otherwise, we use the standard biWidth field of the BITMAPINFOHEADER structure.

This conditional check is necessary because DIBs can come in different formats, and the structure used to store information may vary.

The code ensures that the correct field is accessed based on the specific format of the provided DIB.

This conditional check ensures that the correct field is accessed based on the specific format of the provided DIB.

Fun Exercise: Accessing Pixel Value at (5, 27):

To access the pixel value at coordinate (5, 27), you need information about the DIB, including its width, height, bit count, row byte length, color table entries, presence of color masks, and compression status.

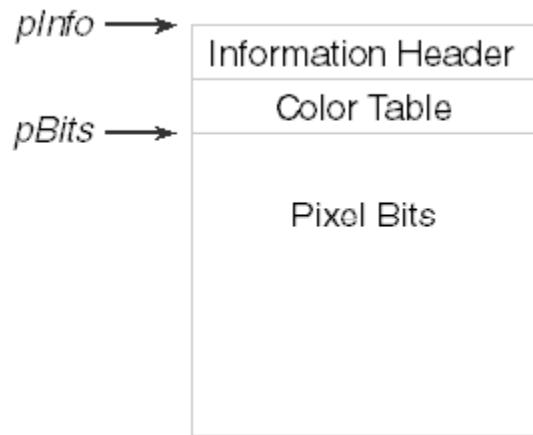
Directly accessing pixels, especially for image-processing tasks, can be time-consuming. A more efficient solution is using a C++ class for DIBs, allowing speedy random access. However, a C solution will be presented in the next chapter.

SetDIBitsToDevice and StretchDIBits Functions:

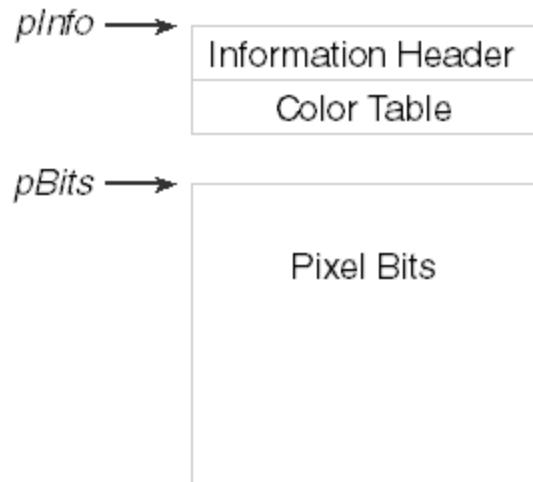
For these functions, you need a pointer to the BITMAPINFO structure of the DIB, comprising the BITMAPINFOHEADER structure and the color table.

Additionally, a pointer to the pixel bits is required. Calculating this pointer is more straightforward with access to the bfOffBits field of the BITMAPFILEHEADER structure, indicating the offset to the pixel bits.

It's important to note that obtaining a pointer to a packed DIB from the clipboard may lack a BITMAPFILEHEADER structure, complicating the process.



The SetDIBitsToDevice and StretchDIBits functions in Windows require two pointers to the DIB because the two sections (header and pixel data) do not have to be in one contiguous block of memory. It is possible to have the DIB stored in two separate blocks of memory.



The **first pointer**, typically referred to as *lpBits*, points to the pixel data section of the DIB. This pointer represents a block of memory that contains the actual pixel information. The pixel data can be organized in various formats, such as uncompressed RGB or compressed representations.

The [second pointer](#), usually referred to as `lpBitmapInfo`, points to the header section of the DIB. This pointer represents a block of memory that contains the DIB's header information, including the header structure (BITMAPINFO or BITMAPINFOHEADER) and, if applicable, the color table.

By providing these two separate pointers to the `SetDIBitsToDevice` or `StretchDIBits` functions, you can specify the location of the pixel data and the header information independently. This allows for flexibility in memory management and the ability to work with DIBs stored in non-contiguous memory blocks.

Here's an example of how these functions can be called:

```
SetDIBitsToDevice(
    hdcDest,           // Destination device context
    xDest, yDest,      // Destination coordinates
    dwWidth, dwHeight, // Width and height of the image
    xSrc, ySrc,        // Source coordinates
    uStartScan,        // Starting scan line
    cScanLines,        // Number of scan lines
    lpBits,            // Pointer to the pixel data
    lpBitmapInfo,      // Pointer to the header information
    iUsage             // Color table usage
);
```

In this example, `lpBits` points to the pixel data, and `lpBitmapInfo` points to the header information. The remaining parameters specify other details of the operation, such as the source and destination coordinates, image dimensions, and color table usage.

Using [two separate pointers](#) allows for flexibility in handling DIBs stored in different memory layouts, enabling efficient rendering and manipulation of DIB images in various scenarios.

Breaking a DIB into two memory blocks using separate pointers is indeed useful and allows for flexibility in memory management. However, [when working with packed DIBs](#), the entire DIB is typically stored in a single memory block.

In addition to the two pointers (`lpBits` and `lpBitmapInfo`), the `SetDIBitsToDevice` and `StretchDIBits` functions typically require the pixel width and height of the DIB. These values are used to determine the size and dimensions of the image being displayed or manipulated.

If you are [displaying only a part of the DIB](#), you may not need to explicitly specify the pixel width and height. However, these values still serve as an upper limit for defining a rectangle within the array of DIB pixel bits. By specifying the pixel width and height, you can ensure that you do not access or modify pixels outside the defined region.

So, even if you are working with a subset of the DIB, it is generally advisable to provide the pixel width and height to the SetDIBitsToDevice or StretchDIBits functions, as they help define the boundaries of the area you are working with within the overall DIB image.

By including the pixel width and height, you ensure that the functions correctly interpret and process the image data within the specified dimensions, preventing any unintended behavior or accessing pixels beyond the intended region.

What are some other functions or methods commonly used when working with DIBs? Can you provide an example of how to define a rectangle within the array of DIB pixel bits? Are there any performance considerations when working with packed DIBs and memory blocks?

When working with DIBs, there are several other functions and methods commonly used in addition to SetDIBitsToDevice and StretchDIBits. Some of these include:

CreateDIBSection: This function allows you to create a DIB and obtain pointers to the pixel data and the header information. It is often used when you need to allocate memory for a new DIB.

GetDIBits: This function retrieves the pixel data from a DIB and copies it into a specified buffer. It is useful when you need to extract pixel information from a DIB for further processing or analysis.

BitBlt: This function performs bit-block transfer operations, allowing you to copy DIBs or parts of DIBs between different device contexts (such as screen, printer, or memory DCs). It can be used for tasks like displaying DIBs, printing, or creating image compositions.

Image processing libraries: There are various libraries and frameworks available that provide advanced image processing capabilities. These libraries often include functions for tasks such as resizing, rotating, filtering, and manipulating DIBs.

To define a rectangle within the array of DIB pixel bits, you typically specify the starting coordinates (x, y) and the width and height of the rectangle. The coordinates (0, 0) usually correspond to the top-left corner of the DIB. For example, to define a rectangle starting at pixel position (10, 20) with a width of 100 pixels and a height of 50 pixels, you would define the rectangle as follows:

```
RECT rect;
rect.left = 10;
rect.top = 20;
rect.right = rect.left + 100;
rect.bottom = rect.top + 50;
```

Regarding performance considerations when working with packed DIBs and memory blocks, there are a few factors to keep in mind:

Memory alignment: Different platforms and compilers may have specific memory alignment requirements. Ensuring proper alignment of the memory blocks containing the DIB data can improve memory access and processing efficiency.

Memory fragmentation: If the memory blocks storing the DIB data become fragmented, it can impact memory access and overall performance. Techniques such as memory pooling or memory defragmentation can be employed to mitigate fragmentation issues.

Memory access patterns: Optimizing memory access patterns, such as accessing memory in a sequential manner or utilizing cache-friendly algorithms, can improve performance when working with packed DIBs.

Image compression: If the DIB utilizes compression, decoding and encoding operations may introduce additional computational overhead. Choosing efficient compression algorithms or optimizing compression settings can help mitigate performance impacts.

Common image processing libraries that provide advanced capabilities for working with DIBs include:

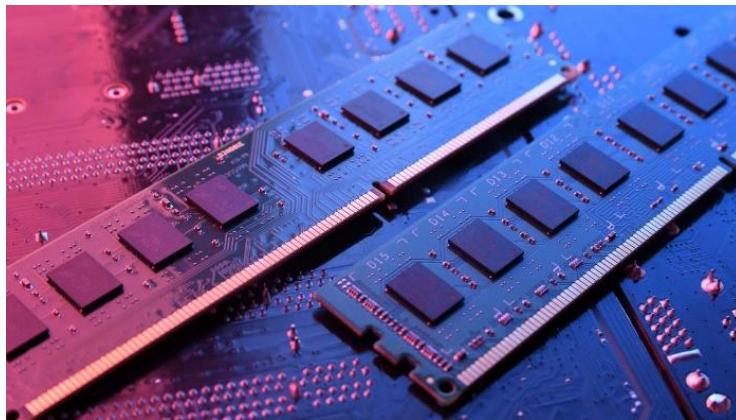
[OpenCV \(Open source Computer Vision Library\)](#) is a popular open-source library that offers a comprehensive set of image processing and computer vision algorithms. It provides functions for image manipulation, feature detection, object recognition, and more. OpenCV supports DIBs and offers efficient processing capabilities.

[ImageMagick](#): ImageMagick is a powerful image processing library that supports a wide range of image formats, including DIBs. It provides functions for image conversion, resizing, filtering, and various other image manipulation operations.

[Pillow](#): Pillow is a friendly fork of the Python Imaging Library (PIL) and offers a simplified API for image processing tasks in Python. It supports DIBs and provides functions for image loading, saving, resizing, and basic manipulation.

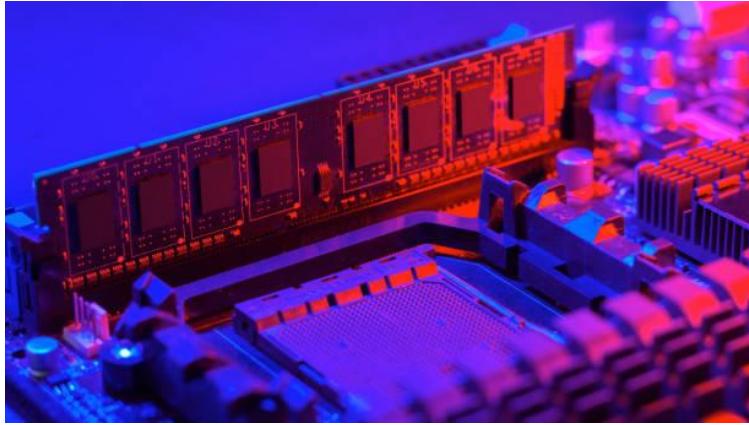
[Skia](#): Skia is a 2D graphics library developed by Google that provides high-performance rendering and image processing capabilities. It is widely used in various applications, including web browsers and mobile platforms, and supports DIBs as an input format.

Memory alignment refers to the way data is stored in memory, respecting certain alignment requirements imposed by the hardware architecture or compiler. Memory alignment can impact performance when working with DIBs due to the way CPUs access memory. CPUs often perform more efficiently when accessing memory that is aligned on specific boundaries, such as 4-byte or 8-byte boundaries.



When memory blocks containing DIB data are properly aligned, CPU cache utilization and memory access patterns are optimized, resulting in improved performance. On the other hand, misaligned memory can lead to additional CPU cycles being required to fetch or update the data, resulting in decreased performance.

To ensure memory alignment when working with DIBs, it's important to adhere to the alignment requirements of the specific hardware architecture or compiler you are using. This typically involves aligning the memory blocks containing the DIB data on appropriate boundaries (e.g., using memory allocation functions that provide aligned memory).

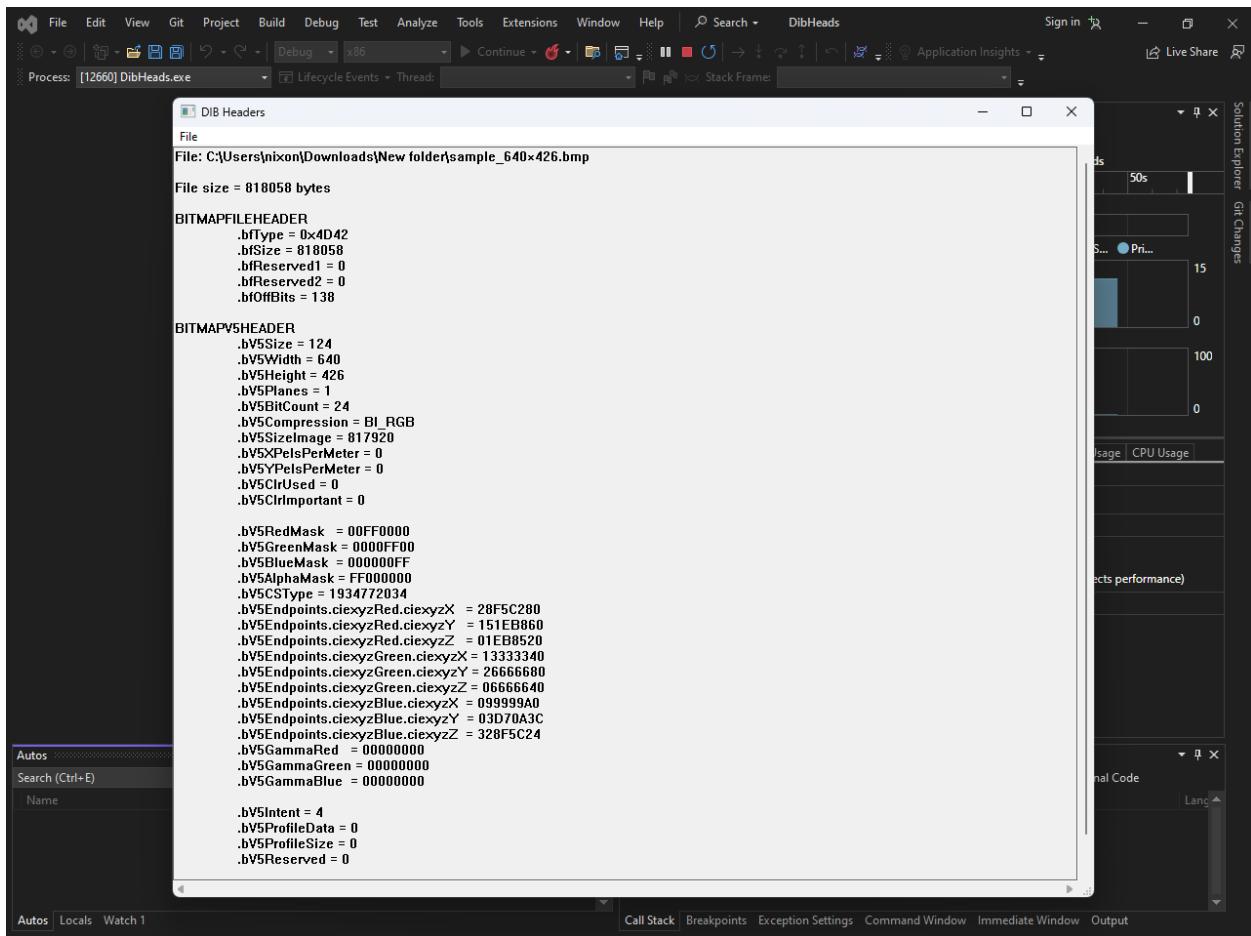


Optimizing memory access patterns when working with packed DIBs can help improve performance. Some techniques and best practices include:

- **Sequential access:** Process the DIB data in a sequential manner whenever possible. Sequential access allows for efficient utilization of CPU caches and minimizes the number of cache misses.
- **Loop unrolling:** Unrolling loops can reduce loop overhead and improve memory access patterns. By processing multiple pixels or scanlines at once within a loop iteration, you can reduce the number of loop iterations and potentially improve performance.
- **Cache utilization:** Minimize random memory access and favor accessing memory in a predictable pattern. This helps maximize cache utilization and reduces the number of cache misses.
- **Data locality:** Arrange data structures and algorithms in a way that maximizes data locality. Keep frequently accessed data close together in memory to improve cache efficiency.
- **SIMD instructions:** Utilize vectorized instructions, such as SIMD (Single Instruction, Multiple Data) instructions, to perform parallel processing on multiple pixels simultaneously. This can significantly boost performance when supported by the hardware and compiler.

These techniques and best practices can vary depending on the specific programming language, platform, and compiler being used. Profiling and benchmarking can help identify performance bottlenecks and guide optimization efforts for memory access patterns when working with packed DIBs.

