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->bmciColors[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.

```c
#include <stdlib.h>
// Define the file header structure
typedef struct tagBITMAPFILEHEADER {
    uint16_t bfType;        // "BM" or 0x4D42
    uint32_t bfSize;        // entire size of file
    uint16_t bfReserved1;   // must be zero
    uint16_t bfReserved2;   // must be zero
    uint32_t bfOffsetBits;  // offset in file of DIB pixel bits
} BITMAPFILEHEADER, *PBITMAPFILEHEADER;

// Define the information header structure
typedef struct tagBITMAPCOREHEADER {
    uint32_t bcSize;        // size of the structure = 12
    uint16_t bcWidth;       // width of image in pixels
    uint16_t bcHeight;      // height of image in pixels
    uint16_t bcPlanes;      // = 1
    uint16_t bcBitCount;    // bits per pixel (1, 4, 8, or 24)
} BITMAPCOREHEADER, *PBITMAPCOREHEADER;
// Define the RGBTRIPLE structure for the color table
typedef struct tagRGBTRIPLE {
    uint8_t rgbtBlue;       // blue level
    uint8_t rgbtGreen;      // green level
    uint8_t rgbtRed;        // red level
} RGBTRIPLE;
// Define the combined structure for DIB with color table
typedef struct tagBITMAPCOREINFO {
    BITMAPCOREHEADER bmciHeader;        // core-header structure
    RGBTRIPLE bmciColors[1];            // color table array
} BITMAPCOREINFO, *PBITMAPCOREINFO;

int main() {
    // Allocate memory for the combined structure including the color table
    PBITMAPCOREINFO pbmci = malloc(sizeof(BITMAPCOREINFO) + 255 * sizeof(RGBTRIPLE));
    // Access the RGBTRIPLE structure within the color table
    RGBTRIPLE color = pbmci->bmciColors[i];
    // Free the allocated memory when done
    free(pbmci);
    return 0;
}
```

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):

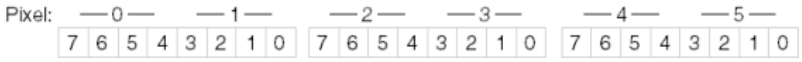
Each pixel enjoys 3 dedicated bytes for red, green, and blue. Rows turn into arrays of RGBTRIPLE structures, encapsulating color intensity. Padding remains key for optimal memory alignment.

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:

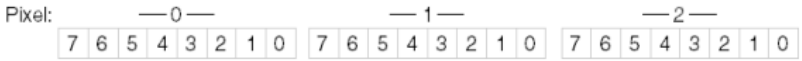
Each pixel can be either a 0 or a 1. A 0 bit means that the color of that pixel is given by the first RGBTRIPLE entry in the color table. A 1 bit is a pixel whose color is the second entry of the color table.

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:



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:



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:

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: GrayLevel = (R + G + 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):

```c
#include <stdio.h>

typedef unsigned char BYTE;  // Assuming BYTE is defined as unsigned char

void DecodeRLE8(BYTE *compressedData, BYTE *pixelData, int width, int height) {
    int x = 0, y = 0;  // Current position in the image

    while (compressedData < end_of_compressed_data) {  // Iterate through compressed data
        BYTE code = *compressedData++;  // Get the RLE code

        if (code == 0) {
            BYTE num_pixels = *compressedData++;  // Get the number of pixels

            if (num_pixels == 0) {  // End of row
                x = 0;
                y++;
            } else if (num_pixels == 1) {  // End of image
                break;
            } else {  // Literal pixels
                for (int i = 0; i < num_pixels; i++) {
                    *pixelData++ = *compressedData++;
                    x++;
                    if (x >= width) {
                        x = 0;
                        y++;
                    }
                }
            }
        } else {  // Repetition code
            BYTE pixel = *compressedData++;  // Get the pixel value
            for (int i = 0; i < code; i++) {
                *pixelData++ = pixel;
                x++;
                if (x >= width) {
                    x = 0;
                    y++;
                }
            }
        }
    }
}

int main() {
    // Example usage
    BYTE compressedData[] = { 0x03, 0x45, 0x32, 0x77, 0x02, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06 };
    int width = 3;
    int height = 3;
    BYTE pixelData[width * height];

    // Call the DecodeRLE8 function
    DecodeRLE8(compressedData, pixelData, width, height);

    // Print the decoded pixel data for demonstration
    for (int i = 0; i < width * height; i++) {
        printf("%02X ", pixelData[i]);
        if ((i + 1) % width == 0) {
            printf("\n");
        }
    }

    return 0;
}
```

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 >> 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**:

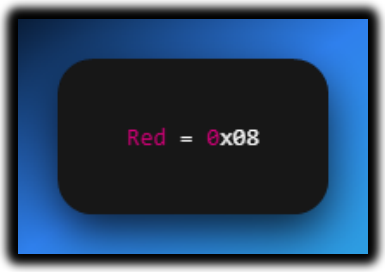
Now the red value is neatly lined up and easy to read.

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

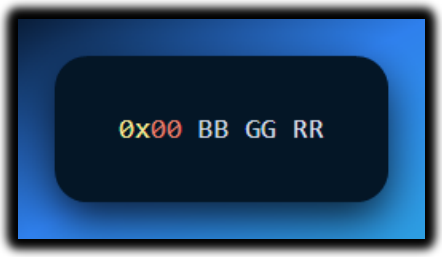
This time, things are **much simpler** than BI_BITFIELDS.

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:

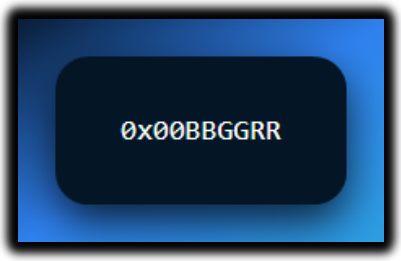
✅ **No left-shifting needed**
The values are already in the 0–255 range.

✅ **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

```c
#include <stdio.h>

typedef unsigned int DWORD;

// Function to calculate right-shift value from a color mask
int MaskToRShift(DWORD dwMask) {
    int iShift = 0;
    if (dwMask == 0)
        return 0;
    while (!(dwMask & 1)) {
        iShift++;
        dwMask >>= 1;
    }
    return iShift;
}

// Function to calculate left-shift value from a color mask
int MaskToLShift(DWORD dwMask) {
    int iShift = 0;
    if (dwMask == 0)
        return 0;
    while (!(dwMask & 1))
        dwMask >>= 1;
    while (dwMask & 1) {
        iShift++;
        dwMask >>= 1;
    }
    return 8 - iShift;
}

int main() {
    // Given color masks for a 16-bit DIB
    DWORD dwMask[3] = {0x00007C00, 0x000003E0, 0x0000001F};

    // Calculate right-shift values
    int iRShift[3];
    iRShift[0] = MaskToRShift(dwMask[0]);
    iRShift[1] = MaskToRShift(dwMask[1]);
    iRShift[2] = MaskToRShift(dwMask[2]);

    // Calculate left-shift values
    int iLShift[3];
    iLShift[0] = MaskToLShift(dwMask[0]);
    iLShift[1] = MaskToLShift(dwMask[1]);
    iLShift[2] = MaskToLShift(dwMask[2]);

    // Example usage with a 16-bit pixel value
    unsigned short wPixel = 0x0048E058;

    // Extract color values
    unsigned char Red = (unsigned char)((dwMask[0] & wPixel) >> iRShift[0]) << iLShift[0];
    unsigned char Green = (unsigned char)((dwMask[1] & wPixel) >> iRShift[1]) << iLShift[1];
    unsigned char Blue = (unsigned char)((dwMask[2] & wPixel) >> iRShift[2]) << iLShift[2];

    // Display the extracted values
    printf("Red: %u\nGreen: %u\nBlue: %u\n", Red, Green, Blue);

    return 0;
}
```

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)

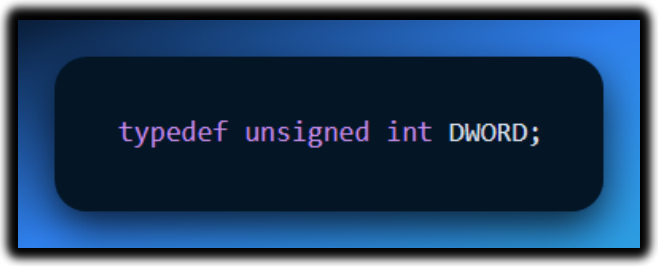
Including the Standard Library

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



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:

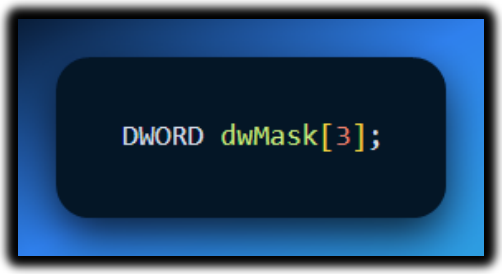
8 - (number of bits used)

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



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

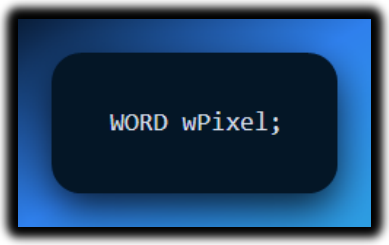
Instead of figuring out shifts every time:

- We calculate them once
- Store them in arrays

This makes extraction:

- Faster
- Cleaner
- Easier to read

Sample Pixel



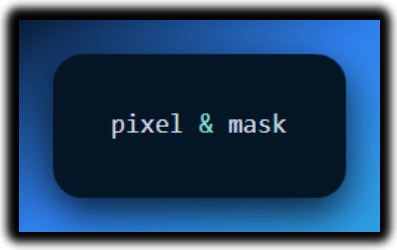
This is a **single 16-bit pixel** that contains:

- 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:

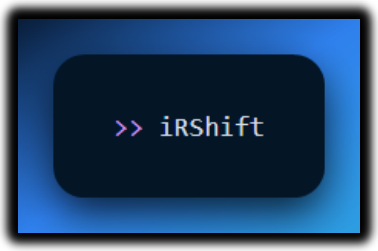
1. Mask It



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

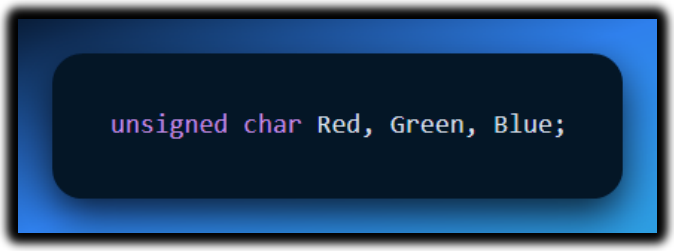
This:

- 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.

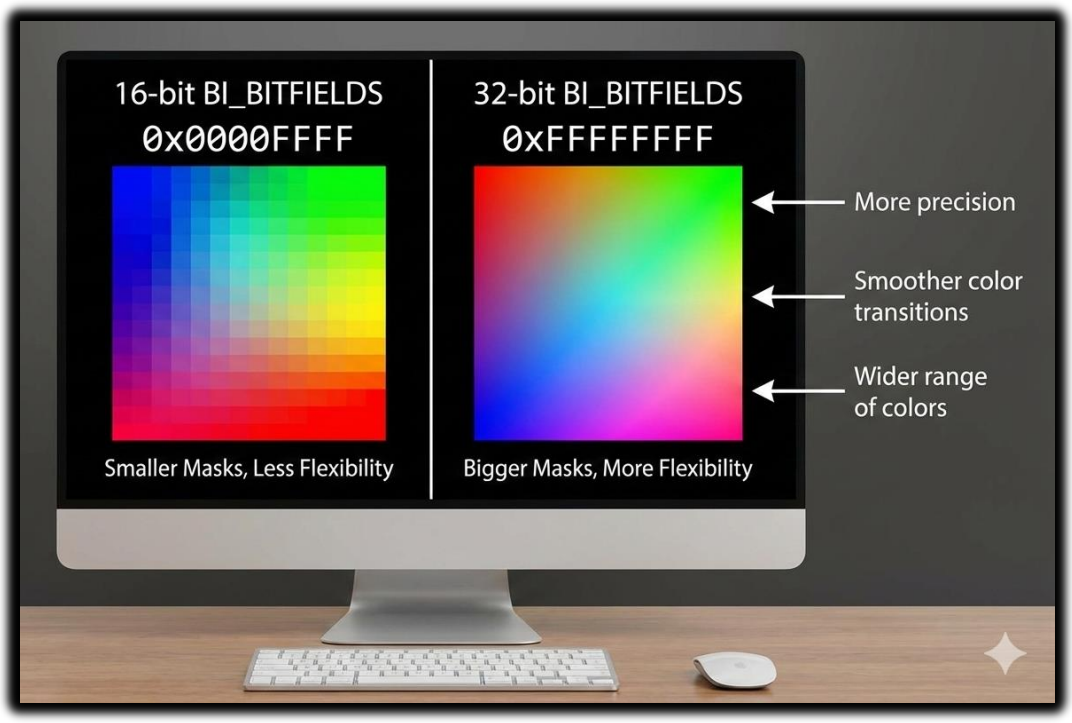
- In 16-bit images, masks usually fit inside 0x0000FFFF
- In 32-bit images, masks can use **many more bits**

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.

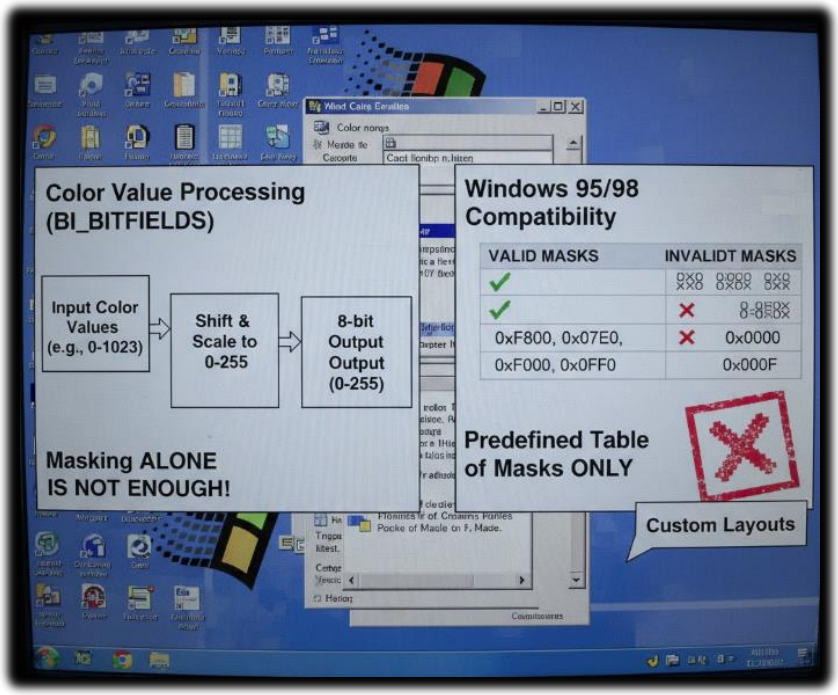
Windows 95 / 98 Compatibility (Old but Important)

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)	0×00007C00	0×000003E0	0×0000001F	5-5-5
16-Bit DIB (5-6-5)	0×0000F800	0×000007E0	0×0000001F	5-6-5
32-Bit DIB (8-8-8)	0×00FF0000	0×0000FF00	0×000000FF	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.

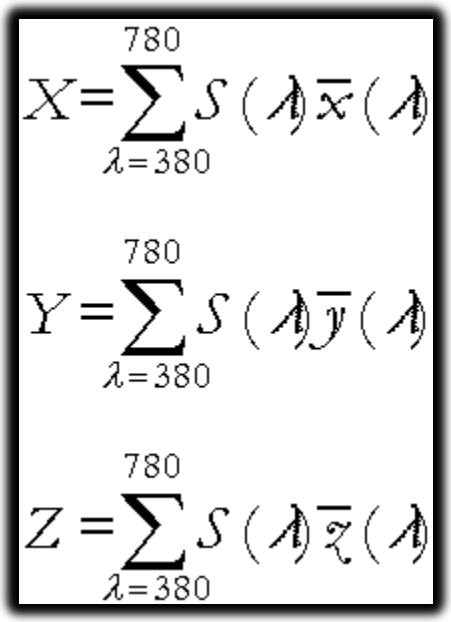
Most of the time, this all happens **quietly in the background** — but understanding it explains *why* modern systems can keep colors looking right almost everywhere.

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



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(λ)**
  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.
- **dλ**
  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.

