CHAPTER 16: THE PALETTE MANAGER - UNVEILING THE WORLD OF 256 COLORS

This chapter delves into the realm of the Windows Palette Manager, a tool born out of necessity due to certain hardware limitations.

A paletter lookup table on video boards works like:



While modern video adapters often support higher color depths like 24-bit or 16-bit, certain setups, particularly on laptops or in high-resolution modes, are constrained to 8 bits per pixel. This limitation translates to a palette of only 256 simultaneous colors.

The question arises: What can be accomplished with a palette of 256 colors? While 16 colors are insufficient for displaying realistic images, and thousands or millions of colors are more than ample for such tasks, the middle ground of 256 colors presents unique challenges.

To effectively showcase real-world images with this limited palette, colors must be carefully selected for each specific image. A one-size-fits-all "standard" set of 256 colors isn't feasible, as it won't cater to the diverse needs of every application.

Enter the Windows Palette Manager. This tool is designed for precisely specifying the colors required by a program when operating in an 8-bit video mode.

If your programs exclusively run in higher bit depths, you may not encounter the need for the Palette Manager. Nevertheless, this chapter holds valuable insights, particularly in tying up loose ends related to bitmap handling.

Key Points:

Hardware Limitations: Certain video adapters restrict color depth to 8 bits per pixel, allowing only 256 colors simultaneously.

Palette Manager's Purpose: Tailored for programs operating in 8-bit video modes, the Palette Manager enables the specification of essential colors.

Color Selection Challenge: Unlike higher color depths, where a standard set suffices, 256 colors require careful curation for each application's unique needs.

Understanding the Palette Manager is crucial for developers navigating the constraints of 8-bit video modes.

While it may not be applicable in all scenarios, its insights into color management are invaluable, especially when working with real-world images in resource-limited environments.

The 20 reserved colors in 256-color video modes:



The 20 reserved colors in 256-color video modes are part of the Windows Palette Manager, and they serve as standard colors that are predefined for system use. These colors are reserved to maintain consistency across applications running in an 8-bit video mode. Here is a description of each of the 20 reserved colors:

1. Black (Pixel Bits: 00000000, RGB Value: 00 00 00)
2. Dark Red (Pixel Bits: 00000001, RGB Value: 80 00 00)
3. Dark Green (Pixel Bits: 00000010, RGB Value: 00 80 00)
4. Dark Yellow (Pixel Bits: 00000011, RGB Value: 80 80 00)
5. Dark Blue (Pixel Bits: 00000100, RGB Value: 00 00 80)
6. Dark Magenta (Pixel Bits: 00000101, RGB Value: 80 00 80)
7. Dark Cyan (Pixel Bits: 00000110, RGB Value: 00 80 80)
8. Light Gray (Pixel Bits: 00000111, RGB Value: C0 C0 C0)
9. White (Pixel Bits: 11111111, RGB Value: FF FF FF)
10. Cyan (Pixel Bits: 11111110, RGB Value: 00 FF FF)
11. Magenta (Pixel Bits: 11111101, RGB Value: FF 00 FF)
12. Blue (Pixel Bits: 11111100, RGB Value: 00 00 FF)
13. Dark Gray (Pixel Bits: 11111000, RGB Value: 80 80 80)
14. Medium Gray (Pixel Bits: 11110111, RGB Value: A0 A0 A4)
15. Cream (Pixel Bits: 11110110, RGB Value: FF FB F0)
16. Sky Blue (Pixel Bits: 11110101, RGB Value: A6 CA F0)
17. Money Green (Pixel Bits: 11110100, RGB Value: C0 DC C0)
18. Reserved
19. Reserved
20. Reserved

In 256-color display modes, Windows manages a system palette that mirrors the video card's hardware color lookup table. This system palette controls the available colors for display.

By default, Windows provides a specific set of colors as the system palette, which is depicted in Figure above.

It holds 256 colors, 20 of which are fixed for system elements, while applications can customize the remaining 236.

Applications can adjust these colors using logical palettes. If multiple applications use logical palettes, Windows prioritizes the active window (the one in the foreground with a highlighted title bar), ensuring its color choices take precedence.

In scenarios where multiple applications are using logical palettes simultaneously, Windows gives the highest priority to the active window.

