

CHAPTER 15: THE DEVICE-INDEPENDENT BITMAP

From DDB to DIB (Why DIBs Exist)

In Windows GDI, a bitmap is usually a **Device-Dependent Bitmap (DDB)**.

A DDB is made for **drawing to the screen**.

It works fast — but it has a big problem.

A DDB depends on the **current display device**.

If you save it to disk and load it later, or move it to another computer:

- colors may change
- pixel format may break
- it may not load correctly

So DDBs are **bad for storing images**.

To solve this, Windows introduced the **Device-Independent Bitmap (DIB)** in Windows 3.0.

A DIB is made for **storage, memory, and exchange**, not just drawing.

DDB vs DIB (Simple Comparison)

I. DDB (Device-Dependent Bitmap)

- Tied to the display device
- Format depends on video hardware
- Not safe to save or reload
- Good only for fast drawing

II. DIB (Device-Independent Bitmap)

- Independent of the display
- Stores pixels + color info together
- Looks the same everywhere
- Safe to save, load, and share

III. Short version:

- DDB → screen
- DIB → memory and files

Why DIBs Are Useful

I. Built for the Windows API

Windows understands DIBs directly.

You can:

- load them into memory
- modify pixels
- display them
- convert them to DDBs for drawing

No external image decoders.

II. Lossless Pixel Data

DIBs are usually **uncompressed**.

That means:

- no quality loss
- exact pixel values

This matters for:

- image editing
- analysis
- learning how graphics really work

III. Full Control

DIBs store **raw pixel memory**.

You can:

- edit pixels directly
- change colors or palettes
- write your own image algorithms

This is hard to do with formats like JPEG or GIF.

IV. Key Takeaway

- **DDB** → fast drawing, bad for saving
- **DIB** → best for storing and manipulating images
- DIBs trade file size for **control and correctness**

That's why serious GDI image work always ends up using DIBs.

DIBs Today

DIBs are not made for the web — and they don't need to be.

They are used when writing:

- image editors
- graphics or CAD tools
- visualization software
- low-level WinAPI programs

Why?

- native Windows support
- raw pixel access
- predictable behavior

Big files. Old format. Still powerful.

I. DIB File Format (Just the Facts)

- Origin: OS/2 1.1 (Presentation Manager bitmap)
- Windows: adopted in Windows 3.0 (1990)
- Never replaced, never removed

This is why WinAPI graphics still feel **bitmap-first**.

II. Core Characteristics

- Extensions: .BMP, .DIB
- Color data stored inside the bitmap
- Stored in memory as a **packed DIB**
(header + color table + pixels)
- Strong WinAPI support
- Easy to process manually

This is why low-level devs and reverse engineers like it.

III. Where DIBs Are Used

- Application images
- Icons and cursors
- Clipboard image transfer
- Custom GDI brushes
- Image processing and experiments

If you touch **GDI**, **BitBlt**, **StretchBlt**, or **raw pixels**, you're already using DIBs.

IV. Remember this

A DIB is:

A block of memory that Windows understands as pixels.

No compression.

No surprises.

No device dependency.

V. File Structure: How the Data is Organized

A DIB file (like a .BMP) is organized into four main parts:

- **File Header (14 bytes):** The "ID card" of the file. It tells the computer this is a bitmap, how big the file is, and where the actual image data starts.
- **DIB Header:** This contains the image details. It stores the width, height, how many colors it uses, and if the image is compressed.
- **Color Table (Optional):** Used if the image has a limited palette (like 256 colors). It's a list that maps numbers to specific colors.
- **Pixel Data:** The actual image. It stores the "dots" (pixels) that make up the picture, usually uncompressed so the quality stays perfect.

VI. Working with DIBs in Memory

- **Packed-DIB Format:** When a program opens a DIB, it often puts the header and pixel data into one continuous block of memory. This makes it faster for the computer to process.
- **Creating from Scratch:** Developers don't just open files; they can create a DIB directly in the computer's memory to build an image from scratch.

VII. Windows & Custom Programming

- **Windows API:** Windows has built-in tools to help display these images on screens/printers or convert them into other formats.
- **Custom Coding:** If you want to do advanced stuff—like adding artistic filters, changing color depth, or high-end editing—you usually have to write your own custom code rather than relying on the basic Windows tools.

VIII. Deep Dive: OS/2-Style DIB Format

This is an older, simpler version of the bitmap format. Here is exactly what is inside:

File Header (14 bytes)

- **bfType:** Must be the letters "BM".
- **bfSize:** Total size of the file.
- **bfOffBits:** How many bytes you have to skip to get to the actual image data.

Information Header (12 bytes)

- **bcWidth / bcHeight:** The size of the image in pixels.
- **bcBitCount:** How many "bits" per pixel (1, 4, 8, or 24). This determines the color quality.

The Pixel Bits (How colors are stored)

The way the computer reads the image depends on the **Bit Count**:

- **1-bit:** 1 byte holds 8 pixels (Black & White).
- **4-bit:** 1 byte holds 2 pixels (16 colors).
- **8-bit:** 1 byte holds 1 pixel (256 colors).
- **24-bit:** 3 bytes hold 1 pixel (Full "True" Color using Red, Green, and Blue).

Key Takeaway: DIBs are great because they work the same way on any screen or device. While Windows provides the basics, programmers can manipulate the raw data to do almost anything with the image.

IX. 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->bmcColors[i];
```

X. Simplified Key Points:

- OS/2 DIBs support 1, 4, 8, or 24 bits per pixel.
- Color tables are used only with 1-, 4-, and 8-bit images.
- Pixel data layout depends on the bit depth.
- Important colors should appear first in the color table.
- Pixel data always starts on a WORD-aligned 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 }

```

This code works with bitmap image headers and related data structures. It starts by including `<stdlib.h>`, which provides functions for dynamic memory management like `malloc()` and `free()`.

After that, the code defines several structures that model different sections of a bitmap image file.

BITMAPFILEHEADER - Defines the basic file-level information for a bitmap, including the file type, total size, and where the pixel data begins.

BITMAPCOREHEADER - Stores the core image details such as width, height, color planes, and bits per pixel (1, 4, 8, or 24).

RGBTRIPLE - Represents a single color entry in the bitmap's color table using red, green, and blue values.

BITMAPCOREINFO - Combines the core header with a color table, forming a complete OS/2-style DIB structure.

In main(), memory is dynamically allocated for a BITMAPCOREINFO structure along with space for the full color table.

The code then accesses entries in the color table (though the index variable is missing in the snippet) and finally frees the allocated memory.

DIB Pixel Order (Bottom-Up)

DIBs store pixel rows **bottom-up**, not top-down.

- The **first row in the file** is the **bottom row of the image**
- The **last row in the file** is the **top row of the image**
- **Top / Bottom rows** → how the image appears visually
- **First / Last rows** → how rows are stored in the file

I. Why This Happens

- OS/2 used a coordinate system where **Y increases upward**
- DIBs followed this system for consistency with OS/2 graphics
- Result: bitmap data is written from bottom to top

II. Practical Impact

- When reading or writing DIBs, **rows must be handled in reverse**
- Image processing code may need to flip rows
- This behavior is normal and expected for classic DIB formats

DIBs are bottom-up by design.

Always account for reversed row order when accessing pixel data.

DIB Pixel Data Layout

I. Pixel storage rules in DIBs:

- **Bottom-up order:**
Pixel rows are stored from the bottom of the image to the top.
- **Left-to-right order:**
Pixels within each row are stored from left to right.
- **Row padding:**
Each row is padded with zero bytes so its size is a multiple of **4 bytes**.

II. Why This Matters

- Code must read rows in reverse to display the image correctly.
- Padding bytes must be skipped when moving between rows.

DIB pixel data is stored **bottom-up, left-to-right**, with **4-byte row alignment**.

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.

What the Diagram Shows

1. File layout

The diagram shows how a DIB file is organized: the file header first, then the information header (and color table when used), and finally the pixel data at the end.

2. Pixel encoding by bit depth

It also shows how pixels are packed depending on the bit count:

- **1-bit:** one byte holds 8 pixels
- **4-bit:** one byte holds 2 pixels (high nibble first)
- **8-bit:** one byte per pixel (color table index)
- **24-bit:** three bytes per pixel (RGB)

3. Bottom-up storage

The diagram places the image's bottom row first to reflect how DIBs store pixel data. This is normal for DIBs, even though it looks backwards compared to most image formats.

4. Working with DIB Pixels

Once you understand how pixels are stored, accessing and modifying them is straightforward. Pixel Access goes like:

- **Find the pixel:** Convert the row and column into a byte offset in the pixel data.
- **Read the value:** Extract the pixel data based on the image's bit depth.

Pixel Modification

- **Change colors:** Update the pixel value directly (eg, flip bits to invert colors).
- **Apply filters:** Loop through pixels and apply calculations for effects like grayscale.
- **Transparency:** For formats with alpha data, adjust transparency along with RGB values.
- Each row may include padding bytes. Account for this when stepping through rows.
- Always check bounds to avoid invalid memory access.

The Windows DIB: Modern Enhancements

Windows 3.0 updated the DIB format to handle larger images and better performance. Here is the breakdown of the changes:

I. File Header

No Change: Still uses the same 14-byte BITMAPFILEHEADER as the OS/2 version.

II. Information Header (BITMAPINFOHEADER)

This is the biggest change. It grew from 12 bytes to **40 bytes** and added these features:

- **Larger Images:** Uses 32-bit values for width and height (allowing for much bigger resolutions).
- **Compression:** Added a field (biCompression) to allow compressed image data.
- **Image Size:** A new field (biSizeImage) specifically for the total byte count of the pixels.
- **Resolution:** Stores "pixels per meter" for horizontal and vertical printing accuracy.
- **Color Control:** Includes fields to show exactly how many colors in the palette are actually being used or are "important."

III. Color Table (RGBQUAD)

- **The Change:** Switched from RGBTRIPLE (3 bytes) to RGBQUAD (4 bytes).
- **The Reason:** The extra byte is "padding" that aligns the data to 32-bit boundaries, making it much faster for modern processors to read.

IV. BITMAPINFO Structure

- This is just a "wrapper" that combines the **Header** and the **Color Table** into one single unit for easier programming.

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

V. Key Summary Points

- **How to tell them apart:** Check the header size. **40 bytes** = Windows; **12 bytes** = OS/2.
- **Better Scaling:** Windows DIBs support larger dimensions and built-in compression.
- **Better Speed:** Uses 4-byte color entries to help 32-bit computers process data faster.
- **Better Printing:** Stores resolution data (pixels per meter) to ensure the image prints at the correct size.

The original Windows 3.0 DIB was simple. But as technology improved (scanners, high-res printers), the format had to evolve. Windows 95 and NT 4.0 took the standard BITMAPINFOHEADER structure and gave the fields new, specific meanings.

1. The Upside-Down Trick (biHeight)

In the classic math world (Cartesian coordinates), the "Y" axis goes **UP**. In the computer reading world (like reading a book), the "Y" axis goes **DOWN**.

- **Positive Height (e.g., 100):** The standard. The image is stored "Bottom-Up." The first byte of data represents the bottom-left pixel.
- **Negative Height (e.g., -100):** The modern twist. The image is stored "Top-Down." The first byte represents the top-left pixel.

Why do this? It makes it easier for video players and DirectX to blast memory straight to the screen without doing math to flip the image. *Note: You cannot compress a top-down DIB. It must be uncompressed.*

2. The "Real World" Size (`biXPelsPerMeter`)

A bitmap is just a grid of pixels (e.g., 100x100). But how big is it?

- On a smartwatch, 100 pixels is tiny.
- On a stadium jumbotron, 100 pixels is huge.

The fields `biXPelsPerMeter` and `biYPelsPerMeter` answer the question: "**If I print this, how many inches should it be?**"

Standard Values:

- 0: "I don't care. Make it whatever size you want."
 - 2835: Approx 72 DPI (Standard Screen).
 - 11811: Approx 300 DPI (Standard Print).
-

3. The Color Table Optimization (`biClrUsed`)

This is the most confusing field for beginners. Its meaning changes depending on the "Bit Depth" (how many colors the image has).

A. For "Palettized" Images (4-bit, 8-bit)

These images use a lookup table (Color Table).

- **Standard (`biClrUsed = 0`):** "I use the full table." (e.g., 256 colors for 8-bit).
- **Optimized (`biClrUsed = 50`):** "I have 256 slots available, but I only actually use the first 50 colors. Don't bother loading the rest."
- **Benefit:** Saves a tiny bit of file size and memory.

B. For "True Color" Images (16-bit, 24-bit, 32-bit)

These images usually store color directly in the pixel (Red/Green/Blue values) and **don't** need a table.

- **Standard (`biClrUsed = 0`):** "No table. Pure pixels."
 - **Compatibility Mode (`biClrUsed = non-zero`):** "I am a True Color image, BUT here is a small list of the 'Most Important' colors in case you try to view me on an ancient 256-color monitor."
-

4. The "Priority" List (biClrImportant)

This field is rarely used today.