From our code dibheads this was the result, I downloaded a bmp file and opened it. Opening other files like jpg was different.



```
File: C:\Users\nixon\Downloads\New folder\sample_640x426.bmp
File size = 818058 bytes

BITMAPFILEHEADER
    .bType = 0x4D42
    .bSize = 818058
    .bReserved1 = 0
    .bReserved2 = 0
    .bOffBits = 138

BITMAPV5HEADER
    .bV5Size = 124
    .bV5Width = 640
    .bV5Height = 426
    .bV5Planes = 1
    .bV5BitCount = 24
    .bV5Compression = BI_RGB
    .bV5SizeImage = 817920
    .bV5XPelsPerMeter = 0
    .bV5YPelsPerMeter = 0
    .bV5CrlUsed = 0
    .bV5CrlImportant = 0

    .bV5RedMask = 00FF0000
    .bV5GreenMask = 0000FF00
    .bV5BlueMask = 000000FF
    .bV5AlphaMask = FF000000
    .bV5CSType = 1934772034
    .bV5Endpoints.ciexyzRed.ciexyzX = 28F5C280
    .bV5Endpoints.ciexyzRed.ciexyzY = 151EB860
    .bV5Endpoints.ciexyzRed.ciexyzZ = 01EB8520
    .bV5Endpoints.ciexyzGreen.ciexyzX = 13333340
    .bV5Endpoints.ciexyzGreen.ciexyzY = 26666680
    .bV5Endpoints.ciexyzGreen.ciexyzZ = 06666640
    .bV5Endpoints.ciexyzBlue.ciexyzX = 099999A0
    .bV5Endpoints.ciexyzBlue.ciexyzY = 03D70A3C
    .bV5Endpoints.ciexyzBlue.ciexyzZ = 328F5C24
    .bV5GammaRed = 00000000
    .bV5GammaGreen = 00000000
    .bV5GammaBlue = 00000000

    .bV5Intent = 4
    .bV5ProfileData = 0
    .bV5ProfileSize = 0
    .bV5Reserved = 0
```

Function Overview:

The [SetDIBitsToDevice](#) function is used to display a Device-Independent Bitmap (DIB) without any stretching or shrinking. Each pixel of the DIB corresponds directly to a pixel on the output device. The orientation of the image is always correct, with the top row displayed at the top. The function takes several arguments, but its usage is generally straightforward.

Function Signature:

```
iLines = SetDIBitsToDevice(
    hdc,          // device context handle
    xDst,         // x destination coordinate
    yDst,         // y destination coordinate
    cxSrc,        // source rectangle width
    cySrc,        // source rectangle height
    xSrc,         // x source coordinate
    ySrc,         // y source coordinate
    yScan,        // first scan line to draw
    cyScans,      // number of scan lines to draw
    pBits,         // pointer to DIB pixel bits
    pInfo,         // pointer to DIB information
    fClrUse       // color use flag
);
```

Arguments Explanation:

- [hdc \(Device Context Handle\)](#): Specifies the device context of the output device where the DIB will be displayed.
- [xDst, yDst \(Destination Coordinates\)](#): Logical coordinates of the output device, indicating the position where the top-left corner of the DIB image will appear.
- [cxSrc, cySrc \(Source Rectangle Dimensions\)](#): Width and height of the source rectangle. You can display the entire DIB or only part of it.
- [xSrc, ySrc \(Source Coordinates\)](#): Logical coordinates of the source rectangle. Typically set to 0 for displaying the entire DIB.
- [yScan, cyScans \(Scan Line Parameters\)](#): Used to display the DIB sequentially, reducing memory requirements. Usually set to 0 and the height of the DIB.
- [pBits \(Pointer to DIB Pixel Bits\)](#): A pointer to the pixel bits of the DIB.
- [pInfo \(Pointer to BITMAPINFO Structure\)](#): A pointer to the BITMAPINFO structure of the DIB, providing information about the bitmap.
- [fClrUse \(Color Use Flag\)](#): Either DIB_RGB_COLORS or DIB_PAL_COLORS. Use DIB_RGB_COLORS if the DIB contains a color table.

When working with DIBs, there are several important considerations to keep in mind:

- **Upside-Down DIB Handling:** DIBs store pixel data upside-down, which means that adjustments are required when specifying source rectangle coordinates. The y-coordinate increases as you move down the DIB, contrary to the conventional coordinate system. Therefore, when working with DIBs, you need to account for this inversion when specifying the source rectangle coordinates.
- **Logical vs. Pixel Coordinates:** The documentation may incorrectly state that the xSrc, ySrc, cxSrc, and cySrc parameters are in logical units. However, in reality, these parameters represent pixel coordinates and dimensions. It's essential to understand that they are measured in pixels, not logical units.
- **Sequential Display:** The yScan and cyScan parameters enable memory-efficient display of DIBs as they are read from storage or transmission. These parameters allow you to specify the starting scanline and the number of scanlines to be processed at a time. This sequential display approach can be beneficial when dealing with large DIBs or limited memory resources.
- **Color Table Handling:** The fClrUse flag indicates whether a DIB contains a color table (DIB_RGB_COLORS) or uses a logical color palette (DIB_PAL_COLORS). This flag helps determine how color information is interpreted when working with the DIB. It is important to correctly set this flag based on the color representation used in the DIB.
- **Mapping Modes and Transforms:** The mapping mode and transformations applied to the device context affect the starting position of the DIB, but they do not alter its size or orientation. These settings control how the DIB is mapped onto the output device and do not directly impact the dimensions or orientation of the DIB itself.
- **BITMAPINFO vs. BITMAPINFOHEADER:** While pInfo typically points to a BITMAPINFO structure, it can also point to BITMAPCOREINFO, BITMAPV4HEADER, or BITMAPV5HEADER structures. The specific structure being used depends on the version and requirements of the DIB. It's important to ensure that the appropriate structure is used and that the data is interpreted correctly.

How to display an entire DIB using SetDIBitsToDevice, incorporating key points and addressing potential confusion:

Essential Information:

Device Context Handle (hdc): Obtain this handle, representing the target output device (screen or printer), using functions like GetDC or CreateDC.

Destination Coordinates (xDst, yDst): Specify the top-left corner of the DIB on the output device using logical coordinates (pixels for common mapping modes).

DIB Dimensions (cxDib, cyDib): Retrieve these from the BITMAPINFO structure, ensuring cyDib is the absolute value of the biHeight field due to DIB's upside-down storage.

DIB Data Pointers (pInfo, pBits): These point to the DIB's information section (header and optional color table) and the pixel data itself, respectively.

Function Call:

```
SetDIBitsToDevice(
    hdc,      // device context handle
    xDst,    // x destination coordinate
    yDst,    // y destination coordinate
    cxDib,   // source rectangle width
    cyDib,   // source rectangle height
    0,        // x source coordinate
    0,        // y source coordinate
    0,        // first scan line to draw
    cyDib,   // number of scan lines to draw
    pBits,   // pointer to DIB pixel bits
    pInfo,   // pointer to DIB information
    0        // color use flag
);
```

The SetDIBitsToDevice function is used to display a Device-Independent Bitmap (DIB) on an output device. It returns the number of scan lines successfully displayed (iLines) as an indication of the operation's success.

Commonly, the arguments xSrc, ySrc, xScan, and cyScan are set to specific values:

- xSrc and ySrc are often set to 0, indicating the top-left corner of the source rectangle.
- yScan is typically set to 0, representing the first scan line to draw.
- cyScan is set to the height of the DIB (cyDib), ensuring that the entire DIB is displayed.

The pInfo and pBits arguments provide the necessary information about the DIB.

Additional Notes:

The **color use flag (fClrUse)** is set to 0 in the function call, indicating the use of DIB_RGB_COLORS. This suggests that the DIB contains a color table.

The **SetDIBitsToDevice function call** is commonly used in programs like the SHOWDIB1 program shown in Figure 15-2. It simplifies the process by setting specific parameter values for displaying the entire DIB.

Key Points:

Return Value: The function returns the number of scan lines that were successfully displayed, providing an indication of the operation's success.

Common Default Arguments: xSrc, ySrc, yScan, and cyScan are often set to default values, such as 0 for displaying the entire DIB from the top-left corner.

Color Use Flag: The color use flag (fClrUse) is set to 0 for DIBs with color tables (DIB_RGB_COLORS) or DIB_PAL_COLORS for DIBs using logical palettes.

Additional Considerations:

Error Handling: It's important to check the return value of the function to ensure successful display and handle any potential errors.

Mapping Modes and Transforms: Consider the effects of mapping modes and transformations on the positioning of the DIB on the output device.

Memory Management: Release any allocated resources (e.g., memory, device contexts) appropriately when finished using them.

SHOWDIB1.C PROGRAM

The code, a Windows application written in C, which displays a Device Independent Bitmap (DIB) in the client area of a window. Let's break down the code and explain its functionality in paragraphs:

The code includes necessary header files, such as `<windows.h>`, `"dibfile.h"`, and `"resource.h"`, and defines a callback function `WndProc` that handles messages sent to the window.

The `WinMain` function is the entry point of the application. It registers a window class, creates a window, and enters the main message loop using `GetMessage` function. This loop processes messages from the operating system and dispatches them to the window procedure (`WndProc`) for handling.

The `WndProc` function is the window procedure that handles various messages sent to the window. It contains a switch statement that handles different message types.

The `WM_CREATE` message is sent when the window is being created. In this case, it calls the `DibFileInitialize` function, which is not shown in the provided code. This function likely initializes any necessary resources for working with DIB files.

The `WM_SIZE` message is sent when the window is resized. The code retrieves the new client area dimensions using the `LOWORD` and `HIWORD` macros and stores them in the variables `cxClient` and `cyClient`.

The `WM_INITMENUPOPUP` message is sent when a menu is about to become active. It enables or disables the "Save" menu item (`IDM_FILE_SAVE`) based on whether a DIB file is loaded (`pbfmh` is non-null).

The `WM_COMMAND` message is sent when a menu item or an accelerator is selected. The code handles specific menu commands, such as opening and saving DIB files.

When the "Open" menu item (`IDM_FILE_OPEN`) is selected, it displays a file open dialog box (`DibFileOpenDlg`) to choose a DIB file. If a file is selected, it loads the DIB into memory using the `DibLoadImage` function and updates the window accordingly.

The loaded DIB is stored in memory using the `pbfmh` variable, which is a pointer to a `BITMAPFILEHEADER` structure.

The `WM_PAINT` message is sent when the window's client area needs to be repainted. In response, the code retrieves a device context (`hdc`) using `BeginPaint`, and if a DIB is loaded (`pbfmh` is non-null), it uses `SetDIBitsToDevice` to draw the DIB onto the device context.

The `WM_DESTROY` message is sent when the window is being destroyed. The code frees any allocated memory (`pbfmh`) and posts a quit message to exit the application.

If none of the handled messages are encountered, the window procedure calls `DefWindowProc` to provide default handling for those messages.

Overall, this code sets up a window that can display DIB images, allows loading and saving of DIB files, and handles the necessary painting operations. It uses various Windows API functions to interact with the operating system and perform the required tasks.

The provided code also consists of two files: "DIBFILE.H" and "DIBFILE.C," which work together to handle DIB (Device-Independent Bitmap) files.

"DIBFILE.H" is a header file containing function declarations and a structure definition. Here's a breakdown of the functions declared in "DIBFILE.H":

void DibFileInitialize(HWND hwnd): This function initializes the global variable ofn of type OPENFILENAME. It sets various members of ofn to default values required for file dialog operations.

BOOL DibFileOpenDlg(HWND hwnd, PTSTR pstrFileName, PTSTR pstrTitleName): This function opens a file dialog box and allows the user to select a file to open. It takes the window handle (hwnd) and pointers to store the selected file name (pstrFileName) and file title (pstrTitleName). It returns a Boolean value indicating the success or failure of the file dialog operation.

BOOL DibFileSaveDlg(HWND hwnd, PTSTR pstrFileName, PTSTR pstrTitleName): This function opens a file dialog box and allows the user to select a destination file for saving. It takes the window handle (hwnd) and pointers to store the selected file name (pstrFileName) and file title (pstrTitleName). It returns a Boolean value indicating the success or failure of the file dialog operation.

BITMAPFILEHEADER *DibLoadImage(PTSTR pstrFileName): This function loads a DIB image file from the specified file path (pstrFileName). It reads the file, allocates memory for storing the image data, and returns a pointer to the BITMAPFILEHEADER structure containing information about the loaded image. If the loading fails, it returns NULL.

BOOL DibSaveImage(PTSTR pstrFileName, BITMAPFILEHEADER *pbmfh): This function saves a DIB image to the specified file path (pstrFileName). It takes a pointer to the BITMAPFILEHEADER structure (pbmfh) that contains the image data. It writes the image data to the file and returns a Boolean value indicating the success or failure of the operation.

Moving on to "DIBFILE.C," this file includes necessary header files, including "windows.h," "commdlg.h," and the custom header file "dibfile.h." It also defines a static global variable ofn of type OPENFILENAME (which was declared in "DIBFILE.H").

The code for the DibFileInitialize function initializes the ofn variable with default values required for file dialog operations.

The `DibFileOpenDlg` function sets the necessary values in `ofn` for opening a file dialog and calls the `GetOpenFileName` function to display the file dialog to the user. It returns a Boolean value indicating whether the user successfully selected a file.

The `DibFileSaveDlg` function sets the necessary values in `ofn` for saving a file dialog and calls the `GetSaveFileName` function to display the file dialog to the user. It returns a Boolean value indicating whether the user successfully selected a destination file.

The [DibLoadImage function](#) opens the specified file and reads its contents into memory. It performs error checks on the file and returns a pointer to the `BITMAPFILEHEADER` structure containing the image data. If the loading fails, it returns `NULL`.

The [DibSaveImage function](#) creates a new file with the specified name and writes the image data from the `BITMAPFILEHEADER` structure to the file. It performs error checks and returns a Boolean value indicating the success or failure of the operation.

Overall, these functions provide a basic framework for handling DIB files, including loading and saving images and displaying file dialogs for file selection and saving.

The provided code consists of excerpts from three files: "SHOWDIB1.RC," "RESOURCE.H," and "DIBFILE.C."

The "[SHOWDIB1.RC](#)" file contains resource script definitions for menus and other resources used by the program. In this case, it includes a menu definition for the "File" menu with two menu items: "Open..." and "Save...". These menu items have associated command IDs (`IDM_FILE_OPEN` and `IDM_FILE_SAVE`) defined in the "RESOURCE.H" file.

The "[RESOURCE.H](#)" file is an include file used by the resource script. It defines the command IDs for the menu items. In this case, `IDM_FILE_OPEN` is defined as 40001, and `IDM_FILE_SAVE` is defined as 40002.

Moving on to the "[DIBFILE.C](#)" file, it contains routines to handle file dialogs, load DIB files into memory, and save DIB files from memory.

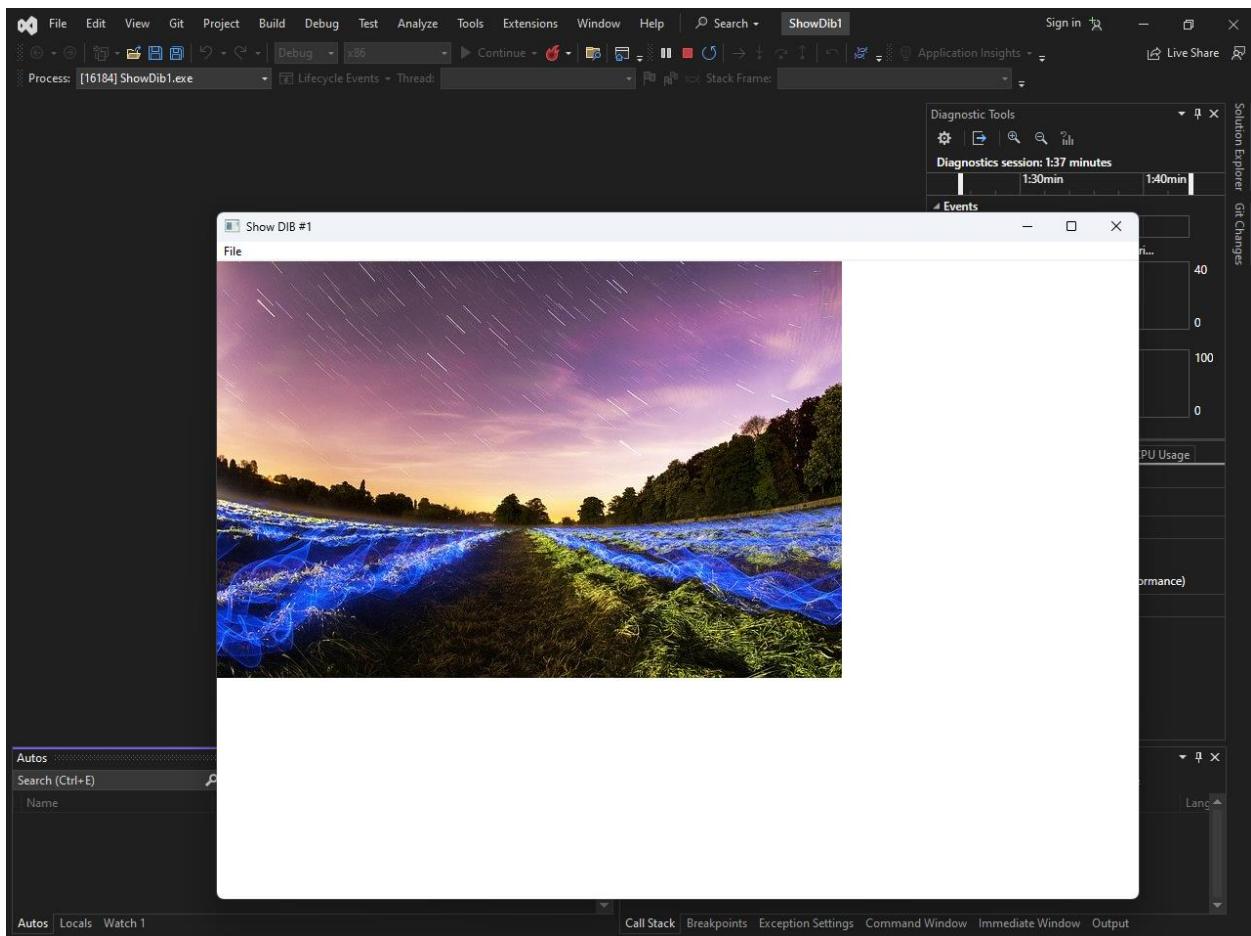
The code in "DIBFILE.C" defines several functions:

- **DibFileInitialize(HWND hwnd):** This function initializes the global variable ofn (defined in "DIBFILE.H") with default values required for file dialog operations.
- **DibFileOpenDlg(HWND hwnd, PTSTR pstrFileName, PTSTR pstrTitleName):** This function opens a file dialog box for selecting a file to open. It takes the window handle (hwnd) and pointers to store the selected file name (pstrFileName) and file title (pstrTitleName). It returns a Boolean value indicating the success or failure of the file dialog operation.
- **DibFileSaveDlg(HWND hwnd, PTSTR pstrFileName, PTSTR pstrTitleName):** This function opens a file dialog box for selecting a destination file for saving. It takes the window handle (hwnd) and pointers to store the selected file name (pstrFileName) and file title (pstrTitleName). It returns a Boolean value indicating the success or failure of the file dialog operation.
- **DibLoadImage(PTSTR pstrFileName):** This function loads a DIB image file from the specified file path (pstrFileName). It reads the file, allocates memory for storing the image data (including the BITMAPFILEHEADER structure), and returns a pointer to the BITMAPFILEHEADER structure. If the loading fails, it returns NULL.
- **DibSaveImage(PTSTR pstrFileName, BITMAPFILEHEADER* pbfm):** This function saves a DIB image to the specified file path (pstrFileName). It takes a pointer to the BITMAPFILEHEADER structure (pbfm) that contains the image data. It writes the image data to the file and returns a Boolean value indicating the success or failure of the operation.