The active window refers to the window that currently has the highlighted title bar and appears in the foreground of all other windows.

This ensures that the active application's color choices take precedence over other applications, providing a consistent and coherent visual experience.

To explore this concept practically, we'll examine a sample program later in this chapter. To align with these examples, consider switching your display to 256-color mode.

Access display settings by right-clicking on your desktop, selecting "Properties," and choosing the "Settings" tab.

Key points:

* System palette: Master color table for Windows in 256-color modes.
* Logical palettes: Application-specific color customizations.
* Active window priority: Windows prioritizes the active application's colors.
* Switching to 256-color mode: Recommended for compatibility with examples.

Additional notes:

* While modern systems often use higher color depths, understanding system palettes is still relevant for legacy applications and specific development scenarios.
* The specific process for adjusting display settings may vary slightly depending on your Windows version.

GRAYS1 PROGRAM



The GRAYS1 program is designed to display 65 shades of gray as a "fountain" of color, ranging from black to white. It does not use the Windows Palette Manager but instead directly creates and fills rectangles with varying shades of gray.

The program starts by defining the necessary headers and function prototypes. It then defines the WinMain function, which is the entry point of the program. Inside WinMain, the program registers a window class, creates a window, and enters the message loop.

The WndProc function is the window procedure for handling messages related to the program's window. It handles messages such as WM\_SIZE, WM\_PAINT, and WM\_DESTROY.

In the WM\_SIZE message case, the cxClient and cyClient variables are updated with the width and height of the client area of the window, respectively.

In the WM\_PAINT message case, the program prepares to paint the window. It begins by obtaining a device context (hdc) and a paint structure (ps) using the BeginPaint function. Then, a loop is executed 65 times to create and fill rectangles with varying shades of gray.

Inside the loop, the rect structure is defined to represent the dimensions of each rectangle. The left and right coordinates of the rectangle are calculated based on the current iteration and the total number of shades. The top and bottom coordinates are set to cover the entire height of the client area.

A brush (hBrush) is created using the CreateSolidBrush function, specifying the RGB values for the gray color. The RGB values are derived from the current iteration to create a gradient effect.

The FillRect function is then used to fill the current rectangle with the gray color represented by the brush. After filling the rectangle, the brush is deleted to release the associated resources.

Finally, the program calls EndPaint to signal the end of the painting process and returns 0 to indicate that the message has been handled.

In the WM\_DESTROY message case, the program posts a quit message to exit the message loop and terminate the program.

If any other messages are received or not handled in the WndProc function, the program calls DefWindowProc to perform the default window procedure for those messages.



*Let's dive deeper into the GRAYS1 program and explore the section where color palettes are mentioned.*

In the given code, the GRAYS1 program does not use the Windows Palette Manager. Instead, it directly creates and fills rectangles with varying shades of gray using the RGB color model.

The concept of color palettes in computer graphics refers to a limited set of colors that are available for use in a particular system or application. In the Windows operating system, a palette is a data structure that holds a fixed number of colors, typically 256 colors. The Windows Palette Manager is responsible for managing and mapping colors from the system palette to the colors used by an application.

However, in the GRAYS1 program, the focus is on displaying shades of gray rather than utilizing a predefined color palette. The program achieves this by dynamically calculating and creating shades of gray using the CreateSolidBrush function and RGB values.

Within the WM\_PAINT message case, a loop is executed 65 times to create 65 rectangles, each representing a different shade of gray. The RGB values for each shade are calculated as min (255, 4 \* i), where i is the current iteration of the loop.

By multiplying i by 4 and clamping the result to a maximum of 255, the program ensures that the RGB values stay within the valid range for a grayscale color. This calculation creates a gradient effect, where the shades of gray become progressively lighter as i increases.

The CreateSolidBrush function is then used to create a brush with the calculated RGB values, representing the current shade of gray. The FillRect function fills the current rectangle with the gray color represented by the brush.

It's important to note that by creating and using brushes directly, the GRAYS1 program bypasses the Windows Palette Manager and the limitations of a fixed color palette. Instead, it dynamically generates and displays the shades of gray as a "fountain" of color in the client area of the window.