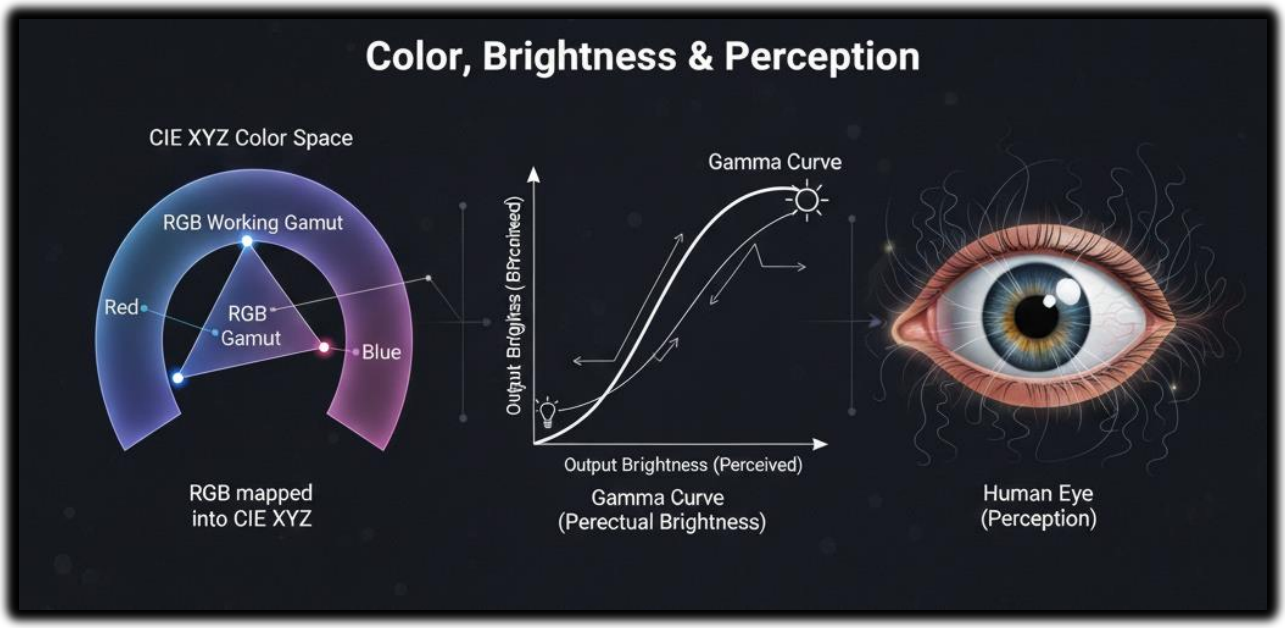
Cameras and gamma
Video cameras often adjust gamma (usually around 0.45) so images look normal and consistent on screens.

```
// 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)^γ** 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 of what we've talked about:

- **CIE XYZ**: A standard color system that works the same on any device.
- **BITMAPV4HEADER**: Stores color space info in image files for consistent colors.
- **bV4Endpoints**: Shows the true meaning of red, green, and blue in XYZ values.
- **Gamma correction**: Fixes uneven brightness from displays and cameras, making colors look natural.
- Video cameras usually use gamma correction to match screen behavior and ensure images look correct.

NONLINEAR RESPONSE AND HUMAN PERCEPTION

When we talk about monitors and images, there's a big idea to understand:

Humans don't see light in a straight line.

A monitor might double the voltage to make a pixel "twice as bright," but our eyes **won't see it as exactly twice as bright**.

Instead, our perception of brightness grows more slowly at higher light levels. This is called a **nonlinear response**, and monitors actually take advantage of it.

CIE Luminance and Lightness (Y and L*)

To describe brightness in a way that matches human vision, the **CIE** defines:

1. **Y** – The actual linear luminance of a color (how much light energy is there)
2. **L*** – The perceived lightness, which approximates how humans see brightness

The formula for L* looks fancy, but here's the **idea**:

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

Yn is the luminance of pure white (the brightest point the system can show)

L* ranges from **0 to 100**, where:

- ✓ 0 = black
- ✓ 100 = white

Each whole number increment in L* roughly represents the **smallest change in brightness your eyes can notice**

Think of L* as a **human-friendly brightness scale**.

Why L* Is Better Than Linear Luminance

Imagine storing pixel brightness directly as **Y values**.

- At very low brightness, tiny differences in Y are noticeable
- At very high brightness, the same differences in Y might not be noticed at all

This wastes **data precision**.

Instead, we store brightness using **perceptual lightness (L)***:

I. Convert pixel value to a normalized intensity:

Intensity = (P / 255) ^{g}

- **P** = pixel value (0–255)
- **g** = monitor gamma
- Assume black = 0

II. Then take the cube root (or apply the L* formula)

- This matches **how humans actually perceive brightness**
- It ensures that changes in pixel values correspond to changes we can see

In short: L* makes every step in pixel brightness **meaningful to our eyes**.

Gamma Correction in BITMAPV4HEADER

The last three fields of **BITMAPV4HEADER** store **gamma values**.
Gamma is basically a number that tells the program:

"How do pixel values relate to perceived brightness?"

Stored as **16-bit integer + 16-bit fraction**

Example:

- 0x10000 = 1.0 (linear)
- 0x23333 ≈ 2.2 (typical for real-world images)

If the image came from a real photo:

- Gamma ≈ 2.2 (matches how cameras and monitors work)

If the image is generated by a program:

- Linear luminance is converted to **perceptual L*** using a power function
- The inverse of that exponent gives the gamma value stored in the DIB

This ensures:

- The monitor shows brightness in a **way our eyes expect**
- The image looks **natural and smooth**, not too dark or too bright

---

Key Takeaways

1. **Human eyes are nonlinear** — we notice small changes in dark areas more than bright areas.
2. **L*** maps linear light to perceptual brightness, making pixel values visually meaningful.
3. **Gamma correction** ensures monitors display images according to how we perceive light.
4. Using L* and gamma makes images **look natural** on any device.

Analogy

Think of it like a **volume knob**:

- A linear knob might go from 0–100, but your ears don't hear a linear increase in volume.
- Gamma correction is like **reshaping the knob** so that each step feels like an equal increase in loudness.
- L* is your **perceptually balanced scale for brightness**, just like adjusting volume so music feels smooth at every level.

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

Coding Pixel Brightness Using Perceptual Lightness

When we deal with images and monitors, it's important to remember:

Humans don't see brightness in a straight line.

A monitor might double the light output, but our eyes **won't see it as twice as bright**. To handle this, we use **perceptual lightness (L*)** instead of raw linear luminance.

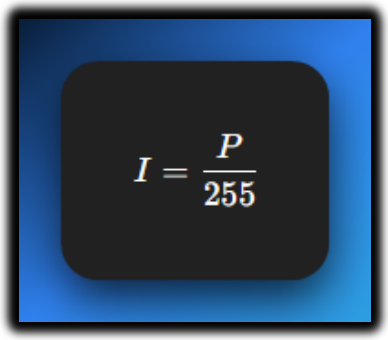
I. Why Perceptual Lightness?

- Using **linear light values** wastes bits in bright areas where the eye can't notice small differences.
- **L*** matches human perception, so each step in pixel value corresponds to a visible change in brightness.
- This also **reduces noise in analog circuitry** and keeps the number of bits needed reasonable.

II. Step-by-Step Process

🧠 **Start with the Pixel Value** - Each pixel has a value **P** between 0 (black) and 255 (white).

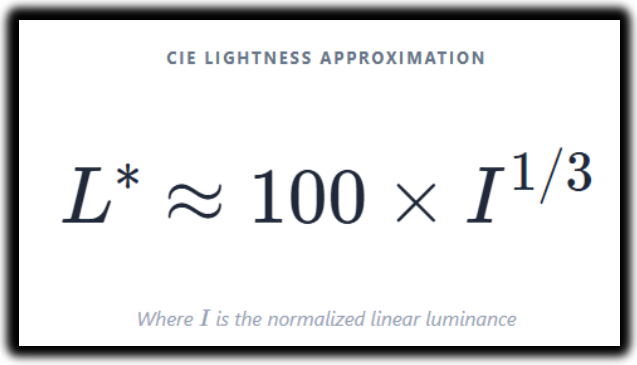
🧠 **Normalize to a Voltage/Intensity -** Convert P to a number between **0.0 and 1.0**:



This is like setting a dimmer on a lamp from 0 (off) to 1 (full brightness). Assuming the monitor's black level is 0, this voltage directly corresponds to the **intensity of light emitted** by the pixel.

🧠 **Account for Human Perception (Gamma / Cube Root)**

- Humans perceive brightness **nonlinearly**. To make the pixel values feel right, we transform intensity to **perceptual lightness (L*)**.
- A good approximation uses the **cube root of intensity**:



OR equivalently, using P directly:

L* ranges from **0 (black) to 100 (white)**, with each step roughly representing the smallest change in light the human eye can notice.

In simpler terms: the cube root "reshapes" brightness so that **dark areas get more precision**, and bright areas don't waste bits.

III. Exponent and Gamma

- The exponent used in the cube root formula is typically around **0.85**, not exactly 1.
- If it were 1, L* would match the linear pixel values exactly, but our eyes don't see linearly, so the cube root gives a **much closer match to human perception**.
- This exponent is closely related to **gamma correction**, which adjusts pixel values for how humans perceive brightness.

IV. How Gamma Works in BITMAPV4HEADER

The last three fields in **BITMAPV4HEADER** let programs **store the gamma value** associated with the pixel data.

Gamma is stored as **16-bit integer + 16-bit fractional** values.

- Example: 0x10000 = gamma 1.0 (linear)
- Example: 0x23333 ≈ gamma 2.2 (common for real photos)

If a DIB is captured from a camera:

- The hardware usually sets gamma (≈ 2.2)

If a DIB is generated by a program:

- Linear luminance is converted to **L*** using a power function
- The inverse of the exponent is then stored as the gamma in the DIB

This ensures that the **image appears correct on a monitor**, matching human perception no matter the device.

V. Big Picture

1. Pixel value P → normalized intensity (0–1)
2. Intensity → cube root → perceptual lightness L*
3. Gamma correction → ensures monitors display it correctly
4. Result → every step in brightness is meaningful to human eyes

VI. Analogy

Think of it like **a dimmer for a lamp**:

- Linear dimmer: halfway doesn't feel half as bright
- Perceptual dimmer (L*): halfway really **feels halfway**, matching human perception
- Gamma = the math behind adjusting the dimmer so the room looks natural

VII. Some additions before we move on

1. LINEAR INTENSITY (GAMMA)

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

P = Pixel Value (0-255) | γ = Display Gamma (Standard: 2.2)

2. CIE PERCEPTUAL LIGHTNESS

$$L^* \approx 100 \cdot I^{1/3}$$

L = Human Perceived Brightness (Scale: 0-100)*

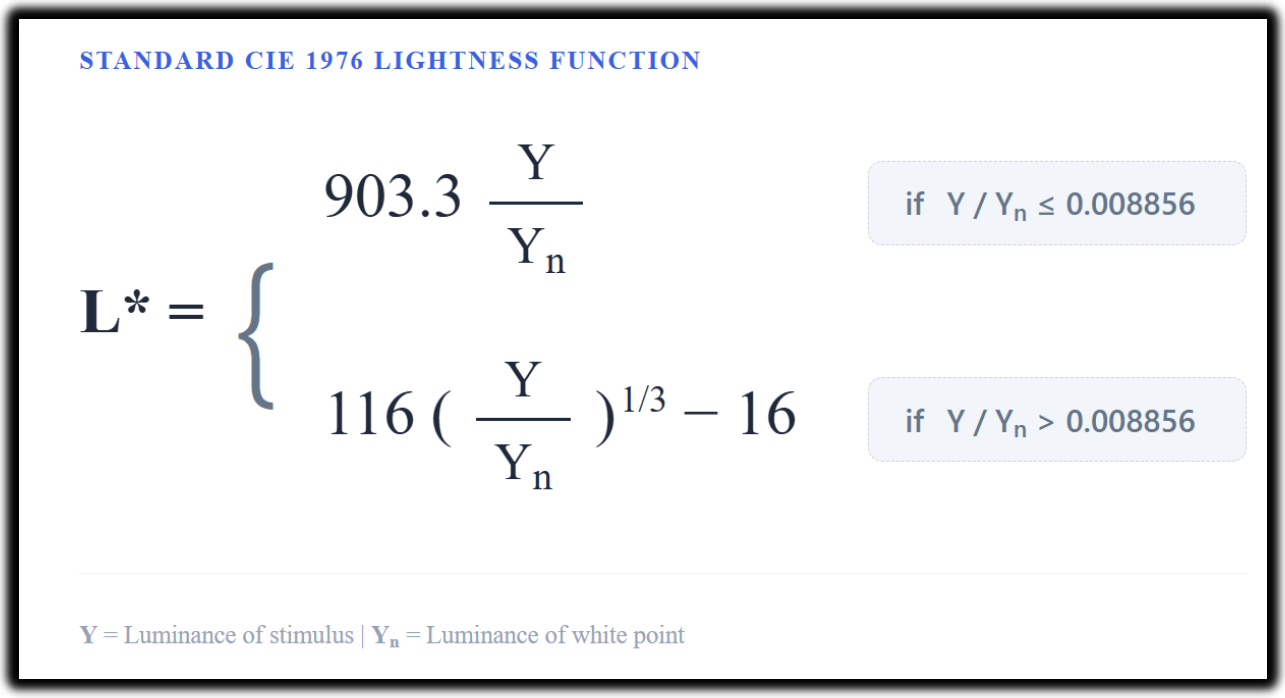
3. PETZOLD'S APPROXIMATION

$$L^* \approx 100 \cdot \left(\frac{P}{255} \right)^{v/5}$$

v = Exponent (~0.85 for 16.16 Fixed Point Gamma)

Let's explain the formulas...

L Lightness Formula (CIE Lab)*



Here's what each part means in simple terms:

- **L*** – The **CIE Lightness**, our final "human-perceived" lightness value. It goes from 0 (black) to 100 (white).
- **Y** – The **linear luminance**, meaning the actual light intensity of a color.
- **Yn** – The **reference white level**, used to normalize values so that the same formula works under different lighting conditions.
- **(Y / Yn)** – Dividing the color's brightness by the white reference gives a proportion of maximum brightness.
- **( )^(1/3)** – The **cube root**, which converts linear light intensity into a value that better matches how we perceive brightness.
- **100** – Just a scaling factor to make the final L* range from 0 to 100, which is easy to work with.

I. Key Points to Remember

1. **Reflects Human Perception:** The cube root makes L* nonlinear, which aligns with how our eyes perceive changes in light.
2. **Normalized Lightness:** By dividing by the reference white **Yn**, we can compare lightness consistently across different scenes or images.
3. **Practical Range:** L* values are scaled from 0–100, giving a straightforward way to measure "how light or dark" a color looks.
4. **Gamma in BITMAPV4HEADER:** Images often store gamma information in this header so that software knows how to interpret the pixel values correctly, ensuring the image **looks right on different monitors**.

II. Analogy 💡

Think of L* like a **volume knob for brightness** that's tuned to human eyes. If linear luminance (Y) were a raw electrical signal, L* adjusts it so that each step feels like the same perceptual difference in light to our vision.

Components of the Pixel Intensity Formula

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

OUTPUT (I)	INPUT (V)	SOURCE (P)
Linear Intensity	***Normalized Voltage***	***8-bit Pixel Value***

NOTE: γ (Gamma) is typically 2.2 for sRGB displays.

- **I (Intensity):** How bright the pixel actually looks on the screen.
- **V (Voltage):** The electrical signal the monitor uses to display that brightness.
- **P (Pixel Value):** The number stored in the image for that pixel, from **0 (black) to 255 (white)**.
- **255:** This is just a normalization factor, turning the pixel value into a number between 0 and 1.
- **Exponent (power):** This controls the nonlinear relationship between the pixel value and the voltage or intensity.

I. What the Exponent Usually Is

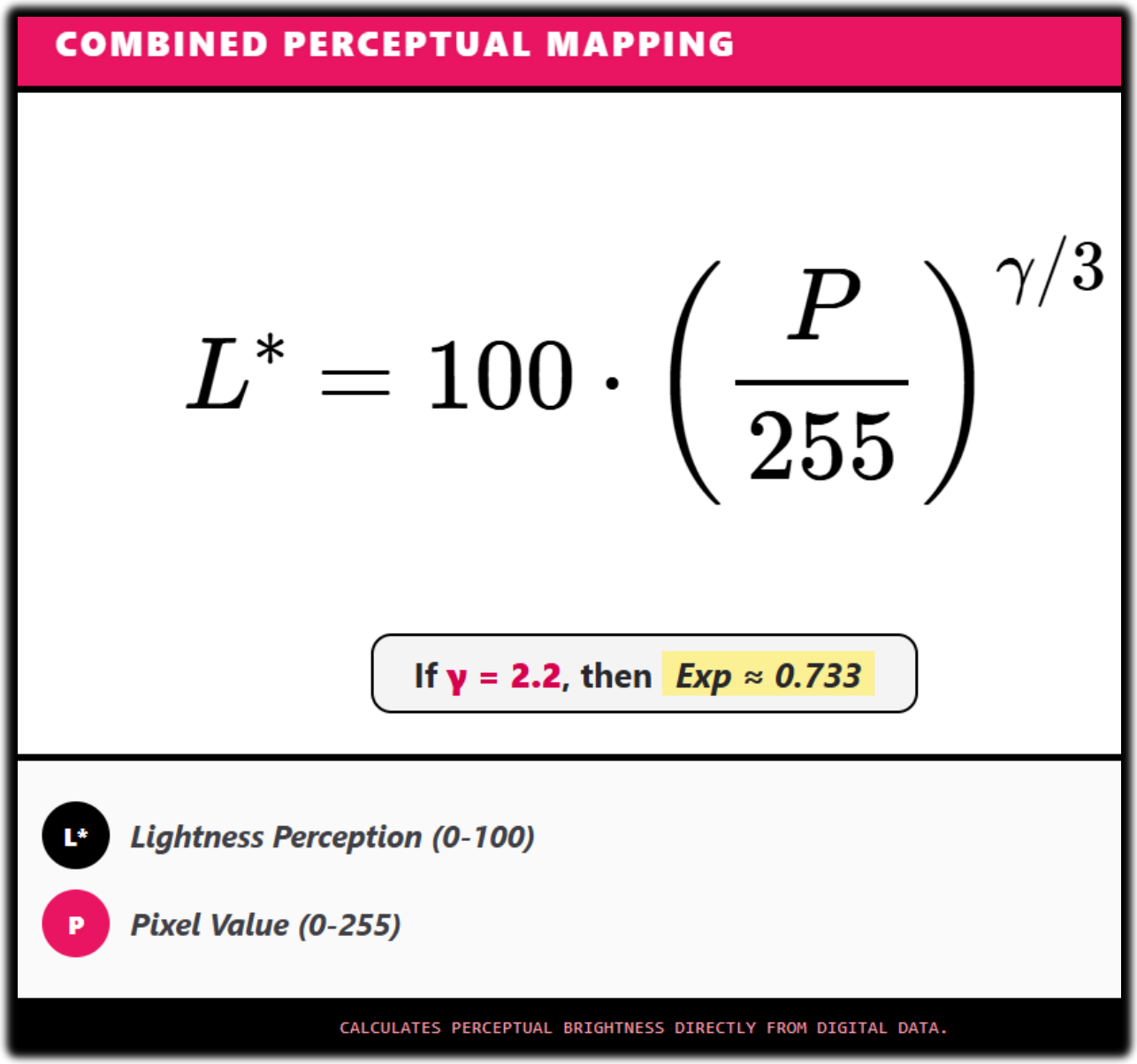
- **Gamma (γ):** Most of the time, this exponent is **gamma**, which is usually around **2.2**. It accounts for the fact that monitors (especially old CRTs) don't respond linearly to voltage. This means a small change in the pixel value doesn't make a small change in brightness—it's more noticeable.
- **Other Exponents:** Depending on the situation, other exponents might be used. Some image processing algorithms tweak this exponent to achieve special effects or correct for specific hardware.

II. Understanding the Formula

The formula turns a file's numbers into the electrical signals that light up your screen.

It uses a specific 'curve' (the exponent) because humans don't see brightness in a straight line—we're more sensitive to changes in shadows than in bright light.

L (Lightness) Formula



I. The Components

- **L*** (Lightness): This is the final result. It's a score from 0 (Pitch Black) to 100 (Blinding White).
- **P** (Pixel Value): This is the raw number from your image file (0 to 255).
- **255**: We divide by this to turn the pixel number into a percentage (between 0 and 1).
- **Exponent (EXP)**: This is the "secret sauce." It's a math trick that curves the brightness so it looks natural to humans.

II. The "Secret Sauce" (The Exponent)

The notes give two options for the math trick:

- **The Cube Root (1/3)**: A simple, old-school way to adjust brightness. Think of it like a "basic" filter.
- **The Gamma Version (gamma/3)**: A more precise method. It accounts for how your specific monitor (like your phone or laptop screen) handles light. Most screens use a gamma of 2.2.

III. The "Why" Behind the 100

- The final L* value usually ends up between 0 and 100. Think of it like a Lightness Percentage.
- If L* = 50, your brain thinks, "That's exactly middle-gray."
- If the computer just used raw numbers, a "50" might look too dark. This formula boosts the middle values so they look correct.

IV. Summary in one sentence

This formula takes a raw pixel number (0–255), turns it into a percentage, and then applies a math "curve" so the brightness you see matches what the computer intended.

BITMAPV5HEADER Overview and Color Management