The code in "DIBFILE.C" provides the basic functionality for opening file dialogs, loading DIB files into memory, and saving DIB files from memory. These functions are used in conjunction with the menu commands defined in the resource script to enable the user to open and save DIB files.

The comment in the excerpt suggests that "SHOWDIB1.C" (not provided) is the main program file. It mentions that when the program processes the "File Open" command, it loads a DIB file using the DibLoadImage function and calculates the offsets of the BITMAPINFOHEADER structure and the pixel bits within the memory block. It also obtains the pixel width and height of the DIB and stores this information in static variables.

During the WM_PAINT message handling, the program displays the loaded DIB by calling the SetDIBitsToDevice function. However, the code is mentioned to have deficiencies, such as lack of scroll bars when the DIB is larger than the client area. These deficiencies are expected to be addressed in subsequent chapters.



Yap, that's the bmp file in that code folder...

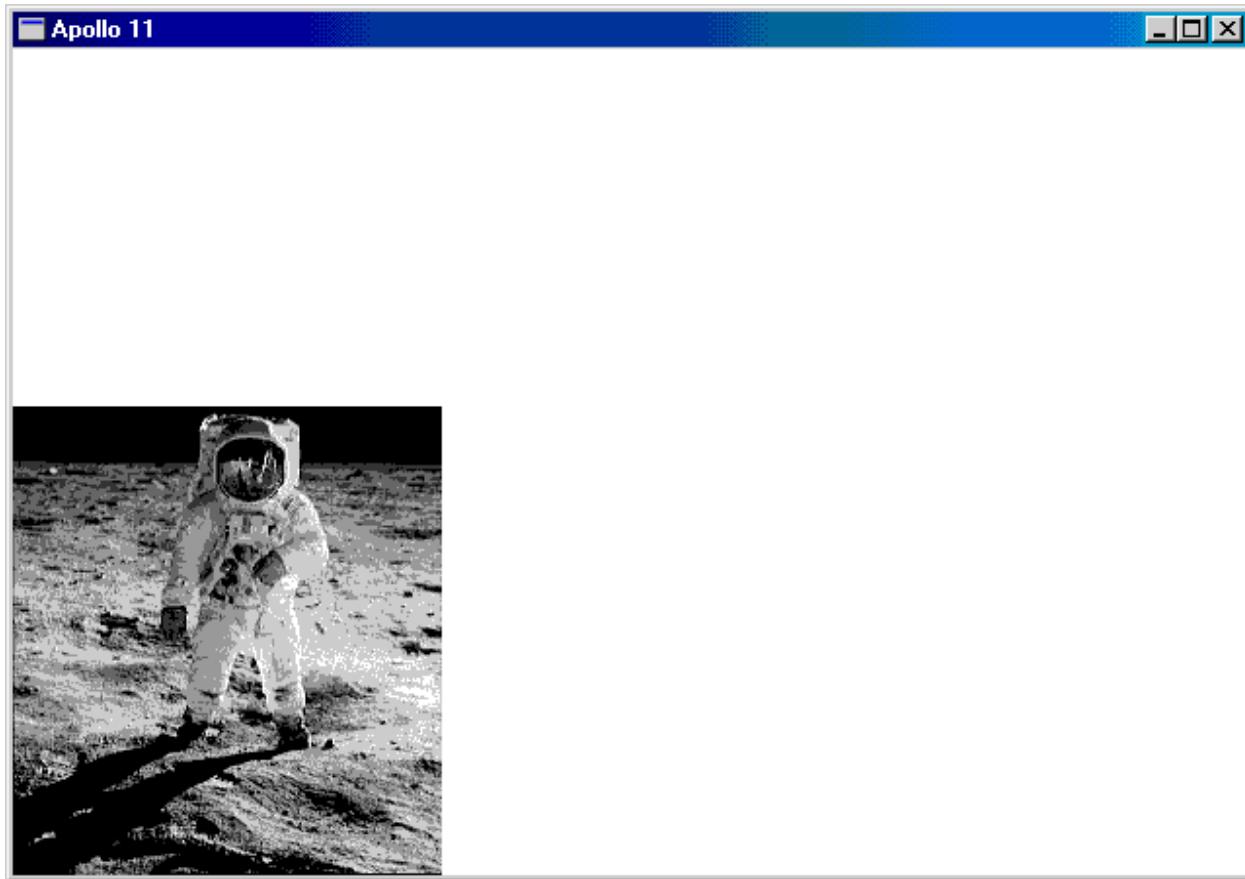
Overall, the code provided sets up the necessary functions and resources to handle DIB files, including file dialogs, loading DIB files into memory, and displaying them on the screen.

The design of application program interfaces for operating systems can be challenging, especially when attempting to fix initial mistakes. This holds true for the concept of Device-Independent Bitmaps (DIBs).

Originally, in the OS/2 Presentation Manager, DIBs were defined with a bottom-up orientation for pixel bits. This decision was somewhat logical because the OS/2 Presentation Manager follows a bottom-left origin convention.

In other words, within a Presentation Manager window, the default origin point (0,0) is located in the lower-left corner. However, this bottom-up orientation may seem counterintuitive to many people, except for mathematicians who are accustomed to this coordinate system.

The bitmap-drawing functions in OS/2 also used lower-left coordinates to specify the destination. Consequently, if you specified a destination coordinate of (0,0) for a bitmap, the image would appear flush against the left and bottom edges of the window, as depicted in Figure 15-3 below (A bitmap as it would be displayed under OS/2 with a (0,0) destination).



The lesson learned here is that when initial decisions are flawed, attempting to patch them later tends to compound the issues further. The bottom-up orientation of DIBs, which made sense within the OS/2 Presentation Manager, created confusion and challenges when interacting with other systems or APIs that followed different conventions.

In subsequent developments and adaptations of DIBs, efforts were made to address these inconsistencies and align them with more widely accepted conventions. This led to changes in how DIBs were handled and displayed in different operating systems and graphical user interfaces.

On slower machines, it was possible to observe the bitmap being drawn from the bottom to the top due to the way DIBs were processed.

Despite the seemingly unconventional coordinate system used in the OS/2 Presentation Manager, it had the advantage of being highly consistent. In OS/2, the origin (0,0) of the bitmap corresponded to the first pixel of the first row in the bitmap file. This pixel was then mapped to the destination coordinate specified in the bitmap-drawing functions.

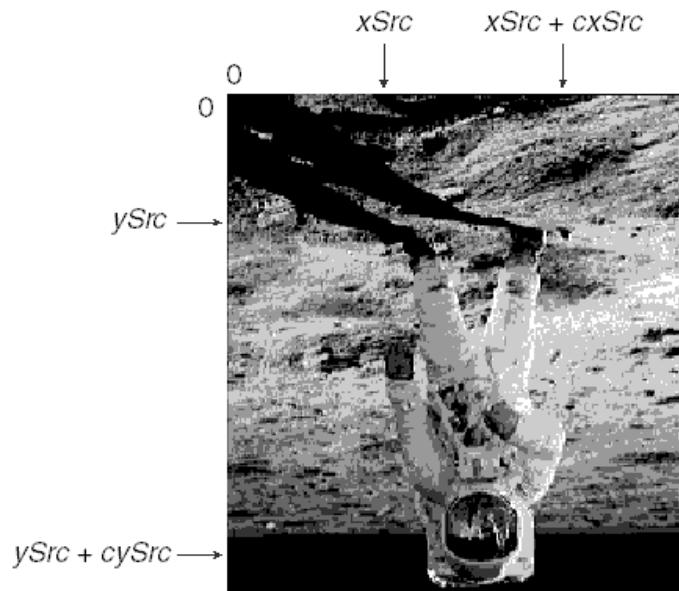
However, Windows introduced inconsistencies in maintaining internal consistency with DIBs. When displaying a rectangular subset of the entire DIB image, you use the arguments $xSrc$, $ySrc$, $cxSrc$, and $cySrc$.

These source coordinates and sizes are relative to the first row of the DIB data, which is the bottom row of the image, similar to OS/2. However, unlike OS/2, Windows displays the top row of the image at the destination coordinate.

Consequently, when displaying the entire DIB image, the pixel displayed at $(xDst, yDst)$ corresponds to the DIB pixel at coordinate $(0, cyDib - 1)$, which represents the last row of DIB data but the top row of the image.

When displaying only a part of the image, the pixel displayed at $(xDst, yDst)$ corresponds to the DIB pixel at coordinate $(xSrc, ySrc + cySrc - 1)$.

To help visualize this behavior, refer to Figure 15-4. The diagram represents the DIB stored in memory, upside-down from how it is typically imagined.



The origin for coordinate measurements aligns with the first bit of pixel data in the DIB. The xSrc argument in SetDIBitsToDevice is measured from the left of the DIB, and cxSrc represents the width of the image to the right of xSrc.

This aspect is straightforward. However, the ySrc argument is measured from the first row of the DIB data (the bottom of the image), and cySrc represents the height of the image from ySrc towards the last row of the data (the top of the image).

If the destination device context uses the default pixel coordinates with the MM_TEXT mapping mode, the relationship between the corner coordinates of the source and destination rectangles follows the table below:

Source Rectangle	Destination Rectangle
(xSrc, ySrc)	(xDst, yDst + cySrc - 1)
(xSrc + cxSrc - 1, ySrc)	(xDst + cxSrc - 1, yDst + cySrc - 1)
(xSrc, ySrc + cySrc - 1)	(xDst, yDst)
(xSrc + cxSrc - 1, ySrc + cySrc - 1)	(xDst + cxSrc - 1, yDst)

The fact that (xSrc, ySrc) does not directly map to (xDst, yDst) is what contributes to the chaotic nature of working with DIBs. However, with any other mapping mode, the point (xSrc, ySrc + cySrc - 1) will still map to the logical point (xDst, yDst), and the image will appear the same as it does in the MM_TEXT mapping mode.

So far, we have discussed the normal case when the biHeight field of the BITMAPINFOHEADER structure is positive. However, if the biHeight field is negative, indicating a top-down arrangement of DIB data, one might think that it resolves all the problems. Unfortunately, that assumption would be naive.

It appears that someone decided that if you take a top-down DIB, flip all the rows around, and then set the biHeight field to a positive value, it should behave the same as a regular bottom-up DIB.

The intention was to ensure that existing code referencing the DIB's rectangle wouldn't require modification. While this objective seems reasonable, it overlooks the fact that programs still need to be modified to handle top-down DIBs and avoid using a negative height.

Furthermore, this decision has peculiar implications. It means that source coordinates within top-down DIBs have an origin at the last row of the DIB data, which is also the bottom row of the image.

This concept is entirely different from anything we have encountered so far. The DIB pixel at the (0,0) origin is no longer the first pixel referenced by the pBits pointer, nor is it the last pixel in the DIB file. It occupies a position somewhere in between.

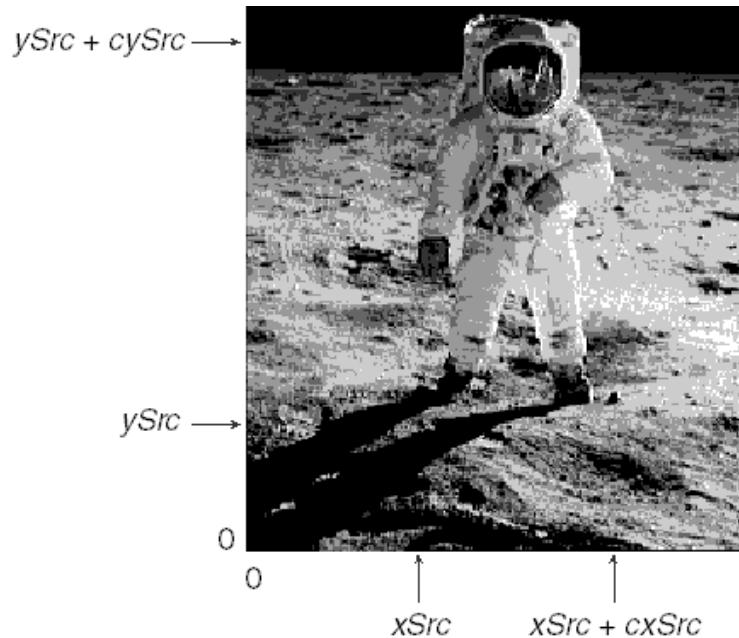


Figure 15-5, illustrates how you specify a rectangle within a top-down DIB. The diagram represents the DIB as it is stored in the file or in memory, providing a visual representation of the arrangement.

Here's the continuation of the notes, focusing on the advantage of the scheme used in the SetDIBitsToDevice function:

One significant advantage of the scheme used in the SetDIBitsToDevice function is that the arguments provided to the function are independent of the orientation of the DIB data.

This means that if you have two DIBs—one bottom-up and the other top-down—that display the same image (with the rows in the DIB files arranged in opposite orders), you can use identical arguments with the SetDIBitsToDevice function to select and display the same portion of the image.

This advantage is demonstrated in the APOLLO11 program, as depicted in Figure 15-6. The program showcases how the same image can be displayed using the SetDIBitsToDevice function, regardless of whether the DIB data is bottom-up or top-down.

By leveraging this consistency in argument usage, developers can work with DIBs of different orientations without needing to modify their code extensively. This flexibility allows for more efficient and streamlined handling of DIBs in various scenarios. Here's the information with the xSrc coordinates provided in code boxes:

```
// Source Rectangle
// Top-left corner of the source rectangle
(xSrc, ySrc)
// Top-right corner of the source rectangle
(xSrc + cxSrc - 1, ySrc)
// Bottom-left corner of the source rectangle
(xSrc, ySrc + cySrc - 1)
// Bottom-right corner of the source rectangle
(xSrc + cxSrc - 1, ySrc + cySrc - 1)
```

When working with MM_TEXT mapping mode (default pixel coordinates), the relationship between the source and destination rectangles is as follows:

```
// Destination Rectangle
// Top-left corner of the destination rectangle
(xDst, yDst + cySrc - 1)
// Top-right corner of the destination rectangle
(xDst + cxSrc - 1, yDst + cySrc - 1)
// Bottom-left corner of the destination rectangle
(xDst, yDst)
// Bottom-right corner of the destination rectangle
(xDst + cxSrc - 1, yDst)
```

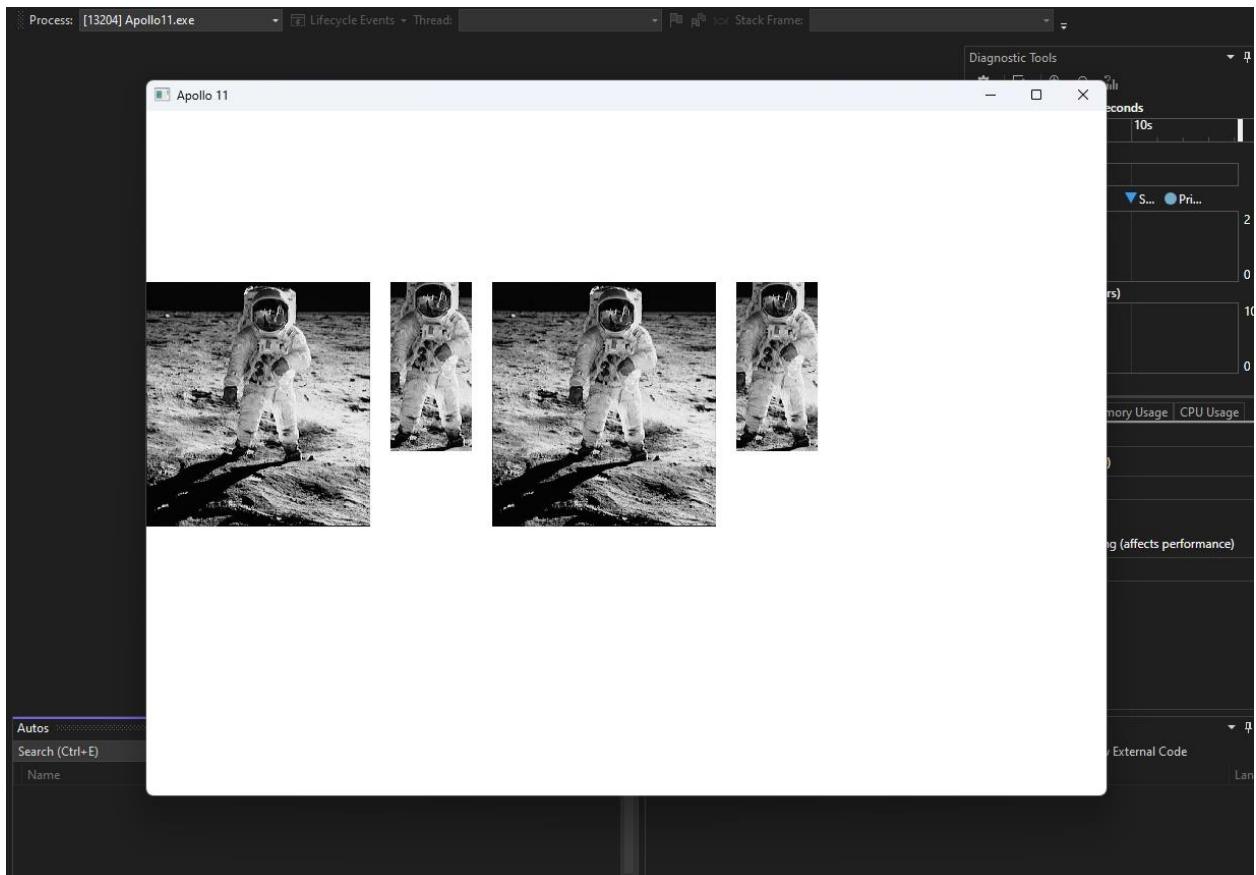
In MM_TEXT mapping mode, it's noted that the point (xSrc, ySrc) does not map directly to (xDst, yDst), which adds complexity to coordinate transformations. However, in other mapping modes, the point (xSrc, ySrc + cySrc - 1) will still map to (xDst, yDst), preserving the appearance of the image.