- It tells the system: "If you can't show all 200 colors, strictly prioritize the first X colors."
 - **Rule of Thumb:** Just set it to 0 (meaning "All colors are important").
-

Explained Like I'm a Teenager: The Image Header

Think of a DIB File like a **container for a Lego set**. The Header is the instruction booklet.

1. **biHeight (Negative):** This is just deciding if you build the Lego set starting from the feet (Bottom-up) or the head (Top-down).
 2. **biXPelsPerMeter:** This is the scale. Is this a miniature Lego set (High DPI) or those giant Duplo blocks (Low DPI)?
 3. **biClrUsed:** This is the piece count.
 - ✓ The box says "Supports 256 colors!"
 - ✓ But if you only built a spaceship using Grey and Black, you write biClrUsed = 2. It tells the user "Don't bother looking for the pink bricks; they aren't in there."
-

Quick Review

Question 1: If biHeight is negative, where is the first pixel of data located in the image?
(Answer: The Top-Left corner. This is a "Top-Down" DIB.)

Question 2: If you have a 24-bit image (millions of colors), why might you still include a Color Table? *(Answer: For backward compatibility. It helps older 256-color monitors pick the "best" colors to display the image roughly.)*

Question 3: What does biClrUsed = 0 mean for an 8-bit image? *(Answer: It means the full standard palette is used ($2^8 = 256$ colors).)*

Real-World DIBs (8-bit and Gray-shade).

In theory, DIBs are simple. In practice, you will encounter weird formats created by old software (like Windows 3.0 Paint) or specialized scanners. This section explains how to handle the two most common "ambiguous" types: **8-bit Gray-shade** vs. **8-bit Color**.

I. The 8-Bit Dilemma

The most common DIB you will find is **8-bit (256 colors)**. The problem? The header (BITMAPINFOHEADER) does **not** tell you if the image is **Color** or **Black & White**. You have to look at the **Color Table** to figure it out.

a) Gray-Shade DIBs (The "Fake" Color)

A grayscale image is just a color image where Red = Green = Blue.

- **64-Level Gray:** Some older scanners only captured 64 shades.
- **256-Level Gray:** The modern standard.

How to Spot It: You iterate through the Color Table. If R == G == B for every single entry, it is a grayscale image.

The Code (Generating a Gray Table): If you are creating a grayscale DIB manually, you need to calculate the palette entries. Here is the formula for a 64-level table:

```
// Generating a 64-level Gray Palette
// We stretch 0-63 range to fit 0-255 RGB range
for (int i = 0; i < 64; i++)
{
    rgb[i].rgbRed   = (i * 256) / 64;
    rgb[i].rgbGreen = (i * 256) / 64;
    rgb[i].rgbBlue  = (i * 256) / 64;
    rgb[i].rgbReserved = 0;
}
```

b) Palettized Color DIBs

These use the color table to store 256 specific colors (e.g., "Mustard Yellow", "Navy Blue").

Full Table: biClrUsed = 0 or 256. The image uses every slot.

The "Safety" Table: Sometimes you see biClrUsed = 236.

Why? Windows reserves 20 colors for system UI (borders, buttons). Old apps avoided overwriting these 20 colors to prevent the screen from flashing weird colors.

2. The Legacy Formats (Rare)

OS/2 DIBs

- **History:** Before Windows dominated, OS/2 (IBM) had its own bitmap format.
- **Difference:** The header structure is different (BITMAPCOREHEADER instead of BITMAPINFOHEADER).
- **Status:** Extremely rare today. Most modern code simply rejects them or converts them.

4-Bit DIBs (16 Colors)

- **History:** From the VGA era (Windows 3.1 Paint).
- **Status:** Surprisingly common in old clip-art libraries.
- **Logic:** Two pixels are packed into one Byte. You need bit-shifting logic ($>> 4$ and & 0x0F) to read the pixels.

3. The "Useless" Fields

When reading DIBs in the wild, you can usually ignore these fields in the header:

biXPelsPerMeter / biYPelsPerMeter:

- ✓ Almost always 0. Even if they have values (like 2835 for Screen Resolution), Windows GDI functions usually ignore them.

biClrImportant:

- ✓ Supposed to tell you "Only the first X colors matter." In reality, it is almost always 0 (meaning "All colors matter").

4. Summary Checklist (Developer's Note)

Flag Missing: There is no header flag for "Grayscale." You must infer it.

Gray Math: $\text{GrayLevel} = (\text{R} + \text{G} + \text{B}) / 3$.

Validation: Always check `biClrUsed`. If it is 0 on an 8-bit image, assume 256.

The Different "Types" of Images

Think of these like different quality settings for a photo:

- **8-bit:** The most common. It can show 256 different colors or shades of gray.
- **4-bit:** Old school and lower quality (only 16 colors). You'd see this in the original version of MS Paint.
- **OS/2-style:** Ancient tech. You'll almost never see this today.

I. 8-Bit Grayscale (Black & White Photos)

This explains how a computer makes a black-and-white image:

- **The Color Table:** This is like a "paint-by-numbers" legend. It lists every shade of gray from pure black to pure white.
- **biClrUsed:** This is just a label that tells the computer how many shades of gray are in that legend (usually 256).
- **The Shortcut:** Because the shades are in order, the computer doesn't have to think hard—a pixel value of "0" is black, and "255" is white.

II. 8-Bit Color (Limited Color Photos)

Instead of shades of gray, this uses a specific set of 256 colors.

- **The Color Table:** Again, this is the "paint-by-numbers" legend, but with colors like Red, Blue, and Green.
- **The 236 Limit:** You might see the number **236** instead of 256. This is because old versions of Windows liked to "reserve" 20 colors for the system (like the colors of the taskbar and buttons), leaving only 236 for the actual picture.

III. 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  
    }  
}
```

The "Useless" Labels

The text lists a few settings that most programmers just ignore:

- **PelsPerMeter:** This is supposed to tell a printer how many pixels fit in a meter. Almost everyone sets this to **0** because the computer usually doesn't care.
- **biClrImportant:** This asks, "Which colors are the most important?" Most people just put **0**, which means "they're all important."

I. Compression (The "Suitcase" Logic)

Compression is just a way to make a file smaller so it doesn't take up too much room on your hard drive.

What is RLE? (The Shortcut)

The text mentions **RLE (Run-Length Encoding)**. It's a very simple trick.

- **Without RLE:** If you have 100 white pixels in a row, the computer writes: "White, White, White..." 100 times. (Waste of space!)
- **With RLE:** The computer just writes: "**100 White.**" (Much smaller!)

II. Which Images Use Compression?

The notes are basically giving you a "rulebook" for different types of images:

IMAGE TYPE	COMPRESSIBLE?	TECHNICAL NOTE
1-bit (Pure B&W)	No	It's already so small it doesn't bother.
4-bit & 8-bit (Indexed)	Yes	They use RLE (Run-Length Encoding) to save space.
24-bit (High Quality)	No	These are "Raw" and stay full-sized for maximum detail.

III. The "16-bit and 32-bit" Mystery

The text mentions "**color masking**" and "**BITFIELDS**." Don't let that confuse you—it's just a fancy way of saying that for very high-quality images (like the ones on your phone today), the computer uses a "mask" to decide exactly how much Red, Green, and Blue to mix for every single pixel.

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

IV. What is this table?

The image shows the "**Escape Codes**" used in Windows bitmap compression (RLE).

When Windows compresses an image, it usually just says "Repeat this color 5 times." But sometimes, it needs to give a special command. It signals a command by setting the first byte to 00. The second byte tells the computer which command to execute:

- **00 00 (End of Row):** Stop drawing this line and move to the start of the next line.
- **00 01 (End of Image):** Stop! The whole picture is finished. Do not read any more data.
- **00 02 (Jump):** Don't draw anything. Just move the "cursor" right by X pixels and down by Y pixels (skipping over a blank spot).
- **00 n (Absolute Mode):** Don't repeat anything. Just copy the next n pixels exactly as they are.

Compression & RLE

The Concept: Run-Length Encoding (RLE) Imagine you have a picture of a blue sky. Instead of saving "Blue pixel, Blue pixel, Blue pixel..." a million times, RLE saves "1,000 Blue pixels." This makes the file much smaller.

I. How 8-bit Compression Works (BI_RLE8)

The computer reads the data in pairs (2 bytes at a time). It looks at the **First Byte**:

- **If the First Byte is a Number (e.g., 5):** This is "Repeat Mode." It means: "Take the color in the *second* byte and draw it 5 times."
- **If the First Byte is Zero (00):** This is "Command Mode." The *second* byte tells the computer to do something special (like stop the line or jump ahead), as shown in the table above.

II. How 4-bit Compression Works (BI_RLE4)

This is the "Hard Mode." In a 4-bit image, one byte holds **two** pixels.

- If you say "Repeat 5 times," the computer has to alternate the colors inside that byte.
- *Example:* If the color byte is Blue/Red, and you repeat it 3 times, you get Blue, Red, Blue. It is confusing math, so most programmers avoid 4-bit compression if they can.

III. The "Transparent" Trick (Delta)

The code 00 02 is a "Jump" command. It tells the computer to skip pixels without drawing anything. This allows you to create non-rectangular images (like a circular character sprite) where the background is transparent because you simply "skipped" drawing it.

IV. The Header Fields (biCompression)

The Header tells Windows which language to speak when opening the file:

- **BI_RGB:** "Raw." No compression. The file is large, but easy to read.
- **BI_RLE8:** "Compressed 8-bit." Use the rules above.
- **BI_BITFIELDS:** "Masking." Used for advanced 16-bit or 32-bit colors (we will discuss this later).

V. Important Limitation

No Top-Down Compression: If you made your image "Upside Down" (by setting biHeight to a negative number), Windows refuses to compress it. You can only compress standard "Bottom-Up" images.

VI. 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.

DIB Structure and Color Masking.

This section explains how Windows packs three colors (Red, Green, Blue) into a single number (a Pixel).

I. The Container (The DIB Structure)

A DIB (Device Independent Bitmap) is just a file that holds a picture. It has two parts:

1. **The Header (BITMAPINFOHEADER):** The ID Card. It tells you "I am 100 pixels wide, and I use 16-bit color."
2. **The Pixel Data:** The actual dots.

The tricky part is the **Header**. Specifically, the field biCompression. This field tells the computer: "*Here is how to slice the pixel data to find the Red, Green, and Blue values.*"

II. The Problem: "Smashing" Colors

In a perfect world, every pixel would have 3 separate bytes: one for Red, one for Green, one for Blue. But computers want to save space. So, they "smash" these colors together into a single integer.

Color Masking is the mathematical tool we use to "un-smash" them.

- **The Mask (&):** A filter that ignores everything except the color we want.
- **The Shift (>>):** Moving the data to the right place so we can read it as a number.

III. Scenario A: 16-Bit Color (BI_RGB)

"The 5-5-5 Format" This format was popular in the 90s (Windows 95/98) to save RAM.

Total space: 16 bits (2 bytes).

The Layout: 1 bit is wasted. Then 5 bits for Red, 5 for Green, 5 for Blue.

- X RRRRR GGGGG BBBBB

How to extract the Red:

1. **The Mask:** The computer applies the Hex mask 0x7C00. This is binary code for "Keep the Red bits, delete the rest."
2. **The Shift:** The Red bits are stuck on the left side (high values). We shift them right by 10 spots ($>> 10$) to move them to the "ones" place.
3. **The Scaling:** 5 bits only gives us numbers from 0 to 31. But modern screens want 0 to 255. So, we multiply the result (Shift Left $<< 3$) to stretch the brightness to the full range.

IV. Scenario B: 32-Bit Color (BI_RGB)

"The Luxury Format" This is the standard today. We don't care about saving space; we want speed.

Total space: 32 bits (4 bytes).

The Layout: 8 bits for Red, 8 for Green, 8 for Blue, and 8 wasted (or used for Alpha transparency).

- 00000000 RRRRRRRR GGGGGGGG BBBBBBBB

How to extract the colors: It is much easier because nothing is "smashed" together tightly. Every color gets its own full byte.

- **Blue:** It's already at the bottom. Just Mask it (0x000000FF). No shifting needed.
- **Green:** It's in the middle. Mask it (0x0000FF00) and shift it right 8 spots ($>> 8$).
- **Red:** It's at the top. Mask it (0x00FF0000) and shift it right 16 spots ($>> 16$).

V. Explaining Like I'm a Teenager: The Combination Lock

Imagine a pixel is a 3-digit combination lock, like "9-5-2".

- **9** is Red.
- **5** is Green.
- **2** is Blue.

But the computer stores it as the single number **952**.

1. **Masking:** If I ask "How much Red is there?", I cover up the "52" with my thumb. Now I only see "900".
2. **Shifting:** "900" is the wrong number. The value is just 9. So I "shift" the decimal point two spots to turn 900 into 9.

That is exactly what Windows does with binary bits (0x7C00 and $\gg 10$).

VI. Quick Review

Question 1: In 16-bit BI_RGB (5-5-5) format, why do we shift the Red value right by 10 bits? (*Answer: Because the Red bits are stored at the top of the 16-bit word. We need to move them to the bottom to read them as a normal number.*)

Question 2: Why is 32-bit color easier for the computer to handle than 16-bit? (*Answer: The colors align perfectly with bytes (8 bits). The computer doesn't have to do as much complex bit-shifting math.*)

Question 3: What is the purpose of "Masking"? (*Answer: To isolate specific bits (like the Red channel) and ignore the others (Green/Blue) inside a packed integer.*)

BI_BITFIELDS

I. The “Mask” (Think: A Stencil)

Imagine you spilled **red, green, and blue paint** all over a piece of paper.

Now imagine you take a **cardboard stencil** that only has holes where the **red paint** is.

- When you place that stencil on the paper, you **only see red**.
- Everything else is blocked out.

That stencil is what computers call a **mask**.

When a computer uses a **Bitwise AND**, it's doing the same thing:

“Show me only the parts I care about. Ignore everything else.”

So, a **Red Mask** means:

“From this number, only keep the bits that represent red.”

Nothing magical — just selective vision.

II. Shifting (Moving Things into the Right Spot)

After using the stencil, the color you found is often **not sitting where you want it**.

It's like finding a word in a sentence, but it's way off to the right.

So, we *move it*.

Right Shift (>>)

- This is like sliding the color **all the way to the left** so you can read it easily.
- Think: "*Move this number into position so I can understand it.*"

Left Shift (<<)

This makes the number **bigger and stronger**.

- It's like turning up the volume.
- Or making the color brighter so it fits the normal **0-255 range** that programs like Photoshop expect.

So:

- **Right shift = move into place**
- **Left shift = scale it up**

III. The Example (The Number 0xABCD)

Let's walk through the example in plain English.

1. The Box

You start with a pixel number:



This single number secretly holds **red, green, and blue** all packed together.

2. The Stencil

You grab the **Red stencil**:



This stencil knows exactly where the red bits live.

3. The Cutout

You place the stencil on the number.

Now:

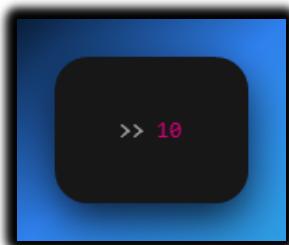
- You **only see the red bits**
- Everything else is gone

This is the **masking** step.

4. The Slide

Those red bits are still sitting off to the side.

So, you **slide them right**:

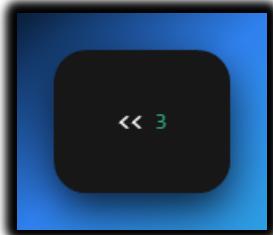


Now the red value is neatly lined up and easy to read.

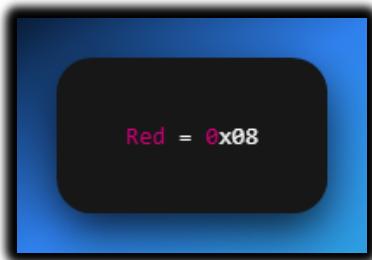
5. The Scale

The red value is still a bit small.

So, you **scale it up**:



Now it matches the normal color range. Final result:



IV. Why Do We Even Do This?

Normally, computers follow **standard color layouts**.

But sometimes a programmer wants to:

- Pack colors differently
- Use fewer bits
- Put blue first instead of red
- Save memory
- Match special hardware

That's where **BI_BITFIELDS** comes in.

It's basically the computer saying:

"Hey, this image doesn't follow the usual rules.
Here's a custom map showing where each color lives."

So **BI_BITFIELDS** is just a **custom instruction manual** for how colors are packed inside a number.

32-bit DIB (BI_RGB) — Extracting Colors the Easy Way

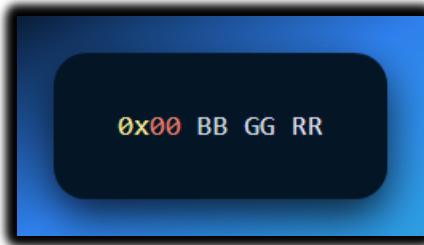
This time, things are **much simpler** than BI_BITFIELDS.

In a **32-bit BI_RGB image**, the computer follows a fixed rule. No guessing, no custom layouts.

I. How One Pixel Is Stored (The Layout)

Each pixel uses **4 bytes** (32 bits total).

They are always arranged like this:



Think of it as four boxes in a row:

- **RR** → Red
- **GG** → Green
- **BB** → Blue
- **00** → Extra byte (unused, always zero)

So, when you see a pixel number, all three colors are already neatly separated into their own byte.

No scaling. No stretching. No tricks.

II. Masks and Shifting (Same Tools, Easier Job)

We still use **masks** and **shifts**, but now it's straightforward because:

- Each color already fits in **8 bits (0-255)**.
- We only need to **move the right byte to the front**.

Extracting Red

Red is stored in the **last byte** (RR).

So, we:

1. Mask everything except the last 8 bits
2. Done — no shifting needed