```
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;
```

I. Purpose:

BITMAPV5HEADER builds on previous DIB headers to give better control over colors and color management. It's 124 bytes long and adds fields for color space, gamma, and ICC profiles.

II. Key Fields:

- bV5CSType: Specifies the color space (sRGB, calibrated RGB, Windows default, or embedded/linked ICC profile).
- bV5Endpoints: Holds XYZ values that define the color space endpoints.
- bV5GammaRed, bV5GammaGreen, bV5GammaBlue: Gamma values for each channel.
- bV5ProfileData & bV5ProfileSize: Handle ICC profile information for device-independent color.

III. Why It Matters:

- BITMAPV5HEADER lets programs store images with precise, device-independent colors.
- ICC profiles ensure colors stay consistent across monitors, printers, and other devices.

Working with DIB Files in Code

A typical Windows program for inspecting DIB headers (like DIBHEADS.c) works as follows:

1. **Open the File:** The program reads the DIB file into memory.
2. **Check the File Size:** Ensures the file is manageable.
3. **Read Headers:** Extracts the BITMAPFILEHEADER and one of the information headers (CORE, INFO, V4, or V5).
4. **Display Information:**
    - ✓ Shows width, height, bit depth, compression, color masks, gamma, and profile info.
    - ✓ For BITMAPV5HEADER, includes advanced color management fields.
5. **User Interface:**
    - ✓ Simple Windows interface with "File → Open" menu.
    - ✓ Opens a file dialog, lets the user select a DIB, and displays header info in a text field.
6. **Error Handling:** Any issues (file missing, read error) are reported gracefully.
7. **Optional Rendering:** To actually display or print the image, Windows functions like SetDIBitsToDevice or StretchDIBits can be used.

DIB Structure at a Glance

1. **BITMAPFILEHEADER:** Basic file info (type, size, pixel offset).
2. **BITMAPINFOHEADER / BITMAPV5HEADER:** Image properties (width, height, bit depth, compression, color masks, gamma, color space).
3. **Color Table (optional):** Only for indexed-color images, mapping pixel values to RGB colors.
4. **Pixel Data:** Actual pixel values, organized according to bit depth and compression.

💡 **Tip:** Most of the technical details (field names, offsets) can be read directly from the code or documentation. The important takeaway is: **BITMAPV5HEADER + ICC profiles = accurate, device-independent color handling.**

DISPLAYING DIBS AND PACKED DIBS

When you display a DIB (Device-Independent Bitmap), you need a few key pieces of information:

- Image width and height
- Color depth (bits per pixel)
- Compression type
- Color masks (if used)
- The actual pixel data

These details let the system correctly interpret and render the image on the screen or other output devices.

I. Packed DIBs: Storing a DIB in Memory

A **packed DIB** is when the entire DIB (except the file header) is stored as a single, continuous block of memory.

The memory starts at the **information header**, followed immediately by the **color table** (if there is one) and then the **pixel data**.

Think of it like this: everything needed to display or manipulate the image is stored in one neat package. This makes it easy to:

- Access image properties like width, height, color depth, or compression
- Loop through pixel data to read or modify colors
- Apply image processing operations like filters or transformations

In code, a packed DIB is usually referenced by a single pointer, such as pPackedDib. By using this pointer, you can navigate through all the sections in memory without worrying about separate blocks.

II. Why Packed DIBs Are Useful

Packed DIBs are common in situations like:

- Copying images through the clipboard
- Creating brushes from a DIB for drawing

They have several advantages:

1. **Compact Storage:** Keeps the image data together, reducing memory overhead.
2. **Easy File Handling:** Matches the structure used in BMP files, so it's easy to save or load directly.
3. **Simple Structure:** With everything in one block, accessing and manipulating pixel data is straightforward.

A packed DIB is basically a “ready-to-use” DIB in memory. It combines the information header, optional color table, and pixel data into one continuous block, making it easy to display, manipulate, or store.

III. 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.

IV. Limitations and Considerations

Even though packed DIBs are widely used, there are some things to keep in mind:

- **Different DIB versions**: Old OS/2 formats and newer versions store information differently. This means your code might need extra checks to read the right data.
- **Accessing pixel data**: Some fields, like pixel width, aren’t always in the same place. Older formats might need special handling.
- **Compatibility matters**: Always check the DIB format before using its fields. Skipping this can lead to errors or crashes.

Tip: Think of it like reading books from different editions — the chapters (fields) might be in different orders, so you need to check the edition before jumping in.

ACCESSING PIXEL WIDTH IN A PACKED DIB

Getting information from a packed DIB isn't always as simple as it looks because DIBs can come in different formats. One common task is retrieving the **pixel width** of the image.

A straightforward approach might look like this:

```
iWidth = ((PBITMAPINFOHEADER) pPackedDib)->biWidth;
```

…but this only works for standard Windows DIB formats. Some DIBs, especially **OS/2-compatible ones**, use a slightly different structure, so a simple cast may not work correctly.

To handle both cases, you need a **conditional check**:

```
if (((PBITMAPCOREHEADER)pPackedDib)->bcSize == sizeof(BITMAPCOREHEADER))
    iWidth = ((PBITMAPCOREHEADER)pPackedDib)->bcWidth; // OS/2 format
else
    iWidth = ((PBITMAPINFOHEADER)pPackedDib)->biWidth;  // Standard Windows format
```

Here's what's happening:

- **bcSize check:** If the size of the header matches BITMAPCOREHEADER, we know it's an OS/2-compatible DIB.
- **OS/2 DIB:** Use bcWidth to get the width in pixels.
- **Windows DIB:** Use biWidth from the BITMAPINFOHEADER structure.

This ensures your code works no matter what kind of DIB you're dealing with. It's a small but crucial step because accessing the wrong field can lead to incorrect image processing or crashes.

Fun Exercise: Accessing a Specific Pixel

Suppose you want the pixel at coordinate **(5, 27)**. To do this properly, you need:

- Image width and height
- Bits per pixel (bit count)
- Row byte length (how many bytes each row takes)
- Color table entries (for indexed images)
- Any color masks (for BI_BITFIELDS formats)
- Compression method (if used)

Directly calculating pixel addresses can get messy, especially for image processing tasks. That's why **C++ classes for DIBs** are handy—they allow **fast random access** to pixel data without writing complex pointer arithmetic.

For now, in plain C, you'll do manual calculations, but we'll cover efficient methods in the next chapter.

SETDIBITSTODEVICE AND STRETCHDIBITS FUNCTIONS

To use these functions, you need:

A pointer to the **BITMAPINFO** structure of the DIB. This includes:

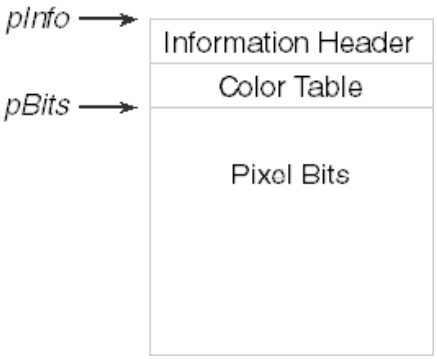
- **BITMAPINFOHEADER** (image metadata)
- The **color table**

A pointer to the **pixel bits**.

Getting the pixel pointer is usually easy if you have the **bfOffBits** field from the BITMAPFILEHEADER, which tells you where the pixel data starts.

Important note: If you get a packed DIB from the clipboard, it might **not** include a BITMAPFILEHEADER. This makes it trickier to find the pixel data.

Tip: Think of bfOffBits like a "map" telling you where the actual pixels live in the file. Without it, you have to search a bit more carefully.



I. Why Two Pointers Are Needed

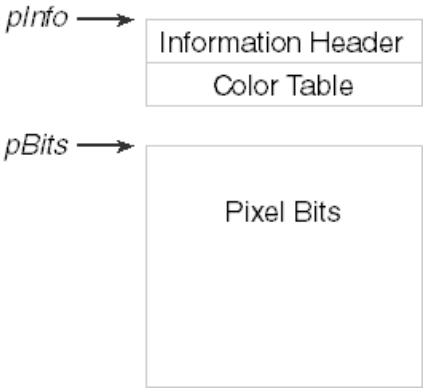
The **SetDIBitsToDevice** and **StretchDIBits** functions need **two separate pointers** to a DIB:

1. One pointer for the **header** (BITMAPINFO + color table)
2. One pointer for the **pixel data**

Why? Because the header and pixel data **don't have to be stored in one continuous block of memory**.

- Sometimes they are in **two separate memory blocks**, and these functions can handle that.

Tip: Think of it like a book where the table of contents and the pages are stored in different locations—you need a pointer to both to read the full story.



When using **SetDIBitsToDevice** or **StretchDIBits**, you work with **two separate pointers**:

lpBits – points to the **pixel data**

- This is the block of memory that holds the actual pixel colors.
- Pixel data can be in different formats, like **uncompressed RGB** or **compressed formats**.

lpBitmapInfo – points to the **header section**

- This memory block contains the DIB's **header information**, like the **BITMAPINFO / BITMAPINFOHEADER** structure.
- It also includes the **color table** if the image uses one.

By giving these two pointers to the functions, you can:

- Specify **exactly where the header and pixel data live**, independently.
- Work with DIBs stored in **non-contiguous memory blocks**, which gives more flexibility in memory management.

Tip: Think of it like a recipe book where the ingredients (header info) are in one folder, and the instructions (pixel data) are in another—you need both to make the dish.

II. Example Call (conceptual):

```
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 functions like **SetDIBitsToDevice** and **StretchDIBits**, you usually pass **two pointers**:

- **lpBitmapInfo** → points to the DIB header (format, size, color info)
- **lpBits** → points to the actual pixel data

Why two pointers?

Because Windows allows flexibility:

- the header and pixels can be in **separate memory blocks**
- or both can live together in **one packed DIB**

This lets the API work with many memory layouts without forcing one structure.

III. Packed DIB vs Split Memory

- **Packed DIB**
  Header + color table + pixels all in one memory block
  (very common, especially when loading BMP files)
- **Split layout**
  Header in one place, pixels somewhere else
  (useful for custom allocation or streaming)

Windows supports both — that's why the API uses two pointers.

IV. Why Width and Height Still Matter

Even when you already have lpBits, Windows still needs:

- **pixel width**
- **pixel height**

These values tell Windows:

- how big the image is
- where pixel rows start and end
- how far it's safe to read or write

Even if you only draw **part** of the image, width and height act as:

hard boundaries for the pixel array

They prevent:

- reading past the buffer
- writing into invalid memory
- corrupted output or crashes

So yes — always pass them.

V. Working with Only Part of a DIB (Rectangle Idea)

Think of the DIB pixel data as a **2D grid stored in memory**.

To work on a rectangle:

- you choose a starting (x, y)
- you choose a rectangle width and height
- Windows uses the full DIB width to move row by row

The full image size defines the **limits**.
Your rectangle must stay inside those limits.

No bounds → undefined behavior.

VI. Common DIB-Related Functions (The Useful Ones)

CreateDIBSection

- Allocates a DIB
- Gives you a **direct pointer to pixel memory**
- Best choice when you want to write pixels yourself

GetDIBits

- Copies pixels **out of** a bitmap
- Useful for analysis or conversion

SetDIBits / SetDIBitsToDevice

- Sends DIB data **to a device**
- Used for drawing without creating a DDB first

StretchDIBits

- Same as above, but with scaling
- Slower than BitBlt, but more flexible

BitBlt

- Very fast
- Works on **DDBs**, not raw DIB memory
- Often used after converting a DIB → DDB

VII. Performance Notes (Important)

- **Packed DIBs are cache-friendly**
  One memory block = fewer cache misses
- **CreateDIBSection is fastest for editing**
  You write pixels directly, no copies
- **StretchDIBits is slower**
  Scaling + format handling costs CPU
- **Avoid repeated conversions**
  DIB ↔ DDB back and forth is expensive

Rule of thumb:

- edit pixels → **DIB**
- draw fast → **DDB**
- convert only when needed

VIII. Final Mental Model (Lock This In)

- lpBitmapInfo = how pixels are described
- lpBits = where pixels actually live
- width + height = safety rails

A DIB is:

structured pixel memory with rules

Break the rules → bugs.
Follow them → clean, predictable graphics.

DEFINING A RECTANGLE INSIDE A DIB

A DIB's pixels are stored in a 2D grid.

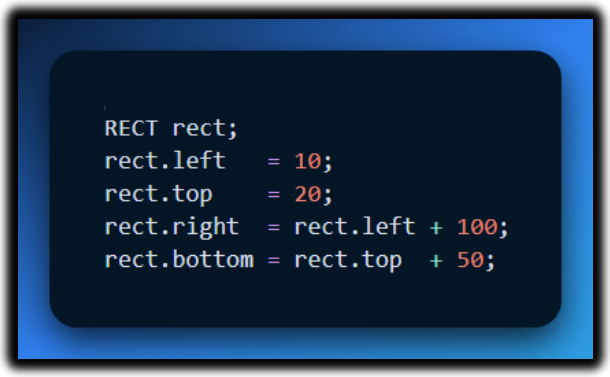
To work with **part of the image**, you define a rectangle using:

- a starting position (x, y)
- a width
- a height

Coordinates (0, 0) refer to the **top-left corner** of the DIB.

Example

A rectangle that starts at (10, 20)
and is 100 × 50 pixels:



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

That's it.

This rectangle must stay **inside the full DIB width and height**, or you'll read past the pixel buffer.

I. What This Really Means in Memory

Even when you draw or process a rectangle:

- the DIB is still one full pixel array
- the rectangle just defines **which pixels you touch**

The full DIB size sets the **limits**.
The rectangle selects a **region inside those limits**.

II. Performance Notes

Packed DIBs

- Stored in **one memory block**
- Cache-friendly
- Fast to scan line by line

This is why they're commonly used.

Memory Access Pattern

Best case:

- process pixels **row by row**
- avoid jumping around memory

Sequential access = fewer cache misses = faster code.

Alignment

- CPUs prefer aligned memory
- Windows DIB scanlines are already **DWORD-aligned**
- If you follow the API rules, you're usually fine

No need to overthink this.

Scaling and Conversion

- **StretchDIBits** is slower (scaling costs CPU)
- **BitBlt** is fast but works on DDBs
- Don't convert DIB ↔ DDB repeatedly unless needed

III. What We're *Not* Doing Here

You do **not** need:

- OpenCV
- ImageMagick
- Pillow
- SIMD theory
- compiler optimization lectures

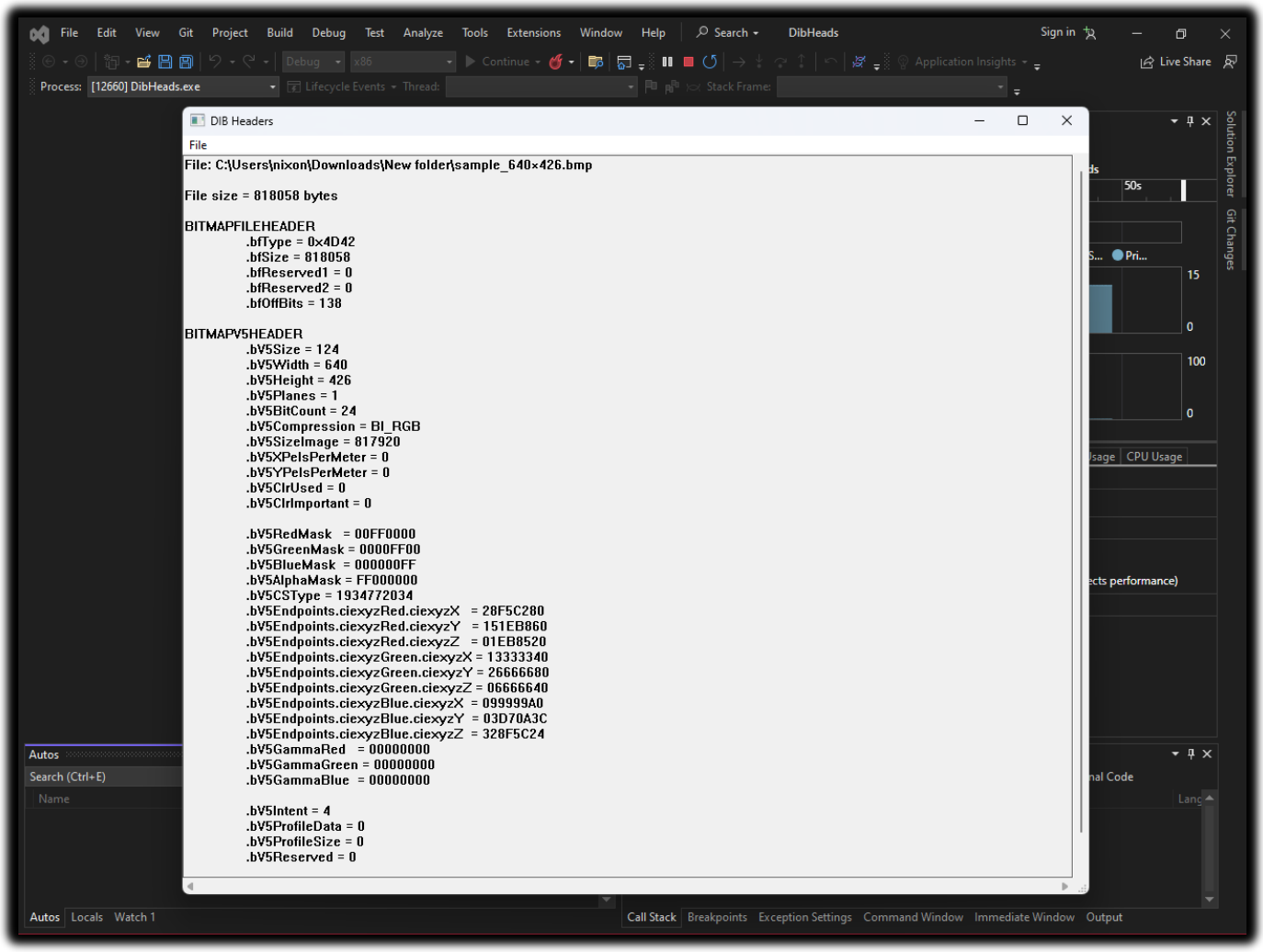
That's outside the scope of **WinAPI DIB fundamentals**.

IV. Mental Model (Keep This)

- DIB = full pixel grid in memory
- Rectangle = limits inside that grid
- Width and height = safety rails
- Sequential access = speed

Anything beyond this is **advanced optimization**, not core knowledge.

DISPLAYING A DIB WITH SETDIBITSTODEVICE



In our code, after reading the **BMP file** and parsing the DIB headers, we ended up with:

- a pointer to the **DIB header + color table**
- a pointer to the **pixel bits**

JPEG and other formats are different — BMP already matches the DIB layout, so it's easier.