For top-down DIBs (when the biHeight field of BITMAPINFOHEADER is negative), source coordinates have an origin at the last row of the DIB data, making it unique compared to bottom-up DIBs. Here's how you specify a rectangle within a top-down DIB:

```
// Top-Down DIB Source Rectangle
// Top-left corner of the source rectangle
(xSrc, ySrc + cySrc - 1)
// Top-right corner of the source rectangle
(xSrc + cxSrc - 1, ySrc + cySrc - 1)
// Bottom-left corner of the source rectangle
(xSrc, ySrc)
// Bottom-right corner of the source rectangle
(xSrc + cxSrc - 1, ySrc)
```

This arrangement considers the peculiarities of top-down DIBs, where the origin is at the last row of the DIB data.

Apollo11 PROGRAM EXPLAINED

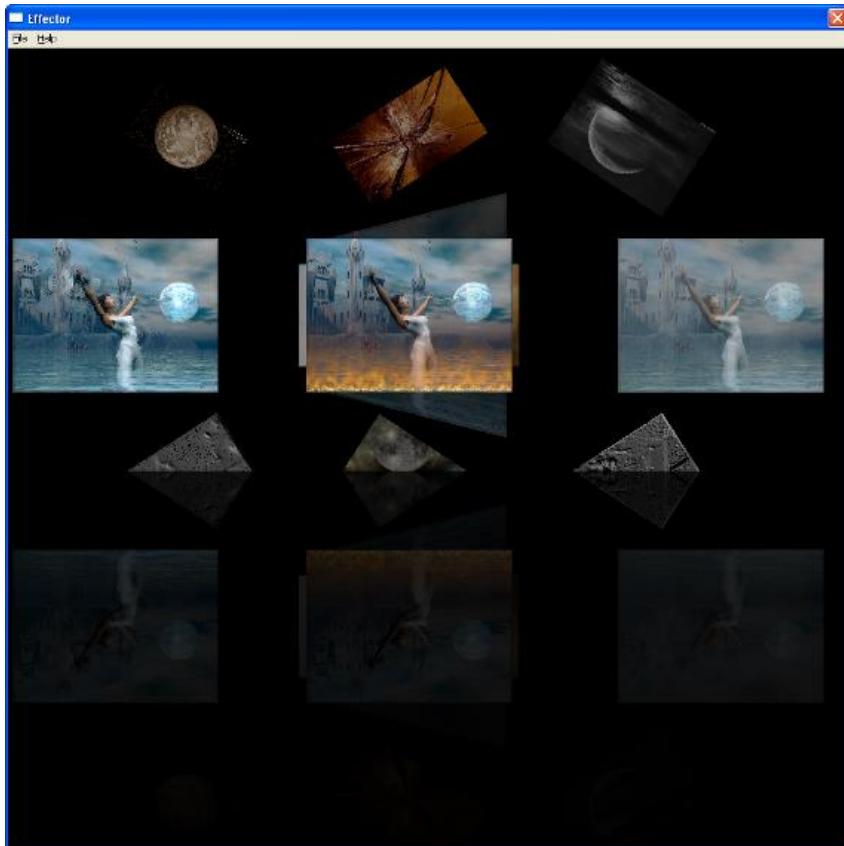


Before we explore the apollo11 program, here are a few scenarios where the SetDIBitsToDevice function can be useful:

Image Rendering: The SetDIBitsToDevice function is commonly used for rendering images on the screen or other output devices. It allows you to efficiently transfer pixel data from a DIB to the specified device context, taking into account the appropriate coordinates and mapping modes. This function is particularly useful when working with DIBs that have different orientations or row orders.



Bitmap Manipulation: The SetDIBitsToDevice function can be used for various bitmap manipulation tasks. For example, you can use it to copy a portion of one bitmap (specified by source rectangle coordinates) onto another bitmap (specified by destination rectangle coordinates). This enables you to perform operations like cropping, resizing, merging, or overlaying bitmaps.



Printing: When printing images or graphics, the SetDIBitsToDevice function can be utilized to transfer DIB data to the printer device context. By specifying the appropriate coordinates and mapping modes, you can accurately render the image on the printed page. This function is often used in conjunction with other printing-related functions to achieve high-quality output.



Image Conversion: The SetDIBitsToDevice function can be employed to convert DIBs between different formats or color spaces. By transferring the pixel data from one DIB to another with the desired format settings, you can perform color space conversions, bit depth adjustments, or even apply image processing algorithms before displaying or saving the modified DIB.



Custom Drawing: In certain cases, you may want to create custom drawings or graphics and directly transfer them to a device context. By constructing a DIB with the desired pixel data and using the SetDIBitsToDevice function, you can efficiently draw complex graphics, patterns, or user-generated content onto the screen or other output devices.



These are just a few examples of how the SetDIBitsToDevice function can be useful in different scenarios. Its versatility and ability to handle DIBs with varying orientations and row orders make it a valuable tool for image manipulation, rendering, and printing tasks.

```
300 //This is simplified code, showing only the important parts
301 #include <windows.h>
302 #include "dibfile.h"
303
304 LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
305
306 static BITMAPFILEHEADER *pbmfh[2];
307 static BITMAPINFO *pbmi[2];
308 static BYTE *pBits[2];
309 static int cxClient, cyClient, cxDib[2], cyDib[2];
310
311 void LoadDIBFiles(HWND hwnd)
312 {
313     pbmfh[0] = DibLoadImage(TEXT("Apollo11.bmp"));
314     pbmfh[1] = DibLoadImage(TEXT("ApolloTD.bmp"));
315
316     if (pbmfh[0] == NULL || pbmfh[1] == NULL)
317     {
318         MessageBox(hwnd, TEXT("Cannot load DIB file"), szAppName, 0);
319         PostQuitMessage(0);
320     }
321
322     pbmi[0] = (BITMAPINFO *) (pbmfh[0] + 1);
323     pbmi[1] = (BITMAPINFO *) (pbmfh[1] + 1);
324     pBits[0] = (BYTE *) pbmfh[0] + pbmfh[0]->bfOffBits;
325     pBits[1] = (BYTE *) pbmfh[1] + pbmfh[1]->bfOffBits;
326
327     cxDib[0] = pbmi[0]->bmiHeader.biWidth;
328     cxDib[1] = pbmi[1]->bmiHeader.biWidth;
329     cyDib[0] = abs(pbmi[0]->bmiHeader.biHeight);
330     cyDib[1] = abs(pbmi[1]->bmiHeader.biHeight);
331 }
332
333 void PaintDIBs(HWND hwnd, HDC hdc)
334 {
335     // Bottom-up DIB full size
336     SetDIBitsToDevice(hdc,
337                         0,                  // xDst
338                         cyClient / 4, // yDst
339                         cxDib[0], // cxSrc
340                         cyDib[0], // cySrc
341                         0,                  // xSrc
342                         0,                  // ySrc
343                         0,                  // first scan line
344                         cyDib[0], // number of scan lines
345                         pBits[0],
346                         pbmi[0],
347                         DIB_RGB_COLORS);
348
349     // Bottom-up DIB partial
350     SetDIBitsToDevice(hdc,
351                         240,                  // xDst
352                         cyClient / 4, // yDst
353                         80,                  // cxSrc
354                         166,                  // cySrc
355                         80,                  // xSrc
356                         60,                  // ySrc
357                         0,                  // first scan line
358                         cyDib[0], // number of scan lines
359                         pBits[0],
360                         pbmi[0],
361                         DIB_RGB_COLORS);
362 }
```

```

363 // Top-down DIB full size
364 SetDIBitsToDevice(hdc,
365     340,           // xDst
366     cyClient / 4, // yDst
367     cxDib[0],      // cxSrc
368     cyDib[0],      // cySrc
369     0,             // xSrc
370     0,             // ySrc
371     0,             // first scan line
372     cyDib[0],      // number of scan lines
373     pBits[0],
374     pbmi[0],
375     DIB_RGB_COLORS);
376
377 // Top-down DIB partial
378 SetDIBitsToDevice(hdc,
379     580,           // xDst
380     cyClient / 4, // yDst
381     80,            // cxSrc
382     166,           // cySrc
383     80,            // xSrc
384     60,            // ySrc
385     0,             // first scan line
386     cyDib[1],      // number of scan lines
387     pBits[1],
388     pbmi[1],
389     DIB_RGB_COLORS);
390 }
391
392 LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
393 {
394     HDC hdc;
395     PAINTSTRUCT ps;
396
397     switch (message)
398     {
399     case WM_CREATE:
400         LoadDIBFiles(hwnd);
401         return 0;
402
403     case WM_SIZE:
404         cxClient = LOWORD(lParam);
405         cyClient = HIWORD(lParam);
406         return 0;
407
408     case WM_PAINT:
409         hdc = BeginPaint(hwnd, &ps);
410         PaintDIBs(hwnd, hdc);
411         EndPaint(hwnd, &ps);
412         return 0;
413
414     case WM_DESTROY:
415         if (pbmfh[0])
416             free(pbmfh[0]);
417         if (pbmfh[1])
418             free(pbmfh[1]);
419
420         PostQuitMessage(0);
421         return 0;
422     }
423
424     return DefWindowProc(hwnd, message, wParam, lParam);
425 }

```

Yes, the code above contains the main parts of the program, including the LoadDIBFiles function, PaintDIBs function, and the WndProc function. Let's go through each of these functions in more detail:

LoadDIBFiles function:

This function is responsible for loading the DIB files ("Apollo11.bmp" and "ApolloTD.bmp") and extracting the necessary information from them.

It uses the DibLoadImage function from the "dibfile.h" library to load the DIB files and store the resulting BITMAPFILEHEADER structures in the pbmfh array.

If the loading of any DIB file fails (returns NULL), a message box is displayed, indicating the failure, and the program calls PostQuitMessage to post a quit message and terminate the application.

The function then extracts the BITMAPINFO structures and the pixel data (pBits) for each DIB from the pbmfh array.

Additionally, it stores the width and height of each DIB in the cxDib and cyDib arrays, respectively.

PaintDIBs function:

This function is responsible for painting the loaded DIBs onto the window's device context (hdc).

It uses the SetDIBitsToDevice function to transfer pixel data from the DIBs to the device context and display them on the window.

Four calls to SetDIBitsToDevice are made in this function, each displaying a different part of the DIBs at specified coordinates on the window.

The parameters of SetDIBitsToDevice are set accordingly to specify the destination position, source dimensions, source data, and color format for each call.

WndProc function:

This function is the window procedure and handles various messages sent to the window.

In the WM_CREATE message case, the LoadDIBFiles function is called to load and extract information from the DIB files.

In the WM_SIZE message case, the client area dimensions (cxClient and cyClient) are updated based on the new size of the window.

In the WM_PAINT message case, painting is initiated by obtaining the device context (hdc) using BeginPaint. Then, the PaintDIBs function is called to display the DIBs on the window. Finally, EndPaint is called to finish the painting process.

In the WM_DESTROY message case, the program frees the memory allocated for the loaded DIBs using free and calls PostQuitMessage to post a quit message and terminate the application.

This code sets up a window, loads DIB files, and displays different parts of the DIBs onto the window using the SetDIBitsToDevice function. The LoadDIBFiles function handles the loading of DIB files and extracting relevant information, while the PaintDIBs function performs the actual rendering of the DIBs onto the window. The WndProc function handles the window messages, including initialization, resizing, painting, and termination of the application.

The program is designed to load and display two DIBs: "APOLLO11.BMP" (in a bottom-up orientation) and "APOLLOTD.BMP" (in a top-down orientation). Both DIBs have a width of 220 pixels and a height of 240 pixels. It's worth noting that when determining the DIB's width and height from the header information structure, the program uses the abs function to obtain the absolute value of the biHeight field.

When displaying the DIBs, whether in full size or in partial views, the xSrc, ySrc, cxSrc, and cySrc coordinates remain the same regardless of which bitmap is being displayed. This means that the specific region of the DIB being shown is consistent across both DIBs.

The program utilizes the SetDIBitsToDevice function to transfer the pixel data from the DIBs to the device context, thereby displaying them on the window. The function is called four times within the PaintDIBs function, each with different parameters to display distinct portions of the DIBs at specified coordinates on the window.

By following this approach, the program successfully loads and renders the two DIBs onto the window, reflecting their respective orientations (bottom-up and top-down).

For a visual representation of the results mentioned above, you can refer to Figure 15-7 below.



When [working with the SetDIBitsToDevice function](#), it's important to keep in mind that certain arguments, such as the "first scan line" and "number of scan lines," should not be modified. It is also advised not to attempt to alter the pBits argument to point to a specific area of the DIB for display.

The [confusion surrounding these aspects of the Windows API](#) does not arise from a lack of effort on the part of developers but rather from inherent inconsistencies in the API's definition. If you find it confusing, it's because it is indeed confusing.

When reading statements in the [Windows documentation](#), such as the one for SetDIBitsToDevice, which states that "the origin of a bottom-up DIB is the lower-left corner of the bitmap; the origin of a top-down DIB is the upper-left corner," it's important to note that this statement is both ambiguous and incorrect.

A [clearer explanation would be as follows](#): The origin of a bottom-up DIB is the bottom-left corner of the bitmap image, representing the first pixel of the first row of bitmap data. On the other hand, the origin of a top-down DIB is also the bottom-left corner of the bitmap image, but in this case, it refers to the first pixel of the last row of bitmap data.

The [challenges become more pronounced](#) when writing functions to access individual bits of a DIB, particularly when consistency with specifying coordinates for displaying partial DIB images is required.

A [suggested solution](#), which will be implemented in a DIB library in Chapter 16, involves consistently referencing DIB pixels and coordinates as if the (0,0) origin corresponds to the leftmost pixel of the top row of the DIB image when it is correctly displayed.

MEMORY EFFICIENCY AND USE CASES:

Sequential display shines when memory conservation is crucial or when working with DIBs from slow sources. It's ideal for:

Handling large DIBs that might strain available memory.

Displaying DIBs as they're being acquired or transmitted, optimizing responsiveness.

Minimizing memory footprint in resource-constrained environments.

Mechanism Under the Hood:

SetDIBitsToDevice holds the key to sequential display. It accepts:

- **pBits:** A pointer to the pixel data to display.
- **yScan:** The starting row within the DIB.
- **cyScans:** The number of rows to display.

Multiple calls to this function, each with different yScan values, progressively reveal the DIB's content.

Memory Management Strategies:

To maximize efficiency, allocate memory strategically:

- Store the DIB's information section (BITMAPINFOHEADER and color table) for reference.
- Allocate memory for only a portion of the pixel data, adjusting based on the current display block.

This approach significantly reduces memory overhead compared to loading the entire DIB at once.

Trade-offs and Considerations:

Close Coupling: Sequential display often necessitates closer integration between data acquisition and display code. This might increase code complexity.

Performance Impact: Alternating between data acquisition and display can introduce overhead, potentially slowing down overall processing.

Functionality Limitations: Currently, only SetDIBitsToDevice supports sequential display. StretchDIBits lacks this feature, so displaying DIBs at different pixel sizes requires alternative approaches.

Alternatives for Rescaling DIBs:

While StretchDIBits doesn't directly support sequential display, explore workarounds:

Make multiple calls to StretchDIBits, each targeting a portion of the DIB.