```
BYTE red = (BYTE)(pixel & 0x000000FF);
```

Think:

“Keep the last box. Throw away the rest.”

Extracting Green

Green is in the **middle byte**.

So, we:

1. Slide the number **right by 8 bits**
2. Mask the last 8 bits

```
BYTE green = (BYTE)((pixel >> 8) & 0x000000FF);
```

Think:

“Slide green to the end, then grab it.”

Extracting Blue

Blue is in the **third byte from the right**.

So, we:

1. Slide right **16 bits**
2. Mask the last 8 bits

```
BYTE blue = (BYTE)((pixel >> 16) & 0x000000FF);
```

Think:

“Slide blue all the way over, then grab it.”

III. Walkthrough Example (Real Numbers)

Pixel Value

```
0x0048E058
```

Break it into bytes:

00	48	E0	58
	BB	GG	RR

So visually:

- **Red** = 0x58
- **Green** = 0xE0
- **Blue** = 0x48

Red Extraction

```
BYTE red = (BYTE)(pixel & 0xFF);
```

Result: Red = 0x58

Blue Extraction

```
BYTE blue = (BYTE)((pixel >> 16) & 0xFF);
```

Result: Blue = 0x48

Everything lines up cleanly.

IV. Why This Is Easier Than BI_BITFIELDS

Key differences:

No left-shifting needed

The values are already in the 0–255 range.

Fixed layout

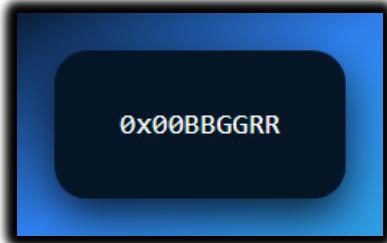
Every pixel follows the same rule: 0x00BBGGRR

Less math

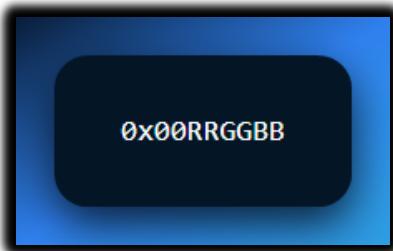
Just shift → mask → done.

V. One Important Gotcha (Windows Detail)

32-bit DIBs store colors as:



But **Windows GDI COLORREF values** use:



So:

- DIBs → **Blue first**
- COLORREF → **Red first**

This is a very common source of confusion.

VI. Big Picture Summary

- 32-bit BI_RGB pixels are **already byte-aligned**
- Each color lives in its own 8-bit box
- Extracting a color = **shift it into place, then mask**
- No scaling, no custom maps, no BI_BITFIELDS headaches

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 }
```

This part of the code is all about **pulling red, green, and blue out of a packed pixel**, even when the colors aren't nicely lined up.

Think of it as:

"Find where the color is hiding, move it into place, and resize it so it looks normal."

i. The Setup (Headers and Types)

Including the Standard Library

The code includes stdio.h so we can:

- Print values to the screen using printf
- See the final red, green, and blue numbers

Nothing special here — just basic C setup.

The DWORD Type

```
typedef unsigned int DWORD;
```

This is just a **shortcut name**.

DWORD means “a 32-bit number”

It's perfect for pixel data because:

- Pixels are just numbers
- We don't want negative values

So anytime you see DWORD, think:

"This holds pixel or mask data."

ii. MaskToRShift — How Far Do We Slide Right?

What This Function Does

This function figures out:

"How far do I need to slide this color to the right so it starts at bit 0?"

In other words:

It finds **where the color begins** inside the pixel.

If the Mask Is Zero

If the mask is 0:

- There is no color
- No shifting is needed
- The function returns 0

Simple safety check.

Finding the First 1 Bit

If the mask isn't zero:

- The function keeps shifting the mask to the right
- It counts how many shifts it takes until it sees the **first 1 bit**

That first 1 means:

"This is where the color starts."

What the Function Returns

The function returns the number of shifts needed.

That number tells us:

"Slide the pixel right this many times to line the color up."

iii. MaskToLShift — How Much Do We Scale It Up?

Why Left Shifting Is Needed

After right-shifting:

- The color is lined up
- But it may still be **too small** (like 5-bit or 6-bit color)

So, we need to **stretch it** to fill a full 8-bit range (0–255).

That's what left shifting does.

If the Mask Is Zero

Same idea as before:

- No mask → no color → no shifting
- Return 0

Step 1: Find Where the Color Starts

The function:

- Shifts right until it finds the first 1 bit
- Counts how many shifts that took

This tells us:

"Where does this color live?"

Step 2: Count How Big the Color Is

It keeps shifting and counting until:

- All the 1 bits are gone

This tells us:

"How many bits does this color actually use?"

Final Calculation

At the end, the function returns:

8 - (number of bits used)

This tells us:

"How much do I need to left-shift to make this an 8-bit color?"

iv. Main Function — Pulling Colors Out of a Pixel

Color Masks

```
DWORD dwMask[3];
```

This array holds:

- Red mask
- Green mask
- Blue mask

Each mask is like a **stencil** that shows where that color lives inside a 16-bit pixel.

Precomputed Shift Values

```
int iRShift[3];
int iLShift[3];
```

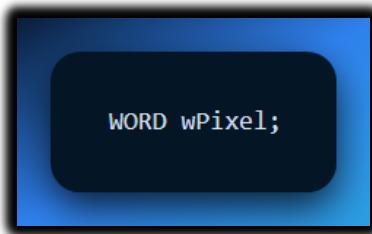
Instead of figuring out shifts every time:

- We calculate them once
- Store them in arrays

This makes extraction:

- Faster
- Cleaner
- Easier to read

Sample Pixel



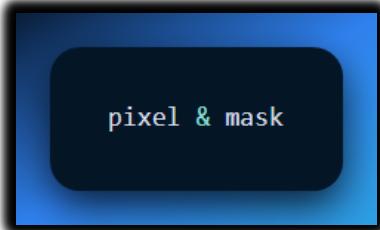
This is a **single 16-bit pixel** that contains:

- Red
 - Green
 - Blue
- All packed together.

V. Extracting Each Color (Same Pattern Every Time)

For **each color** (red, green, blue), we do the same three steps:

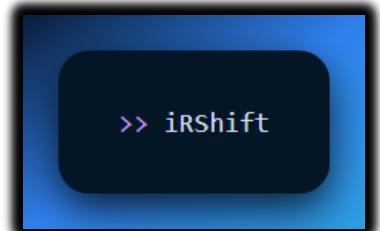
1. Mask It



This:

- Keeps only the bits we care about
- Blocks everything else

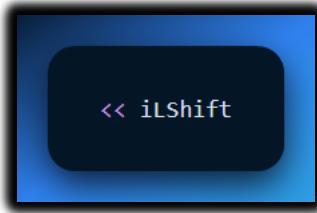
2. Right Shift It



This:

- Slides the color to the right
- Lines it up neatly

3. Left Shift It



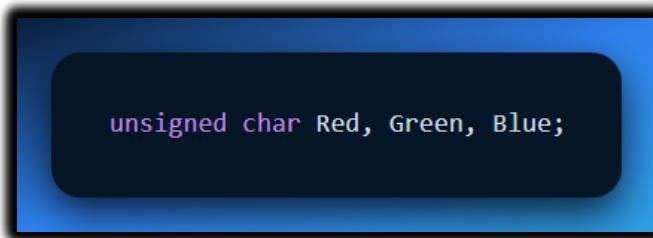
This:

- Scales the color up
- Makes it fit into an 8-bit range (0–255)

Now the color is ready to use.

VI. Final Result

The extracted colors are stored as:



Each one now represents:

- A normal color value
- From **0 to 255**

Finally, printf prints the values so we can see them.