I. What SetDIBitsToDevice Does

SetDIBitsToDevice draws a **DIB directly** to a device (screen or printer).

Important:

- **No scaling**
- **1 DIB pixel = 1 device pixel**
- Image orientation is handled correctly by Windows

This function is for **simple, exact display**.

```
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
);
```

II. The Only Arguments That Really Matter

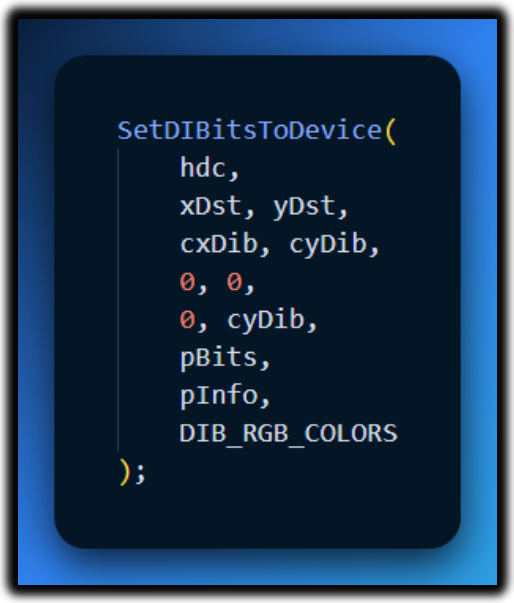
Here's what you actually need to understand:

- **hdc**
  The device context to draw to (screen, window, printer).
- **xDst, yDst**
  Where the image appears on the output device (top-left corner).
- **cxSrc, cySrc**
  Width and height of the source image (or part of it), in **pixels**.
- **xSrc, ySrc**
  Where to start reading pixels inside the DIB.
  Usually 0, 0.

- **yScan, cyScan**
  Which scanlines to draw.
  Usually draw everything.
- **pBits**
  Pointer to the pixel data.
- **pInfo**
  Pointer to BITMAPINFO (header + color table).
- **fClrUse**
  Almost always DIB_RGB_COLORS.

III. Typical Call (Entire DIB)

Most programs use **these exact values**:



That draws the **entire image**, starting from the top-left.

IV. About the "Upside-Down" DIB Thing

Yes, DIB pixel data is stored **bottom-up**.

But:

- SetDIBitsToDevice already understands this
- you do **not** flip anything manually for normal use

Just remember:

- cyDib = abs(biHeight)

That's enough.

V. Pixel Coordinates vs Logical Coordinates

Despite what some docs say:

- xSrc, ySrc, cxSrc, cySrc are **pixel values**
- not logical units
- not affected by mapping modes

Mapping modes affect **where** the image is placed,
not its **size or orientation**.

VI. Sequential Scanline Drawing (Advanced)

The parameters:

- yScan
- cyScan

exist so Windows can:

- draw a DIB **piece by piece**
- save memory for very large images

Example use case:

- loading a huge BMP from disk
- streaming image data

For normal programs:

- yScan = 0
- cyScan = image height

Ignore this unless memory is tight.

VII. Color Table Flag (fClrUse)

- Use DIB_RGB_COLORS for normal DIBs
- Use DIB_PAL_COLORS only when working with logical palettes

Most modern code:

VIII. What pInfo Can Point To

Usually:

- BITMAPINFO

But Windows also supports:

- BITMAPCOREINFO
- BITMAPV4HEADER
- BITMAPV5HEADER

As long as the header is valid, Windows figures it out.

You don't need to special-case this unless writing loaders.

IX. Return Value (Don't Ignore This)

SetDIBitsToDevice returns:

- number of scanlines drawn

If it returns 0:

- drawing failed

Always check it.

X. Final Mental Model

- DIB already describes its own pixels
- SetDIBitsToDevice just **copies them to a device**
- no scaling
- no format conversion
- no magic

If you want:

- scaling → StretchDIBits
- speed → convert to DDB + BitBlt
- pixel editing → CreateDIBSection

SHOWDIB1.C — WHAT ACTUALLY MATTERS

The SHOWDIB1 program is a **minimal DIB viewer**.

Its purpose is simple:

- load a BMP (DIB) from disk
- keep it in memory
- display it in a window using SetDIBitsToDevice

Everything else is just standard Win32 boilerplate.

What This Program Demonstrates (That You Should Learn)

1. BMP = DIB-Friendly

BMP files already store data in **DIB layout**.

That's why:

- BMP is easy to load
- JPEG is not (needs decoding)

This example is about **DIB handling**, not image formats.

2. DIB Is Loaded as One Memory Block

The image is loaded into memory as:

- BITMAPFILEHEADER
- followed by DIB header
- followed by pixel bits

The program then computes:

- pointer to BITMAPINFO
- pointer to pixel data
- image width and height

This is the **key DIB pattern**.

3. Display = One API Call

During WM_PAINT, the program simply calls:

- SetDIBitsToDevice
- using the stored pointers
- with no scaling
- no conversion

This shows the **purest DIB → screen path**.

4. File I/O Is Isolated (Good Design)

All file handling lives in:

- DIBFILE.C
- DIBFILE.H

That code:

- opens file dialogs
- loads the DIB into memory
- saves it back to disk

The window code does **not** care how files work.

That separation is intentional.

5. Menus Trigger DIB State Changes

- **File → Open**
  Loads a DIB into memory
- **File → Save**
  Writes the same memory block back to disk

Menu enable/disable logic just checks:

- "Do we currently have a DIB?"

Nothing fancy.

6. This Program Is Intentionally Incomplete

Petzold even tells you this.

Known limitations:

- no scrollbars
- no scaling
- no clipping
- no large-image handling

Those are **teaching hooks** for later chapters.

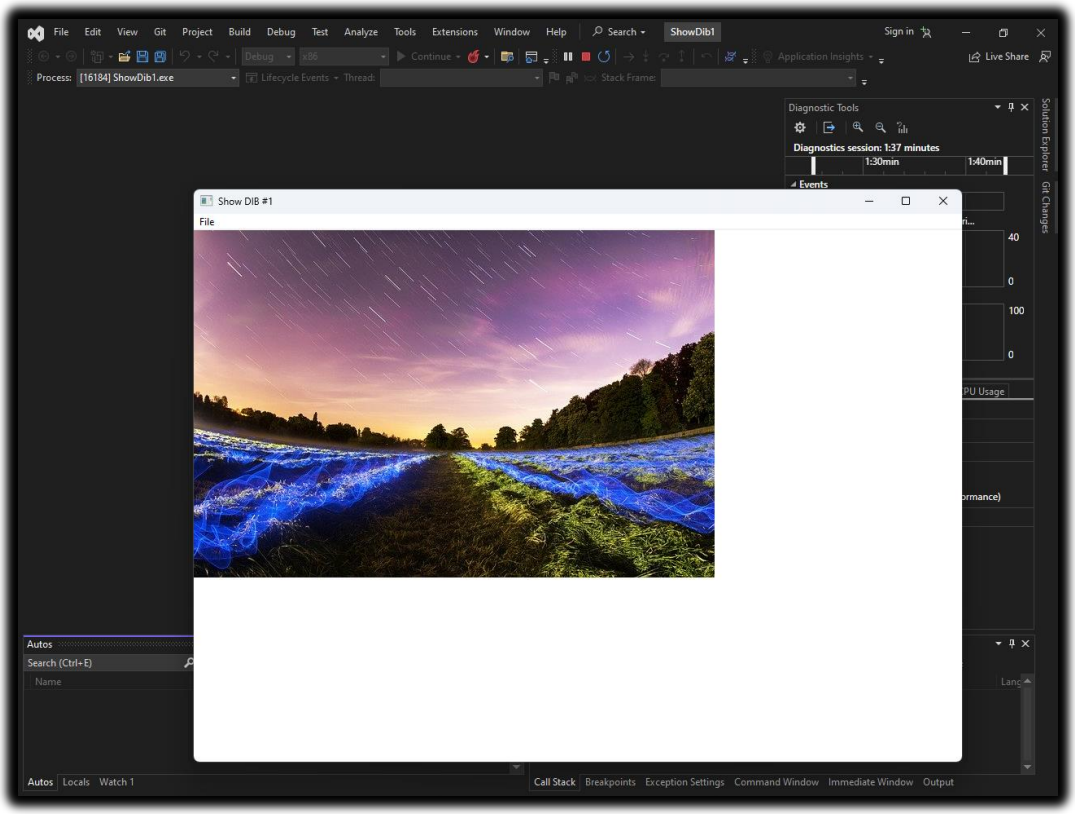
7. The Only Mental Model to Keep

SHOWDIB1 proves one thing:

A DIB is just memory, and Windows knows how to draw it.

Load memory → compute pointers → draw.

That's it.



DIBS: WHEN EARLY DESIGN DECISIONS COME BACK TO HAUNT YOU

Designing OS APIs is hard — **fixing bad early decisions is worse**.
Device-Independent Bitmaps (DIBs) are a perfect example.

Where the Problem Started (OS/2) 🧭

In **OS/2 Presentation Manager**:

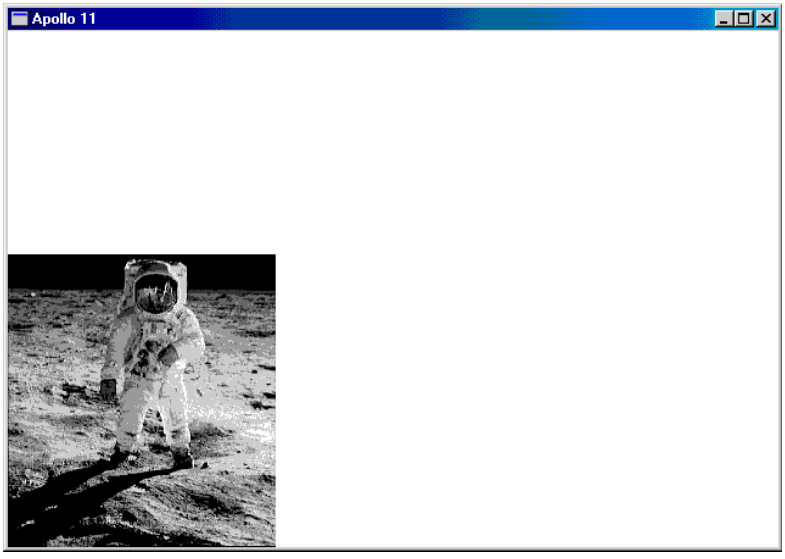
- The coordinate origin **(0,0)** is at the **bottom-left**
- Bitmaps were stored **bottom-up**
- Bitmap drawing functions also used **bottom-left coordinates**

So, if you drew a bitmap at (0,0), it appeared flush against:

- the **left**
- the **bottom** of the window

This was **internally consistent** and made sense *inside OS/2* — even if it felt weird to anyone not thinking like a mathematician.

✓ First pixel in bitmap data
✓ First row in memory
✓ Origin (0,0)
→ all lined up nicely



I. The Lesson ⚠

If the foundation is wrong, patching later just multiplies the mess.

The bottom-up DIB design worked in OS/2,
but became **confusing and painful** when mixed with other systems and APIs.

On slow machines, you could *literally see* the bitmap being drawn **from bottom to top**.

II. Windows Enters the Scene (and Breaks Consistency)

Windows kept **bottom-up DIB storage**, but changed how things are displayed.

Source coordinates (still OS/2-style):

- xSrc, ySrc are measured from the **first row of DIB data**
- The **first row is the bottom of the image**

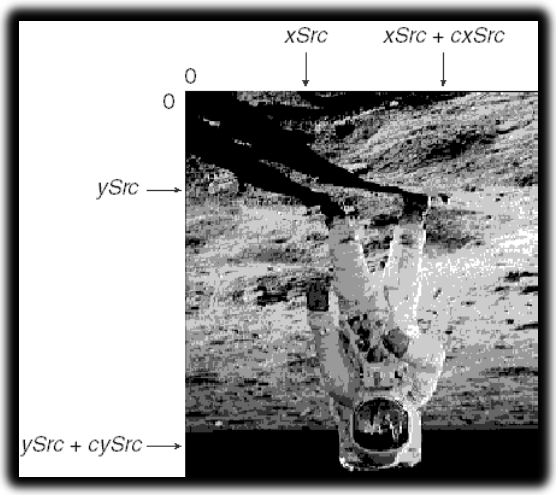
Destination behavior (Windows-style):

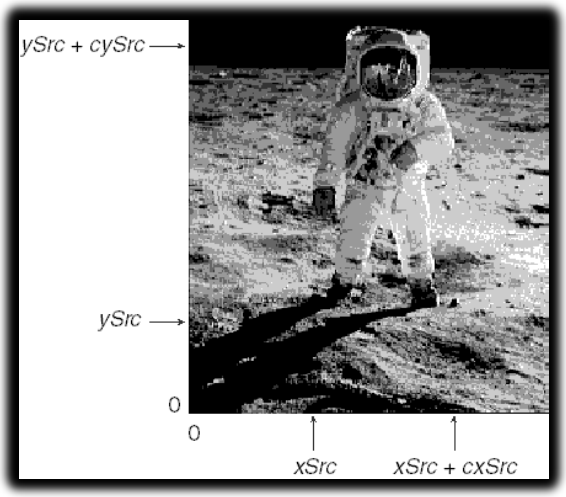
- Windows displays the **top row of the image** at (xDst, yDst)

So now:

- (xSrc, ySrc) **does NOT map** to (xDst, yDst)
- Instead, (xSrc, ySrc + cySrc − 1) maps to (xDst, yDst)

This is the core reason **DIBs feel insane** in WinAPI.





III. Why This Feels Chaotic

- DIB data starts at the **bottom**
- Image is displayed from the **top**
- Coordinates refer to **data order**, not visual order

Even worse:

- This behavior stays the same **regardless of mapping mode**
- MM_TEXT or not — the image looks the same, but the math stays cursed

IV. Top-Down DIBs (Negative biHeight): Not the Fix You Think 🔁

If biHeight is **negative**, the DIB is **top-down**.

Sounds like a solution, right?
Nope. 😐

Someone decided:

- Flip the rows
- Set biHeight positive
- Pretend nothing changed

The goal:
👉 *Don't break existing code*

The result:

- Programs **still** must handle top-down DIBs specially
- Source coordinates now have an origin at the **last row of data**
- (0,0) is:
    - ✓ not the first pixel
    - ✓ not the last pixel
    - ✓ somewhere awkward in the middle

This is a **brand-new mental model**, and not in a good way.

V. The One Smart Thing: SetDIBitsToDevice

Here's the **one genuine win** in this whole mess 👇

✅ Orientation-Independent Arguments

SetDIBitsToDevice uses arguments that **don't care** whether the DIB is:

- bottom-up **or**
- top-down

If two DIBs:

- contain the same image
- but have rows stored in opposite order

You can use the **exact same arguments** to draw the same part of the image.

🔥 This is why it works cleanly in the **APOLLO11** example.

VI. Final Takeaway (Straight Talk)

OS/2 was weird but **consistent**

Windows tried to be compatible and became **inconsistent**

DIBs are confusing because:

- data order ≠ visual order
- source ≠ destination

SetDIBitsToDevice is the rare API that **does the right thing**

MM_TEXT Mapping Mode and Coordinate Behavior

Because the arguments are used the same way every time, you don't need to rewrite your code for different DIB orientations.

Whether the image is top-down or bottom-up, the function calls stay the same.

This makes your code **simpler, cleaner, and easier to reuse** when working with DIBs in different situations.

Below is the same information, with the **xSrc coordinates shown 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 using **MM_TEXT** mapping mode (the default pixel-based mode), source and destination rectangles don't map in a perfectly intuitive way.

```
// 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** mode, the source point **(xSrc, ySrc)** does **not** directly map to **(xDst, yDst)**.

This makes coordinate calculations a bit more confusing compared to other mapping modes.

In other mapping modes, the mapping behaves more predictably:

- The point **(xSrc, ySrc + cySrc − 1)** maps to **(xDst, yDst)**.
- This preserves the visual appearance of the image, even though the coordinate systems differ.

Top-Down vs Bottom-Up DIBs

For **top-down DIBs**, the **biHeight** field in **BITMAPINFOHEADER** is **negative**.

This changes how source coordinates work:

- The origin starts at the **first row of pixel data**, not the last.
- This is different from **bottom-up DIBs**, where the origin is at the bottom row.

To specify a rectangle inside a **top-down DIB**, you use source coordinates like this:

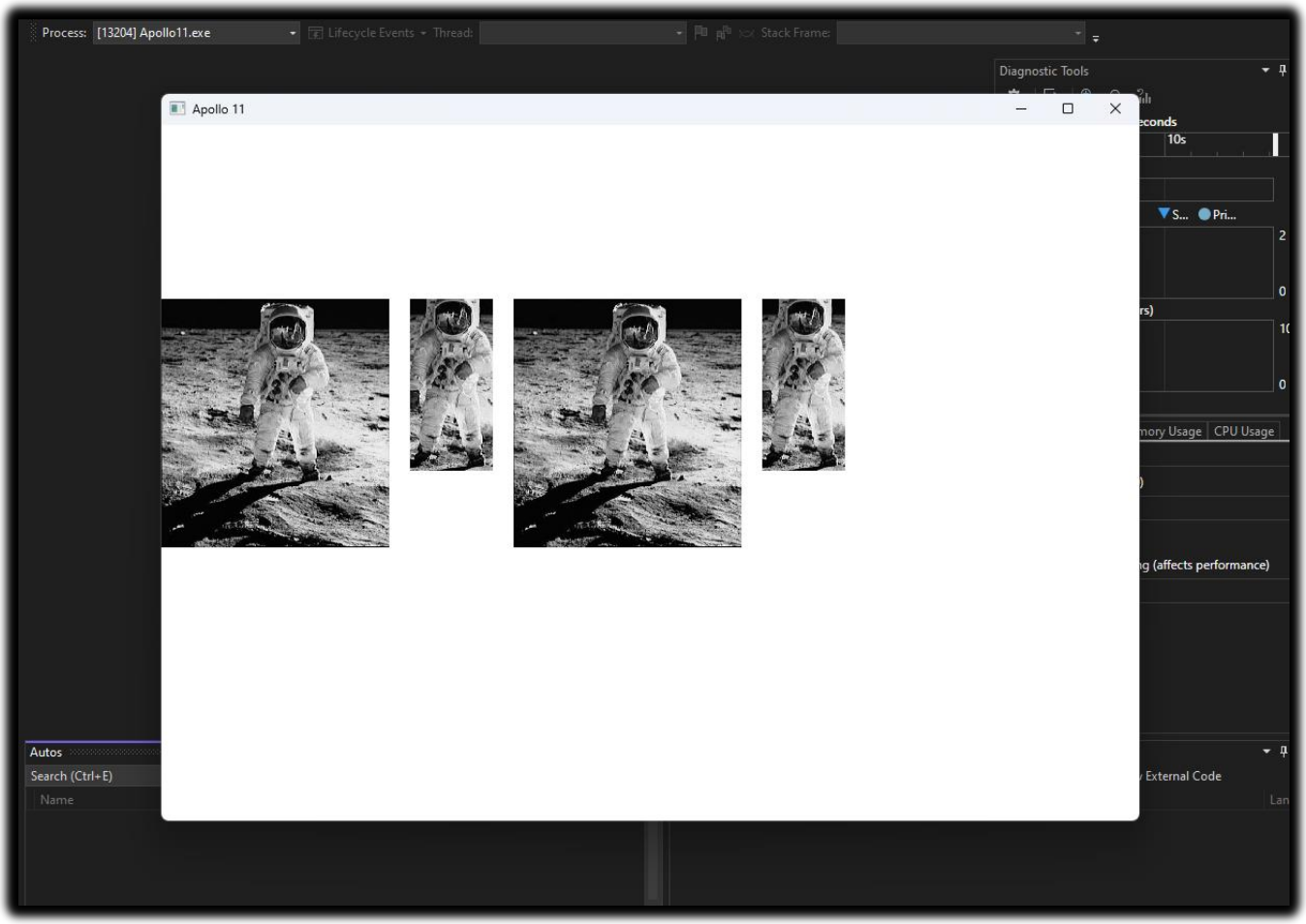
```
// Top-Down DIB Source Rectangle
// Top-left corner of the source rectangle
(xSrc, ySrc + cySrc - 1)
// Top-right corner of the source rectangle
(xSrc + cxSrc - 1, ySrc + cySrc - 1)
// Bottom-left corner of the source rectangle
(xSrc, ySrc)
// Bottom-right corner of the source rectangle
(xSrc + cxSrc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

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

No coordinate flipping is needed because the image data is already stored top-down.