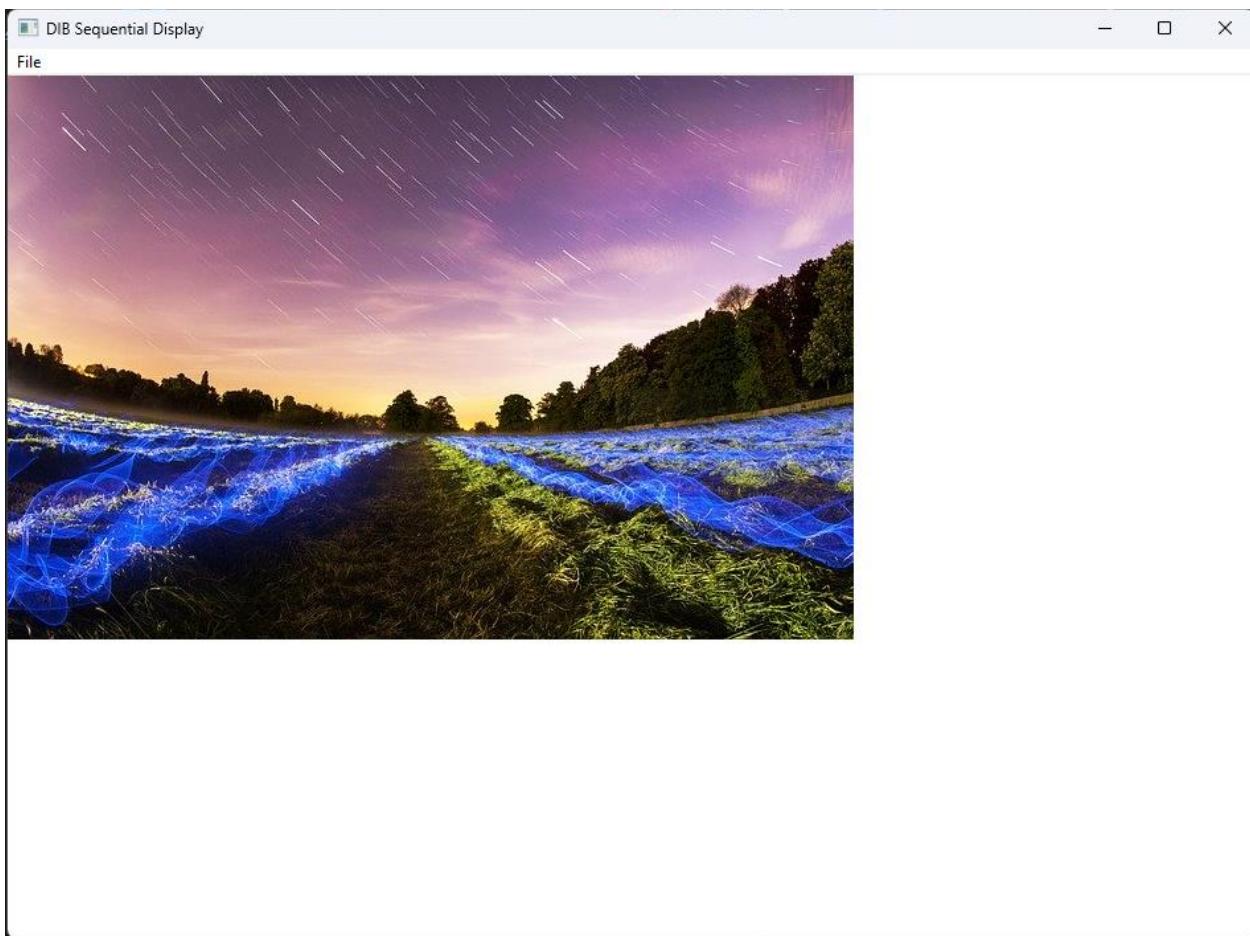
Modify the BITMAPINFOHEADER structure within each call to achieve the desired scaling effect.

Practical Demonstration:

The SEQDISP program, offers a hands-on example of sequential display implementation. Refer to it for practical guidance.

Sequential display provides a powerful technique for memory-efficient DIB handling.

Understanding its benefits, trade-offs, and implementation details empowers informed decisions when working with bitmap images in various scenarios.



The SEQDISP program, authored by Charles Petzold, is designed for the sequential display of Device Independent Bitmaps (DIBs). Upon opening a Bitmap (BMP) file, the program reads its header and pixel data, accommodating variations in DIB formats, including top-down orientation. It then sequentially displays the image row by row on the window, considering the DIB's specific characteristics such as width, height, and bit count. The program provides visual feedback during the file loading process and reverts to a standard arrow cursor upon completion. In contrast to the previous program, APOLLO11, which focused on displaying specific portions of DIBs and addressing complications in DIB handling, SEQDISP prioritizes the sequential display of entire DIBs, showcasing a different approach to handling and presenting bitmap images.

I then observed the bmp file being loaded from bottom to top very quickly?

The observed bottom-to-top loading behavior in the SEQDISP program is a consequence of the program's handling of top-down DIBs. In certain DIB formats, the pixel data starts from the last row (bottom) and progresses upward, contrary to the conventional bottom-up orientation where pixel data begins at the top. This peculiarity is particularly relevant when dealing with image data that is stored in a top-down manner, as indicated by the negative height in the DIB header. During the loading process, the program adjusts its display strategy to correctly render the image, resulting in the observed fast rendering from bottom to top. This behavior underscores the program's flexibility in accommodating different DIB formats and orientations.

Additionally, when opening a bitmap file in the SEQDISP program, you should note its ability to dynamically handle various bitmap formats and orientations. The program is designed to interpret the bitmap file's structure, accounting for both bottom-up and top-down pixel data arrangements. It efficiently reads the file, adapts to the specified DIB format, and displays the image appropriately on the window. The SEQDISP program showcases a robust implementation that accommodates the complexities of different bitmap configurations, offering a seamless visual representation regardless of the specific details encoded in the file.

Stretching DIBs to Fit with StretchDIBits

StretchDIBits allows you to stretch or shrink a DIB to fit a desired size on the output device.

It offers more flexibility than SetDIBitsToDevice, which only performs pixel-to-pixel display.

Syntax:

```
iLines = StretchDIBits(
    hdc,           // Device context handle
    xDst, yDst,   // Destination rectangle coordinates
    cxDst, cyDst, // Destination rectangle width and height
    xSrc, ySrc,   // Source rectangle coordinates
    cxSrc, cySrc, // Source rectangle width and height
    pBits,        // Pointer to DIB pixel bits
    pInfo,        // Pointer to DIB information
    fClrUse,      // Color use flag
    dwRop         // Raster operation
);
```

Key Differences from SetDIBitsToDevice:

Destination Rectangle Size: StretchDIBits includes parameters for destination rectangle width and height (cxDst, cyDst).

Sequential Display: StretchDIBits doesn't have the option to reduce memory usage by displaying the DIB sequentially.

Raster Operation: StretchDIBits includes a raster operation parameter (dwRop) to control how pixels are combined with the output device.

Signed Width and Height: Source and destination widths and heights (cxSrc, cySrc, cxDst, cyDst) can be negative to flip the DIB horizontally or vertically.

Mapping Modes and Flipping:

StretchDIBits is fully affected by the mapping mode of the destination device context.

You can use negative widths or heights to compensate for flipping effects caused by mapping modes.

```
// Stretch a 2x2 DIB to a 4x4 size:
StretchDIBits(hdc, 0, 0, 4, 4, 0, 0, 2, 2, pBits, pInfo, DIB_RGB_COLORS, SRCCOPY);
```

The -1 terms in the destination rectangle table are approximate and can vary depending on stretch and transforms.

The Sign functions in the expressions for flipping aren't standard C functions; their implementation is context-specific.

Flipping DIBs Using Mapping Modes and Width/Height:

Mapping Modes (xMM, yMM):

- Indicate the orientation of the mapping mode.
- xMM is 1 for values of x increasing to the right, -1 for increasing to the left.
- yMM is 1 for values of y increasing downward, -1 for increasing upward.

Flipping Expressions:

- `if (!Sign(xMM × cxSrc × cxDst))`: DIB is flipped on its vertical axis (mirror image).
- `if (!Sign(yMM × cySrc × cyDst))`: DIB is flipped on its horizontal axis (upside down).

You can also use negative widths or heights in `StretchDIBits` to flip the DIB intentionally.

Additional Considerations:

The `Sign` functions mentioned in the expressions for flipping aren't standard C functions. Their implementation is likely specific to the context in which they're used.

SHOWDIB2 PROGRAM:

The provided code is a Windows application written in C that displays Device Independent Bitmaps (DIBs) in various ways, such as normal, centered, stretched, or isotropically stretched. Let's break down the DIB-related portions of the code:

DIB Loading and Initialization:

The program utilizes functions from a file named "dibfile.h" for handling DIB operations. In the `WM_COMMAND` message handler, when the user selects "Open" from the File menu (`IDM_FILE_OPEN`), the program prompts the user to choose a DIB file using `DibFileOpenDlg`. It then loads the selected DIB into memory using `DibLoadImage`. The program checks whether the DIB is loaded successfully and retrieves pointers to the DIB's information structure (`pbmi`) and pixel data (`pBits`). The width and height of the DIB are obtained from the DIB header.

Displaying DIBs:

The actual rendering of the DIB on the window occurs in the WM_PAINT message handler. The ShowDib function is called, passing the device context (hdc), DIB information structure (pbmi), pixel data (pBits), DIB width and height (cxDib and cyDib), and client area dimensions (cxClient and cyClient). The way the DIB is displayed depends on the value of wShow, which is set based on the user's choice from the "Show" menu.

Printing DIBs:

If the user selects the "Print" option from the File menu (IDM_FILE_PRINT), the program prepares for printing. It checks whether the printer supports bitmaps and obtains the printer's device context. The ShowDib function is then called to print the DIB on the printer's page.

Copying and Cutting DIBs:

The program allows the user to copy (IDM_EDIT_COPY) or cut (IDM_EDIT_CUT) the DIB to the clipboard. The selected operation allocates a global memory block (hGlobal) and copies the packed DIB into it. The program then transfers this data to the clipboard, making it available for pasting into other applications.

Deleting and Freeing Resources:

If the user chooses to delete (IDM_EDIT_DELETE) the DIB, the allocated memory is freed, and the client area is invalidated for an update. Additionally, when the program is closed (WM_DESTROY), it checks if a DIB is loaded and frees the associated memory.

In the above explanations I only focused on the DIBs portion of the program, coz the rest is simple stuff. Let's see this program in a video...



Stretching bitmap
files.mp4

Displaying DIBs:

The ShowDib function is highlighted as a key element in the program, responsible for displaying the DIB in different ways based on menu selections. It uses SetDIBitsToDevice for normal and centered display and StretchDIBits for stretched and isotropically stretched options. The distinction between these modes affects how the DIB is presented in the client area.

Copying DIBs to Clipboard:

The process of copying a DIB to the clipboard is explained. It involves making a copy of the packed-DIB memory block in global shared memory, and the clipboard data type used is CF_DIB. However, the note mentions that the code does not show how to copy a DIB from the clipboard, indicating that this functionality may require additional logic. This is a common scenario when working with clipboard operations.

Deficiencies and Future Improvements:

The notes point out certain deficiencies in the program. For instance, there are potential issues when displaying DIBs in a 256-color video mode, where colors may not appear correctly. This is attributed to the need for handling the palette, which is a topic reserved for the next chapter. Additionally, a speed problem, especially in Windows NT, is acknowledged, and the resolution of this issue is promised in the upcoming chapter. The note also hints at addressing the display of larger DIBs with the addition of scroll bars.

Palette Handling and Color Display:

The mention of deficiencies related to 256-color video modes indicates that the program currently lacks proper handling for displaying colors in such environments. The note hints at the need to work with palettes to address this issue, suggesting that this aspect will be covered in more detail in the subsequent chapter.

Performance Considerations:

The note acknowledges a speed problem, particularly in Windows NT, without providing specific details. It promises a solution or optimization strategy in the next chapter. This indicates an awareness of potential performance issues and a commitment to addressing them in future discussions.

COLOR CONVERSION, PALETTES, AND PERFORMANCE IN DIB DISPLAY: A DEEP DIVE

Following the Pixel Bits: Understanding color conversion is crucial for optimizing bitmap display performance. The DIB format differs from video display memory, necessitating conversions during SetDIBitsToDevice or StretchDIBits calls. Millions of pixels might be involved, impacting performance significantly.

Trivial Conversions:

24-bit DIB on 24-bit display: minimal byte reordering if necessary.

16-bit DIB on 24-bit device: bit-shifting and padding.

24-bit DIB on 16-bit device: bit-shifting and truncation.

4/8-bit DIB on 24-bit device: color table lookup, potential byte reordering.

Challenging Conversions:

High-bit DIBs (16/24/32-bit) on 4/8-bit displays: nearest-color search for each pixel, involving loops and calculations. This is computationally expensive compared to simpler conversions.

Nearest-color search: finding the closest device color to the DIB pixel within the RGB color cube (represented as a 3D space). Distance metric involves complex calculations, although square root is not required for RGB comparisons.

8-bit DIB on 8-bit device: nearest-color search only needed for unique DIB colors, not every pixel.

Performance Optimization:

Avoid displaying high-bit DIBs on 4/8-bit displays using SetDIBitsToDevice/StretchDIBits.

Pre-convert DIBs to 8-bit DIBs or DDBs for faster BitBlt/StretchBlt display on any video mode.

DDBs offer significant performance boosts compared to DIBs for most display purposes.

8-bit Palette Limitations:

Windows in 8-bit mode restricts displayed DIBs to a maximum of 20 colors.

The Palette Manager (discussed later) plays a crucial role in exceeding this limit.

Windows NT Performance Differences:

Windows NT's client/server architecture can lead to slower DIB display compared to Windows 98 for similar video modes due to increased data transfer overhead.

Converting DIBs to DDBs or using CreateDIBSection (introduced later) can improve performance in such scenarios.

Key Takeaways:

Color conversion between DIBs and video memory significantly impacts performance.

Understanding conversion types and their complexities is crucial for optimization.

Pre-converting to lower bit depths (8-bit DIB or DDB) offers substantial performance gains.

Windows NT architecture requires specific considerations for optimal DIB display.

Let's take these notes a bit deeper...

To achieve optimal performance in bitmap display, it is crucial to understand the process of color conversion, especially when dealing with Device Independent Bitmaps (DIBs) and their presentation on device-dependent video displays.

The analogy to "Follow the pixel bits" is drawn, emphasizing the importance of tracking how color conversion occurs and when it takes place during functions like SetDIBitsToDevice or StretchDIBits.

The DIB exists in a device-independent format, while the video display memory is likely to be in another format.

During the mentioned function calls, each pixel, potentially numbering in the millions, must be converted from the device-independent format to the device-dependent format. The complexity of this conversion varies depending on the color depths involved.

For instance:

Displaying a 24-bit DIB on a 24-bit video display may require a simple rearrangement of the red, green, and blue bytes.

Presenting a 16-bit DIB on a 24-bit device involves bit-shifting and padding.

Displaying a 4-bit or 8-bit DIB on a 24-bit device requires a lookup in the DIB's color table and potential byte reordering.

However, when attempting to display a higher bit-depth DIB (e.g., 16-bit, 24-bit, or 32-bit) on a lower bit-depth video display (e.g., 4-bit or 8-bit), a different type of color conversion is necessary.

A nearest-color search must be performed for each pixel, involving a loop and a calculation. The RGB color space can be visualized as a cube, and finding the closest color involves determining the shortest distance between two points in this cube.

The distance formula between two colors R1G1B1 and R2G2B2 is given by:

$$\sqrt{(R2 - R1)^2 + (G2 - G1)^2 + (B2 - B1)^2}$$

However, in practice, the square root part of the calculation is often omitted for efficiency. Each pixel must be compared to all the colors on the device to find the closest match, making this process computationally intensive.

For performance reasons, the note advises against displaying high bit-depth DIBs on low bit-depth video displays using `SetDIBitsToDevice` or `StretchDIBits`.

Instead, conversion to an 8-bit DIB or, for even better performance, to an 8-bit Device Dependent Bitmap (DDB) is recommended. Converting most DIBs to DDBs and utilizing `BitBlt` and `StretchBlt` for display purposes can significantly enhance speed.

In situations where Windows is operating in an 8-bit video display, there may be limitations in displaying DIBs with their full range of colors, restricting them to just 20 colors. Overcoming this limitation involves utilizing the Palette Manager, a topic to be covered in the next chapter.

Additionally, the note addresses performance differences in displaying large DIBs between Windows 98 and Windows NT on the same machine.

Windows NT's client/server architecture incurs a penalty for handling large amounts of data across the API.

The recommended solution is, once again, to convert the DIB to a DDB. The `CreateDIBSection` function, discussed later, is introduced as a tool specifically designed to assist in this scenario.

DDBs AND DIBs MEET(union)

The advantages and limitations of working with Device-Independent Bitmaps (DIBs) and Device Dependent Bitmaps (DDBs) and suggests a possible solution to overcome these limitations.

Here's a deeper explanation:

DIBs offer direct access to every bit, byte, and pixel of the image, providing flexibility in examining and modifying the data. Functions like SetDIBitsToDevice and StretchDIBits allow you to draw DIBs on output devices by directly manipulating the pixel data.

However, there are some limitations to working with DIBs. Unlike DDBs, there doesn't appear to be a straightforward way to draw images directly on DIBs using GDI functions. In other words, you can't use functions like BitBlt and StretchBlt to draw on DIBs as efficiently as you can with DDBs.