VII. Big Picture Summary

- Pixels pack colors tightly to save space
- Masks tell us **where each color is**
- Right shifts move colors into position
- Left shifts scale them to full brightness
- The code works for **any 16-bit format**, not just one layout

THE ROLE OF COLOR MASKS (WHAT THEY REALLY DO)

In **Device-Independent Bitmaps (DIBs)**, color masks are basically **instructions** that explain how colors are packed inside a pixel.

A pixel is just a number.

Color masks tell us:

“Which bits belong to red, which belong to green, and which belong to blue?”

This is especially important when the colors are **not stored in a fixed order**, like in **BI_BITFIELDS**.

Think of color masks as a **map or blueprint**. Without them, the pixel data would just look like random bits.

What BI_BITFIELDS Means (Why Masks Matter Here)

When the biCompression field is set to **BI_BITFIELDS**, the image is saying:

“Don’t assume a standard color layout. I’ll tell you exactly where each color is.”

Instead of using a fixed format (like **BI_RGB**), **BI_BITFIELDS** provides:

- One mask for **Red**
- One mask for **Green**
- One mask for **Blue**

Each mask is a 32-bit number that clearly marks:

- Where that color starts
- How many bits it uses

This gives a lot of flexibility:

- Colors can be packed tighter
- Bit sizes can vary
- The order of colors can change

But it also means:

You *must* use the masks to decode the pixel correctly.

How Colors Are Decoded Using Masks

I. Get the Masks from the Header

Before touching the pixel data, the program first reads:

- The **red mask**
- The **green mask**
- The **blue mask**

These come directly from the DIB header and describe the pixel layout. No guessing involved.

II. Figure Out How Much to Shift

The color bits are usually:

- Not aligned nicely
- Not already in the 0–255 range

So, we calculate two things for each color:

- **Right shift:**
How far to slide the bits so the color starts at bit 0
- **Left shift:**
How much to stretch the color so it fits into an 8-bit value

The helper functions (MaskToRShift and MaskToLShift) do this work automatically based on the mask.

III. Mask and Shift the Pixel

Now comes the actual extraction:

1. **Mask the pixel**
This keeps only the bits for one color and removes everything else.
2. **Right shift**
This moves the color into the correct position so it can be read.
3. **Left shift**
This scales the color up to a normal 8-bit range (0–255).

After these steps, the color value is clean, readable, and ready to use.

IV. Big Idea to Remember

- BI_BITFIELDS = **custom color layout**
- Color masks = **the rules for decoding it**
- Mask → shift right → shift left
- Result = standard red, green, and blue values

In short:

Color masks turn packed, confusing pixel data into real colors you can actually use.

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 Things to Keep in Mind (32-bit BI_BITFIELDS)

When working with **32-bit DIBs** that use **BI_BITFIELDS**, the idea is very similar to the 16-bit case — but with **more room to work with**.

The steps are the same:

- Read the masks
- Use them to extract colors
- Shift and scale the values

What changes is **how big and flexible those masks can be**.

I. Bigger Masks, More Flexibility

In **32-bit BI_BITFIELDS**, color masks can be much larger than in 16-bit images.

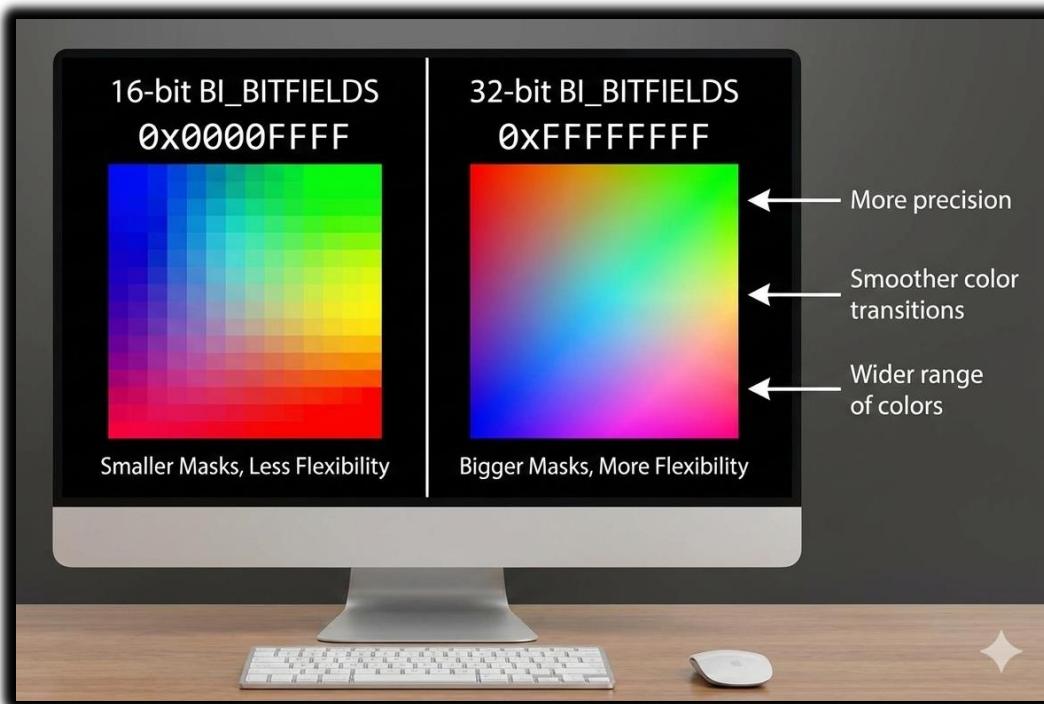
- In 16-bit images, masks usually fit inside 0x0000FFFF
- In 32-bit images, masks can use **many more bits**

This means:

- More precision
- Smoother color transitions
- A wider range of colors if needed

In short:

32-bit BI_BITFIELDS gives you more freedom in how colors are stored.



II. Color Values Can Go Beyond 255

With BI_BITFIELDS (both 16-bit and 32-bit):

- Color values are **not guaranteed** to be in the 0–255 range
- Some colors may use more or fewer bits

That's why:

- Masking alone isn't enough
- Shifting and scaling are required

Once extracted and adjusted, the values are converted back into the familiar **8-bit (0–255)** range that most graphics systems use.

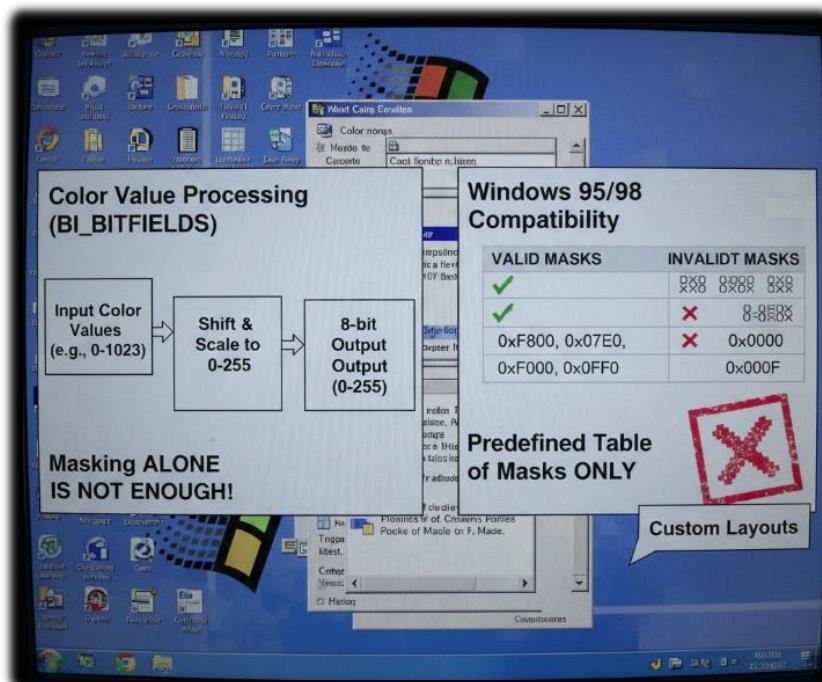
Windows 95 / 98 Compatibility (Old but Important)

Older versions of Windows — especially **Windows 95 and Windows 98** — had **strict rules** about which mask values were allowed.

- Only specific mask patterns were supported
- Anything outside those rules could fail or display incorrectly

Because of this:

- Developers had to follow a predefined table of valid masks
- Custom layouts were limited



Modern Windows Is Much More Relaxed

On newer systems like **Windows 10 and Windows 11**:

- Mask restrictions are far less strict
- Developers can choose masks more freely
- Custom color layouts are easier to use

However:

If you care about backward compatibility with very old systems, you still need to respect those older rules.

The safest approach:

- Use modern masks when targeting modern Windows
 - Follow Microsoft's documented mask tables when legacy support matters
-

Why Use Custom Color Layouts?

BI_BITFIELDS isn't just about flexibility — it's about **optimization**.

Here are some real reasons to customize color masks.

I. Subsampling (Using Bits Where They Matter Most)

Not all colors matter equally to the human eye.

For example:

- Green is often more noticeable than blue
- You might give green more bits
- And give blue fewer bits

This can:

- Reduce file size
- Keep images looking good
- Use bits more efficiently

II. Matching Color Spaces

Custom masks can be designed to better match:

- sRGB
- Adobe RGB
- Or other color spaces

This helps with:

- Accurate color reproduction
- Color-managed workflows
- Professional graphics pipelines

III. Channel Priority

Sometimes one color channel is more important than others.

Examples:

- Medical imaging
- Scientific visualization
- Night vision or thermal imaging

By giving certain channels more bits:

- Important details are preserved
- Less important data can be compressed

Advanced Uses of Color Masks

Once you understand masking, you're not limited to just reading pixels.

You can *manipulate* them.

I. Color Correction

Masks allow precise control over individual channels, enabling:

- White balance fixes
- Removing color casts
- Gamma adjustments

II. Color Effects

Custom masking makes it possible to:

- Apply effects to specific colors only
- Create sepia or grayscale effects
- Perform selective color grading

III. Image Segmentation

By checking color ranges:

- Objects can be isolated
- Backgrounds can be removed
- Regions can be processed independently

This is common in:

- Computer vision
- Object detection
- Image analysis

IV. Color Quantization

By limiting available color values:

- Bit depth can be reduced
- Posterization effects can be created
- Indexed or stylized images can be generated

Beyond the Basics

I. Custom Mask Design

BI_BITFIELDS allows developers to:

- Design their own color layouts
- Optimize for specific image types
- Balance quality, performance, and size

This can even reduce file size by:

- Using fewer bits where precision isn't needed
- Packing colors efficiently

II. Why This Knowledge Matters

Understanding masks gives you:

- Full control over pixel data
- The ability to decode *any* BI_BITFIELDS image
- A foundation for advanced image processing

In short:

Masks turn pixels from “mystery numbers” into something you fully control.

BITMAPV4HEADER AND ADVANCED COLOR

The standard BITMAPINFOHEADER was great for the 1980s, but it had a flaw: it assumed "Red" on my monitor looked exactly like "Red" on your printer. (Spoiler: It never does.) Windows 95 introduced BITMAPV4HEADER to fix this by adding **Color Management**.

I. The Custom Masks (BI_BITFIELDS)

In the previous section, we saw the standard "5-5-5" layout (5 bits for Red, Green, Blue). But what if you want to use that wasted bit?

Developers realized the human eye is most sensitive to **Green**. So, they created the "**5-6-5**" format, giving Green an extra bit of detail.

To use this, you set *biCompression* to BI_BITFIELDS and define custom masks in the V4 Header.

- **16-Bit (5-5-5):** The classic. Wastes 1 bit. 0x7C00 (Red), 0x03E0 (Green), 0x001F (Blue).
- **16-Bit (5-6-5):** The "Green" optimization. Uses all 16 bits. 0xF800 (Red), 0x07E0 (Green), 0x001F (Blue).
- **32-Bit (8-8-8):** Pure luxury. 8 bits per color. 0x00FF0000 (Red), 0x0000FF00 (Green), 0x000000FF (Blue).

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

I. The Color Management Problem (ICM)

The Problem: RGB is "Device Dependent." If you tell a cheap monitor "Show Red 255," it glows a weak orange-red. If you tell a high-end laser projector "Show Red 255," it burns your retinas with deep crimson. The numbers are the same, but the *color* is different.

The Solution: CIE XYZ (The Universal Passport) The V4 header includes a "Color Space" field (bV4CSType). It translates your RGB values into **CIE XYZ**—a scientific, mathematical map of all visible light created in 1931.

- **RGB:** "Go 5 blocks east." (Depends on how big a 'block' is in your city).
- **CIE XYZ:** "Go to GPS coordinate 40.7128° N." (Exact same spot everywhere).

III. The "Gamma" Knobs

The V4 header also adds bV4GammaRed, bV4GammaGreen, and bV4GammaBlue. These are not just "Brightness" sliders. They are **Linearity Correction Curves**.

