CHAPTER 15: THE DEVICE-INDEPENDENT BITMAP

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 Saving DDBs to disk and loading them back into memory is cumbersome and outdated due to their device-dependent nature.

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

Device Dependence vs. Device Independence:

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

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

Benefits of DIBs:

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



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



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



DIBs in the Modern Landscape:

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

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

Remember:

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

DELVING DEEPER INTO THE DIB FILE FORMAT: A COMPREHENSIVE EXPLORATION

Origins and Evolution:

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

Key Characteristics:

File Extensions: .BMP and .DIB serve as the common file extensions associated with DIB files.

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

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

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

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

Common Applications:

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

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

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

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

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

File Structure:

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

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

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

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

In-Memory Representation:

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

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

Windows API Support:

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

Beyond Built-in Functions:

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

Key Takeaways:

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

DELVING INTO THE OS/2-STYLE DIB FORMAT

File Structure:

File Header (14 bytes):

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

Information Header (12 bytes):

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

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

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

Pixel Bits:

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

Code Examples:

Allocating memory for an 8-bit DIB information structure:



Accessing a color table entry:



Key Points:

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



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

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

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

BITMAPFILEHEADER:

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

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

BITMAPCOREHEADER:

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

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

RGBTRIPLE:

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

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

BITMAPCOREINFO:

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

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

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

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

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

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

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

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

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

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

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

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

Terminology:

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

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

Why the Bottom-Up Order?

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

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



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



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

Consequences:

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

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

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

DECODING THE LANGUAGE OF DIB PIXEL BITS: A CLOSER LOOK

Revealing the Pixel Grid:

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

Bottom-Up Climb:

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

Left-to-Right March:

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

Padding for Efficiency:

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

Bit Depth Guides Pixel Encoding:

1-bit DIBs (Simple Black and White):

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



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:



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

1. File Structure:

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

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

2. Pixel Encoding:

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

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

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

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

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

3. Bottom-Up Order:

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

Navigating the Pixel Landscape:

Unraveling DIBs: Decoding and Tweaking Pixels with Confidence

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

Understanding Pixel Access:

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



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

Pixel Manipulation Magic:

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

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

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

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

Remember:

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

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

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

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

1. File Header:

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

2. Information Header:

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

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

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

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

4. BITMAPINFO Structure:

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



Key Points:

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

DEMYSTIFYING THE EXPANDED WINDOWS DIB: A SIMPLIFIED EXPLORATION

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

New Fields and Features:

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

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

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

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

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

Color Table Twists:

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

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

Pixel Bits:

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

Remember:

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

Pixels per Meter: Revealing Real-World Dimensions

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

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

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

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

Common Values:

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

Tailoring the Color Palette: The Power of biClrUsed

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

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

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

Essential Guidelines:

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

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

Windows 95's Colorful Twist:

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

Understanding DIBs (Device-Independent Bitmaps):

Key Points:

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

Important Fields in the Header:

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

Color Table:

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

Pixel Data:

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

Additional Notes:

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

1. Overview of DIBs

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

2. Evolution of DIBs

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

3. Common 8-bit DIBs

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

4. Gray-shade DIBs

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



5. Variations in Gray-shade DIBs

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

6. Palletized 8-bit Color DIBs

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

7. Uncommon Fields

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

8. Potential Enhancement

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

Key Points:

Common DIB Types:

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

8-Bit DIBs:

Gray-Shade DIBs:

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

Palletized Color DIBs:

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

Code Examples:

Generating Gray-Shade Color Tables:



Accessing Pixel Values in Gray-Shade DIBs:



Additional Notes:

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

COMPRESSION IN DEVICE-INDEPENDENT BITMAPS (DIBS):

1. Overview

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

2. Compression Constants

The biCompression field can take four constants: BI\_RGB, BI\_RLE8, BI\_RLE4, or BI\_BITFIELDS, represented by values 0 through 3 in the WINGDI.H header. For 4-bit and 8-bit DIBs, it indicates whether the pixel bits are compressed using run-length encoding. For 16-bit and 32-bit DIBs, it signifies the use of color masking introduced in Windows 95.