This approach allows for greater flexibility in displaying a wider range of shades and gradients, as it leverages the full RGB color space rather than being constrained by a predefined palette.

Here's a clearer and more concise explanation of the program's behavior in 256 color mode:

During the WM\_PAINT message, the program paints 65 rectangles using different gray shades, ranging from black to white.

Dithering: To achieve these shades, Windows employs a technique called "dithering." It blends combinations of the four pure colors available in the system palette (black, dark gray, light gray, and white) to simulate additional shades, resulting in a grainy pattern.

Lines and text: In contrast, lines and text in Windows are typically drawn using only the pure colors, without dithering.

Bitmaps: When displaying bitmaps in 256-color mode, Windows approximates them using the 20 standard system colors, often leading to color inaccuracies. Dithering is not typically applied to bitmaps.

Palette Manager functions and messages: The GRAYS2 program, featured in Figure 16-3, demonstrates key concepts of color management in Windows through the use of Palette Manager functions and messages.

Key takeaways:

Limited color palette: In 256-color mode, Windows has a restricted set of colors.

Dithering for filled areas: Windows uses dithering to simulate additional colors within filled areas.

Pure colors for lines and text: Lines and text are drawn using only the pure colors available in the system palette.

Color approximations for bitmaps: Bitmaps are approximated using the standard system colors, often resulting in inaccuracies.

Palette Manager for color control: The Palette Manager functions and messages provide a way for applications to manage color palettes in Windows.

The GRAYS2 program is a Windows application that displays a gradient of gray shades using the Palette Manager. Let's break down its functionality into paragraphs:

GRAYS2 PROGRAM





Application Structure:

The program is structured as a Windows application with a standard WinMain function and a window procedure (WndProc).

It defines a window class (WNDCLASS) with basic attributes such as the window procedure, background brush, icon, and cursor.

Palette Initialization (WM\_CREATE):

Upon window creation, the program sets up a LOGPALETTE structure to define a custom palette containing 65 shades of gray.

Each palette entry is assigned RGB values representing varying intensities of gray.

The CreatePalette function is then called to create a logical palette based on the provided information.

Handling Window Size (WM\_SIZE):

The program handles the WM\_SIZE message to update the client area dimensions whenever the window is resized. This information is crucial for drawing the gradient.

Painting the Gradient (WM\_PAINT):

In response to the WM\_PAINT message, the program begins painting by obtaining a device context (HDC) using BeginPaint.

It selects the custom palette into the device context and realizes the palette to make it effective for the current device.

The gradient is drawn by creating a series of rectangles, each filled with a solid brush of a specific gray shade. The shades vary from black to white.

Palette Interaction (WM\_QUERYNEWPALETTE, WM\_PALETTECHANGED):

The program handles palette-related messages to ensure proper interaction with the system and other applications.

WM\_QUERYNEWPALETTE is used to respond to a request for palette selection, indicating whether the application has a palette to select.

WM\_PALETTECHANGED is used to handle changes in the system palette, updating the application's palette if necessary.

Cleanup on Window Destruction (WM\_DESTROY):

Upon window destruction, the program deletes the logical palette using DeleteObject and posts a quit message to terminate the application.

The Palette Manager plays a crucial role in managing colors in Windows applications, particularly when dealing with limited color environments such as 8-bit video modes.

The GRAYS2 program exemplifies the use of the Palette Manager to create a logical palette and display a gradient of gray shades. Let's delve into the detailed explanation:



Creating a Logical Palette:

The initial step involves creating a logical palette using the CreatePalette function.

In the WM\_CREATE message handler, the program initializes a LOGPALETTE structure to define its custom palette. The structure includes a version, the number of entries, and an array of PALETTEENTRY structures.

Each PALETTEENTRY structure represents an RGB color value. The program sets up 65 shades of gray, calculating the appropriate intensity values and storing them in the palette.

Palette Initialization Details:

The LOGPALETTE structure is defined with compatibility information and an array of PALETTEENTRY structures.

Each PALETTEENTRY structure consists of red, green, and blue color components, each represented by a byte. The peFlags field is set to 0.

Memory is allocated for the LOGPALETTE structure and additional PALETTEENTRY structures to accommodate the desired number of shades.