- **Gamma 1.0:** If you double the pixel number, the screen emits double the light (Linear).
- **Gamma 2.2 (Standard):** Screens are dark by nature. You need to boost the mid-tones significantly to make them look "normal" to the human eye. The V4 header lets the image file carry its own instructions on how to perform this boosting.

IV. Explain Like I'm a Teenager: The Instagram Filter

Think of BITMAPV4HEADER like posting a photo to Instagram with a **Filter** baked in.

- **Old Bitmap:** You send a raw photo. It looks great on your phone, but dark and green on your friend's laptop.
- **V4 Bitmap:** You attach a note:
"This photo was taken on an iPhone with 'Vivid' settings."

Your friend's laptop reads the note (ICM Profile) and automatically adjusts the colors so they look exactly like they did on your phone.

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;
```

Beyond the Basics — Color Management (ICM) Made Simple

Up to now, we've talked about **where colors are stored** and **how to extract them**. This part explains how Windows makes sure those colors **look right on different screens and devices**.

That's where **ICM (Image Color Management)** comes in.

I. ICM in Real Use (What Those Extra Header Fields Do)

In a **BITMAPV4HEADER**, the last four fields are all about color management.

Together, they tell Windows:

- What the colors *really mean*
- How they should be displayed
- How to adjust them for different devices

You can think of these fields as:

A set of instructions that helps Windows translate colors correctly for each screen, printer, or device.

Without ICM:

- Colors might look fine on one monitor
- But too dark, too bright, or slightly off on another

II. The XYZ Color Space (A Neutral Meeting Place)

The **XYZ color space** is a device-independent color system.

That means:

- Colors aren't tied to any specific monitor or printer
- They're defined based on **human vision**, not hardware

Think of XYZ as:

A neutral reference point where all devices agree on what a color actually is.

When a color is defined in XYZ:

- Windows can accurately convert it
- Any device can display it as closely as possible

III. Gamma Values (How Brightness Is Balanced)

Gamma controls how brightness and contrast behave. Different devices:

- Respond to brightness differently
- Display dark and light colors in their own way

Gamma values tell Windows:

"Here's how this device handles brightness — adjust the colors accordingly."

The result:

- Shadows don't get crushed
- Highlights don't get blown out
- Colors feel natural instead of harsh

IV. Why This Matters

Without ICM:

- The same image can look different everywhere
- Designers, photographers, and users lose consistency

With ICM:

- Colors stay visually consistent
- Images look closer to how they were intended
- Devices speak the same “color language”

V. Going Deeper (If You Want To)

ICM goes far beyond simple color correction.

It enables:

- Accurate color workflows.
- Professional image and print matching.
- Reliable color reproduction across devices.

Most of the time, this all happens **quietly in the background** — but understanding it explains *why* modern systems can keep colors looking right almost everywhere.

VI. Big Takeaway

- Masks and bitfields define **where colors live**
- ICM defines **what those colors mean**
- XYZ provides a shared color reference
- Gamma fine-tunes how colors appear

In short: *ICM makes sure colors stay honest, no matter where they're displayed.*

Understanding the CIE XYZ Color Equations

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

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

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

The CIE XYZ equations are used to describe a color in a **device-independent way** — meaning the color is defined by **human vision**, not by a screen, printer, or camera.

Each color is described using **three values**:

- X
- Y
- Z

Together, these three numbers fully describe how a color is perceived by the human eye.

I. The Big Idea Behind the Equations

Light is made up of many **wavelengths** (like a rainbow).

To figure out a color's XYZ values, the equations:

1. Look at **how much of each wavelength** is present
2. Weigh each wavelength by **how sensitive human vision is to it**
3. Add everything together

That's it.

II. What the Symbols Mean (Once, Clearly)

- **Σ (Sigma)**
Means “add everything up.”
- **λ (Lambda)**
Represents wavelength, covering the visible range (about 380–780 nm).
- **$S(\lambda)$**
Tells us how strong each wavelength is in the light source.
- **$\bar{x}(\lambda), \bar{y}(\lambda), \bar{z}(\lambda)$**
These are **human vision filters**.
Each one models how sensitive our eyes are to different wavelengths.
- **$d\lambda$**
A very small slice of the spectrum — like sampling the rainbow one tiny step at a time.

What Each Equation Does

i) X Equation — Color Balance

The **X value** measures how strongly the light matches one part of human color perception.

It's influenced by:

- Red and green wavelengths
- Overall color balance

ii) Y Equation — Brightness (The Important One)

The **Y value** is special.

It closely matches:

- **How bright the color looks to humans**

That's why Y is often used for:

- Luminance
- Grayscale conversions
- Brightness calculations

In simple terms: Y answers the question: *“How bright does this color look?”*

iii) Z Equation — Blue Influence

The **Z value** captures another dimension of vision, with a stronger link to:

- Blue wavelengths
- Blue–yellow color differences

Z doesn't map directly to brightness like Y does, but it completes the color description.

Why We Need All Three (X, Y, Z)

Each equation looks similar, but:

- They use **different human vision filters**
- Each captures a **different aspect of perception**

Together:

X, Y, and Z form a complete, three-dimensional description of color.

This makes XYZ:

- Accurate
- Consistent
- Independent of hardware

A Simple Analogy

Think of:

- **RGB** as ingredients (red, green, blue)
- **XYZ** as a scientific measurement of how the finished color actually looks
- **ICM** as the system that translates those measurements so every device shows the color correctly

Or:

XYZ tells us *what the color really is*, and ICM makes sure devices respect that.

Why This Matters

Because of these equations:

- The same color can look consistent across screens and printers
- Colors can be converted reliably between color spaces
- Professional color workflows are possible

Without XYZ:

- Every device would “guess” what a color means
- Color accuracy would fall apart

Final Takeaway

- The equations add up light across all wavelengths
- Human vision determines the weighting
- X, Y, and Z together fully define a color
- The result is a **true, device-independent color description**

Clean, scientific, and reliable — even if the math looks intimidating at first.

CIEXYZTRIPLE AND CIEXYZ: BASIC COLOR-BUILDING BLOCKS FOR DEVICE-INDEPENDENT COLOR

CIEXYZTRIPLE

This is a group of three colors: red, green, and blue.

Each color is described using the CIE XYZ color system.

It helps computers show colors in a way that matches how humans see them.

CIEXYZ

This describes one color using three numbers (X, Y, Z).

These numbers tell exactly where the color is in the CIE XYZ system.

FXPT2DOT30

This is just a way to store numbers accurately while using less space.

BITMAPV4HEADER: Color Management

bV4CSType

This tells what color system the image uses.

When it is set to **LCS_CALIBRATED_RGB**, it means colors should look correct on any device.

bV4Endpoints

These are the XYZ values for red, green, and blue.

They help make sure colors look the same on different screens.

Gamma Correction: Making Colors Look Right

Gamma (γ)

Gamma fixes the problem that screens do not show brightness in a straight, simple way. A number sent by the computer does not always match how bright the screen looks.

Why this happens

Older screens (like CRTs) and modern displays naturally change brightness in a curved way, not a straight line.

Formula

The formula explains how voltage turns into brightness on the screen.

Cameras and gamma

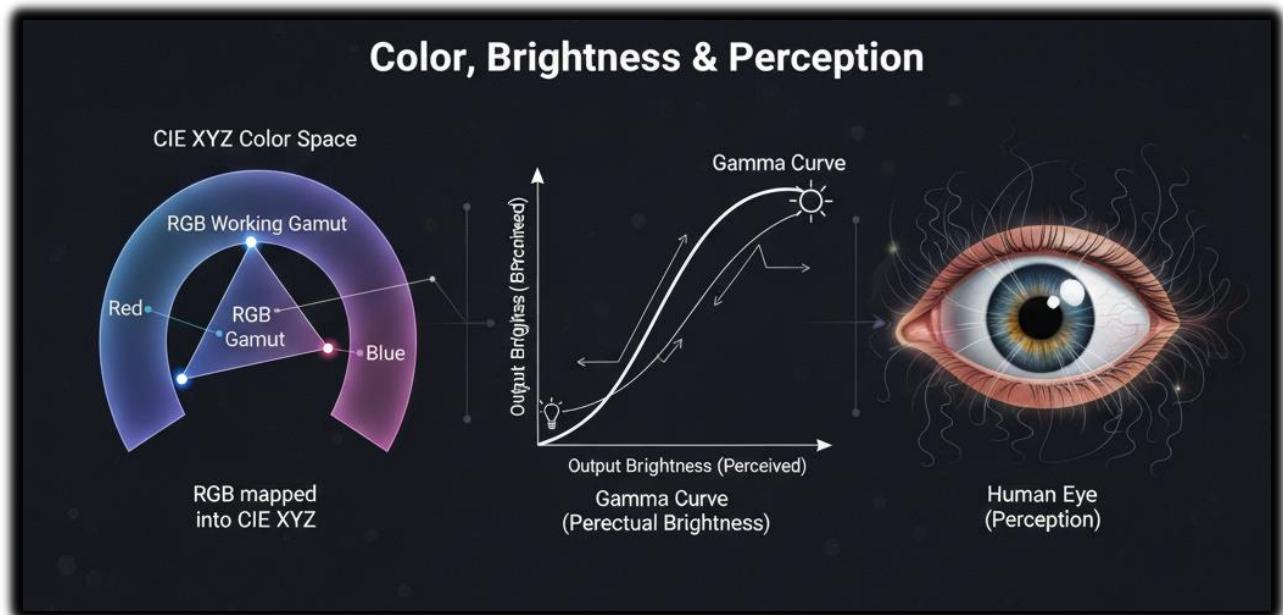
Video cameras often adjust gamma (usually around 0.45) so images look normal and consistent on screens.

```
// 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** describe how humans see color.
They are based on how our eyes react to light from violet (380 nm) to red (780 nm).
- **Y** is special because it shows how bright the color is overall.
- In **BITMAPV5HEADER**, setting **bV4CSType** to **LCS_CALIBRATED_RGB** (0) means the image uses a standard color system that works the same on all devices.
- **CIEXYZTRIPLE** stores three colors: red, green, and blue.
Each one is described using X, Y, and Z values.
- **FXPT2DOT30** is just a precise way to store numbers using fixed-point math.
- **bV4Endpoints** gives XYZ values for pure red, green, and blue
→ (255,0,0), (0,255,0), and (0,0,255).
These values define what RGB really means, no matter the device.
- The last three fields in **BITMAPV4HEADER** control **gamma**.
- **Gamma correction** fixes the problem that screens don't show brightness evenly.
- The formula $I = (V + e)^{\gamma}$ explains how brightness is calculated.
- Cameras usually use a gamma of **0.45**, which matches screens that have a gamma around **2.2**.
- This makes images look natural and consistent on different displays.



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.

0^{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.

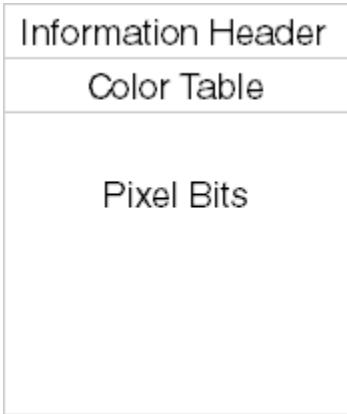
By utilizing the information from these sections, the displaying functions can properly decode and render the image, ensuring accurate representation on the screen or output device.

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.

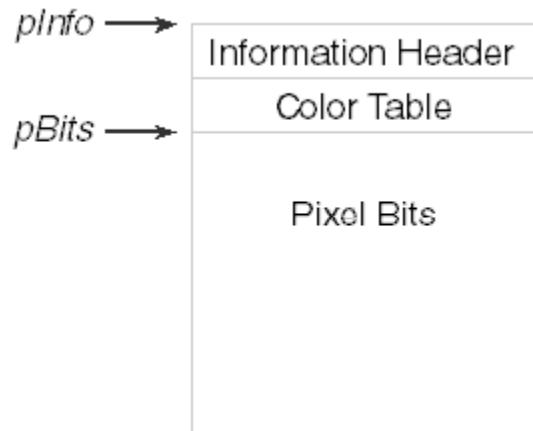
Directly accessing pixels, especially for image-processing tasks, can be time-consuming. A more efficient solution is using a C++ class for DIBs, allowing speedy random access. However, a C solution will be presented in the next chapter.

SetDIBitsToDevice and StretchDIBits Functions:

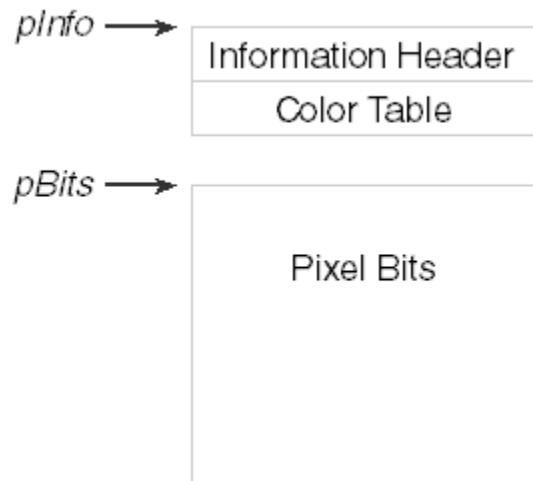
For these functions, you need a pointer to the BITMAPINFO structure of the DIB, comprising the BITMAPINFOHEADER structure and the color table.