Key idea:
MM_TEXT mode makes source-to-destination mapping less direct, but Windows keeps the image looking correct. Top-down DIBs simplify source coordinates by matching how we naturally think about images (top-left origin).

APOLLO11 Program — What It's Actually Demonstrating 🚀



Before looking at the **APOLLO11** program, it helps to understand **why SetDIBitsToDevice exists and when it's useful**.

This function is not magic — it's a **reliable pipe** for pushing raw DIB pixel data onto a device context **without caring about DIB orientation**.

That's the whole point.

I. Image Rendering (The Main Use)

SetDIBitsToDevice is commonly used to draw images directly to:

- the screen
- a window
- any device context (DC)

It efficiently copies pixel data from a DIB into the DC while correctly handling:

- bottom-up DIBs
- top-down DIBs
- mapping modes

This is exactly why APOLLO11 uses it.

II. Bitmap Copying & Sub-Regions ✂

You can also use SetDIBitsToDevice to:

- draw only **part** of a bitmap
- select a rectangular region using:
  - ✓ xSrc, ySrc, cxSrc, cySrc
- place it at a destination rectangle

This enables:

- cropping
- partial rendering
- overlaying one bitmap region onto another

No manual row flipping needed.

III. Printing 🖨️

When printing images:

- the printer exposes a **device context**
- SetDIBitsToDevice can copy DIB pixels directly to it

By choosing correct coordinates and mapping modes, the image prints exactly as expected. This function is often used alongside other GDI printing calls.

IV. Format Conversion & Processing 🎨

Although not its primary role, the function can be used as part of a pipeline to:

- convert color formats
- change bit depth
- move data between DIBs
- apply processing before display or storage

The key idea:
👉 *you control the DIB, and this function just pushes pixels.*

V. Custom Drawing 🧩

If you manually generate pixel data:

- procedural graphics
- effects
- user-generated content

You can:

1. build a DIB in memory
2. call SetDIBitsToDevice
3. blast it onto the screen or DC efficiently

This avoids slow per-pixel drawing calls.

VI. Why APOLLO11 Matters

The **APOLLO11** program exists to prove one thing:

You can use the **same arguments** to display the **same image region**, regardless of whether the DIB is **top-down or bottom-up**.

```
//This is simplified code, showing only the important parts
#include <windows.h>
#include "dibfile.h"

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

static BITMAPFILEHEADER *pbmfh[2];
static BITMAPINFO *pbmi[2];
static BYTE *pBits[2];
static int cxClient, cyClient, cxDib[2], cyDib[2];

void LoadDIBFiles(HWND hwnd)
{
    pbmfh[0] = DibLoadImage(TEXT("Apollo11.bmp"));
    pbmfh[1] = DibLoadImage(TEXT("ApolloTD.bmp"));

    if (pbmfh[0] == NULL || pbmfh[1] == NULL)
    {
        MessageBox(hwnd, TEXT("Cannot load DIB file"), szAppName, 0);
        PostQuitMessage(0);
    }

    pbmi[0] = (BITMAPINFO *)(pbmfh[0] + 1);
    pbmi[1] = (BITMAPINFO *)(pbmfh[1] + 1);
    pBits[0] = (BYTE *)pbmfh[0] + pbmfh[0]->bfOffBits;
    pBits[1] = (BYTE *)pbmfh[1] + pbmfh[1]->bfOffBits;

    cxDib[0] = pbmi[0]->bmiHeader.biWidth;
    cxDib[1] = pbmi[1]->bmiHeader.biWidth;
    cyDib[0] = abs(pbmi[0]->bmiHeader.biHeight);
    cyDib[1] = abs(pbmi[1]->bmiHeader.biHeight);
}
```

```
void PaintDIBs(HWND hwnd, HDC hdc)
{
    // Bottom-up DIB full size
    SetDIBitsToDevice(hdc,
                      0,            // xDst
                      cyClient / 4, // yDst
                      cxDib[0],     // cxSrc
                      cyDib[0],     // cySrc
                      0,            // xSrc
                      0,            // ySrc
                      0,            // first scan line
                      cyDib[0],     // number of scan lines
                      pBits[0],
                      pbmi[0],
                      DIB_RGB_COLORS);

    // Bottom-up DIB partial
    SetDIBitsToDevice(hdc,
                      240,          // xDst
                      cyClient / 4, // yDst
                      80,           // cxSrc
                      166,          // cySrc
                      80,           // xSrc
                      60,           // ySrc
                      0,            // first scan line
                      cyDib[0],     // number of scan lines
                      pBits[0],
                      pbmi[0],
                      DIB_RGB_COLORS);

```

```
    // Top-down DIB full size
    SetDIBitsToDevice(hdc,
                      340,          // xDst
                      cyClient / 4, // yDst
                      cxDib[0],     // cxSrc
                      cyDib[0],     // cySrc
                      0,            // xSrc
                      0,            // ySrc
                      0,            // first scan line
                      cyDib[0],     // number of scan lines
                      pBits[0],
                      pbmi[0],
                      DIB_RGB_COLORS);

    // Top-down DIB partial
    SetDIBitsToDevice(hdc,
                      580,          // xDst
                      cyClient / 4, // yDst
                      80,           // cxSrc
                      166,          // cySrc
                      80,           // xSrc
                      60,           // ySrc
                      0,            // first scan line
                      cyDib[1],     // number of scan lines
                      pBits[1],
                      pbmi[1],
                      DIB_RGB_COLORS);
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;

    switch (message)
    {
    case WM_CREATE:
        LoadDIBFiles(hwnd);
        return 0;

    case WM_SIZE:
        cxClient = LOWORD(lParam);
        cyClient = HIWORD(lParam);
        return 0;

    case WM_PAINT:
        hdc = BeginPaint(hwnd, &ps);
        PaintDIBs(hwnd, hdc);
        EndPaint(hwnd, &ps);
        return 0;

    case WM_DESTROY:
        if (pbmfh[0])
            free(pbmfh[0]);
        if (pbmfh[1])
            free(pbmfh[1]);

        PostQuitMessage(0);
        return 0;
    }

    return DefWindowProc(hwnd, message, wParam, lParam);
}
```

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:

APOLLO11 — Code Structure Explained (Clean Version)

The APOLLO11 program is built around **three key functions**:

- LoadDIBFiles
- PaintDIBs
- WndProc

Each has a very clear job.

I. LoadDIBFiles — Loading the Bitmaps

Purpose:
Load two DIB files and extract everything needed to draw them later.

What it does:

- Loads:
  - ✓ APOLLO11.BMP (bottom-up DIB)
  - ✓ APOLLOTD.BMP (top-down DIB)
- Uses DibLoadImage (from dibfile.h)
- Stores results in the pbmfh[] array (BITMAPFILEHEADER*)

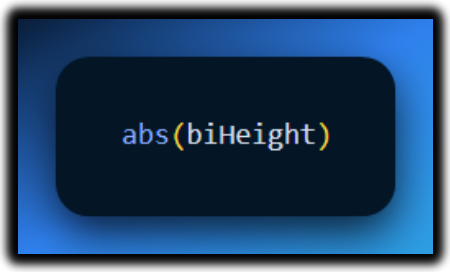
Error handling:

- If loading fails:
  - ✓ shows a message box
  - ✓ calls PostQuitMessage
  - ✓ program exits cleanly

Extracted data:

- BITMAPINFO* for each DIB
- pointer to pixel bits (pBits)
- width (cxDib[])
- height (cyDib[])

⚠ Height is stored using:



This handles both **top-down** and **bottom-up** DIBs correctly.

II. PaintDIBs — Drawing the Images

Purpose:
Render the loaded DIBs onto the window.

How it works:

- Uses SetDIBitsToDevice
- Called **four times**
- Each call:
  - ✓ draws a different region
  - ✓ places it at a different position in the window

Key point (VERY important):

The same xSrc, ySrc, cxSrc, cySrc values are used
for **both** the bottom-up and top-down DIBs.

That's the entire lesson of APOLLO11.

Orientation does **not** change how you select the image region.

III. WndProc — Window Message Handling

WM_CREATE

- Calls LoadDIBFiles
- Loads both DIBs at startup

WM_SIZE

- Updates:
  - ✓ cxClient
  - ✓ cyClient
- Tracks window size changes

WM_PAINT

- Calls BeginPaint
- Calls PaintDIBs
- Calls EndPaint

WM_DESTROY

- Frees allocated DIB memory
- Calls PostQuitMessage
- Clean shutdown

IV. What This Program Proves

- Two DIBs:
  - ✓ same image
  - ✓ same dimensions (220 × 240)
  - ✓ opposite row order
- Same source rectangle arguments
- Same drawing function
- **Same visual result**

👉 SetDIBitsToDevice hides the orientation madness. That's why this function exists.

V. One-Line Mental Rule (Keep This)

With SetDIBitsToDevice, source coordinates describe the image — not the memory layout.

Sequential Display and SetDIBitsToDevice.

This section deals with two major topics:

1. **The API Headache:** Why SetDIBitsToDevice coordinates are confusing.
2. **The Memory Hack:** How to display huge images on low-memory computers ("Sequential Display").

I. The Coordinate Confusion

If you read the official Microsoft documentation for SetDIBitsToDevice, you will get a headache. It talks about "Origins" and "Bottom-Up vs. Top-Down" in a way that contradicts itself.

The Reality:

- **Standard DIBs (Bottom-Up):** The file stores the bottom row first.
- **Top-Down DIBs (Negative Height):** The file stores the top row first.

The Problem: When you tell Windows "Draw this at (0,0)," does that mean the Top-Left of the screen? Or the Bottom-Left? And does it align with the Top-Left of the image or the Bottom-Left? The API is inconsistent.

The Engineering Fix: Ignore the confusion. When writing your own DIB wrapper functions (like the ones in Chapter 16), **force** consistency.

Always write your code assuming (0,0) is the **Top-Left** of the image and the **Top-Left** of the screen. Let your wrapper function handle the ugly math underneath.

II. Sequential Display (The "Streaming" Technique)

Imagine you need to print a 500MB high-resolution poster, but your computer only has 50MB of RAM.

You cannot load the whole file.

The Solution: You "Stream" the pixels to the screen.

1. Read the first 50 lines from the disk.
2. Blast them to the screen immediately.
3. Delete them from memory.
4. Read the next 50 lines.

The Tool: SetDIBitsToDevice This function is built for this exact scenario.

It has two specific arguments:

- uStartScan: "I am starting at row X."
- cScanLines: "I am giving you Y rows of data."

You call this function inside a while loop, painting the image in bands (strips) rather than all at once.

III. The SEQDISP Program

This program demonstrates the "Streaming" technique.

The Weird Observation: When you run SEQDISP, you see the image appear **from Bottom to Top** (like a rising curtain).

Why does this happen? It is not a bug; it is the file format.

1. Standard BMP files are stored **Upside Down** (Bottom-Up).
2. The first bytes in the file represent the **Bottom** row of pixels.
3. SEQDISP reads the file sequentially (start to finish).
4. Therefore, it reads (and draws) the bottom of the picture first.

If you used a "Top-Down" DIB (negative height), the image would paint from Top to Bottom.

IV. StretchDIBits: The Upgrade

SetDIBitsToDevice is fast, but it is dumb. It only does 1:1 pixel mapping. If you want to Zoom, Shrink, or Flip, you need StretchDIBits.

Key Differences:

1. **Scaling:** It asks for a *Destination Size* (cxDst, cyDst) and a *Source Size* (cxSrc, cySrc). If they are different, Windows stretches/shrinks the image.
2. **Flipping:** If you pass a **negative width** or **height**, Windows mirrors the image instantly.
3. **Raster Ops (dwROP):** You can use logic like SRCINVERT to combine the image with the background (e.g., for transparency effects), which SetDIBitsToDevice cannot do.

The Trade-off: It is very hard to do "Sequential Display" (Streaming) with StretchDIBits. To stretch an image accurately, the computer usually needs all the surrounding pixels to blend them (interpolation).

V. Explained Like ur a Teenager: Streaming

Standard Loading: Like downloading a movie completely before watching it. It takes a lot of storage, and you wait a long time.

Sequential Display: Like Netflix. You buffer 10 seconds, watch it, and dump it. You don't need to store the whole movie on your phone at once.

The Glitch: Because standard BMPs are "Upside Down," it's like watching the movie starting from the end credits and rewinding to the beginning!

VI. Quick Review

Question 1: Why does the SEQDISP program draw the image from the bottom of the screen upwards?*(Answer: Because standard DIB files are stored "Bottom-Up," so the program reads the bottom rows from the disk first.)*

Question 2: Which function should you use if you need to display a 100x100 pixel bitmap into a 50x50 pixel box?*(Answer: StretchDIBits, because it supports scaling/resizing.)*

Question 3: What are the arguments uStartScan and cScanLines used for?*(Answer: They allow you to update just a specific "band" or slice of the image, enabling memory-efficient sequential display.)*

Mapping Modes, StretchDIBits & SHOWDIB2 (Clean Notes)

I. Mapping Modes & Flipping (Core Idea)

StretchDIBits **respects the mapping mode** of the destination device context.

That means:

- If the mapping mode flips axes → **your image flips**
- Windows does **not protect you** from this

You either:

- compensate
- or use it intentionally

II. Stretching Example (Simple)

```
// Stretch a 2×2 DIB into 4×4
StretchDIBits(
    hdc,
    0, 0, 4, 4,      // destination (4×4)
    0, 0, 2, 2,      // source (2×2)
    pBits,
    pInfo,
    DIB_RGB_COLORS,
    SRCCOPY
);
```

What happens:

- Source pixels are scaled
- Orientation depends on **mapping mode + signs of width/height**

III. Flipping Using Mapping Modes

Mapping mode direction flags:

- xMM = 1 → x increases to the right
- xMM = -1 → x increases to the left
- yMM = 1 → y increases downward
- yMM = -1 → y increases upward

These affect how the bitmap appears.

When does flipping happen?

- **Vertical flip (mirror image):**
    - ✓ Happens if x direction is reversed
- **Horizontal flip (upside-down):**
    - ✓ Happens if y direction is reversed

Windows decides this internally based on:

- mapping mode
- source size
- destination size

⚠ The Sign() logic mentioned in the book is **conceptual**, not real C code.

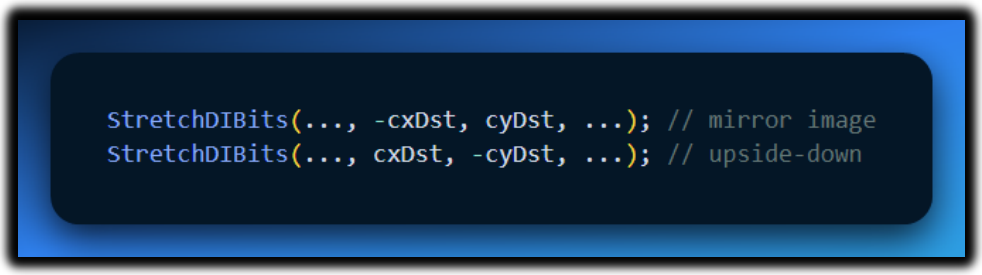
IV. Intentional Flipping (The Clean Trick)

You don't need mapping modes to flip images.

Just do this:

- Negative width → **horizontal flip**
- Negative height → **vertical flip**

Example idea:



Simple. Reliable. Common.

SHOWDIB2 Program — What Actually Matters

Forget the menu noise. Focus on DIB logic.

ShowDib Program: The "Engine Room"

This program is basically a Swiss Army Knife for DIBs.

FEATURE	WINAPI FUNCTION	WHAT IT ACTUALLY DOES
Normal Display	`SetDIBitsToDevice`	Blits pixels 1:1. No scaling, just fast copying.
Stretching	`StretchDIBits`	Resizes the DIB to fit your window.
Isotropic	`SetMapMode(MM_ISOTROPIC)`	Forces the DIB to keep its aspect ratio (no "fat" or "thin" distortion).
Clipboard	`SetClipboardData(CF_DIB, hMem)`	Puts the "raw" DIB data into Windows memory so Word or Paint can "Paste" it.

I. DIB Loading (Nothing Special)

- Uses dibfile.h
- Opens a DIB file via dialog
- Loads it using DibLoadImage
- Extracts:
    - ✓ BITMAPINFO
    - ✓ pBits
    - ✓ width & height

That's it.

II. Displaying the DIB (Important Part)

Everything goes through **ShowDib**.

Based on menu choice, it uses:

- SetDIBitsToDevice
  - ✓ normal
  - ✓ centered
- StretchDIBits
  - ✓ stretched
  - ✓ isotropic stretch (keep aspect ratio)

👉 **This is the real lesson**:

- SetDIBitsToDevice → no scaling
- StretchDIBits → scaling + mapping mode effects

III. Printing

- Gets printer DC
- Calls ShowDib
- Same drawing logic
- Different device

Nothing magical.

IV. Clipboard (Minimal Understanding)

- Copies packed DIB into global memory
- Uses clipboard format CF_DIB
- Paste support is **not implemented**

This is common in WinAPI samples.

V. Known Problems (Author Admits Them)

- ❌ 256-color mode issues (palette not handled yet)
- ❌ Slow rendering (especially Windows NT)
- ❌ No scrollbars for large DIBs

All of these are postponed to later chapters.