Palette Selection and Realization:

During the WM\_PAINT message processing, the program selects the logical palette into the device context using SelectPalette. The logical palette is then "realized" in the device context using RealizePalette.

Selecting a palette into the device context is crucial for the system to map the colors to the actual physical palette of the video board.

The SelectPalette function returns the handle of the previous logical palette, allowing for restoration if needed.

Color Specification and Use of PALETTERGB:

The program continues to use the familiar RGB macro to specify colors in the logical palette. However, it introduces the concept of "Palette RGB" values using the PALETTERGB macro.

A "Palette RGB" color is similar to an RGB color but with the high byte of the COLORREF value set to 2.

Rules are provided for using colors in the logical palette, emphasizing the need to specify colors using Palette RGB values or Palette Index values when working with a selected palette.

Handling Palette Management Support:

Notably, the program does not explicitly check whether the video display driver supports palette management.

In scenarios where palette management is not supported (non-256 color video modes), the program functions similarly to a version (GRAYS1) that does not utilize the Palette Manager.

In essence, GRAYS2 demonstrates a meticulous process of creating, selecting, and realizing a logical palette to effectively manage and display a gradient of gray shades, showcasing the nuanced interaction with the Palette Manager in a Windows environment.

LOGICAL PALETTE DEFINITION:



In the provided code snippet, we're dealing with the implementation of the Windows Palette Manager in the context of the GRAYS2 program. Let's break down the code and discuss its functionality in the context of logical palettes, color representation, and palette management.

The LOGPALETTE structure is a fundamental part of Windows Palette Manager. It holds information about a logical palette, including the palette version, the number of entries, and an array of PALETTEENTRY structures. Each PALETTEENTRY structure represents an RGB color value and flags.

Palette Entry Structure:

The PALETTEENTRY structure defines individual entries within the logical palette. It consists of fields representing the red, green, and blue color components, along with additional flags.

Logical Palette Creation:

The GRAYS2 program dynamically allocates memory for the LOGPALETTE structure, including space for an array of PALETTEENTRY structures. It then initializes the logical palette by setting the version, the number of entries, and RGB color values for each entry in a loop. The logical palette is created using the CreatePalette function, and memory is freed after its creation.

WM\_PAINT Handling:

When handling the WM\_PAINT message, the program begins painting by obtaining the device context (hdc) using BeginPaint. It then selects and realizes the logical palette in the device context using SelectPalette and RealizePalette. This step is crucial for mapping the colors in the logical palette to the system palette, aligning them with the physical palette of the video board.

The program proceeds to draw a "fountain of grays" by iterating over the 65 shades defined in the logical palette. For each shade, it creates a solid brush using the CreateSolidBrush function and the PALETTERGB macro. This macro is used to specify colors from the logical palette, ensuring that the additional colors are utilized. The drawn rectangle is then filled with the brush, and the brush is appropriately deleted.

Finally, the painting process is concluded using EndPaint.

Explanation Summary:

In summary, the provided code demonstrates the creation and utilization of a logical palette in the GRAYS2 program. This logical palette is essential for managing and displaying a range of gray shades, and the program ensures proper integration with the Windows Palette Manager during the painting process.

PALETTE MANAGER PRIORITY AND COLOR MAPPING:

In a multi-program environment using the Palette Manager, the active window takes precedence over the palette. The system prioritizes the most recently active window and subsequent windows accordingly.

When a new program becomes active, the Windows Palette Manager may need to reorganize the system palette table.

If a program defines a color in its logical palette identical to one of the 20 reserved colors, Windows will map that logical palette entry to the corresponding reserved color.

Moreover, if multiple applications specify the same color in their logical palettes, they will share the system palette entry.

The program can influence this behavior by using the PC\_NOCOLLAPSE flag in the peFlags field of the PALETTEENTRY structure.

Palette Manager Messages:

Two crucial messages, WM\_QUERYNEWPALETTE and WM\_PALETTECHANGED, are integral to organizing the system palette.

WM\_QUERYNEWPALETTE:

Sent to a main window when it is about to become active, this message requires processing by programs utilizing the Palette Manager.

In the context of GRAYS2, the program handles this message by obtaining a device context handle, selecting the palette into it, calling RealizePalette, and then invalidating the window to trigger a WM\_PAINT message.