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:

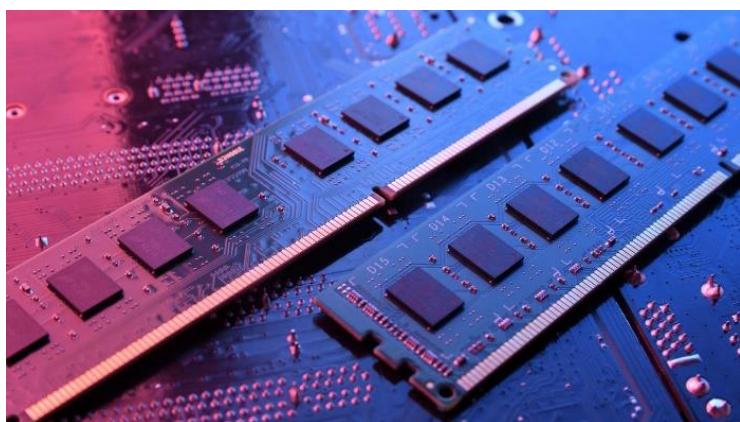
[OpenCV \(Open source Computer Vision Library\)](#) is a popular open-source library that offers a comprehensive set of image processing and computer vision algorithms. It provides functions for image manipulation, feature detection, object recognition, and more. OpenCV supports DIBs and offers efficient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

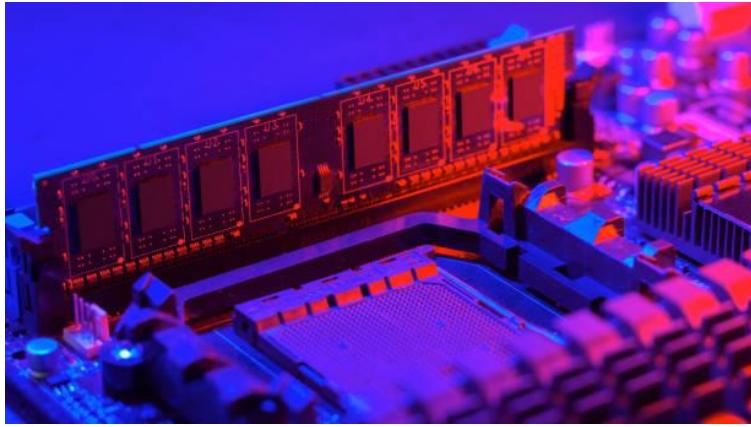
[Skia](#): Skia is a 2D graphics library developed by Google that provides high-performance rendering and image processing capabilities. It is widely used in various applications, including web browsers and mobile platforms, and supports DIBs as an input format.

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).

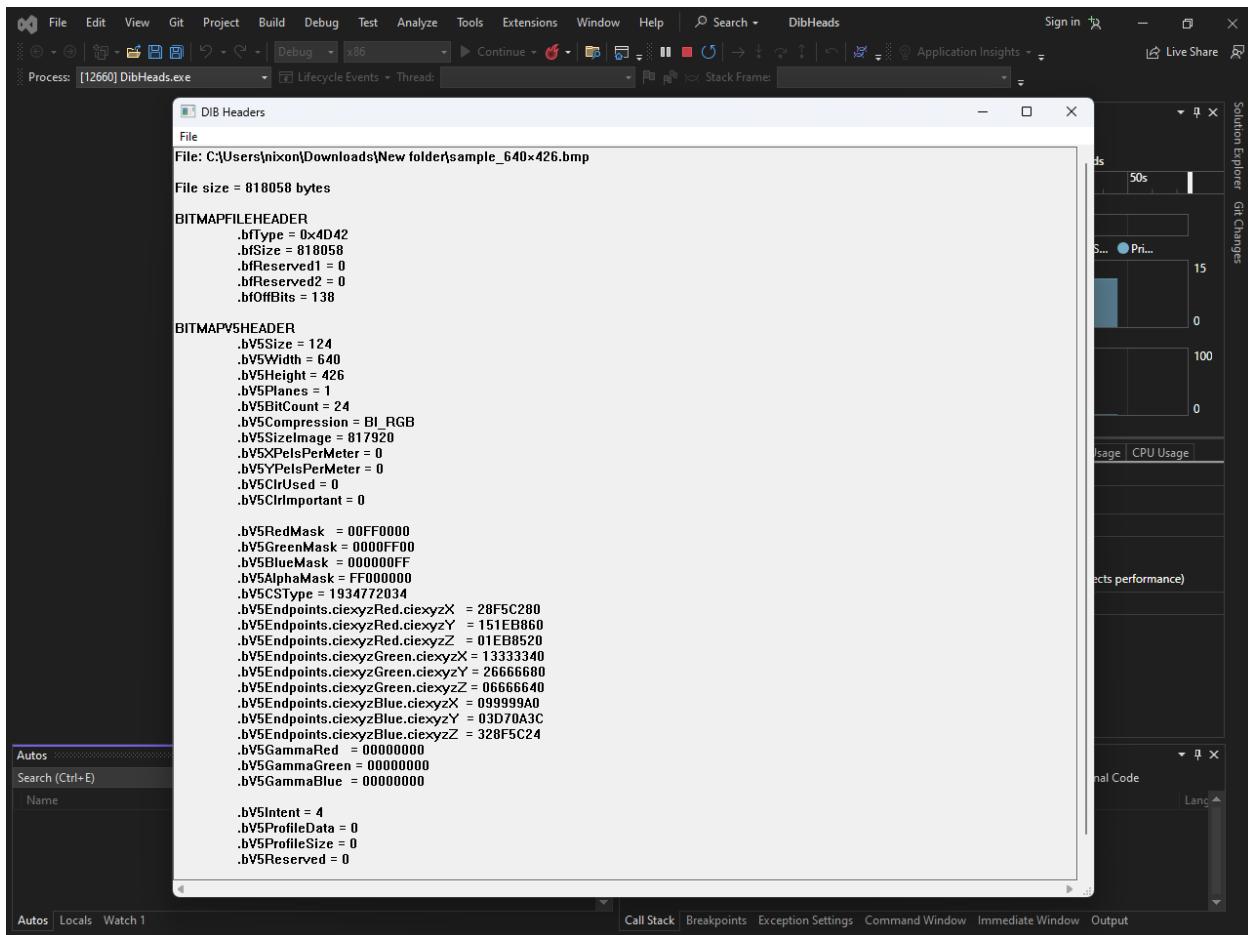


Optimizing memory access patterns when working with packed DIBs can help improve performance. Some techniques and best practices include:

- **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.



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 `HIGHWORD` 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 (pbmfh 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 (pbmfh) 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:

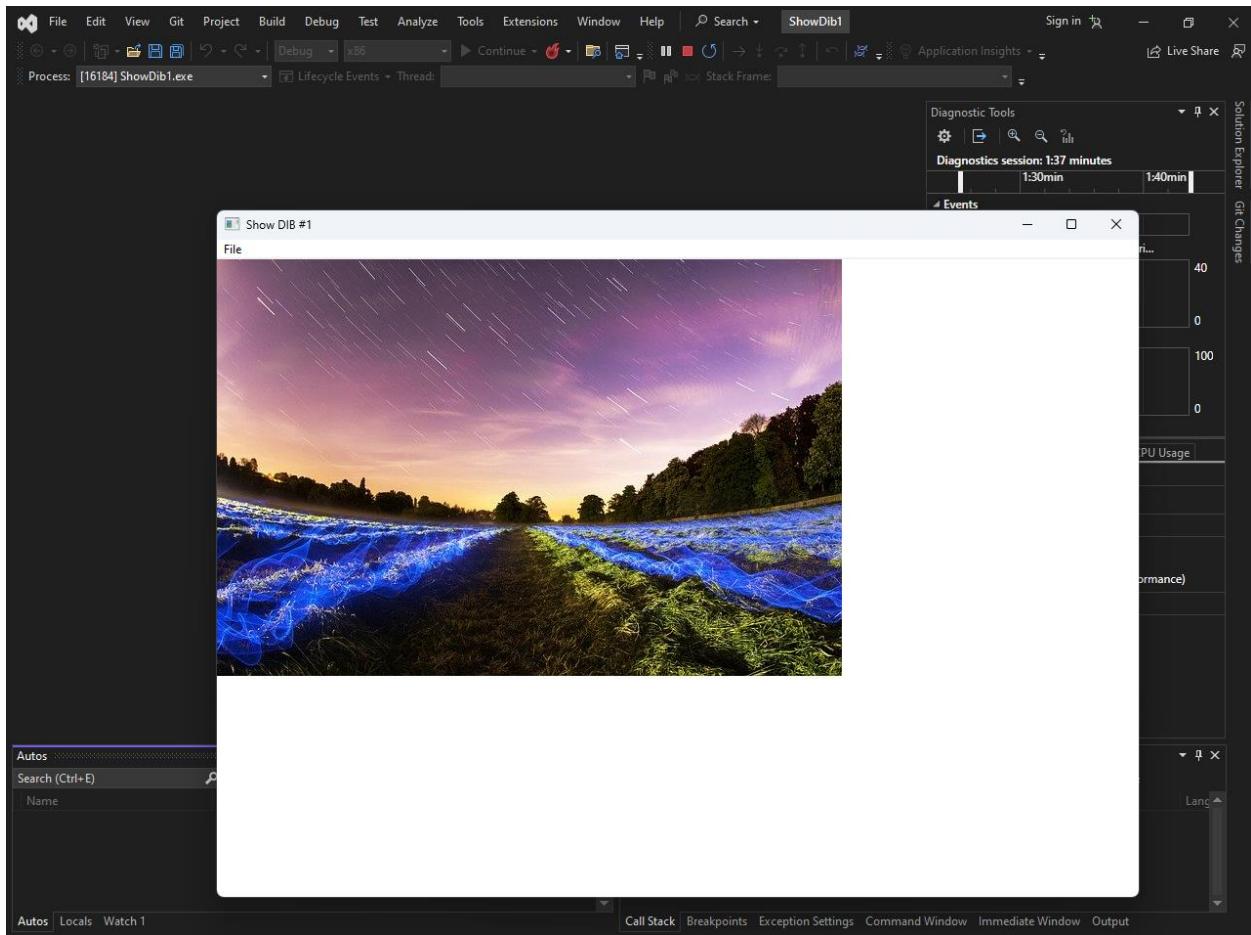
- **DibFileInitialize(HWND hwnd):** This function initializes the global variable ofn (defined in "DIBFILE.H") with default values required for file dialog operations.
- **DibFileOpenDlg(HWND hwnd, PTSTR pstrFileName, PTSTR pstrTitleName):** This function opens a file dialog box for selecting a file 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* 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.

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...

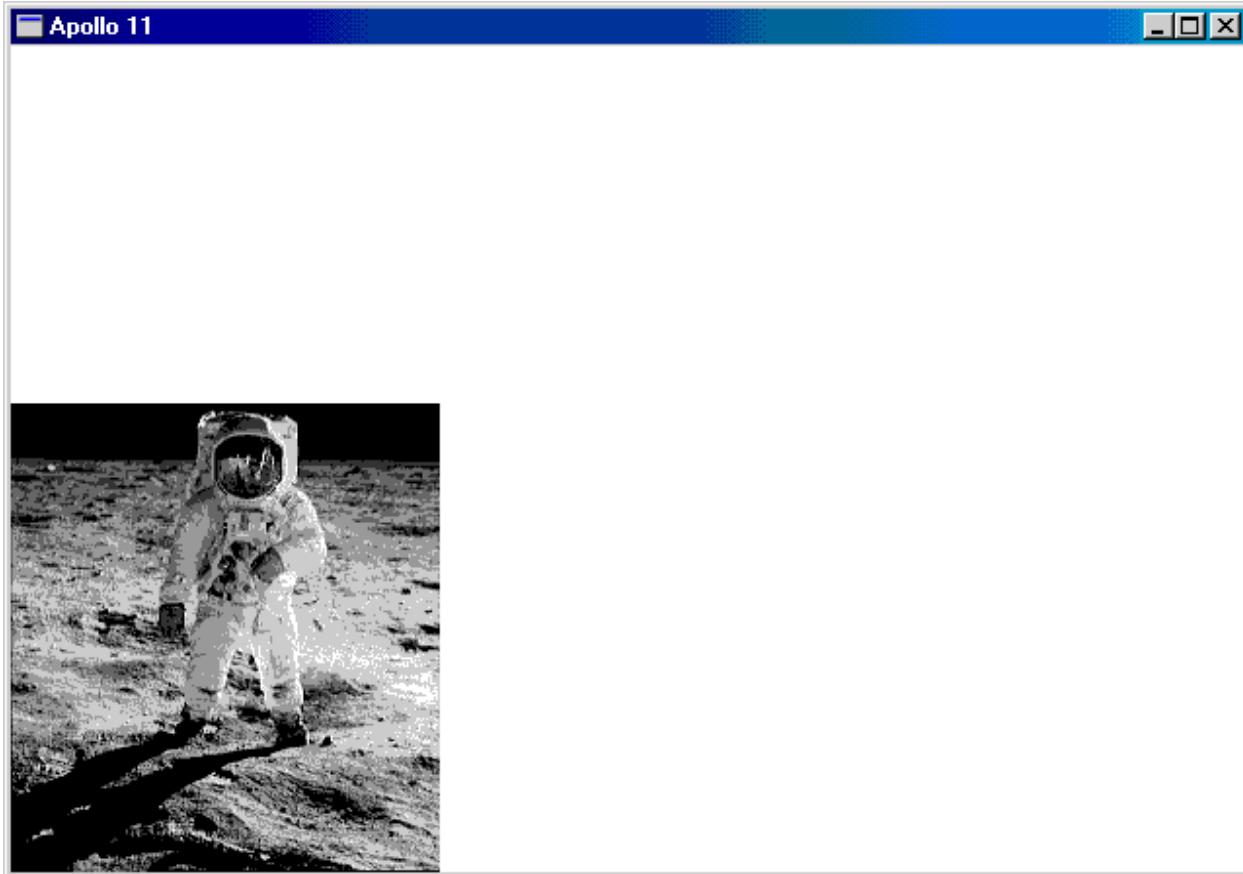
Overall, the code provided sets up the necessary functions and resources to handle DIB files, including file dialogs, loading DIB files into memory, and displaying them on the screen.