Additionally, the performance of SetDIBitsToDevice and StretchDIBits functions may not be as fast as BitBlt and StretchBlt, especially on systems like Windows NT and when there are many nearest-color searches involved (e.g., displaying 24-bit DIBs on 8-bit video boards).

To overcome these limitations, one possible solution is to convert DIBs into DDBs. By converting a DIB into a DDB, you can take advantage of the faster BitBlt and StretchBlt functions for drawing and displaying the image.

Converting a DIB to a DDB involves creating a bitmap object that matches the characteristics of the DIB, such as size and color depth. Then, you can use functions like SetDIBitsToDevice to transfer the pixel data from the DIB to the DDB.

Once the DIB is converted to a DDB, you can use BitBlt and StretchBlt functions for efficient drawing and displaying operations on output devices.

Converting between DIBs and DDBs can be advantageous when you need to display a DIB multiple times or when performance is crucial, especially on systems where DIB drawing functions may be slower.

It's important to note that the decision to convert between DIBs and DDBs depends on specific requirements and trade-offs between flexibility and performance. Depending on the situation, you may choose to work directly with DIBs or opt for conversion to DDBs for improved drawing efficiency.

Creating a (DDB) from a (DIB)

Creating a Device-Dependent Bitmap (DDB) from a Device-Independent Bitmap (DIB) involves using the CreateDIBitmap function in Windows. There are two approaches to achieve this, and the syntax for the CreateDIBitmap function includes several parameters that dictate the characteristics of the resulting bitmap.

Using CreateCompatibleBitmap:

One way to create a DDB from a DIB is to use CreateCompatibleBitmap. This involves creating a GDI bitmap object of the same size as the DIB and compatible with the video display. The process includes the following steps:

- Use CreateCompatibleBitmap to create a GDI bitmap object.
- Select the bitmap object into a memory device context.
- Call SetDIBitsToDevice to draw on the memory DC, resulting in a DDB with the same image as the DIB but with color organization compatible with the display.

Using CreateDIBitmap:

Alternatively, the CreateDIBitmap function can be employed for the same purpose with fewer steps. The function's syntax is as follows:

```
hBitmap = CreateDIBitmap(
    hdc,           // device context handle
    pInfoHdr,     // pointer to DIB information header
    fInit,         // 0 or CBM_INIT
    pBits,         // pointer to DIB pixel bits
    pInfo,         // pointer to DIB information
    fClrUse       // color use flag
);
```

- `pInfoHdr` is a pointer to a BITMAPINFOHEADER structure.
- `pInfo` is a pointer to a BITMAPINFO structure (includes BITMAPINFOHEADER and color table).
- The `fClrUse` parameter can be DIB_RGB_COLORS or DIB_PAL_COLORS.

Notably, despite its name, CreateDIBitmap does not create a Device-Independent Bitmap; instead, it creates a Device-Dependent Bitmap from a Device-Independent specification. The function returns a handle to a GDI bitmap object, similar to other bitmap creation functions like CreateBitmap, CreateBitmapIndirect, and CreateCompatibleBitmap.

The simplest form of calling CreateDIBitmap involves passing only a pointer to a BITMAPINFOHEADER structure (without the color table), resulting in the creation of a monochrome GDI bitmap object.

```
hBitmap = CreateDIBitmap(NULL, pbmih, 0, NULL, NULL, 0);
```

Another form includes providing a device context (hdc), creating a DDB that is compatible with the specified device context.

```
hBitmap = CreateDIBitmap(hdc, pbmih, 0, NULL, NULL, 0);
```

In the simplified forms of CreateDIBitmap, where the third argument is CBM_INIT, the pixel bits of the resulting Device-Dependent Bitmap (DDB) remain uninitialized.

In this case, Windows uses the last three arguments of CreateDIBitmap to initialize the bitmap bits.

The pInfo argument is a pointer to a BITMAPINFO structure that includes a color table, and the pBits argument is a pointer to an array of bits in the color format indicated by the BITMAPINFO structure.

These bits are then converted to the color format of the device, similar to what happens in SetDIBitsToDevice.

The following code demonstrates how the entire CreateDIBitmap function could be implemented based on this logic:

```
440 HBITMAP CreateDIBitmap(
441     HDC hdc,
442     CONST BITMAPINFOHEADER *pbmih,
443     DWORD fInit,
444     CONST VOID *pBits,
445     CONST BITMAPINFO *pbmi,
446     UINT fUsage
447 )
448 {
449     HBITMAP hBitmap;
450     int cx, cy, iBitCount;
451     if (pbmih->biSize == sizeof(BITMAPCOREHEADER))
452     {
453         cx = ((PBITMAPCOREHEADER)pbmih)->bcWidth;
454         cy = ((PBITMAPCOREHEADER)pbmih)->bcHeight;
455         iBitCount = ((PBITMAPCOREHEADER)pbmih)->bcBitCount;
456     }
457     else
458     {
459         cx = pbmih->biWidth;
460         cy = pbmih->biHeight;
461         iBitCount = pbmih->biBitCount;
462     }
463     if (hdc)
464         hBitmap = CreateCompatibleBitmap(hdc, cx, cy);
465     else
466         hBitmap = CreateBitmap(cx, cy, 1, 1, NULL);
467     if (fInit == CBM_INIT)
468     {
469         HDC hdcMem = CreateCompatibleDC(hdc);
470         SelectObject(hdcMem, hBitmap);
471         SetDIBitsToDevice(hdcMem, 0, 0, cx, cy, 0, 0, 0, cy, pBits, pbmi, fUsage);
472         DeleteDC(hdcMem);
473     }
474     return hBitmap;
475 }
```

This code checks whether the bitmap header is of type BITMAPCOREHEADER or BITMAPINFOHEADER and then creates a compatible bitmap using CreateCompatibleBitmap or CreateBitmap.

If the initialization flag is set to CBM_INIT, it creates a compatible device context, selects the bitmap into the device context, and uses SetDIBitsToDevice for initialization.

It's crucial to note that if you're displaying a DIB only once and are concerned about the performance of SetDIBitsToDevice, using CreateDIBitmap followed by BitBlt or StretchBlt may not offer significant advantages.

Both SetDIBitsToDevice and CreateDIBitmap involve color conversion, making the overall time required comparable. However, if you intend to display the DIB multiple times, especially when processing WM_PAINT messages, the conversion performed by CreateDIBitmap can become more efficient.

DIBCONV PROGRAM

The DIBCONV program, depicted in Figure 15–10, provides an example of using SetDIBitsToDevice to convert a DIB file to a DDB, illustrating practical usage of the color conversion process.

Let's delve into the specific portion of the code that handles the conversion of a Device-Independent Bitmap (DIB) to a Device-Dependent Bitmap (DDB) in the DIBCONV.C program.

DIB to DDB Conversion Process:

1. File Open Dialog:

The application begins by initializing the Open File Dialog (ofn) in the WM_CREATE message. This dialog is later used to prompt the user for a Bitmap file.

2. File Open Command:

When the user selects the "Open" option from the File menu (IDM_FILE_OPEN), the program triggers the File Open Dialog. Once the user selects a Bitmap file, the program proceeds to convert the DIB to a DDB.

3. DIB File Reading:

The selected Bitmap file is opened, and its contents are read into memory. The program performs necessary validation checks, ensuring the file is a valid Bitmap file (BM signature in the BITMAPFILEHEADER and proper file size).

4. DIB to DDB Conversion:

The `CreateBitmapObjectFromDibFile` function is then called. This function takes the following steps:

- **File Handling:** The file is opened and read into memory. Various checks are performed to ensure the validity of the file, such as verifying the file type and size.
- **Create DDB:** Using the `CreateDIBitmap` function, a DDB is created based on the DIB data. The function initializes the DDB with the color information and pixel data from the DIB.

5. Painting the DDB:

Upon successful creation of the DDB, the program uses the `BitBlt` function to paint the DDB onto the client area of the window. This is done in the `WM_PAINT` message.

6. Error Handling:

If any issues occur during the file reading or DIB to DDB conversion process, appropriate error messages are displayed using message boxes.

7. Resource Definitions:

The resource file (DIBCONV.RC) contains definitions for the File menu and its items, including the "Open" option.

8. Resource Header Definitions:

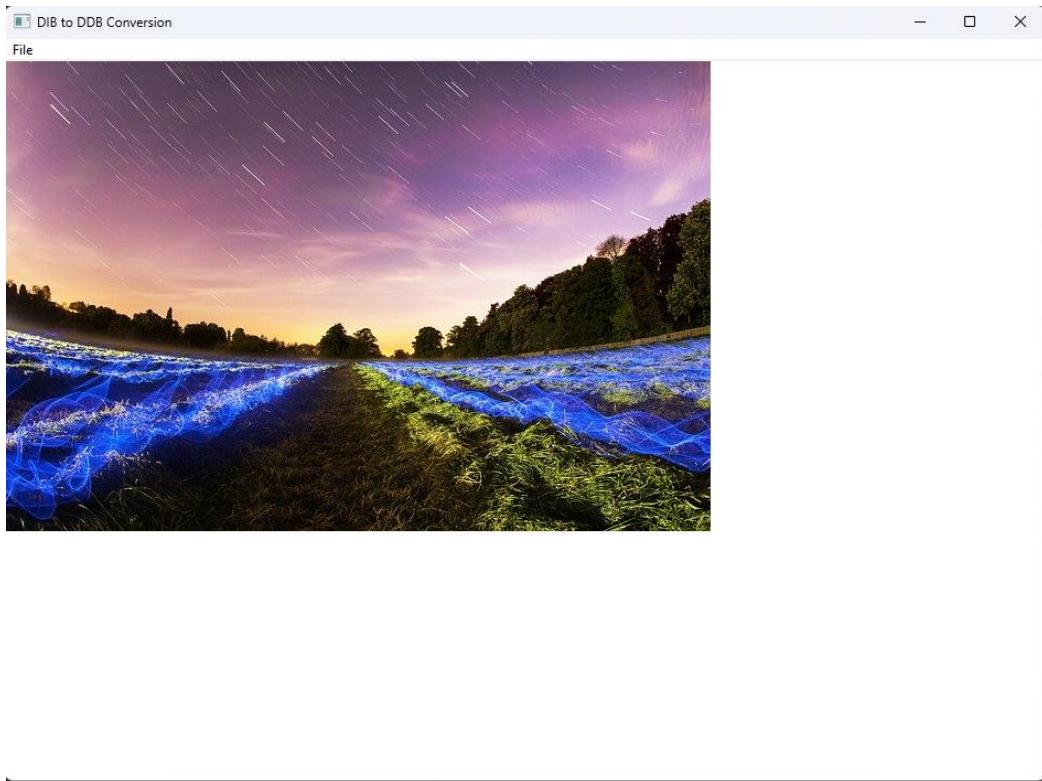
The resource header file (RESOURCE.H) includes symbolic constants for menu items, such as `IDM_FILE_OPEN`.

DIB to DDB Conversion Explanation:

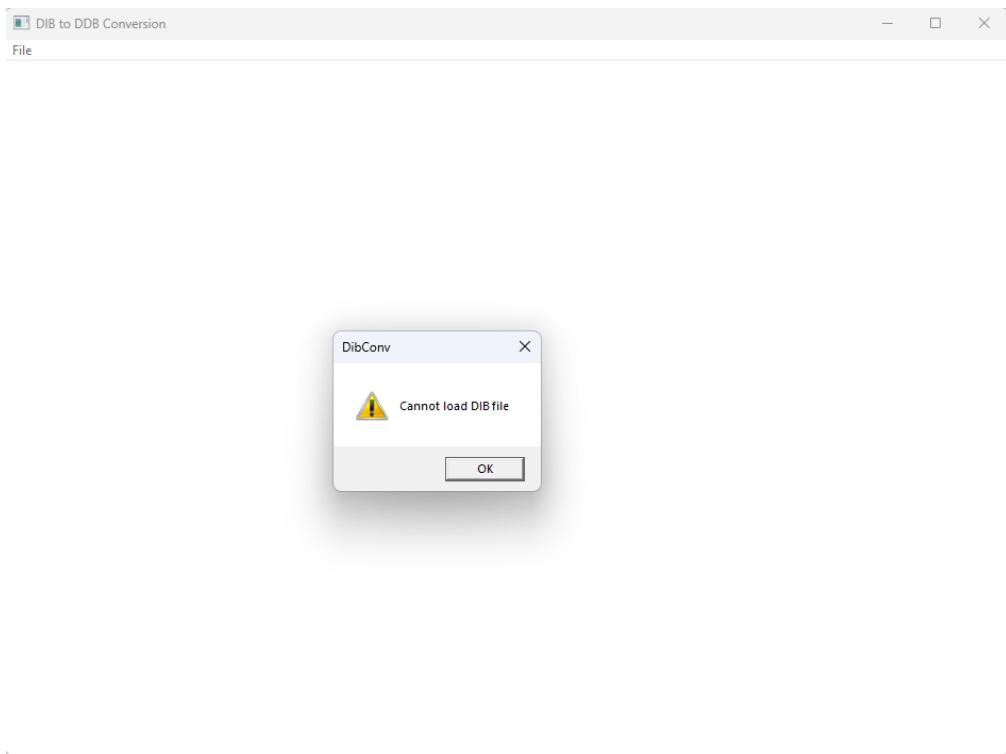
The conversion process involves creating a DDB from a DIB obtained from a Bitmap file. The crucial function, `CreateBitmapObjectFromDibFile`, encapsulates the steps to handle file I/O, validate the Bitmap file, and convert the DIB to a DDB using the `CreateDIBitmap` function.

The DDB is then painted onto the window's client area in response to the "Open" command. This conversion is vital for displaying images efficiently, as DDBs are device-dependent and more suitable for direct rendering on the screen. The process encapsulates essential concepts in handling image data in Windows applications.

.bmp file loads okay.



Other files don't load.



Why doesn't it load any other file apart from bmp files?

The limitation to loading only BMP files is inherent in the design and implementation of the CreateBitmapObjectFromDibFile function. Let's look at the relevant part of the code:

```
// Verify the file
if (!bSuccess || (dwBytesRead != dwFileSize) || (pbmfh->bfType != *(WORD *)"BM") || (pbmfh->bfSize != dwFileSize)) {
    free(pbmfh);
    return NULL;
}
// Create the DIB
hBitmap = CreateDIBitmap(hdc, (BITMAPINFOHEADER *)pbmfh + 1, CBM_INIT, (BYTE *)pbmfh + pbmfh->bfOffBits, (BITMAPINFO *)pbmfh + 1, DIB_RGB_COLORS);
free(pbmfh);
return hBitmap;
```

In this code, the [verification step checks whether the file is a valid BMP file](#) by comparing the bfType field in the BITMAPFILEHEADER structure with the expected value "BM".

If this check fails, the function immediately returns NULL, indicating that the file is not a valid BMP file.

For a more [general-purpose file loader](#) that can handle various image formats, you would need a more robust method of determining the file type and parsing the image data accordingly.

BMP files have a specific structure, and this code assumes that structure. To support other formats like JPEG, PNG, or GIF, you would need to implement additional logic to identify and handle the structure of those file types.

This typically involves reading the file headers and potentially using external libraries or APIs designed for image format recognition and decoding.

DIBCONV.C CONTINUED: A Self-Contained Image Conversion Program

DIBCONV.C stands as a self-contained program that necessitates no prior files for execution. Its main functionality lies in responding to the sole menu command, "File Open," where the WndProc function invokes the CreateBitmapObjectFromDibFile function.

This function, in turn, [reads the entire file into memory](#) and forwards pointers to the memory block to the CreateDIBitmap function. The function then returns a handle to the bitmap, allowing the memory block containing the Device-Independent Bitmap (DIB) to be subsequently freed.

During the handling of the WM_PAINT message, WndProc selects the bitmap in a compatible memory device context and employs BitBlt instead of SetDIBitsToDevice to display the bitmap on the client area. The dimensions of the bitmap are acquired by calling GetObject with the BITMAP structure on the bitmap handle.

Noteworthy is the fact that the initialization of the Device-Dependent Bitmap (DDB) pixel bits is unnecessary during the creation of the bitmap through CreateDIBitmap. Instead, this can be deferred and accomplished later by utilizing the SetDIBits function.

This function, presented below, accepts parameters for the device context handle, bitmap handle, starting scan line, the number of scan lines to convert, a pointer to pixel bits, DIB information, and a color use flag.

```
iLines = SetDIBits(
    hdc,          // Device context handle
    hBitmap,      // Bitmap handle
    yScan,        // First scan line to convert
    cyScans,      // Number of scan lines to convert
    pBits,        // Pointer to pixel bits
    pInfo,        // Pointer to DIB information
    fClrUse       // Color use flag
);
```

In this process, the function employs the color table in the BITMAPINFO structure to perform the conversion of bits into the device-dependent format.

The requirement for the device context handle is contingent upon whether the last argument is set to DIB_PAL_COLORS.

This approach allows for flexibility in initializing the pixel bits, offering control over the color conversion process in a manner conducive to the specific requirements of the application.