The window procedure returns TRUE if it realizes its logical palette and FALSE otherwise.

WM\_PALETTECHANGED:

When the system palette changes due to a WM\_QUERYNEWPALETTE message, Windows sends the WM\_PALETTECHANGED message to all main windows, starting with the most active window.

The wParam value passed to the window procedure is the handle of the active window.

A program should process this message only if wParam is not equal to the program's window handle.

Generally, programs using a customized palette call SelectPalette and RealizePalette while processing this message.

System Palette Organization:

Upon subsequent calls to RealizePalette during the WM\_PALETTECHANGED message, Windows checks for matches of RGB colors in the logical palette with those already loaded in the system palette.

If two programs require the same color, the same system palette entry is shared. If no unused system palette entries exist, the color in the logical palette is mapped to the closest color from the 20 reserved entries.

Handling WM\_PALETTECHANGED:

For programs concerned about the appearance of the client area when inactive, processing the WM\_PALETTECHANGED message is essential.

GRAYS2 demonstrates one approach by obtaining a device context, selecting the palette into it, and calling RealizePalette.

Instead of calling InvalidateRect, GRAYS2 opts for UpdateColors, a function typically more efficient than repainting the window.

It changes the values of pixels in the window to help preserve the previous colors.

GRAYS3 PROGRAM



In the GRAYS3 program, several changes and enhancements have been introduced compared to GRAYS2, primarily focusing on how colors are managed and displayed using the Palette Manager. Let's delve into the key modifications:



Palette Management Approach:

Instead of using the PALETTERGB macro as in GRAYS2, GRAYS3 employs the PALETTEINDEX macro during the WM\_PAINT message processing. This macro is used to specify colors based on the index within the logical palette.

Palette Entry Specification:

The program still creates a logical palette using the LOGPALETTE structure, similar to GRAYS2. The palette consists of 65 entries, each representing a different shade of gray. The entries are specified with varying levels of red, green, and blue components, providing a gradient effect.

Palette Realization in WM\_PAINT:

During the WM\_PAINT message processing, the program selects the logical palette into the device context and calls RealizePalette to map the colors to the system palette. However, in GRAYS3, the colors are drawn using the PALETTEINDEX macro, indicating an index within the logical palette, which simplifies color specification.

Drawing with PALETTEINDEX:

The fountain of grays is drawn in the client area of the window using a loop. For each iteration, a rectangle representing a segment of the window is filled with a solid brush created using the PALETTEINDEX macro. This approach directly references the index in the logical palette to determine the color.

Handling Palette Change Messages:

The program responds to the WM\_QUERYNEWPALETTE message, which is sent when the window is about to become active. In this case, it obtains the device context, selects and realizes the palette, and invalidates the window to trigger a repaint.

Additionally, the WM\_PALETTECHANGED message is processed, ensuring that the program updates its colors if another window's palette has changed. This involves selecting and realizing the palette in response to the system palette change.

Efficient UpdateColors Function:

GRAYS3 introduces the use of the UpdateColors function instead of directly repainting the window in response to a palette change.

This function efficiently updates the pixel values in the window to preserve the previous colors without the need for a full repaint.

Palette indices and Palette RGB colors represent different ways of specifying colors in Windows applications.

A Palette Index color consists of a high byte set to 1, and the low byte represents an index into the logical palette currently selected in the device context.

In the case of GRAYS3, the logical palette used has a total of 65 entries, with indices ranging from 0 to 64.

Each index corresponds to a specific color in the palette, allowing for a wide range of shades.

For example, PALETTEINDEX(0) represents black, PALETTEINDEX(32) represents medium gray, and PALETTEINDEX(64) represents white.

Using palette indices offers efficiency advantages over using RGB values because Windows does not need to perform a nearest-color search.

Instead of specifying the RGB values directly, the program can simply reference the desired color by its corresponding palette index.

This approach eliminates the need for Windows to search for the closest matching color in the palette, resulting in faster color retrieval and rendering.

By utilizing palette indices, Windows applications can leverage the predefined logical palette and directly access the desired colors without additional calculations.

This approach is particularly beneficial in scenarios where palette manipulation is supported and a limited color depth, such as 256 colors, is used.