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.

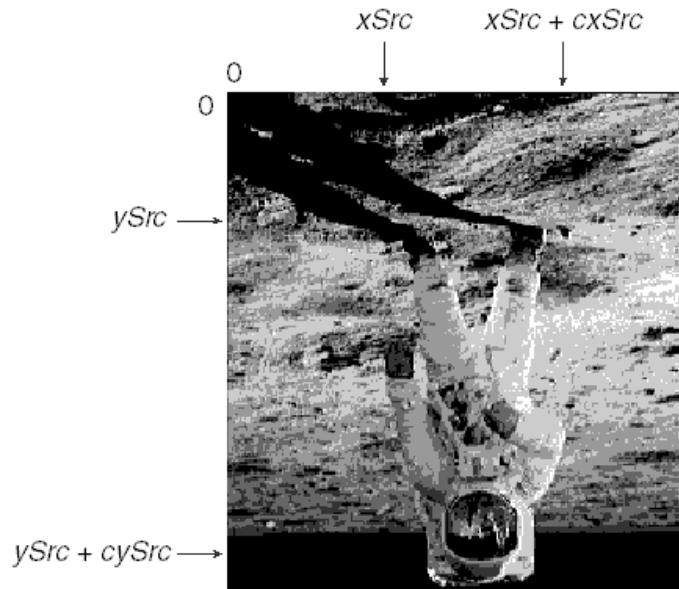
However, Windows introduced inconsistencies in maintaining internal consistency with DIBs. When displaying a rectangular subset of the entire DIB image, you use the arguments `xSrc`, `ySrc`, `cxSrc`, and `cySrc`.

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.

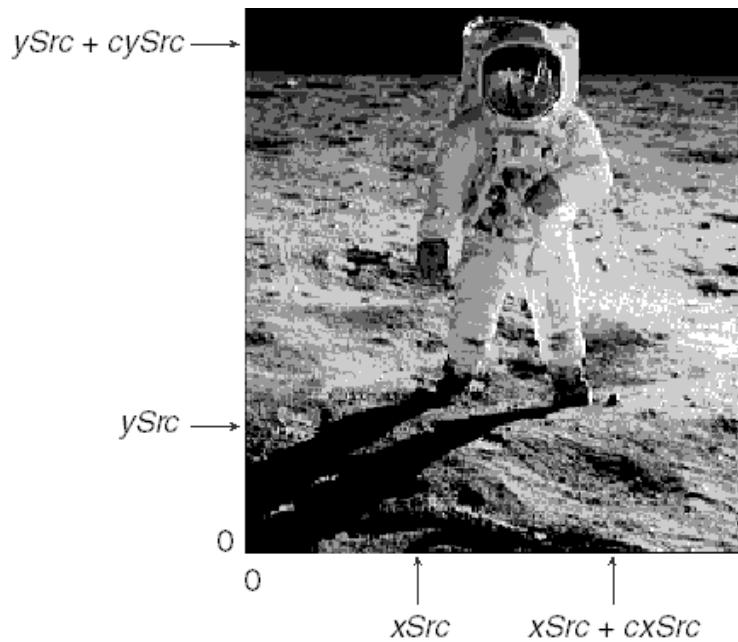


Figure 15-5, illustrates how you specify a rectangle within a top-down DIB. The diagram represents the DIB as it is stored in the file or in memory, providing a visual representation of the arrangement.