Breakdown of the SetDIBits function:

Purpose: Transfers color data (pixel bits) from a device-independent bitmap (DIB) to a device-dependent bitmap (DDB) within a specified device context (DC).

```
int SetDIBits(
    HDC hdc,          // Handle to the device context
    HBITMAP hBitmap, // Handle to the bitmap
    UINT yScan,       // First scan line to set
    UINT cyScans,     // Number of scan lines to set
    CONST VOID *pBits, // Pointer to the pixel bits
    CONST BITMAPINFO *pInfo, // Pointer to the bitmap information
    UINT fClrUse     // Color use flag
);
```

Parameters:

- **hdc:** The handle to the device context where the bitmap is displayed.
- **hBitmap:** The handle to the DDB receiving the pixel data.
- **yScan:** The starting scan line (row) within the bitmap to begin the transfer.
- **cyScans:** The number of scan lines to transfer.
- **pBits:** A pointer to the array of pixel bits within the DIB.
- **pInfo:** A pointer to a BITMAPINFO structure containing information about the DIB's format and colors.
- **fClrUse:** A flag indicating the color format of the DIB (usually DIB_RGB_COLORS or DIB_PAL_COLORS).

Return Value: The number of scan lines successfully transferred.

Safety Considerations:

- **Memory Management:** Ensure proper allocation and deallocation of memory for bitmap structures and pixel data to avoid memory leaks and potential security vulnerabilities.
- **Data Validation:** Validate the integrity of input data (e.g., pBits and pInfo) to prevent buffer overflows or invalid memory access.
- **Error Handling:** Implement robust error handling mechanisms to gracefully handle unexpected situations and prevent program crashes or data corruption.
- **Usage Within Scope:** Use SetDIBits only within its intended scope and avoid misuse for unintended purposes.

SetDIBits is primarily used for DIB-to-DDB transfers. For DDB-to-DIB transfers, consider GetDIBits. Always refer to official documentation for detailed usage guidelines and safety considerations.

Breakdown of the GetDIBits function

The GetDIBits function is similar to SetDIBits but serves the reverse purpose. It allows you to retrieve the pixel data and DIB information from a bitmap (DDB) and store it in a DIB.

```
int WINAPI GetDIBits(
    HDC hdc,           // Handle to the device context containing the DDB
    HBITMAP hBitmap,   // Handle to the DDB itself
    UINT yScan,        // First scan line to retrieve (0-based)
    UINT cyScans,      // Number of scan lines to retrieve
    LPVOID pBits,       // Pointer to a buffer to receive the pixel data
    LPBITMAPINFO pInfo, // Pointer to a BITMAPINFO structure to receive DIB information
    UINT fClrUse       // Color use flag (DIB_RGB_COLORS or DIB_PAL_COLORS)
);
```

The function takes several parameters:

- **hdc:** The handle to the device context associated with the bitmap.
- **hBitmap:** The handle to the bitmap.
- **yScan:** The index of the first scan line to retrieve.
- **cyScans:** The number of scan lines to retrieve.
- **pBits:** A pointer to the buffer where the pixel data will be stored (output).
- **pInfo:** A pointer to the structure where the DIB information will be stored (output).
- **fClrUse:** A flag indicating how to interpret color indices.

When converting a DDB to a DIB using GetDIBits, it's important to note that some information may be lost during the conversion. The amount of information lost depends on the specific video mode under which Windows is running during the conversion.

In general, it's not recommended to rely on GetDIBits as a way to obtain the original bitmap data if the bitmap handle is the only available information. This is because the conversion process from DDB to DIB and back may not yield the exact original bitmap due to potential information loss.

GetDIBits can be useful in situations such as screen captures, where you want to retrieve the pixel data and DIB information from the screen bitmap.

The function is not demonstrated in detail, but you can refer to Knowledge Base article Q80080 for more information on screen captures and the usage of GetDIBits.

It's worth noting that in many cases, automatic conversion to DIBs is provided by the clipboard, reducing the need to use GetDIBits for bitmap data retrieval.

GetDIBits is not a perfect reversal of SetDIBits. Converting a DIB to a DDB and back can lead to information loss due to device-specific optimizations and color space adjustments.

The actual information loss depends on the video mode and hardware capabilities.

GetDIBits is most useful for capturing screen data (like BLOWUP in Chapter 14) or specific scenarios where the original DIB information is unavailable.

The clipboard often provides automatic DIB conversion, making GetDIBits unnecessary for clipboard operations.

In summary, GetDIBits is a function that allows you to convert a DDB to a DIB by retrieving the pixel data and DIB information.

However, due to potential information loss during the conversion, it may not be suitable for obtaining the exact original bitmap. Its main application is in screen captures. Below is just a sample code to think about...

```

1 #include <Windows.h>
2 void ConvertDDBtoDIB(HBITMAP hBitmap)
3 {
4     HDC hdc = GetDC(NULL); // Get a handle to the screen device context
5     HDC hdcMem = CreateCompatibleDC(hdc); // Create a compatible device context
6     // Get the bitmap information
7     BITMAP bmp;
8     GetObject(hBitmap, sizeof(BITMAP), &bmp);
9     // Create a DIB section to store the pixel data
10    BITMAPINFO bmi = { 0 };
11    bmi.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
12    bmi.bmiHeader.biWidth = bmp.bmWidth;
13    bmi.bmiHeader.biHeight = bmp.bmHeight;
14    bmi.bmiHeader.biPlanes = 1;
15    bmi.bmiHeader.biBitCount = bmp.bmBitsPixel;
16    bmi.bmiHeader.biCompression = BI_RGB;
17    // Retrieve the pixel data from the DDB
18    BYTE* pPixels = nullptr;
19    HBITMAP hDibBitmap = CreateDIBSection(hdcMem, &bmi, DIB_RGB_COLORS, (void**)&pPixels, NULL, 0);
20    SelectObject(hdcMem, hDibBitmap);
21    GetDIBits(hdcMem, hBitmap, 0, bmp.bmHeight, pPixels, &bmi, DIB_RGB_COLORS);
22    // Release resources
23    DeleteDC(hdcMem);
24    ReleaseDC(NULL, hdc);
25    // Now you have a DIB stored in pPixels, which you can use as needed
26    // Clean up the DIB
27    DeleteObject(hDibBitmap);
28 }
29
30 int main()
31 {
32     // Assuming you already have an HBITMAP handle named hBitmap
33     ConvertDDBtoDIB(hBitmap);
34     return 0;
35 }

```

What are some alternative methods for obtaining the original bitmap data without relying on GetDIBits? Can you provide an example of how the clipboard automatically converts to DIBs? Are there any specific scenarios where the original DIB information is typically unavailable?

There are alternative methods for obtaining the original bitmap data without relying on GetDIBits. Here are a few:

Directly accessing the bitmap data: If you have access to the original bitmap creation code or the source from which the bitmap was loaded, you can retrieve the pixel data directly from the source without the need for conversion functions. This method ensures that you have the exact original bitmap data.

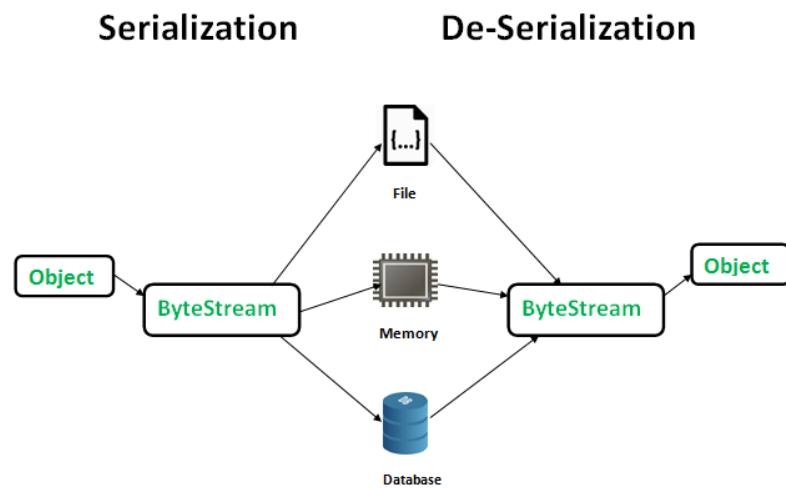


Storing the bitmap data during creation: When creating a bitmap, you can store a copy of the pixel data in a separate buffer or data structure. This way, you have access to the original bitmap data whenever needed, without relying on conversion functions.



Don't store it in a server please!! 🚫

Serializing and deserializing the bitmap: You can convert the bitmap data into a serialized format (such as a file or a binary stream) during creation. Later, when you need to retrieve the original bitmap data, you can deserialize it back into a bitmap object.



Regarding the automatic conversion of clipboard data to DIBs, when you place a bitmap on the clipboard, Windows automatically converts the bitmap into various formats, including DIBs. When you retrieve the bitmap data from the clipboard, you can request the data in DIB format using the CF_DIB or CF_DIBV5 format identifier. The clipboard API functions, such as OpenClipboard, GetClipboardData, and CloseClipboard, handle the conversion process internally.

There are scenarios where the original DIB information is typically unavailable. Some examples include:

Bitmaps received from external sources: If you receive a bitmap from an external source, such as via network communication, the source may not provide the original DIB information along with the bitmap. In such cases, you may need to rely on conversion functions like GetDIBits to obtain the pixel data and DIB information.



Bitmaps created or modified by third-party libraries or applications: If a bitmap is created or modified by a third-party library or application, the original DIB information may not be accessible to you. In these scenarios, you might need to resort to conversion methods like GetDIBits to retrieve the bitmap data.



It's important to consider the specific context and requirements of your application to determine the most appropriate method for obtaining the original bitmap data.

How can I directly access the bitmap data when creating a bitmap? Can you provide an example? Are there any other clipboard format identifiers besides CF_DIB and CF_DIBV5 that can be used to retrieve bitmap data? Explain how the serialization and deserialization process works for bitmap data?

Direct Access to Bitmap Data in Windows Programming: An Example

In Windows programming, direct access to bitmap data is often facilitated through the use of the GetDIBits and SetDIBits functions. These functions allow you to retrieve and set pixel data for a device-independent bitmap (DIB). Here's a basic example demonstrating how to create a bitmap, directly access its data, and then set it back:

```
1 #include <Windows.h>
2 void CreateAndAccessBitmap() {
3     // Create a bitmap (for demonstration, using a simple 100x100 monochrome bitmap)
4     HBITMAP hBitmap = CreateBitmap(100, 100, 1, 1, NULL);
5     // Get the device context of the bitmap
6     HDC hdc = GetDC(NULL);
7     HDC hdcMem = CreateCompatibleDC(hdc);
8     SelectObject(hdcMem, hBitmap);
9     // Get information about the bitmap
10    BITMAP bmp;
11    GetObject(hBitmap, sizeof(BITMAP), &bmp);
12    // Allocate memory for the pixel data
13    BYTE* pBits = new BYTE[bmp.bmHeight * bmp.bmWidthBytes];
14    // Retrieve the bitmap data
15    GetDIBits(hdcMem, hBitmap, 0, bmp.bmHeight, pBits, (BITMAPINFO*)&bmp, DIB_RGB_COLORS);
16    // Now you can directly manipulate the pixel data in the pBits array
17    // For demonstration, invert the colors
18    for (int i = 0; i < bmp.bmHeight * bmp.bmWidthBytes; ++i) {
19        pBits[i] = 255 - pBits[i];
20    }
21    // Set the modified data back to the bitmap
22    SetDIBits(hdcMem, hBitmap, 0, bmp.bmHeight, pBits, (BITMAPINFO*)&bmp, DIB_RGB_COLORS);
23    // Clean up
24    delete[] pBits;
25    DeleteDC(hdcMem);
26    ReleaseDC(NULL, hdc);
27    DeleteObject(hBitmap);
28 }
29
30 int main() {
31     CreateAndAccessBitmap();
32     return 0;
33 }
```

This example creates a monochrome bitmap, retrieves its pixel data using GetDIBits, modifies the data, and then sets it back using SetDIBits. The modification here is a simple inversion of colors for demonstration purposes.

Clipboard Format Identifiers for Bitmap Data:

Besides CF_DIB and CF_DIBV5, other common clipboard format identifiers related to bitmap data include:

- **CF_BITMAP:** A standard Windows bitmap format.
- **CF_PALETTE:** A handle to a color palette.

Serialization and Deserialization of Bitmap Data:

Serialization and deserialization of bitmap data involve converting the pixel data into a format suitable for storage or transmission and then reconstructing the bitmap from that serialized data. Common methods include encoding the bitmap as a file (e.g., BMP, PNG, JPEG) or using a custom format for transmission.

For instance, you can serialize a bitmap by saving it to a file:

```
void SerializeBitmap(HBITMAP hBitmap, const char* filename) {
    BITMAP bmp;
    GetObject(hBitmap, sizeof(BITMAP), &bmp);

    HDC hdc = GetDC(NULL);
    HDC hdcMem = CreateCompatibleDC(hdc);
    SelectObject(hdcMem, hBitmap);

    // Assuming BMP format for simplicity
    SaveDIBAsBMP(hdcMem, filename);

    ReleaseDC(NULL, hdc);
    DeleteDC(hdcMem);
}
```

Deserialization involves loading the serialized data back into a bitmap:

```
HBITMAP DeserializeBitmap(const char* filename) {
    // Assuming BMP format for simplicity
    return LoadImage(NULL, filename, IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE);
}
```

Serialization methods may vary based on the desired format and use case.

UNDERSTANDING DIB SECTIONS:

Understanding CreateDIBSection: A Deep Dive into Bitmap Creation

By now, we've delved into the disparities between device-dependent bitmaps (DDB) and device-independent bitmaps (DIB), appreciating their distinct characteristics and applications.

However, the [introduction of CreateDIBSection](#) in 32-bit versions of Windows brings about a function that seems to challenge the established norms. Let's dissect CreateDIBSection and demystify its peculiarities.

Hybrid Bitmap Type: A DIB Section blends aspects of both DDBs (device-dependent bitmaps) and DIBs (device-independent bitmaps).

Memory Access and GDI Handling: It provides direct memory access to pixel data like a DIB, while also being a GDI bitmap object like a DDB.

Creation via CreateDIBSection: This function generates a DIB Section, returning a bitmap handle.

Distinct Features:

Direct Memory Access: You can directly manipulate pixel data in memory, unlike with DDBs.

DIB Handle Flexibility: The handle can be used with GDI functions that typically require DDBs.

Potential Performance Benefits: Direct memory access can lead to performance improvements in certain scenarios.

Key Function Parameters:

pInfo: Pointer to a BITMAPINFO structure defining the DIB's characteristics.

ppBits: Pointer to a pointer variable that will receive the memory address of the pixel bits.

hSection and dwOffset: Optional parameters for sharing memory with other processes (advanced usage).

Usage Steps:

Define Variables: Create variables for BITMAPINFOHEADER, BYTE pointer, and bitmap handle.

Initialize BITMAPINFOHEADER: Set fields like width, height, bit count, and compression.

Call CreateDIBSection: Pass the BITMAPINFOHEADER and ppBits pointer, receiving the bitmap handle and pixel data memory address.

Access Pixel Data: Use the pBits pointer to directly access and manipulate pixel data.

Use Bitmap Handle: Employ the handle with GDI functions as needed.

Differences from CreateDIBitmap:

Handle Type: The handle returned by CreateDIBSection represents a DIB Section, while CreateDIBitmap creates a traditional DDB.

Pixel Data Access: CreateDIBSection offers direct memory access, whereas CreateDIBitmap involves GDI for pixel operations.

Additional Considerations:

Performance: Consider DIB Sections for performance-critical bitmap operations.

Multiple Processes: Use hSection and dwOffset for memory sharing between processes.

Error Handling: Implement error checking for CreateDIBSection calls.

Memory Management: Properly free allocated memory when finished using the DIB Section.

Let's go in-depth...

```
HBITMAP hBitmap = CreateDIBSection(
    hdc,           // Device context handle
    pInfo,         // Pointer to DIB information
    fClrUse,       // Color use flag
    ppBits,        // Pointer to pointer variable
    hSection,      // File-mapping object handle
    dwOffset       // Offset to bits in file-mapping object
);
```

CreateDIBSection, while crucial, may initially appear esoteric due to its unique characteristics. It stands out in bitmap handling functions by creating a "DIB section," essentially a memory block dedicated to storing pixel bits of a bitmap.

Return Value and Bitmap Handle:

The return value, `hBitmap`, is a handle to a GDI bitmap object. This is where the function deviates from expectations. Despite its similarity in name, `CreateDIBSection`'s return is fundamentally different from other bitmap-creation functions encountered so far.

Simplifying `CreateDIBSection`:

To simplify its use, you can set the last two arguments, `hSection` and `dwOffset`, to `NULL` and `0`, respectively, unless you delve into specific advanced scenarios discussed later.

The `hdc` parameter is relevant only when `fColorUse` is set to `DIB_PAL_COLORS`. If `fColorUse` is `DIB_RGB_COLORS` (or `0`), `hdc` is ignored.

In its basic form, `CreateDIBSection` requires only the second and fourth arguments. The second argument is a pointer to a `BITMAPINFO` structure.

The fourth argument, `ppBits`, might seem unusual—it's a pointer to a pointer variable. However, its usage becomes clear when you understand the function's purpose.

Practical Example:

Let's consider a scenario where you want to create a 384×256 -bit DIB with 24 bits per pixel.

In this case, the color format is 24-bit, simplifying the structure to a `BITMAPINFOHEADER`. Initialization involves setting the relevant fields, and then the function is called as follows:

```
BITMAPINFOHEADER bmih;
BYTE* pBits;
HBITMAP hBitmap;

// Initialize BITMAPINFOHEADER (details omitted for clarity)
// ...

// Create DIB section
hBitmap = CreateDIBSection(NULL, (BITMAPINFO*)&bmih, 0, &pBits, NULL, 0);
```

The function examines the structure, allocates memory for the DIB pixel bits, and stores a pointer to this memory block in `pBits`.