In summary, palette indices provide a more efficient means of specifying colors in Windows applications compared to RGB values.

By referencing the desired color through the index, the need for a nearest-color search is eliminated, resulting in improved performance and streamlined color retrieval.

GRAYS3 refines the color management process by utilizing the PALETTEINDEX macro, simplifying color specification, and maintains responsiveness to palette changes with efficient updates, offering an enhanced visual experience compared to GRAYS2.

QUERY PALETTE SUPPORT

In Windows applications, the Palette Manager provides efficient color mapping and manipulation capabilities, particularly in video display modes with limited color depths like 256 colors.

To determine whether the device driver supports the Palette Manager, the program can use the GetDeviceCaps function with the RASTERCAPS parameter.

By performing a bitwise AND operation between the return value and the constant RC\_PALETTE, the program can check if the value is nonzero, indicating support for palette manipulation.

The GetDeviceCaps function also provides other important information related to the palette.

The call GetDeviceCaps (hdc, SIZEPALETTE) returns the total size of the palette table on the video board, typically 256 for 8 bits per pixel video display modes.

The function call GetDeviceCaps (hdc, NUMRESERVED) returns the number of colors in the palette table that the device driver reserves for system purposes, usually 20.

These reserved colors are the only pure colors that a Windows application can use without invoking the Palette Manager in a 256-color video mode.

The GetDeviceCaps (hdc, COLORRES) function call provides the resolution (in bits) of the RGB color values loaded into the hardware palette table, indicating the precision of the digital-to-analog converters (ADCs) used by the video display adapter. For example, a value of 18 indicates the usage of 6-bit ADCs, while a value of 24 indicates the usage of 8-bit ADCs.

By examining the color resolution value, a Windows program can make informed decisions about color usage.

For instance, if the color resolution is 18, requesting 128 shades of gray would be unnecessary because only 64 discrete shades of gray are possible.

Requesting more shades would lead to redundant entries in the hardware palette table.

In summary, by querying the Palette Manager support and using the GetDeviceCaps function, Windows applications can determine palette capabilities, such as support for palette manipulation, the size of the palette table, the reserved colors, and the color resolution of the hardware palette. This information assists in making optimal color choices and utilizing the available colors effectively in various video display modes.

*How does the Palette Manager handle situations where the device driver does not support palette manipulation? Can you provide an example of how the Palette Manager functions are used to work with the palette indices in GRAYS3? What are some other advantages of using palette indices instead of RGB values in Windows applications?*

When the device driver does not support palette manipulation, the Palette Manager functions gracefully adapt, and Windows continues to function without palette enhancements.

In such cases, the application essentially operates as if the Palette Manager were not utilized.

The advantages provided by the Palette Manager, such as accessing additional colors beyond the standard system palette, become relevant only in video modes that support palette management, typically those with 256 colors.

Let's delve into an example of how the Palette Manager functions are employed to work with palette indices in GRAYS3.

In this program, the logical palette is created with 65 entries during the WM\_CREATE message.

Later, during the WM\_PAINT message, the selected and realized palette is used to draw a series of rectangles representing different shades of gray.

Notably, the PALETTEINDEX macro is employed to specify the color of the solid brush when creating these rectangles.

Here's an excerpt of the relevant code from GRAYS3:



In this code, the PALETTEINDEX macro is employed to specify the color when creating a solid brush (CreateSolidBrush). The index 'i' is used to access different shades of gray from the logical palette. This method is more efficient than using RGB values, as Windows does not need to perform a nearest-color search, and it directly references the index in the logical palette.

The advantages of using palette indices over RGB values in Windows applications extend beyond efficiency. Some notable advantages include:

Reduced Memory Usage: Palette indices typically require less memory than RGB values, as they refer to predefined colors in a palette rather than storing full RGB color information.



Improved Performance: Since palette indices directly map to entries in the logical palette, color selection and rendering are faster compared to the computation involved in matching RGB values.



Compatibility: Palette indices facilitate compatibility with legacy systems and older display technologies that rely on palette-based color representation.



Consistent Color Mapping: Using palette indices ensures consistent color mapping, especially when dealing with limited color environments, leading to predictable and reliable color rendering.