VI. Final Mental Model (KEEP THIS)

- SetDIBitsToDevice → **orientation-safe**
- StretchDIBits → **mapping-mode-sensitive**
- Negative width/height → **manual flipping**
- Palettes matter in 256-color modes

That's it.
Everything else is UI noise.

COLOR CONVERSION, PALETTES & PERFORMANCE 🎨

I. Why Color Conversion Exists (Core Truth)

A **DIB is device-independent**.
Your **video card is not**.

So, when you call:

- SetDIBitsToDevice
- StretchDIBits

Windows must:
➡ **convert every single pixel**
from the DIB format → video memory format.

If the bitmap is big:

- that's **millions of conversions**
- performance can die fast

II. Easy (Fast) Conversions

These are mostly cheap bit operations:

24-bit DIB → 24-bit display

- almost free (maybe byte order swap)

16-bit DIB → 24-bit display

- shift bits, pad with zeros

24-bit DIB → 16-bit display

- drop lower bits

4-bit / 8-bit DIB → 24-bit display

- palette lookup per pixel

These are **not scary**.

III. Expensive (Slow) Conversions 🚨

The real killer:

High-color DIB on low-color display

Examples:

- 24-bit DIB → 8-bit display
- 16-bit DIB → 4-bit display

What Windows must do:

- For **each pixel**
- Find the **closest available color**
- Using RGB distance math

That's:

- loops
- comparisons
- tons of CPU work

This is why it feels slow.

IV. Nearest-Color Search

Think of colors as points in a 3D RGB box.

Windows asks:

"Which of my limited colors is closest to this pixel?"

- Square root not required
- Still expensive
- Still happens **a lot**

V. Special Case: 8-bit DIB → 8-bit Display

Good news:

- Windows only does nearest-color search **once per unique color**
- Not per pixel

Much faster than high-bit → low-bit cases.

VI. The Big Performance Rule ⚠️

❌ Do NOT display high-bit DIBs on 4/8-bit displays using SetDIBitsToDevice.

Instead:

✅ Convert **once**, display fast forever.

Options:

- Convert to **8-bit DIB**
- Convert to **DDB**
- Use **BitBlt / StretchBlt**

VII. DIB vs DDB (Why DDB Is Faster)

DIB

- Converted every draw
- Flexible
- Slower

DDB

- Already in device format
- Blits fast
- Much better for repeated drawing

👉 For real apps: **convert once, blit many times**

VIII. 8-bit Palette Limitation (Hidden Trap)

In 8-bit video mode, windows only gives you **~20 system colors.** Everything else fights for palette slots.

Without palette management:

- colors look wrong
- images look ugly

Palette handling comes later (rightfully postponed).

IX. Windows NT Is Slower (Why)

Windows NT uses: **client/server graphics model**

Result:

- more data copied
- more overhead
- slower DIB drawing than Win98

Fixes:

- Convert to **DDB**
- Use **CreateDIBSection** (later chapter)

X. Final Mental Model (Burn This In)

- DIB → display = **conversion cost**
- High-bit → low-bit = **very expensive**
- Convert once, draw many times
- DDBs are fast
- Palettes matter in 8-bit modes
- NT needs extra care

That's all you need.

DIB TO DDB CONVERSION

I. The Bottleneck: High-Color Images on Low-Color Screens

Displaying a high-quality image (24-bit) on a low-quality screen (8-bit) is computationally expensive.

The Easy Case: If you display a 24-bit image on a 24-bit screen, Windows just copies the bytes (maybe swapping Red and Blue). It is fast.

The Hard Case: If you display a 24-bit image on an 8-bit screen (256 colors), Windows cannot just copy pixels. It must perform a **Nearest-Color Search** for *every single pixel*.

The Math: It calculates the distance between the pixel's color and every single color in the device's palette using the 3D distance formula:

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

The Problem: Doing this calculation millions of times (for every pixel) freezes the CPU.

The Solution: Do not use SetDIBitsToDevice every time you repaint the screen. Instead, convert the DIB to a **DDB (Device Dependent Bitmap)** *once*. This caches the color conversion. Afterward, you can draw the DDB instantly using BitBlt.

II. DIBs vs. DDBs: The Trade-off

Why do we have two types of bitmaps?

	DIB (DEVICE INDEPENDENT)	DDB (DEVICE DEPENDENT)
Storage	Good Perfect for saving to disk (.bmp). Works on any computer.	Bad Only works on your specific video card settings.
Pixel Access	Easy You have a pointer to the raw bytes. You can read/write pixels easily.	Hard Pixels are locked by the driver/GPU.
Drawing Speed	Slow Windows must translate colors every time you draw.	Fast It is already in the video card's native format.
GDI Support	Limited You cannot use `BitBlt` or draw text on a DIB easily.	Full You can select it into a DC and draw on it.

The Workflow:

1. Load the **DIB** from disk.
2. Convert it to a **DDB** (handles the slow color conversion).
3. Throw away the DIB.
4. Use the DDB for painting the window (Fast).

III. How to Convert (CreateDIBitmap)

The function CreateDIBitmap is poorly named. It does **not** create a DIB. It **reads** a DIB and **creates** a DDB.

```
HBITMAP CreateDIBitmap(
    HDC hdc,                        // Handle to the device context
    CONST BITMAPINFOHEADER *pbmih,  // Pointer to DIB header
    DWORD fdwInit,                  // Initialization flag (usually CBM_INIT)
    CONST VOID *pbInit,             // Pointer to raw DIB pixel bits
    CONST BITMAPINFO *pbmi,         // Pointer to DIB info (header + colors)
    UINT fuUsage                    // Color type (DIB_RGB_COLORS)
);
```

Approach A: The "Lazy" Monochrome DDB

If you only pass the header and skip the initialization (pixels), Windows creates a basic, empty monochrome bitmap.

```
// Creates an uninitialized monochrome bitmap (rarely useful)
hBitmap = CreateDIBitmap(NULL, &info_header, 0, NULL, NULL, 0);
```

Approach B: The "Full" Conversion (Recommended)

This tells Windows: "Here is the raw DIB data (pBits) and the color table (pInfo). Please convert it to match the current screen (hdc) and give me a handle to the new DDB."

```
// 1. Get the HDC so Windows knows what "Device" we are matching
HDC hdc = GetDC(hwnd);

// 2. Create the DDB
// CBM_INIT tells Windows: "Use the provided bits to paint the new bitmap immediately."
HBITMAP hBitmap = CreateDIBitmap(hdc,
                                 &pInfo->bmiHeader,
                                 CBM_INIT,
                                 pBits,
                                 pInfo,
                                 DIB_RGB_COLORS);

// 3. Cleanup
ReleaseDC(hwnd, hdc);
```

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 }
```

What This Code Is Doing - It **takes a DIB from a BMP file**, converts it into a **DDB**, then draws it fast using BitBlt. That's it.

I. Header Type Check (Why It Exists)

The code first checks:

- BITMAPCOREHEADER (old format)
- or BITMAPINFOHEADER (modern format)

Why?

- Because Windows supports **both**
- And bitmap creation depends on which one you have

Based on this, it uses:

- CreateCompatibleBitmap **or**
- CreateBitmap

This is just **format compatibility**, nothing fancy.

II. CBM_INIT Flag (Important Detail)

If CBM_INIT is used:

- Windows **initializes the bitmap immediately**
- Pixel data is copied during creation

How it works internally:

1. Create a memory DC
2. Select bitmap into it
3. Call SetDIBitsToDevice once
4. Bitmap is now fully initialized

After that?
➞ You **stop using DIB drawing**
➞ You just BitBlt

III. Performance Reality Check ⚠

If you draw the image only once:

- SetDIBitsToDevice
- or CreateDIBitmap + BitBlt

➞ **Almost same speed**
➞ Both do color conversion

No magic win here.

If you draw the image many times (WM_PAINT):

✅ Convert **once** using CreateDIBitmap
✅ Draw many times using BitBlt

Now:

- Conversion cost = **paid once**
- Repaints = **fast**

This is where DDBs shine.

IV. What the DIBCONV Program Demonstrates

The program exists to show **one thing**:

How to convert a DIB file into a DDB and display it efficiently.

Nothing more.

V. DIBCONV Flow (Straight Line)

Program starts - Initializes Open File dialog

User clicks File → Open

- File dialog appears
- User selects a .bmp

File is loaded

Reads file into memory

Checks:

- BM signature
- file size
- basic validity

DIB → DDB conversion

CreateBitmapObjectFromDibFile is called

Inside it:

- file already loaded
- CreateDIBitmap creates a **DDB**
- pixel data copied during creation

Painting

- In WM_PAINT
- Uses BitBlt
- Fast rendering

Errors

- If file is invalid → message box
- If conversion fails → message box

VI. Resources (Nothing Special)

- .RC file defines **File → Open**
- RESOURCE.H defines IDM_FILE_OPEN

Standard WinAPI boilerplate.

VII. Important Limitation (Why Only BMP Works)

“.bmp file loads okay. Other files don’t load.”

That’s expected.

Why?

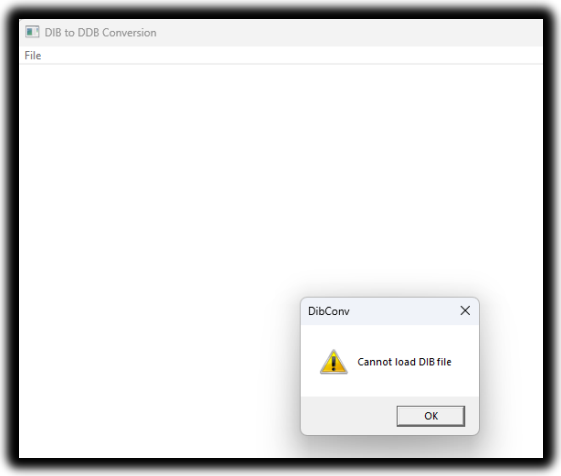
- Code assumes **BMP/DIB format**
- No JPEG, PNG, GIF parsing
- No codecs
- No GDI+

This program is about **DIB mechanics**, not image formats.

VIII. Final Mental Model

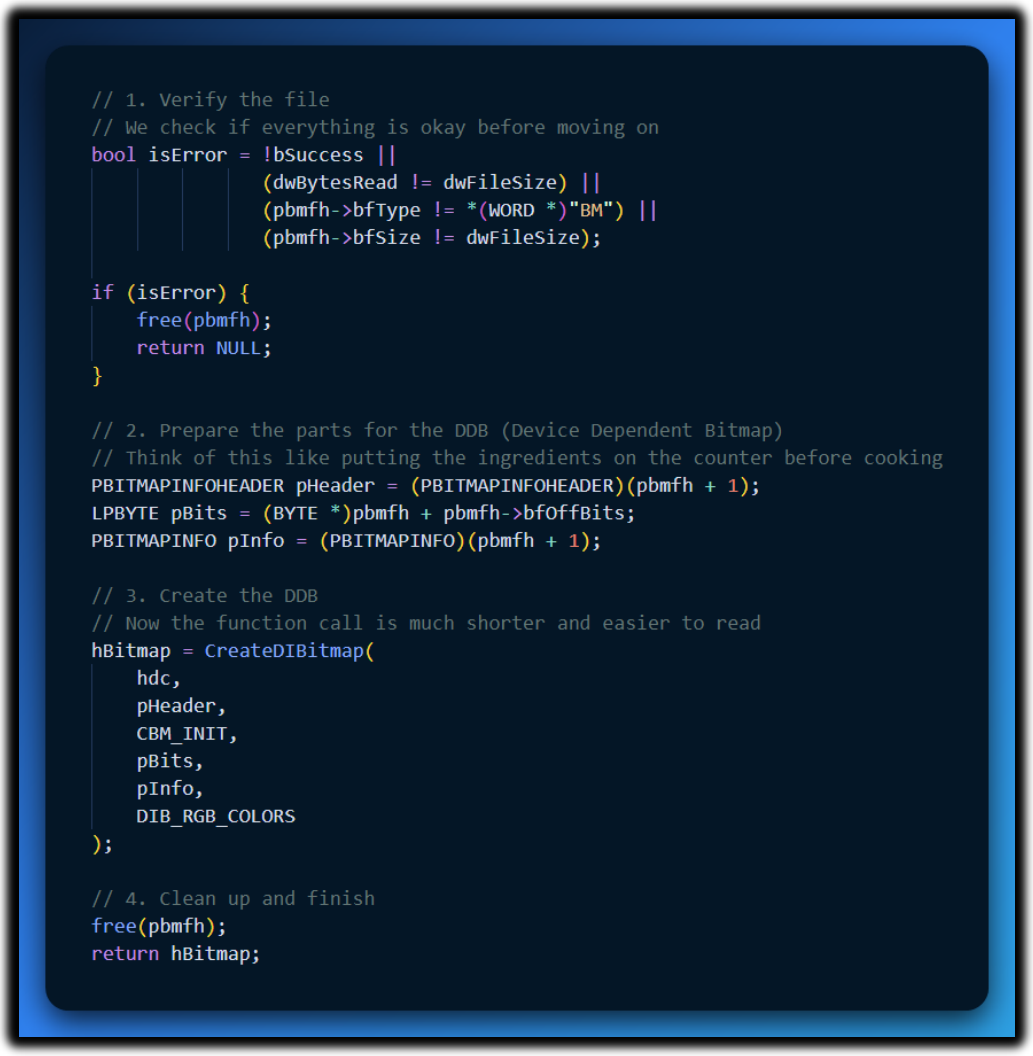
- DIB = flexible, slow to draw repeatedly
- DDB = fast, device-specific
- Convert once → draw many times
- CreateDIBitmap is for performance
- BMP only, by design

That’s the lesson.
Everything else is noise.



It only loads BMP files because that's how the CreateBitmapObjectFromDibFile function was designed and implemented.

If you look at the relevant part of the code, you'll see that it specifically handles BMP data and doesn't include support for any other file formats.



```
// 1. Verify the file
// We check if everything is okay before moving on
bool isError = !bSuccess ||
               (dwBytesRead != dwFileSize) ||
               (pbmfh->bfType != *(WORD *)"BM") ||
               (pbmfh->bfSize != dwFileSize);

if (isError) {
    free(pbmfh);
    return NULL;
}

// 2. Prepare the parts for the DDB (Device Dependent Bitmap)
// Think of this like putting the ingredients on the counter before cooking
PBITMAPINFOHEADER pHeader = (PBITMAPINFOHEADER)(pbmfh + 1);
LPBYTE pBits = (BYTE *)pbmfh + pbmfh->bfOffBits;
PBITMAPINFO pInfo = (PBITMAPINFO)(pbmfh + 1);

// 3. Create the DDB
// Now the function call is much shorter and easier to read
hBitmap = CreateDIBitmap(
    hdc,
    pHeader,
    CBM_INIT,
    pBits,
    pInfo,
    DIB_RGB_COLORS
);

// 4. Clean up and finish
free(pbmfh);
return hBitmap;
```

The code only checks for BMP files by looking for the "BM" signature in the file header. If that signature isn't found, the function just returns NULL and stops. In other words, anything that isn't a BMP is ignored.

This works because the code assumes the file follows the BMP layout. Other formats like JPEG or PNG are structured differently, so they won't load unless extra handling is added for them.

DIBCONV.C is a standalone program that runs without any extra files. When the user selects **File Open**, WndProc calls CreateBitmapObjectFromDibFile, which loads the entire file into memory and passes it to CreateDIBitmap. Once the bitmap handle is created, the original memory buffer can be freed.

When painting the window, WndProc selects the bitmap into a memory device context and displays it using BitBlt. The bitmap's size is obtained with GetObject.

The bitmap's pixel data doesn't have to be fully set when CreateDIBitmap is called. That can be done later with SetDIBits, which fills in the pixel data when needed.

```
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
);
```

During this process, the function uses the color table from the BITMAPINFO structure to convert the pixel data into a device-dependent format.

A device context is only needed if the last parameter is set to DIB_PAL_COLORS.

This makes it possible to set up the pixel data later and control how colors are converted, depending on what the application needs.

```
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
);
```

SetDIBits copies pixel data from a DIB into an existing DDB so the bitmap can be used or displayed.

GetDIBits pulls pixel data from a DDB and converts it into a DIB.
It's mainly useful for things like screen captures. The conversion isn't perfect, so you shouldn't expect to get the original bitmap back exactly.

```
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)
);
```

```
#include <Windows.h>
void ConvertDDBtoDIB(HBITMAP hBitmap)
{
    HDC hdc = GetDC(NULL); // Get a handle to the screen device context
    HDC hdcMem = CreateCompatibleDC(hdc); // Create a compatible device context
    // Get the bitmap information
    BITMAP bmp;
    GetObject(hBitmap, sizeof(BITMAP), &bmp);
    // Create a DIB section to store the pixel data
    BITMAPINFO bmi = { 0 };
    bmi.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
    bmi.bmiHeader.biWidth = bmp.bmWidth;
    bmi.bmiHeader.biHeight = bmp.bmHeight;
    bmi.bmiHeader.biPlanes = 1;
    bmi.bmiHeader.biBitCount = bmp.bmBitsPixel;
    bmi.bmiHeader.biCompression = BI_RGB;
    // Retrieve the pixel data from the DDB
    BYTE* pPixels = nullptr;
    HBITMAP hDibBitmap = CreateDIBSection(hdcMem, &bmi, DIB_RGB_COLORS, (void**)&pPixels, NULL, 0);
    SelectObject(hdcMem, hDibBitmap);
    GetDIBits(hdcMem, hBitmap, 0, bmp.bmHeight, pPixels, &bmi, DIB_RGB_COLORS);
    // Release resources
    DeleteDC(hdcMem);
    ReleaseDC(NULL, hdc);
    // Now you have a DIB stored in pPixels, which you can use as needed
    // Clean up the DIB
    DeleteObject(hDibBitmap);
}

int main()
{
    // Assuming you already have an HBITMAP handle named hBitmap
    ConvertDDBtoDIB(hBitmap);
    return 0;
}
```

When You Don't Want to Use GetDIBits

GetDIBits exists to **pull pixel data back out of a bitmap**.
But most of the time… you **shouldn't need it**.

Better options (simpler + faster):

Keep the pixels yourself

- When you load or create the bitmap, **store the raw DIB data**.
- Then you already have the pixels — no extraction needed.

Read from the original source

- If the bitmap came from a file, memory buffer, or stream:
- Just re-read that source instead of asking GDI to convert it back.

Save the bitmap yourself

- Serialize the DIB (header + pixels) to disk or memory.
- Later, reload it exactly as-is.
- No guessing, no conversion.
- *(And yes — locally, not on some server 🤣)*

👉 Core idea: **If you own the pixels, don't ask Windows to give them back to you.**

Clipboard → DIB (Automatic Conversion) 📋

When you copy a bitmap to the clipboard, windows **automatically converts it to a DIB, y**ou don't have to do anything special. You can retrieve it as:

- CF_DIB (classic)
- CF_DIBV5 (newer, richer header)

So:

- Copy bitmap
- Open clipboard
- Ask for CF_DIB
- You get a packed DIB (header + pixels)

Windows already did the work.

When You Don’t Have the Original DIB ❓

Sometimes you **never had the pixels** to begin with:

Examples:

- Screenshot / screen capture
- Bitmap received over the network
- Bitmap created by the OS
- Bitmap created or modified by a third-party library

In these cases:

- You only have an HBITMAP
- You don’t know the original format
- You don’t know the pixel layout

👉 **This is when GetDIBits actually makes sense**

It’s the *last-resort extractor*.

Directly Accessing Bitmap Data (Simple Explanation)

If you want to work with bitmap pixels **directly** (read or write them yourself), the easiest way in Windows is to create a **DIB (Device-Independent Bitmap)** that gives you a pointer to its memory.

Instead of creating a bitmap first and then asking Windows for the pixels, you can create it **and get the pixel buffer at the same time**.

The most common way to do this is with **CreateDIBSection**.

I. Example: Create a Bitmap and Access Its Pixels

This example creates a 32-bit bitmap and gives you direct access to its pixel data.