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)
```

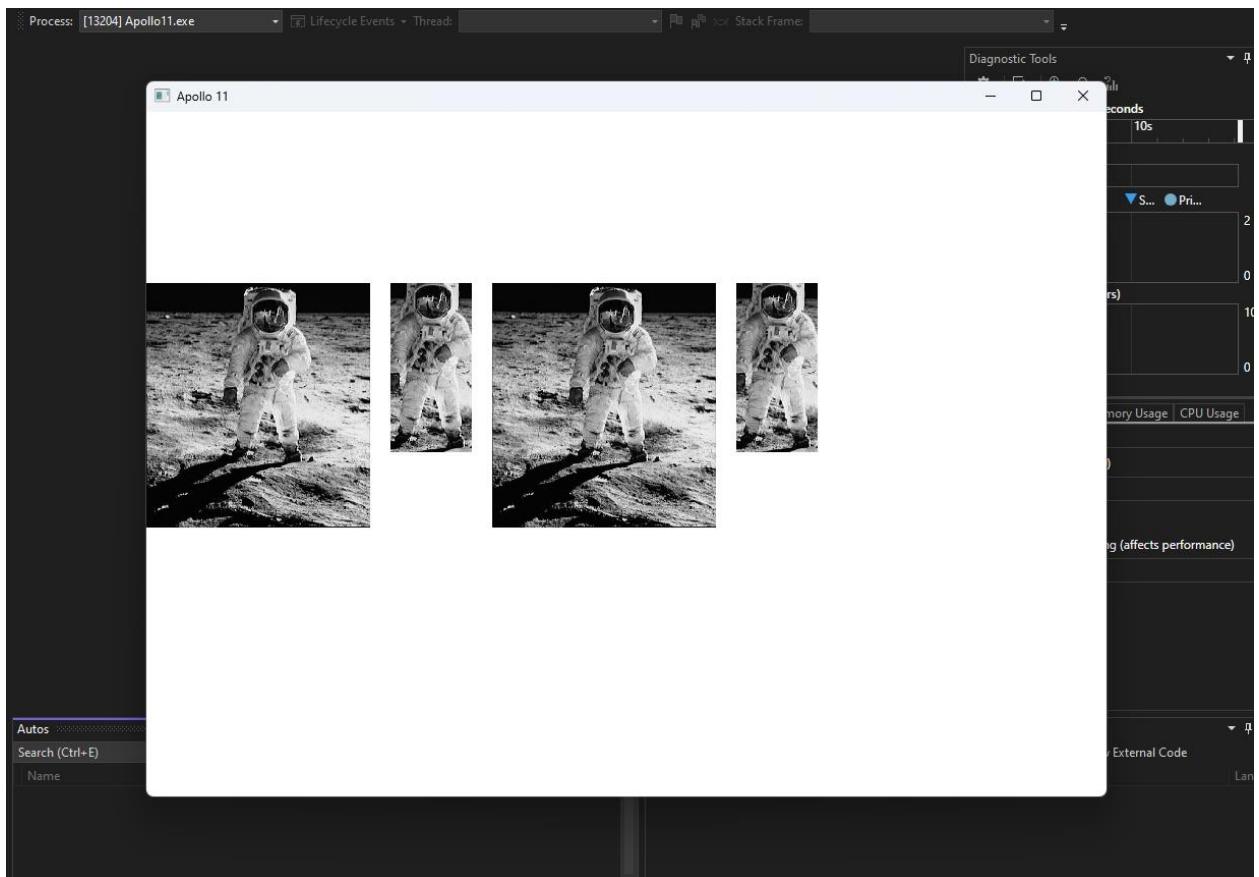
In MM_TEXT mapping mode, it's noted that the point (xSrc, ySrc) does not map directly to (xDst, yDst), which adds complexity to coordinate transformations. However, in other mapping modes, the point (xSrc, ySrc + cySrc - 1) will still map to (xDst, yDst), preserving the appearance of the image.

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



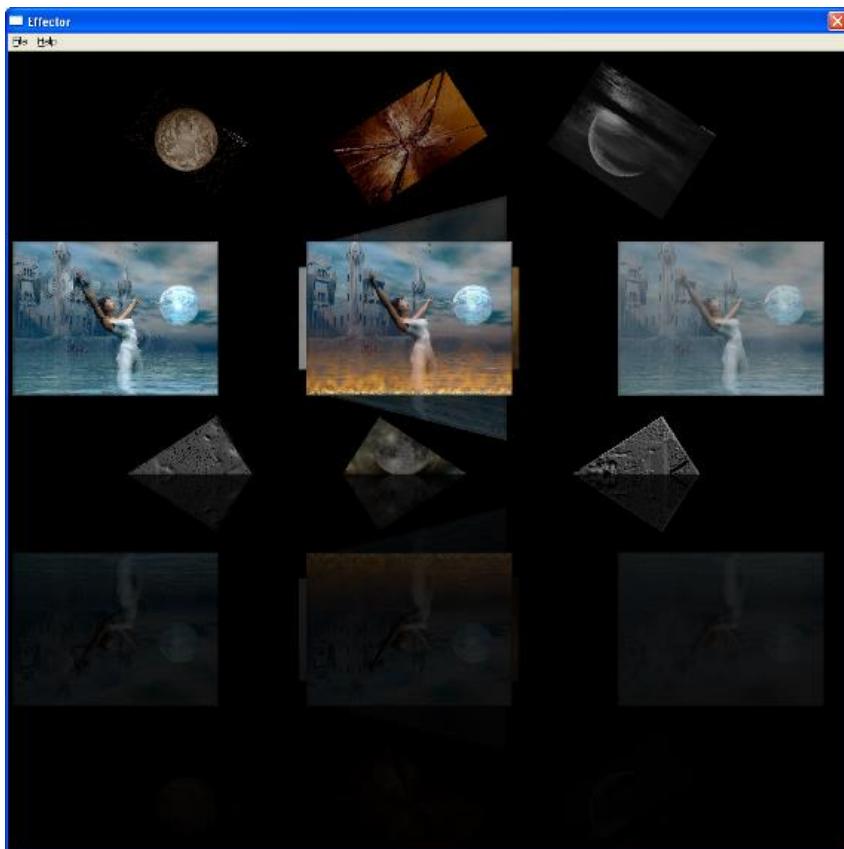
Before we explore the apollo11 program, here are a few scenarios where the SetDIBitsToDevice function can be useful:

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 hWnd, UINT uMsg, WPARAM wParam, LPARAM 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.

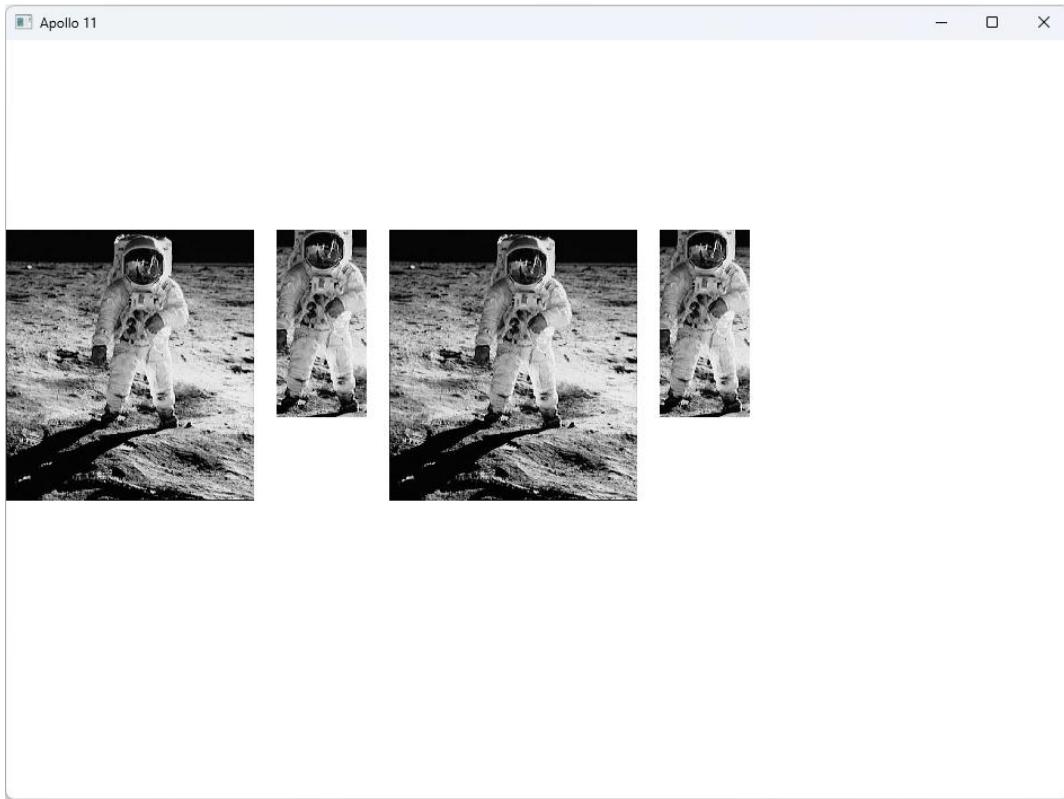
The program is designed to load and display two DIBs: "APOLLO11.BMP" (in a bottom-up orientation) and "APOLLOTD.BMP" (in a top-down orientation). Both DIBs have a width of 220 pixels and a height of 240 pixels. It's worth noting that when determining the DIB's width and height from the header information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

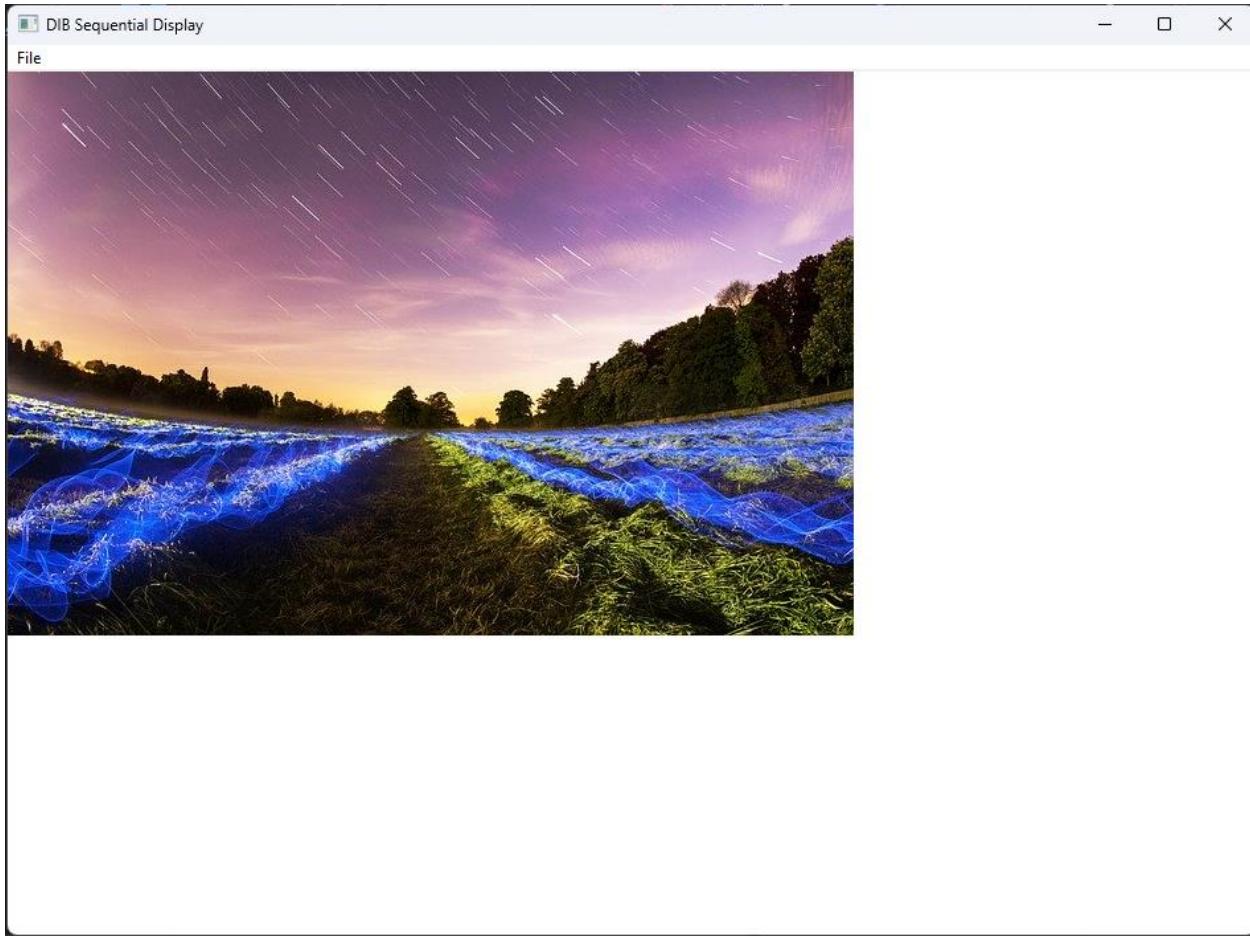
Modify the BITMAPINFOHEADER structure within each call to achieve the desired scaling effect.

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 Conversations:

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.

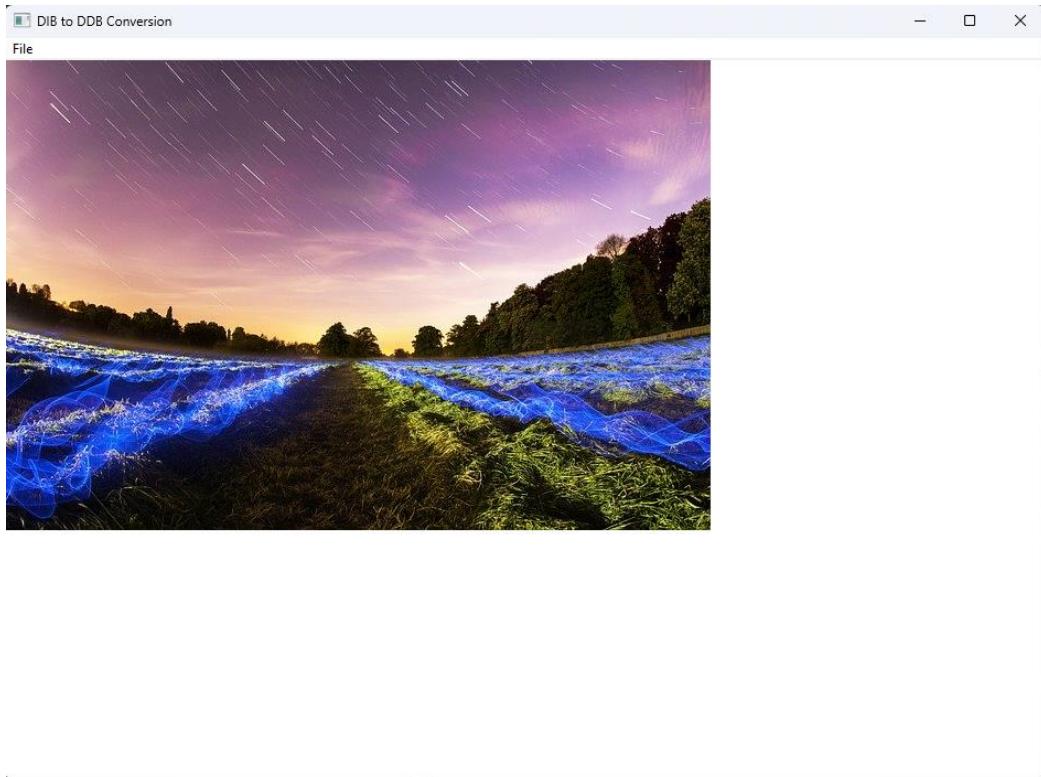
DIB to DDB Conversion Explanation:

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

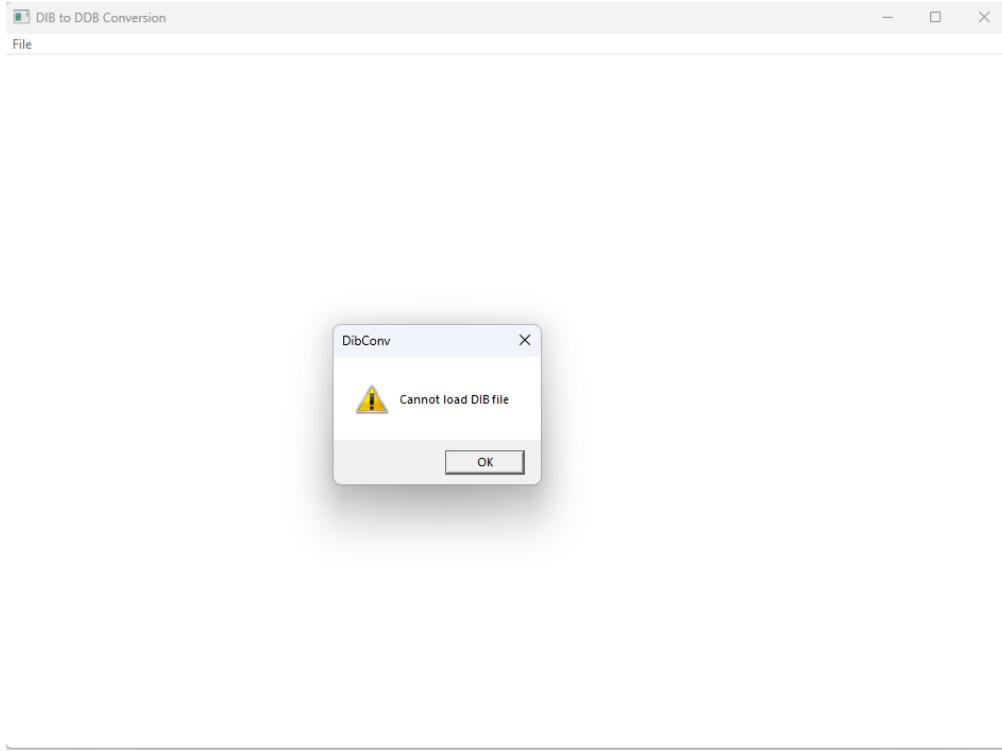
I/O, validate the Bitmap file, and convert the DIB to a DDB using the CreateDIBitmap function.

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 DDB
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.

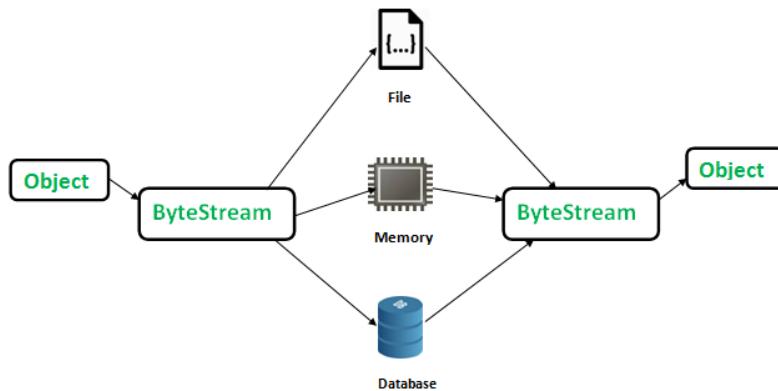


Don't store it in a server please!! ⚡

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.

Serialization

De-Serialization



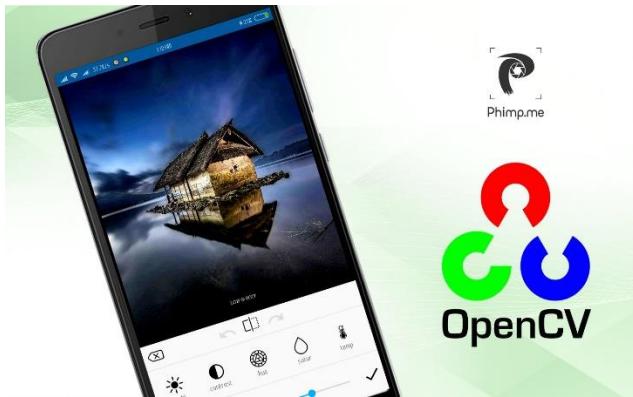
Regarding the automatic conversion of clipboard data to DIBs, when you place a bitmap on the clipboard, Windows automatically converts the bitmap into various formats, including DIBs. When you retrieve the bitmap data from the clipboard, you can request the data in DIB format using the CF_DIB or CF_DIBV5 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 GetDIBBits 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 $(\text{bmfh}.bfOffBits + 3) \& \sim 3$, 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