In summary, the Palette Manager, when used with palette indices, offers efficiency, reduced memory usage, improved performance, and compatibility advantages in Windows applications, particularly in scenarios where palette manipulation is supported by the display hardware.

*How can a Windows program utilize the Palette Manager functions to access the remaining 236 colors in a 256-color video mode? What are some examples of decisions a Windows program can make based on the color resolution value? Can you provide more information on how the Palette Manager functions work in Windows applications?*

A Windows program can utilize the Palette Manager functions to access the remaining 236 colors in a 256-color video mode by creating and managing a logical palette. Here are the general steps involved:

Create a Logical Palette:

Allocate memory for a LOGPALETTE structure that includes information about the palette.

Set the palVersion field to 0x0300 for Windows 3.0 compatibility.

Set the palNumEntries field to the number of entries in the palette table, typically 256 for a 256-color video mode.

Populate the palPalEntry array with RGB values for each color in the palette.

Create the Palette:

Use the CreatePalette function to create a logical palette based on the information in the LOGPALETTE structure.

The function returns a handle to the logical palette.

Select and Realize the Palette:

During the WM\_PAINT message, use SelectPalette to select the logical palette into the device context.

Call RealizePalette to map the colors to the system palette, corresponding to the physical palette of the video board.

Use Palette Indices:

To access colors from the logical palette, use palette indices or the PALETTEINDEX macro instead of normal RGB values.

Palette indices range from 0 to 255 (or the total size of the palette table), allowing access to the 236 additional colors beyond the 20 reserved colors.

Examples of decisions a Windows program can make based on the color resolution value (COLORRES) obtained from GetDeviceCaps:

Optimal Color Choices:

If the color resolution is lower (e.g., 18 bits), the program might prioritize using fewer distinct colors to avoid filling the limited color space with redundant entries.

Color Depth Adjustments:

The program can dynamically adjust its use of colors based on the available color depth. For instance, it might use more colors in higher color depth scenarios.

Palette Efficiency:

In situations with lower color resolution, the program might limit the number of requested shades, optimizing the use of the available color range.

*Regarding how the Palette Manager functions work in Windows applications:*

SelectPalette Function:

Selects a logical palette into a device context.

Only one logical palette can be selected into a device context at a time.

RealizePalette Function:

Causes Windows to realize the logical palette in the device context by mapping colors to the system palette.

The actual work, including determining window activity and notifying other windows about palette changes, occurs during this function call.

Palette Messages (WM\_QUERYNEWPALETTE and WM\_PALETTECHANGED):

Windows sends these messages to main windows to manage palette changes.

Programs using the Palette Manager should process these messages to ensure proper palette realization and updating.



The system palette in Windows is directly linked to the hardware palette lookup table on the video adapter board, although the color resolution of the hardware palette may be lower than that of the system palette.



The function GetSystemPaletteEntries allows a program to retrieve individual or multiple RGB entries from the system palette, provided that the video adapter mode supports palette manipulation.

The function's parameters include the device context handle (hdc), an unsigned integer indicating the index of the first palette entry (uStart), an unsigned integer specifying the number of palette entries to retrieve (uNum), and a pointer to a PALETTEENTRY structure (&pe).

There are various ways to use the GetSystemPaletteEntries function. A program can define a single PALETTEENTRY structure and make multiple calls to the function, incrementing the index (i) from 0 to the value returned by GetDeviceCaps with the SIZEPALETTE index, typically 255.

Alternatively, a program can retrieve all system palette entries by defining a pointer to a PALETTEENTRY structure, allocating memory to accommodate the required number of PALETTEENTRY structures based on the palette size.

The function essentially allows detailed examination of the hardware palette table. The order of entries in the system palette corresponds to the increasing values of pixel bits used to represent colors in the video display buffer.

This information can be valuable for understanding the organization of colors in the palette and may be utilized for various purposes within a program.

For a more practical demonstration, let's consider an example where a program iterates through the system palette entries, gaining insights into the arrangement of colors in the hardware palette table.

In summary, the Palette Manager functions enable Windows programs to efficiently manage and utilize color palettes in scenarios where there are limitations on available colors, such as in 256-color video modes.

Programs can make informed decisions based on the color resolution value to optimize color usage and provide a better visual experience.