```
BITMAPINFO bmi = {};
bmi.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);
bmi.bmiHeader.biWidth = 200;
bmi.bmiHeader.biHeight = -200; // top-down bitmap
bmi.bmiHeader.biPlanes = 1;
bmi.bmiHeader.biBitCount = 32;
bmi.bmiHeader.biCompression = BI_RGB;

void* pixels = nullptr;

HBITMAP hBitmap = CreateDIBSection(
    NULL,
    &bmi,
    DIB_RGB_COLORS,
    &pixels,
    NULL,
    0
);

// Now you can directly write to the bitmap memory
if (pixels)
{
    uint32_t* p = (uint32_t*)pixels;
    p[0] = 0x00FF0000; // first pixel = red (BGRA)
}
```

What's important here:

- pixels points directly to the bitmap's memory
- You can read and write pixels like a normal array
- No GetDIBits call is needed

II. Alternative: GetDIBits / SetDIBits (Less Direct)

Another way is:

1. Create a bitmap
2. Call GetDIBits to copy pixel data into your own buffer
3. Modify the buffer
4. Call SetDIBits to copy it back

This works, but it's slower and more code. CreateDIBSection is usually better.

III. Clipboard Bitmap Formats (Besides CF_DIB and CF_DIBV5)

Yes, there are other clipboard formats related to bitmaps:

CF_BITMAP

- Contains an HBITMAP
- You *don't* get raw pixel bytes directly
- Older and less flexible

CF_DIB

- Bitmap data without alpha (older format)

CF_DIBV5

- Newer DIB format
- Supports alpha channel and color profiles
- Best choice for modern apps

Registered formats (e.g. PNG)

- Apps can register formats like "PNG" using RegisterClipboardFormat
- Data is serialized (compressed image bytes)

Direct Access vs Copying Bitmap Data

I. Direct Access (Easy & Fast)

- Use **CreateDIBSection**.
- You get a **pointer to the bitmap memory** right away.
- You can **read and write pixels like a normal array**.
- No need for GetDIBits or SetDIBits.

II. Copying Pixels (Slower & More Work)

Another way is using GetDIBits and SetDIBits:

- Create a bitmap.
- Call GetDIBits to copy pixel data into your buffer.
- Modify your buffer.
- Call SetDIBits to copy it back into the bitmap.

Works, but slower and more code.

CreateDIBSection is usually better for modern apps.

III. Clipboard Bitmap Formats

Besides CF_DIB and CF_DIBV5, here are other clipboard formats you might see:

FORMAT	WHAT IT CONTAINS	NOTES
`CF_BITMAP`	Just an HBITMAP handle	*No direct pixel data; old style*
`CF_DIB`	Device-independent bitmap, no alpha	*Old but works*
`CF_DIBV5`	Newer DIB format with alpha & color profiles	*Best choice for modern apps*
`"PNG"`	Registered custom formats	*Usually compressed image bytes*

How Bitmap Serialization & Deserialization Works

Think of it like **packing a suitcase**.

I. Serialization (Saving or Sending)

- Collect everything needed to rebuild the bitmap later:
    1. Bitmap header (size, width, height, format)
    2. Color table (if needed)
    3. Raw pixel data
- Pack it into a continuous block of bytes.
- You can write this to a file, send it over a network, or put it on the clipboard.

II. Deserialization (Loading or Using)

- Open the "suitcase" (file or buffer).
- Read the header to know the bitmap format.
- Allocate memory for the pixels.
- Copy the pixel data into memory.
- Create a bitmap object from it.

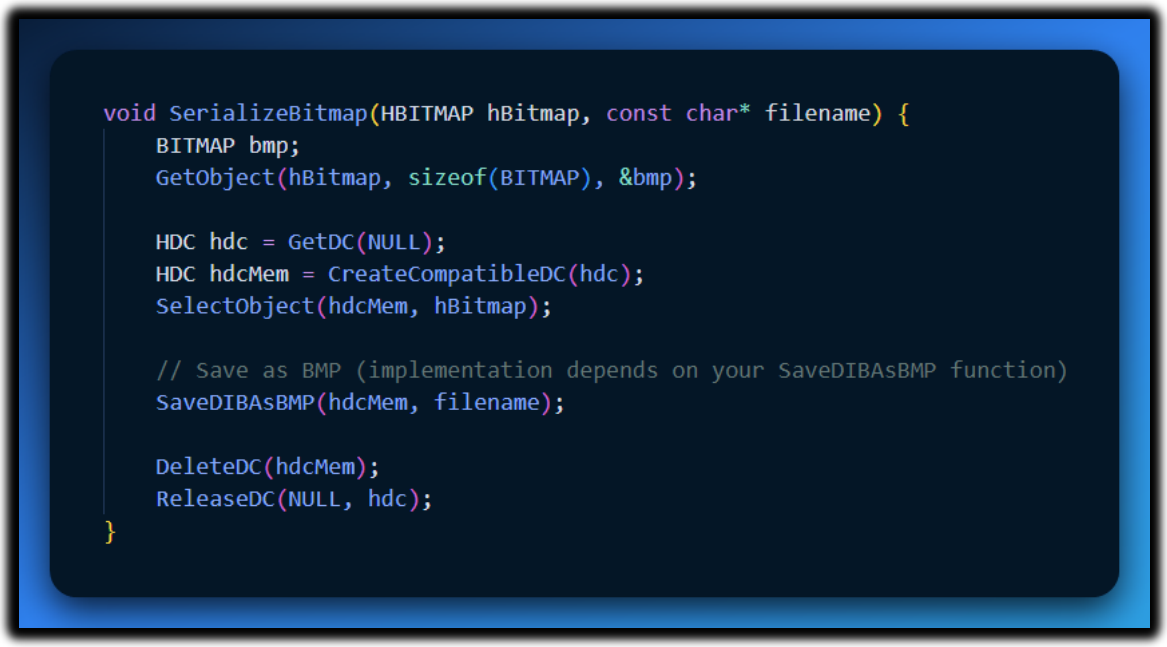
For clipboard formats like CF_DIB, Windows expects this **exact memory layout**.

Simple Code Examples

Deserialize a Bitmap (Load from File)

```
HBITMAP DeserializeBitmap(const char* filename) {
    // Load BMP file into an HBITMAP
    return LoadImage(NULL, filename, IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE);
}
```

Serialize a Bitmap (Save to 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);

    // Save as BMP (implementation depends on your SaveDIBAsBMP function)
    SaveDIBAsBMP(hdcMem, filename);

    DeleteDC(hdcMem);
    ReleaseDC(NULL, hdc);
}
```

Quick Recap

- **Direct access:** CreateDIBSection → fast, simple, pointer to memory.
- **Copying pixels:** GetDIBits / SetDIBits → slower, more work.
- **Clipboard formats:** CF_BITMAP (old), CF_DIB (no alpha), CF_DIBV5 (modern, alpha + color profiles), or custom formats like PNG.
- **Serialization:** turn bitmap into bytes (header + pixels + color info).
- **Deserialization:** rebuild bitmap from those bytes.

💡 **Analogy:** Serialization = packing a suitcase. Deserialization = unpacking it and putting your stuff back where it belongs.

I. The "Best of Both Worlds" Solution

Before CreateDIBSection (introduced in Windows NT/95), you had to choose:

- **Use a DIB:** You can edit pixels directly (fast math), but drawing it to the screen is slow.
- **Use a DDB:** Drawing is fast, but you cannot touch the pixels.

The DIB Section is a hybrid.

- **To Windows:** It looks like a **DDB**. You get an HBITMAP handle that works with fast functions like BitBlt.
- **To You:** It looks like a **DIB**. Windows gives you a pointer to the raw RAM (ppBits) so you can read/write pixels instantly without asking the OS for permission.

I. How it Works (CreateDIBSection)

This function does two things at once:

Allocates RAM for the image pixels.

Returns two keys to that memory:

- ✓ HBITMAP (The Handle): For passing to GDI functions.
- ✓ void* (The Pointer): For your C++ code to modify pixels directly.

The Syntax:

```
HBITMAP CreateDIBSection(
    HDC hdc,                // Device Context (optional)
    CONST BITMAPINFO *pbmi, // DIB Header & Colors
    UINT iUsage,            // Color Type (DIB_RGB_COLORS)
    VOID **ppvBits,         // **OUTPUT**: Windows puts the pixel pointer here
    HANDLE hSection,        // File Mapping (Advanced: for sharing memory between apps)
    DWORD dwOffset          // Offset (Advanced)
);
```

II. Usage: The "Magic" Workflow

Here is how modern GDI code works using DIB Sections. It eliminates the need to constantly convert between DIB and DDB.

Initialize the Header: Set up a BITMAPINFOHEADER (Width, Height, 32-bit).

Create the Section:

```
BYTE* pPixels = NULL; // This will hold our raw access
HBITMAP hBitmap = CreateDIBSection(NULL, &bmi, DIB_RGB_COLORS, (void**)&pPixels, NULL, 0);
```

Draw Manually (The "DIB" Way): You can now write a loop to change pixels directly.

```
// Make the first pixel Red
pPixels[0] = 0;   // Blue
pPixels[1] = 0;   // Green
pPixels[2] = 255; // Red
```

Draw to Screen (The "DDB" Way): Because you have an HBITMAP, you can give it to GDI.

```
SelectObject(memDC, hBitmap);
BitBlt(screenDC, 0, 0, w, h, memDC, 0, 0, SRCCOPY);
```

The result: You modified the memory directly, and BitBlt saw the changes instantly because they share the same memory. No copying required!

III. Comparison: CreateDIBitmap vs. CreateDIBSection

The names are confusingly similar. Here is the difference:

FUNCTION	OUTPUT	PIXEL ACCESS	SPEED
`CreateDIBitmap`	**DDB** (Device Dependent)	None Pixels are locked in driver/GPU memory.	Mixed Slow creation, but Fast drawing (Blitting).
`CreateDIBSection`	**DIB Section** (Hybrid)	Direct Pixels are in System RAM, you get a pointer.	Fast Fast creation, Fast modification, Good drawing speed.

IV. Explain Like I'm a Teenager: The Shared Notebook

- **DIB:** A notebook you keep at home. You can write in it, but the teacher (Windows) can't see it until you photocopy it and mail it.
- **DDB:** A notebook locked in the teacher's desk. The teacher can read it instantly, but you aren't allowed to touch it.
- **DIB Section:** A Google Doc. You can type in it (modify pixels) *and* the teacher can project it on the board (draw to screen) at the same time.

V. Quick Review

Question 1: What is the most valuable thing CreateDIBSection gives you that CreateDIBitmap does not? *(Answer: A pointer (ppvBits) that allows direct read/write access to the pixel memory.)*

Question 2: If you want to perform fast image processing (like adding a blur filter) and then display the result, which type should you use? *(Answer: A DIB Section. You can process the pixels using the pointer, then display it immediately using the handle.)*

Question 3: What are the last two parameters (hSection, dwOffset) used for? *(Answer: They allow two different programs to share the exact same image memory (File Mapping), but in 99% of cases, you just set them to NULL and 0.)*

VI. CreateDIBSection Cheat Sheet

The Weird Argument (ppBits):

- **Why is it **ppBits?** Because the function needs to *give you* a memory address. In C, if a function changes a variable's value (the address), you must pass the address of that variable.
- **Translation:** You declare BYTE *pBits; and pass &pBits.

The Useless Argument (hdc):

- If you are using standard RGB colors (which is 99% of the time), the function ignores the hdc (Device Context). It's just there for legacy Palette support.

The "Advanced" Arguments (hSection, dwOffset):

- Just set them to NULL and 0.
- They are only used if you want to perform "File Mapping" (sharing the image memory between two separate running programs).

Conclusion: It returns a **Handle** (for Windows to use) and fills your **Pointer** (for you to use). The pixels start out as "garbage" (uninitialized), so you must fill them with color data yourself.

Understanding DIBSECT.C – Simplified

1. Loading a DIB File

- CreateDibSectionFromDibFile opens a BMP/DIB file using CreateFile.
- Checks the file is valid by confirming the **"BM"** signature in the header.
- Reads bitmap info into a BITMAPINFO structure.

2. Creating the DIB Section

- CreateDIBSection allocates memory for the bitmap pixels and returns a handle.
- Pixels can now be accessed directly via the pointer returned by CreateDIBSection.
- RGB colors are used for simplicity (DIB_RGB_COLORS).

3. Reading Bitmap Bits

- Pixel data from the file is copied into the memory allocated for the DIB Section.
- Size = total file size minus the offset to pixel data (bfOffBits).

4. Cleanup

- Frees the BITMAPINFO memory.
- Closes the file handle.
- Returns the handle to the DIB Section.

Window Procedure (WndProc) – Key Points

Opening a File

- Handles WM_COMMAND → IDM_FILE_OPEN.
- Shows File Open dialog (GetOpenFileName).
- Deletes existing bitmap if needed.
- Creates a new DIB Section from the selected file.

Painting the Window

- Handles WM_PAINT.
- Creates a compatible device context (hdcMem).
- Uses BitBlt to draw the DIB Section onto the window.

Cleanup

- Handles WM_DESTROY.
- Deletes the DIB Section handle to prevent memory leaks.

Extending the Program – Ideas

Support More Formats

- Add PNG, JPEG, or GIF support via Windows codecs.
- Update the file dialog filter to include these formats.

Image Manipulation

- Access pixels directly via the DIB Section pointer.
- Apply filters, adjust colors, rotate images, etc.

Integrate Other Features

- Save images, handle user input, or send images over a network.
- Build simple image editing, animation, or sharing tools.

Customize the UI

- Add buttons, sliders, or text boxes for controlling image display or effects.

Learn and Experiment

- Explore Windows graphics APIs and libraries.
- Try tutorials and MSDN examples to deepen understanding.

Summary

- The program opens a BMP/DIB file.
- Creates a **DIB Section** to access pixels directly.
- Displays the image in the window.
- Properly cleans up memory.
- Can be extended for other formats, image editing, or UI enhancements.

Comparing DIBCONV vs DIBSECT Functions

Both CreateBitmapObjectFromDibFile (DIBCONV) and CreateDibSectionFromDibFile (DIBSECT) create bitmaps from DIB files, but they handle memory, access, and flexibility very differently. Here's the breakdown:

I. File Reading & Memory Allocation

DIBCONV (CreateDIBitmap)

- Reads the **entire DIB file** in one go.
- Passes the memory pointer directly to CreateDIBitmap.
- Memory is **owned and controlled by the system**.

DIBSECT (CreateDIBSection)

- Reads the file **in steps**:
    1. BITMAPFILEHEADER first.
    2. Allocates memory for BITMAPINFO.
    3. Reads the structure into memory.
- Creates a DIB Section with a **pointer to the pixels (pBits)** that the application can modify.
- Memory for pixel bits is still system-managed but **accessible directly by the app**.

II. Memory Ownership & Access

FEATURE	DIBCONV	DIBSECT
Memory control	System	System owns memory, but app can modify via `pBits`
Direct pixel modification	❌ No	✅ Yes, pointer `pBits` allows direct access
Bitmap handle	Typical handle, format conversion happens during creation	Handle references system memory; format conversion happens when drawing (`BitBlt` / `StretchBlt`)

III. Accessing & Modifying Pixel Bits

DIBCONV:

- Application **cannot** directly change pixel data.
- Any modifications require extra copying or GDI operations.

DIBSECT:

- pBits lets you **read and write pixels directly**.
- Very useful for filters, effects, or image processing.

```
// Example: invert all pixels in a 32-bit DIB Section
for (int i = 0; i < width * height; i++) {
    uint32_t* pixel = (uint32_t*)pBits + i;
    *pixel = 0x00FFFFFF - (*pixel & 0x00FFFFFF); // invert RGB, keep alpha
}
```

Tip: On Windows NT, batch GDI calls first, then call GdiFlush() before manipulating bits manually.

IV. Format Conversion

- **DIBCONV:** Happens **during CreateDIBitmap**.
- **DIBSECT:** Happens **when drawing** via BitBlt or StretchBlt.
- This means CreateDIBSection handles **system memory bitmap conversion lazily**, but you can still access raw pixels directly.

V. Color Organization

- **DIBCONV:** Color order follows the device context (HDC) used to create the bitmap.
- **DIBSECT:** Color order is defined by BITMAPINFOHEADER.
- You can still select a DIB Section into a compatible memory DC to match the screen display.

VI. Row Byte Length Differences

- CreateDIBSection sometimes gives unexpected bmWidthBytes due to **alignment padding**.
- Example: 24-bit bitmap, width = 2 pixels → bmWidthBytes may be **8 bytes** instead of 6.
- This is normal and due to DWORD alignment of rows in DIB Sections.

VII. DIBSECTION Structure

CreateDIBSection lets you retrieve **full bitmap info** using GetObject:

```
DIBSECTION ds;
GetObject(hBitmap, sizeof(DIBSECTION), &ds);

// Access info
BITMAP bmp = ds.dsBm;
BITMAPINFOHEADER bih = ds.dsBmih;
BYTE* pixelPtr = ds.dsBm.bmBits;
```

- Combines **BITMAP** + **BITMAPINFOHEADER**
- Gives complete details about the DIB Section.

VIII. Color Table Access

You can read or modify the color table of a DIB Section:

```
COLORREF colors[256];
int numColors = GetDIBColorTable(hdcMem, 0, 256, colors);

// Modify first color
colors[0] = RGB(255, 0, 0);
SetDIBColorTable(hdcMem, 0, 256, colors);
```

- Useful for paletted bitmaps.
- Only available with DIB Sections, not CreateDIBitmap.

✅ Summary of Differences

FEATURE	DIBCONV (CREATEDIBITMAP)	DIBSECT (CREATEDIBSECTION)
Memory control	System	System + pointer to pixels for app
Direct pixel access	❌ No	✅ Yes via `pBits`
Format conversion	During creation	During drawing
Color table access	Limited	Full access via `GetDIBColorTable`
Row alignment	Matches device	May include padding for `DWORD` alignment
Use case	Simple bitmap creation	Pixel-level control or custom manipulation

Use CreateDIBitmap for **quick, simple bitmap loading**.