Simultaneously, it returns a handle to the bitmap. Importantly, the bitmap pixel bits are uninitialized at this point and require further handling.

Understanding `CreateDIBSection` involves recognizing its unique role in creating a dedicated memory section for bitmap pixel bits, a departure from conventional bitmap creation functions.

UNDERSTANDING DIB SECTION CREATION FROM DIB FILE IN DIBSECT.C PROGRAM:

File Loading and Validation:

The function `CreateDibSectionFromDibFile` is responsible for loading a DIB (Device-Independent Bitmap) file and creating a DIB Section from it. It begins by opening the file using `CreateFile`, ensuring it has read access and prohibiting write access. The function validates the file format by checking the `BITMAPFILEHEADER` for the "BM" signature.

Memory Allocation for BITMAPINFO:

The DIB file contains information about the bitmap's characteristics, stored in the `BITMAPINFO` structure. The function allocates memory for this structure based on the information read from the file. It checks for successful allocation and reads the structure from the file.

DIB Section Creation:

With the `BITMAPINFO` structure obtained, the function proceeds to create a DIB Section using `CreateDIBSection`. This function allocates memory for the bitmap pixel bits and returns a handle to the DIB Section. The `DIB_RGB_COLORS` flag indicates the use of RGB color values in the color table.

Reading Bitmap Bits:

After successfully creating the DIB Section, the function reads the actual bitmap bits from the file and stores them in the allocated memory. The size of the bits is determined by subtracting the offset to the bits (`bfOffBits`) from the total file size (`bfSize`).

Cleaning Up:

Finally, the function frees the memory allocated for the `BITMAPINFO` structure and closes the file handle. The resulting DIB Section handle is returned.

Window Procedure (WndProc) in DIBSECT.C:

File Open Command:

In the window procedure (WndProc), the WM_COMMAND message is handled to respond to user commands. When the "Open" menu item is selected (IDM_FILE_OPEN), the File Open dialog is displayed (GetOpenFileName). If a valid file is chosen, the existing bitmap is deleted, and a new DIB Section is created from the selected file using the previously discussed function.

Painting the Client Area:

In the WM_PAINT message, the program paints the client area by creating a compatible device context (hdcMem) and using BitBlt to transfer the content of the DIB Section to the screen.

Cleanup on Window Close:

When the window is closed (WM_DESTROY), the DIB Section handle is deleted to prevent memory leaks.

The provided code primarily focuses on displaying a loaded BMP file within the application window. However, understanding its underlying functionality can be a valuable foundation for further exploration and customization. Here are some ways you can work with the code beyond merely opening a BMP file:

1. Explore different file formats:

Modify the CreateDibSectionFromDibFile function to handle additional image formats like GIF, PNG, or JPEG by utilizing appropriate libraries or Windows codecs.

Update the file filter string (szFilter) in the OPENFILENAME structure to include relevant extensions for these formats.

2. Implement image manipulation:

Leverage direct access to the bitmap data within the DIB Section to manipulate pixel values.

You can write code to apply filters, modify colors, rotate the image, or perform other image processing operations.

3. Integrate with other functionalities:

Combine this code with other features like image saving, user input handling, or network communication to create more interactive or complex applications.

Develop tools for image editing, basic animation, or image sharing.

4. Further customization:

Modify the window appearance and behavior beyond its current basic structure.

Add UI elements like buttons, sliders, or text boxes for user interaction and control over the displayed image or processing options.

5. Research and learn:

Dive deeper into Windows graphics APIs and image processing libraries to unlock further capabilities and techniques.

Explore resources like MSDN documentation, tutorials, and code examples to expand your knowledge and skillset in working with bitmap data and image manipulation.

In summary, the program allows users to open a DIB file, creates a DIB Section from it, and displays the resulting image in the client area. The handling of file loading, DIB Section creation, and proper cleanup are essential aspects of the implementation.

COMPARISON OF DIBCONV AND DIBSECT FUNCTIONS:

In the analysis of the `CreateBitmapObjectFromDibFile` function in DIBCONV and the `CreateDibSectionFromDibFile` function in DIBSECT, distinct differences emerge in their approaches to handling DIB files and creating bitmap objects. Let's delve into the nuanced disparities:

File Reading and Memory Allocation:

DIBCONV (CreateBitmapObjectFromDibFile): Reads the entire file in one step and passes pointers to the DIB memory block to `CreateDIBitmap`.

DIBSECT (CreateDibSectionFromDibFile): Reads the `BITMAPFILEHEADER` first, determines the size of the `BITMAPINFO` structure, allocates memory for it, and then reads it in a second `ReadFile` call. Passes pointers to the `BITMAPINFO` structure and the pointer variable `pBits` to `CreateDIBSection`.

Memory Ownership and System Control:

DIBCONV: The DIB memory block is implicitly owned by the system as it is handed over to `CreateDIBitmap`. The system controls the memory.

DIBSECT: The memory block pointed to by `pBits` is owned by the system. The system automatically frees this memory when the bitmap is deleted using `DeleteObject`. However, the program retains the capability to alter DIB bits directly through the `pBits` variable.

Accessing and Modifying DIB Bits:

DIBCONV: The system retains exclusive control over the DIB bits, limiting direct modification by the application.

DIBSECT: The application can use the pointer `pBits` to directly alter DIB bits, providing a significant advantage in scenarios where manual manipulation of pixel data is necessary.

Format Conversion and Bitmap Handle Differences:

DIBCONV: The format conversion, if needed, occurs during the `CreateDIBitmap` call.

DIBSECT: The format conversion occurs during subsequent `BitBlt` or `StretchBlt` calls. The bitmap handle returned from `CreateDIBSection` references a DIB stored in system-managed memory that undergoes conversion when displayed.

Handling Bitmap Handle Differences:

DIBCONV: The bitmap handle returned has typical behavior, and any format conversion is mostly handled during creation.

DIBSECT: The bitmap handle from CreateDIBSection references a DIB in system memory with application access. Direct drawing on it reflects changes in the DIB pixel bits through the pBits variable.

Batching GDI Calls and Manual Access:

DIBSECT: When drawing on the memory device context, GDI calls are batched under Windows NT. It's advisable to call GdiFlush after drawing before manually accessing the bits.

Retaining Access to pBits:

DIBSECT: If direct bit manipulation is necessary, retaining the pBits variable is crucial. Once discarded, there seems to be no later way to obtain the bits pointer after the CreateDIBSection call.

In essence, while both functions achieve the creation of bitmap objects from DIB files, **DIBCONV** relies on **CreateDIBitmap** for simplicity, **DIBSECT**, utilizing **CreateDIBSection**, offers more control over DIB bits and their direct manipulation. The differences in memory ownership and system access highlight the nuanced capabilities of each approach.

Color Organization Discrepancy:

CreateDIBitmap: The bitmap handle shares the color organization with the device specified by the hdc parameter.

CreateDIBSection: The bitmap handle reflects the color organization specified in the BITMAPINFOHEADER structure. Despite this, the handle can be selected into a memory device context compatible with the video display, allowing for versatile usage.

Byte Length Anomaly in Rows:

CreateDIBSection: When using a BITMAPINFOHEADER with 24 bits per pixel and a width of 2 pixels, the bmWidthBytes field obtained from GetObject may not align with the conventional rules. For example, it might be 8 instead of the expected 6. This discrepancy stems from the distinct nature of DIB section handles.

Accessing Bitmap Information with DIBSECTION:

[CreateDIBSection](#): Introduces the DIBSECTION structure, retrievable via GetObject with a specific format:

```
typedef struct tagDIBSECTION {  
    BITMAP dsBm;                      // BITMAP structure  
    BITMAPINFOHEADER dsBmih;           // DIB information header  
    DWORD dsBitfields[3];              // Color masks  
    HANDLE dshSection;                // File-mapping object handle  
    DWORD dsOffset;                   // Offset to bitmap bits  
} DIBSECTION, *PDIBSECTION;
```

This structure combines both BITMAP and BITMAPINFOHEADER information, providing comprehensive details about the DIB section.

Color Table Access:

[CreateDIBSection](#): Enables extraction of the color table by utilizing the GetDIBColorTable function. This is particularly useful when the DIB section handle is selected into a memory device context. Example:

```
HDC hdcMem = CreateCompatibleDC(NULL);  
SelectObject(hdcMem, hBitmap);  
GetDIBColorTable(hdcMem, uFirstIndex, uNumEntries, &rgb);  
DeleteDC(hdcMem);
```

Similarly, modifications to the color table can be achieved using [SetDIBColorTable](#).

In summary, the [peculiarities of DIB section bitmap handles](#), especially those returned from CreateDIBSection, offer a unique blend of color flexibility, information encapsulation through DIBSECTION, and nuanced byte alignment in rows. These features empower developers with greater control and insights into the intricacies of bitmap manipulation and presentation.

FILE-MAPPING APPROACH IN CREATEDIBSECTION: ADDRESSING LIMITATIONS

The concept of file mapping in conjunction with CreateDIBSection is an intriguing avenue for optimizing memory usage, especially for large DIBs.

However, a key limitation arises from the requirement that the offset (dwOffset) within the file mapping object must be a multiple of the size of a DWORD.

Unfortunately, the size of the bitmap file header (BITMAPFILEHEADER) is not always a multiple of 4, leading to complications.

Here is an in-depth exploration of the file-mapping option and its intricacies:

File Mapping Overview:

A file-mapping object treats a file as if it were in memory, allowing efficient access without loading the entire file into RAM.

This approach is particularly beneficial for large DIBs, where the pixel bits can be kept on disk but still accessed as if in memory.

Function Description:

The function CreateDibSectionMappingFromFile attempts to create a DIB section using file mapping and an offset within the file.

It employs CreateFileMapping to establish a file-mapping object, and CreateDIBSection is called with this mapping and the specified offset.

Limitation and DwOffset Constraint:

CreateDIBSection mandates that dwOffset must be a multiple of the size of a DWORD.

The size of the bitmap file header (BITMAPFILEHEADER) is 14 bytes, making bmfh.bfOffBits not always a multiple of 4.

Complication and Resolution:

Due to the non-compliance of `bmfh.bfOffBits` with the required multiple-of-4 constraint, the straightforward implementation encounters issues.

To address this, a potential solution involves aligning `bmfh.bfOffBits` to the nearest multiple of 4 before passing it as the `dwOffset`.

Here is a refined version of the function considering the alignment constraint:

```
HBITMAP CreateDibSectionMappingFromFile(PTSTR szFileName) {
    // ... (previous function code)

    // Align bfOffBits to the nearest multiple of 4
    DWORD dwAlignedOffset = (bmfh.bfOffBits + 3) & ~3;

    // Create the DIB Section with the aligned offset
    hBitmap = CreateDIBSection(NULL, pbmi, DIB_RGB_COLORS, &pBits, hFileMap, dwAlignedOffset);

    // ... (rest of the function code)

    return hBitmap;
}
```

By aligning `bmfh.bfOffBits` to the nearest multiple of 4 using `(bmfh.bfOffBits + 3) & ~3`, we adhere to the constraints imposed by `CreateDIBSection` and improve the likelihood of success in creating a DIB section from a file mapping object.

Key Points:

File-Mapping Option: Allows treating a file as memory-mapped, reducing memory usage for large DIBs.

Challenge for DIB Files: Offset to pixel bits (`bfOffBits`) within a DIB file isn't guaranteed to be a multiple of 4, as required by `CreateDIBSection`'s `dwOffset` argument.

Solution: Use a separate file containing only pixel bits, ensuring a DWORD-aligned offset.

```

1 HBITMAP CreateDibSectionMappingFromSeparateFile(PTSTR szFileName) {
2     BITMAPFILEHEADER bmfh;
3     BITMAPINFO *pbmi;
4     BYTE *pBits;
5     BOOL bSuccess;
6     DWORD dwInfoSize, dwBytesRead;
7     HANDLE hFile, hFileMap;
8     HBITMAP hBitmap;
9
10    hFile = CreateFile(szFileName, GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);
11    if (hFile == INVALID_HANDLE_VALUE)
12        return NULL;
13
14    // Read only BITMAPFILEHEADER and BITMAPINFO structures
15    bSuccess = ReadFile(hFile, &bmfh, sizeof(BITMAPFILEHEADER), &dwBytesRead, NULL);
16    if (!bSuccess || (dwBytesRead != sizeof(BITMAPFILEHEADER)) || (bmfh.bfType != *(WORD *)"BM")) {
17        CloseHandle(hFile);
18        return NULL;
19    }
20
21    dwInfoSize = bmfh.bfOffBits - sizeof(BITMAPFILEHEADER);
22    pbmi = malloc(dwInfoSize);
23    bSuccess = ReadFile(hFile, pbmi, dwInfoSize, &dwBytesRead, NULL);
24    if (!bSuccess || (dwBytesRead != dwInfoSize)) {
25        free(pbmi);
26        CloseHandle(hFile);
27        return NULL;
28    }
29
30    // Create file mapping for a separate file containing only pixel bits
31    // Assuming a separate file with DWORD-aligned pixel data exists
32    hFileMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, NULL); // Replace with appropriate file path
33
34    hBitmap = CreateDIBSection(NULL, pbmi, DIB_RGB_COLORS, &pBits, hFileMap, 0); // Offset is 0 for separate file
35
36    free(pbmi);
37    return hBitmap;
38}

```

Separate File Preparation: Ensure the separate file containing pixel bits has a DWORD-aligned offset for correct mapping.

Performance: File-mapping might introduce performance overhead, so evaluate its benefits against memory requirements.

Error Handling: Implement robust error handling for file operations and function calls.

Memory Management: Free allocated memory and close handles appropriately.

UNDERSTANDING PERFORMANCE PENALTIES AND ALTERNATIVES FOR ACCESSING DIB PIXEL BITS FROM DISK

1. Potential Performance Penalties:

Utilizing a file-mapping object to access DIB pixel bits from disk introduces potential performance penalties.

Disk I/O operations are inherently slower than memory access, leading to increased latency in reading pixel data.

Large DIBs may result in frequent disk reads, impacting real-time processing and responsiveness.

2. Alternative Methods for Accessing DIB Pixel Bits:

Memory-Mapped Files: Instead of direct file mapping, consider memory-mapped files. This approach allows the operating system to manage caching and paging, potentially improving performance.

Buffered Reading: Implement buffered reading techniques to minimize the number of disk reads. Load portions of the DIB incrementally, reducing the impact on performance.

Code Illustration - Memory-Mapped Files:

```
1  HBITMAP CreateDibSectionMappingFromFile_MemoryMapped(PTSTR szFileName)
2  {
3      // ... (previous function code)
4
5      // Create a memory-mapped file
6      hFileMap = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
7      if (hFileMap == NULL)
8      {
9          free(pbmi);
10         CloseHandle(hFile);
11         return NULL;
12     }
13
14     // Create DIB section with memory-mapped file
15     hBitmap = CreateDIBSection(NULL, pbmi, DIB_RGB_COLORS, (void**)&pBits, hFileMap, alignedOffset);
16
17     // ... (rest of the function code)
18
19     return hBitmap;
20 }
```

3. Limitations and Considerations:

Alignment Requirement: Ensure proper alignment of the offset to a multiple of 4 to meet CreateDIBSection's constraint.

Memory Usage: Memory-mapping large DIBs may consume substantial virtual memory, affecting overall system performance.

Disk Contention: Concurrent disk access by multiple processes may lead to contention, impacting performance.

File Locking: Consider file locking mechanisms to prevent inconsistencies in multi-process scenarios.

In-Depth Code Explanation:

The modified code aligns the offset to meet CreateDIBSection requirements.

Introduces memory-mapped file usage as an alternative for potential performance improvement.

Addresses considerations related to memory usage, disk contention, and file locking.

By considering alternative methods and being mindful of limitations, you can optimize DIB access from disk while mitigating potential performance penalties.

SUMMARY: CHOOSING THE RIGHT APPROACH FOR DIBS IN WINDOWS PROGRAMMING

When working with Device-Independent Bitmaps (DIBs) in Windows programming, the choice of method depends on the size of the DIB and the desired manipulations. Here's a summary of the approaches discussed:

Small DIBs and Frequent Pixel Manipulations:

Method: Use SetDIBitsToDevice and StretchDIBits for frequent pixel manipulations.

Consideration: Suitable for small DIBs, but performance issues may arise, especially on 8-bit video displays and under Windows NT.

Conversion to DDB (Device-Dependent Bitmap):

Method: Convert a DIB to a DDB using CreateDIBitmap and SetDIBits.

Advantages: Faster display using BitBlt and StretchBlt.

Drawback: No direct access to device-independent pixel bits after conversion.

CreateDIBSection - A Balanced Compromise:

Method: Use CreateDIBSection for a compromise between performance and pixel bit access.

Advantages: Better performance under Windows NT compared to SetDIBitsToDevice and StretchDIBits. Retains access to DIB pixel bits.

Consideration: The bitmap handle references a DIB stored in system-managed memory, allowing direct pixel bit access.

In Conclusion:

For small DIBs with frequent manipulations, SetDIBitsToDevice may suffice but could face performance issues.

Converting to a DDB with CreateDIBitmap provides faster display but sacrifices direct access to pixel bits.

CreateDIBSection strikes a balance, offering good performance under Windows NT, efficient display with BitBlt and StretchBlt, and continued access to DIB pixel bits.

Next Chapter 16:

The exploration of bitmaps will conclude in the next chapter with a focus on the Windows Palette Manager. This will provide a comprehensive understanding of bitmap handling in Windows programming.

End of Chapter 15...the largest chapter