3. RLE Compression

For 1-bit DIBs: biCompression is always BI\_RGB.

For 4-bit DIBs: biCompression can be either BI\_RGB or BI\_RLE4.

For 8-bit DIBs: biCompression can be either BI\_RGB or BI\_RLE8.

For 24-bit DIBs: biCompression is always BI\_RGB.

4. Run-Length Encoding (RLE) Overview

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

5. RLE Compression for 8-bit DIBs

The table below illustrates how pixel bits are encoded when biCompression is BI\_RLE8.



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

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

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

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

6. Compression for 4-bit DIBs

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

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

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

7. Handling DIB Image Areas

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

8. Size Information

When biCompression is BI\_RLE4 or BI\_RLE8, biSizeImage indicates the size of DIB pixel data in bytes.

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

9. Note on Compression

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

Understanding Compression Fields in-depth:

* biCompression: This field within the BITMAPINFOHEADER structure reveals the compression method applied to the DIB's pixel data. It can hold four possible values:
* BI\_RGB: No compression (standard storage for 1-bit, 4-bit, 8-bit, and 24-bit DIBs).
* BI\_RLE8: Run-length encoding for 8-bit DIBs.
* BI\_RLE4: Run-length encoding for 4-bit DIBs.
* BI\_BITFIELDS: Color masking for 16-bit and 32-bit DIBs (introduced in Windows 95).
* biSizeImage: Specifies the size of compressed pixel data in bytes (when compression is used). It's typically 0 for uncompressed DIBs (BI\_RGB), but can be set to image height multiplied by row byte length.

Run-Length Encoding (RLE):

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

Implementation:

8-bit RLE:

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

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

Key Considerations:

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

Additional Insights:

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

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

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

Code Example (8-bit RLE Decoding):



Pg 585 book.

UNDERSTANDING THE DIB STRUCTURE AND COLOR MASKING:

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

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

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

Color Masking Techniques:

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

BI\_RGB Compression:

16-bit DIBs:

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

32-bit DIBs:

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

BI\_BITFIELDS Compression:

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

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

This isolates the relevant bit pattern for each color component.

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

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

Visualizing Color Extraction:

16-bit DIB (BI\_RGB):

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

32-bit DIB (BI\_RGB):

Extracting Color Components

1. *Pixel Data Layout:*

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



Blue occupies the first byte (BB).

Green occupies the second byte (GG).

Red occupies the third byte (RR).

The fourth byte is always 0.

1. *Applying Masks and Shifting:*

*To extract the red value:*

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

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

*To extract the green value:*

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

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

*To extract the blue value:*

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

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



1. *Illustrative Example:*

Pixel value: 0x0048E058

*Red value extraction:*

(0x00FF0000 & 0x0048E058) = 0x00004800

(0x00004800) >> 16 = 0x48

*Green value extraction:*

(0x0000FF00 & 0x0048E058) = 0x0000E000

(0x0000E000) >> 8 = 0xE0

*Blue value extraction:*

(0x000000FF & 0x0048E058) = 0x00000058



Visual Representation:

Key Points:

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

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

Color masking techniques:



Header and Typedef:

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

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

MaskToRShift Function:

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

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

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

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

MaskToLShift Function:

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

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

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

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

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

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

Main Function and Pixel Value Extraction:

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

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

Extracting Color Components:

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

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

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

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

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

Displaying Extracted Values:

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

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

Overall Functionality:

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

The Role of Color Masks:

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

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

Understanding BI\_BITFIELDS Compression:

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

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

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

Decoding Colors Using Masks:

Retrieving the Masks:

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

Calculating Shift Values:

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

Applying Masks and Shifting:

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

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

Illustrative Example (16-bit DIB with BI\_BITFIELDS):



Key Considerations:

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

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



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

Windows 95/98 Mask Restrictions:

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

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

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

*Custom color layouts in DIBs with BI\_BITFIELDS can optimize color representation in various ways. Here are a few examples:*

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



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



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



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

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



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



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



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



Beyond the Basics:

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

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



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