Use CreateDIBSection when you want **direct pixel access, color table control, and full flexibility**.

FILE-MAPPING APPROACH IN CREATEDIBSECTION

File mapping can be a neat way to handle **large bitmaps** without loading the entire file into memory. But there's a catch: CreateDIBSection has a **DWORD-aligned offset requirement**, which can make things tricky.

1. What is File Mapping?

- Treat a file like it's in memory.
- Access pixels **directly from disk** without reading the whole file.
- Great for **large DIBs** that would otherwise use a lot of RAM.

2. How It Works

- Use CreateFileMapping to make a memory-mapped file.
- Call CreateDIBSection using the mapping and a **dwOffset** for where pixel data starts.
- Problem: dwOffset **must be a multiple of 4** (DWORD-aligned).

3. The Alignment Problem

- Standard bitmap file header (BITMAPFILEHEADER) is **14 bytes**.
- bfOffBits (offset to pixel data) might **not be divisible by 4**.
- Passing it directly to CreateDIBSection can fail.

4. The Solution: Align the Offset

Round bfOffBits up to the next multiple of 4 using:

```
DWORD dwAlignedOffset = (bmfh.bfOffBits + 3) & ~3;
```

- Then use dwAlignedOffset in CreateDIBSection.

5. Example Function

```
HBITMAP CreateDibSectionMappingFromFile(PTSTR szFileName, BYTE** pBits) {
    // Open file
    HANDLE hFile = CreateFile(szFileName, GENERIC_READ, FILE_SHARE_READ, NULL,
                              OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE) return NULL;

    // Map file
    HANDLE hFileMap = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
    if (!hFileMap) {
        CloseHandle(hFile);
        return NULL;
    }

    // Read BITMAPFILEHEADER
    BITMAPFILEHEADER bmfh;
    DWORD bytesRead;
    ReadFile(hFile, &bmfh, sizeof(BITMAPFILEHEADER), &bytesRead, NULL);

    // Read BITMAPINFO
    BITMAPINFO* pbmi = ReadBitmapInfo(hFile); // assume helper function

    // Align offset to DWORD
    DWORD dwAlignedOffset = (bmfh.bfOffBits + 3) & ~3;

    // Create DIB Section
    HBITMAP hBitmap = CreateDIBSection(NULL, pbmi, DIB_RGB_COLORS, pBits, hFileMap, dwAlignedOffset);

    // Cleanup
    CloseHandle(hFile); // file map remains open until bitmap is deleted

    return hBitmap;
}
```

6. Key Points to Remember

- **File-Mapping Option:** Treat files as memory-mapped, reducing RAM usage for huge DIBs.
- **DWORD Alignment:** dwOffset must be a multiple of 4. Standard headers may not align naturally.
- **Solution:** Align the offset with (bfOffBits + 3) & ~3 or use a file containing only pixel bits.
- **Performance:** File mapping is efficient for large files but may add minor overhead.
- **Error Handling:** Always check handles and function results.
- **Memory Management:** Close handles and free any allocated memory properly.

💡 **Tip:**
File mapping is mostly useful when **bitmaps are huge** or you want to **share pixel memory between processes**. For normal BMPs, standard CreateDIBSection with in-memory loading is simpler.

ADVANCED DIB HANDLING – FILE MAPPING AND PERFORMANCE

This section explores **file-mapping approaches** for DIBs, potential performance issues, alternatives, and how to choose the right method in Windows programming.

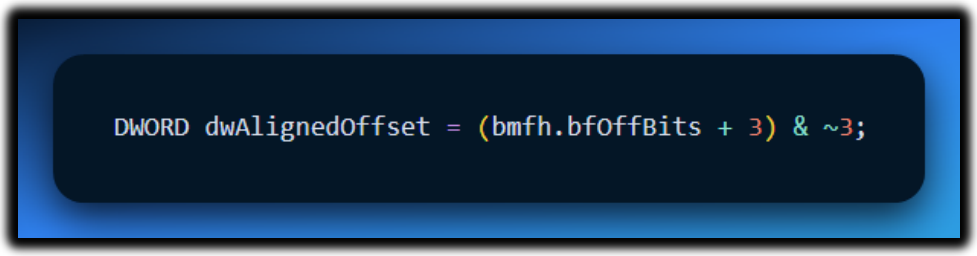
1. File-Mapping Option for DIBs

File mapping lets you **treat a file as memory**, reducing RAM usage when working with large DIBs. You don't need to load the entire bitmap into memory—you can access pixel bits directly from disk.

Key Limitation

- CreateDIBSection requires the dwOffset argument (where pixel bits start) to be a multiple of 4 (DWORD-aligned).
- Standard BMP headers (BITMAPFILEHEADER) are **14 bytes**, so bfOffBits may not be aligned.
- If not aligned, CreateDIBSection will fail.

Solution - Align the offset with:



Alternatively, create a **separate file containing only pixel bits** with a DWORD-aligned offset.

2. Memory-Mapped File Example

Memory-mapped files allow the **OS to manage caching** and paging, improving performance for large files.

```
HBITMAP CreateDibSectionMappingFromFile_MemoryMapped(PTSTR szFileName, BYTE** pBits) {
    HANDLE hFile = CreateFile(szFileName, GENERIC_READ, FILE_SHARE_READ, NULL,
                              OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE) return NULL;

    HANDLE hFileMap = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
    if (!hFileMap) {
        CloseHandle(hFile);
        return NULL;
    }

    BITMAPFILEHEADER bmfh;
    DWORD bytesRead;
    ReadFile(hFile, &bmfh, sizeof(BITMAPFILEHEADER), &bytesRead, NULL);

    BITMAPINFO* pbmi = ReadBitmapInfo(hFile); // Assume helper function

    DWORD dwAlignedOffset = (bmfh.bfOffBits + 3) & ~3;

    HBITMAP hBitmap = CreateDIBSection(NULL, pbmi, DIB_RGB_COLORS, pBits,
                                       hFileMap, dwAlignedOffset);

    CloseHandle(hFile); // File map remains valid until bitmap is deleted
    return hBitmap;
}
```

3. Separate Pixel File Approach

For maximum simplicity and DWORD alignment, create a separate file containing only the pixel bits. Offset = 0.

```
HBITMAP CreateDibSectionFromSeparateFile(PTSTR szFileName, BYTE** pBits) {
    HANDLE hFile = CreateFile(szFileName, GENERIC_READ, FILE_SHARE_READ, NULL,
                              OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE) return NULL;

    BITMAPFILEHEADER bmfh;
    DWORD bytesRead;
    if (!ReadFile(hFile, &bmfh, sizeof(BITMAPFILEHEADER), &bytesRead, NULL) || bmfh.bfType != 0x4D42) {
        CloseHandle(hFile);
        return NULL;
    }

    BITMAPINFO* pbmi = ReadBitmapInfo(hFile); // Read BITMAPINFO
    if (!pbmi) { CloseHandle(hFile); return NULL; }

    HANDLE hFileMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, NULL);
    if (!hFileMap) { free(pbmi); CloseHandle(hFile); return NULL; }

    // Offset = 0 because file contains only pixel bits
    HBITMAP hBitmap = CreateDIBSection(NULL, pbmi, DIB_RGB_COLORS, pBits, hFileMap, 0);

    free(pbmi);
    return hBitmap;
}
```

✅ Advantage: Ensures DWORD alignment, avoids complications with bfOffBits.

4. Performance Considerations

FACTOR	IMPACT	NOTES
Disk I/O	Slower than memory	Frequent access to pixel bits may lag.
Large DIBs	High virtual memory usage	Memory-mapped files help, but monitor usage.
Disk Contention	Multiple processes	Could reduce performance.
File Locking	Needed for safety	Prevents concurrent write conflicts.

Alternatives to improve performance:

1. **Buffered Reading** – Load portions of the DIB incrementally.
2. **Memory-Mapped Files** – OS manages caching and paging.
3. **Direct Memory DIB** – Use CreateDIBSection normally for smaller DIBs.

5. Choosing the Right DIB Approach

SCENARIO	METHOD	NOTES
Small DIB, frequent pixel changes	`SetDIBitsToDevice` / `StretchDIBits`	*Easy, but can be slow on older systems.*
Fast display, no pixel access needed	`CreateDIBitmap` → DDB	*Great for BitBlt, loses pixel-level access.*
Balance speed and pixel access	`CreateDIBSection`	*Efficient, allows direct pixel manipulation, better for Windows NT.*

Tip:

- For huge bitmaps that won't fit entirely in RAM, **memory-mapped files** or **separate pixel files** are the way to go.
- For small to medium bitmaps with frequent processing, stick with CreateDIBSection.

---

6. Summary

1. **File Mapping** reduces memory usage but needs DWORD-aligned offsets.
2. **Memory-Mapped Files** allow OS-managed caching, improving performance.
3. **Separate Pixel File** ensures proper alignment and simplicity.
4. **Performance trade-offs** exist with disk access vs RAM usage.
5. **CreateDIBSection** provides the **best compromise** between access and speed for most applications.

This closes Chapter 15. The next chapter will explore the **Windows Palette Manager**, which ties into displaying and manipulating bitmaps efficiently.

CHAPTER 15 DIB CHEAT-SHEET – WINDOWS PROGRAMMING

1. Direct Access to Bitmap Pixels

- Use **CreateDIBSection** to get a pointer to pixel bits (pBits).
- Pixels behave like a normal array; you can read/write directly.
- Alternative (slower): GetDIBits / SetDIBits.

```
HBITMAP hBitmap = CreateDIBSection(hdc, &bmi, DIB_RGB_COLORS, &pBits, NULL, 0);

// Invert colors example:
for (int i = 0; i < height * widthBytes; ++i) {
    pBits[i] = 255 - pBits[i];
}
```

Key: pBits points directly to memory. No need for extra calls.

2. Clipboard Bitmap Formats

FORMAT	DESCRIPTION	PIXEL ACCESS
`CF_BITMAP`	Standard HBITMAP	No raw pixels
`CF_DIB`	Older DIB, no alpha	Pixels accessible after loading
`CF_DIBV5`	Modern DIB, alpha & color profiles	Best for modern apps
`Registered`	App-defined (PNG, etc.)	Serialized/compressed bytes

3. Bitmap Serialization & Deserialization

Serialization (saving/copying):

- Save header, color table, raw pixels in memory.
- Store as file or clipboard data.

Deserialization (loading/pasting):

- Read header to understand format.
- Allocate memory.
- Copy pixels and create bitmap.

```
HBITMAP DeserializeBitmap(const char* filename) {
    return (HBITMAP)LoadImage(NULL, filename, IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE);
}

void SerializeBitmap(HBITMAP hBitmap, const char* filename) {
    BITMAP bmp;
    GetObject(hBitmap, sizeof(BITMAP), &bmp);
    // Helper function to save as BMP
    SaveDIBAsBMP(hBitmap, filename);
}
```

Analogy: Serialization = packing clothes; deserialization = unpacking them.

4. DIBCONV vs DIBSECT

FEATURE	DIBCONV (CREATEBITMAPFROMDIB)	DIBSECT (CREATEDIBSECTIONFROMDIB)
Memory Reading	Reads entire file at once	Reads header, then `BITMAPINFO`, then pixels
Memory Ownership	**SYSTEM** owns memory	**SYSTEM** owns pBits; app can modify
Pixel Modification	**Not allowed**	Direct via `pBits`
Format Conversion	Done during `CreateDIBitmap`	Done during `BitBlt` / `StretchBlt`
Handle Behavior	Standard `HBITMAP`	`DIBSECTION` handle gives pixel access and color flexibility
Color Table	N/A	Can read/write via `GetDIBColorTable` / `SetDIBColorTable`
Byte Alignment	Standard	Row bytes may be padded (e.g., 24bpp, 2 pixels → 8 bytes)

Key: DIBSECT = control + pixel access, DIBCONV = simplicity.

5. DIBSECTION Structure

Get via GetObject(hBitmap, sizeof(DIBSECTION), &dibSec);

Combines **BITMAP** + **BITMAPINFOHEADER** info.

Allows:

- Access to pixels via pBits
- Row alignment info
- Color table operations

```
DIBSECTION dibSec;
GetObject(hBitmap, sizeof(DIBSECTION), &dibSec);
BYTE* pixels = (BYTE*)dibSec.dsBm.bmBits;
```

6. File-Mapping & Memory-Mapped Bitmaps

Why Use File Mapping?

- Treat large bitmap files like memory without loading all at once.
- Good for large DIBs that would eat RAM.

DWORD Alignment Issue

- CreateDIBSection requires **dwOffset % 4 == 0**
- Standard BMP header (BITMAPFILEHEADER) is 14 bytes → often unaligned.
- Fix:

```
DWORD dwAlignedOffset = (bmfh.bfOffBits + 3) & ~3;
```

Memory-Mapped File Example:

```
HBITMAP CreateDibSectionMappingFromFile(PTSTR szFileName, BYTE** pBits) {
    HANDLE hFile = CreateFile(szFileName, GENERIC_READ, FILE_SHARE_READ, NULL,
                              OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    HANDLE hFileMap = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);

    BITMAPFILEHEADER bmfh;
    DWORD bytesRead;
    ReadFile(hFile, &bmfh, sizeof(BITMAPFILEHEADER), &bytesRead, NULL);

    BITMAPINFO* pbmi = ReadBitmapInfo(hFile);

    DWORD dwAlignedOffset = (bmfh.bfOffBits + 3) & ~3;
    HBITMAP hBitmap = CreateDIBSection(NULL, pbmi, DIB_RGB_COLORS, pBits,
                                       hFileMap, dwAlignedOffset);

    CloseHandle(hFile); // Mapping remains valid
    return hBitmap;
}
```

Separate Pixel File Approach

- Store only pixel bits in a separate file.
- Offset = 0 → avoids alignment issues.
- Ideal for large files or shared memory.

```
HBITMAP CreateDibSectionFromSeparateFile(PTSTR szFileName, BYTE** pBits) {
    HANDLE hFile = CreateFile(szFileName, GENERIC_READ, FILE_SHARE_READ, NULL,
                              OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    BITMAPINFO* pbmi = ReadBitmapInfo(hFile);
    HANDLE hFileMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, 0, NULL);
    HBITMAP hBitmap = CreateDIBSection(NULL, pbmi, DIB_RGB_COLORS, pBits, hFileMap, 0);
    free(pbmi);
    return hBitmap;
}
```

7. Performance Tips

- Disk I/O = slower than memory → frequent pixel access may lag.
- Large DIBs = high virtual memory usage.
- Memory-mapped files = OS-managed caching → faster than manual reads.
- Use buffered reading if memory-mapping isn't feasible.
- Consider file locking when multiple processes may access the same file.

8. Choosing the Right DIB Method

SCENARIO	METHOD	NOTES
Small DIB, frequent pixel edits	`SetDIBitsToDevice / StretchDIBits`	*Easy to implement, but can be slow on older systems.*
Fast display, no pixel access	`CreateDIBitmap → DDB`	*BitBlt is extremely fast, but you lose raw pixel-level access.*
Balanced speed & pixel access	`CreateDIBSection`	*Best for Windows NT; allows direct access and is BitBlt compatible.*
Huge DIB, memory savings	`Memory-mapped / Separate pixel file`	*OS manages caching via virtual memory; requires careful **DWORD** alignment.*

Rule of Thumb:

- Small + frequent edits → SetDIBitsToDevice
- Fast display only → DDB
- Need pixel access + performance → CreateDIBSection
- Huge files → memory-mapped / pixel file

CHAPTER 15 – DIB QUESTIONS (50)

Direct Access & Basics

1. What function allows direct access to a bitmap's pixel data in Windows?
2. How do you obtain a pointer to the pixels when creating a DIB?
3. What type of bitmap allows direct pixel access: DDB or DIBSECTION?
4. Why is CreateDIBSection preferred over GetDIBits/SetDIBits for direct pixel access?
5. How do you modify pixels once you have a pBits pointer?
6. What happens if you try to access pixels of a standard HBITMAP without a DIBSECTION?
7. What is the difference between DIB_RGB_COLORS and DIB_PAL_COLORS?
8. Give a simple C++ example of inverting the colors of a bitmap using pBits.
9. What is the role of the bmWidthBytes field in the BITMAP structure?
10. Explain why row padding might occur in DIBSECTIONs.

Clipboard Formats

11. Name three standard clipboard formats for bitmaps.
12. What is the key limitation of CF_BITMAP?
13. Which clipboard format supports alpha channels?
14. How do registered formats like "PNG" differ from CF_DIB?
15. What kind of data is stored in CF_DIB?
16. Why is CF_DIBV5 preferred for modern applications?

17. Can you modify pixel data directly from CF_DIB?
18. How does Windows expect memory to be organized for CF_DIB?
19. What function can you use to register a custom clipboard format?
20. Explain in simple terms the difference between DIB and DDB clipboard formats.

Serialization & Deserialization

21. What is bitmap serialization?
22. What is bitmap deserialization?
23. Name the three components typically included when serializing a bitmap.
24. Give a C++ example of deserializing a bitmap from a BMP file.
25. Give a C++ example of serializing a bitmap to a BMP file.
26. Why is serialization compared to packing a suitcase?
27. What function is typically used to load a BMP from disk in Windows?
28. Why is it important to read the header first when deserializing a bitmap?
29. What is the purpose of allocating memory during deserialization?
30. How does the system handle pixel data when using CF_DIB clipboard format?

DIBCONV vs DIBSECT

31. How does DIBCONV read a DIB file?
32. How does DIBSECT read a DIB file?
33. Which approach allows direct modification of pixel bits?
34. What function does DIBCONV rely on to create a bitmap?
35. What function does DIBSECT rely on to create a bitmap?
36. How does memory ownership differ between DIBCONV and DIBSECT?
37. When does format conversion occur in DIBCONV?
38. When does format conversion occur in DIBSECT?
39. How can you access the color table in DIBSECT?

40. Explain why bmWidthBytes might not match expected values in a 24bpp DIBSECTION.

File-Mapping & Memory-Mapped Bitmaps

41. What is the main benefit of using a file-mapped DIB?
42. Why must the offset (dwOffset) be a multiple of 4 in CreateDIBSection?
43. What is the typical size of a BITMAPFILEHEADER?
44. How can you align bfOffBits to meet the DWORD multiple requirement?
45. Give a C++ snippet for creating a DIBSECTION with a memory-mapped file.
46. What is the advantage of using a separate file containing only pixel bits?
47. Name two performance considerations when using file-mapped DIBs.
48. How can memory-mapped files improve performance over direct file access?
49. What precautions should you take when multiple processes access a memory-mapped DIB?
50. Summarize why CreateDIBSection is considered a balanced approach between speed and pixel access.

✅ End of Chapter 15 Cheat-Sheet ✅

Next up in Chapter 16: **Windows Palette Manager**, which ties into color handling and bitmap presentation.