THE RASTER OPERATIONS PROBLEM

Understanding Raster Operations (ROPs):

Purpose: Raster operations, or ROPs, are techniques in GDI (Graphics Device Interface) for combining pixels of different objects during drawing operations.

Types:

* Binary ROPs involve two objects (e.g., line and background).
* Tertiary ROPs involve three objects (e.g., source bitmap, brush, and destination bitmap).

SetROP2 Function: Sets the drawing mode for binary ROPs.

Bitwise Operations: ROPs function by performing bitwise operations (AND, OR, XOR, etc.) on the pixel bits of the involved objects.

The Raster-Op Problem:

Root Cause: ROPs manipulate pixel bits directly, often without regard to their corresponding colors in the system palette.

Color Distortions: When the palette is changed, the relationship between pixel bits and colors can be altered, leading to unexpected color results in ROP-based drawing operations.

Example: Dragging a sizing border in GRAYS2 or GRAYS3 might produce random colors due to inverted pixel bits mapping to incorrect palette entries.

Reserved Colors and Limitations:

Reserved Colors: The system palette reserves 20 standard colors to minimize ROP issues.

Black and White Guarantee: Only black (pixel bits 0) and white (pixel bits 1) are guaranteed to behave predictably in ROPs.

Predicting Results: To anticipate ROP behavior with colors, examine the system palette table to understand the RGB values associated with various pixel bit combinations.

Key Considerations for Handling Raster-Op Issues:

Respect Reserved Colors: Avoid modifying the 20 reserved colors to maintain some ROP consistency.

Exercise Caution with Palette Changes: Be mindful of potential color distortions when altering the palette.

Prioritize Black and White: Use black and white whenever possible for reliable ROP results.

Analyze Palette for Color ROPs: Inspect the system palette table to anticipate color outcomes when using ROPs with non-black/white colors.

Consider Alternative Approaches: Explore techniques that don't rely heavily on ROPs for color-sensitive drawing operations.

Additional Insights:

Figure 16-1: (1st image) May visually illustrate the arrangement of reserved colors in the system palette.

GRAYS2 and GRAYS3 Programs: (Covered) Could be used for hands-on experimentation with ROP behavior.

Remember: Understanding the interplay between ROPs, palettes, and pixel bits is crucial for achieving predictable and visually accurate drawing results in GDI.

The raster operations in GDI, governed by functions like SetROP2 and used in drawing lines and filling areas, involve binary or tertiary operations on pixel bits.

The numeral "2" in SetROP2 signifies a binary raster operation, while tertiary raster operations are employed with functions like BitBlt.

These operations determine how the pixels of the drawn object interact with the pixels of the surface, allowing for diverse effects such as bitwise exclusive-OR combinations.

In the context of changing the palette, the raster operations become particularly nuanced.

These operations manipulate pixel bits, which may not necessarily align with actual colors.

An illustrative example can be observed by running the GRAYS2 or GRAYS3 program and dragging the top or bottom sizing border.

Windows typically uses a raster operation to invert the background pixel bits, ensuring the dragged sizing border remains visible.

However, with these programs, the result may manifest as various random colors, corresponding to unused entries in the palette table resulting from the inversion of pixel bits.

It's crucial to note that only the pixel bits are inverted, not the visible color.

Figure 16-1 demonstrates that the 20 standard reserved colors in the system palette, positioned at the top and bottom, maintain normalcy in the results of raster operations.

However, as palette changes occur, especially if reserved colors are altered, raster operations on colored objects can lose their intended meaning.

A fundamental guarantee is that raster operations will consistently work with black and white.

Black, the initial entry in the system palette with all pixel bits set to 0, and white, the final entry with all pixel bits set to 1, remain unchangeable.

To anticipate the outcomes of raster operations on colored objects, one can retrieve the system palette table and inspect the RGB color values associated with different pixel-bit configurations.

This provides a means to predict the impact of raster operations when palette alterations are in play.

SYSPAL1 PROGRAM





Purpose and Overview:

The SYSPAL1 program is designed to display the contents of the system palette in a Windows environment. The system palette, essentially a logical palette, is critical for handling colors across various programs.

The goal is to visualize the RGB values of the system palette, aiding in the debugging of palette-related applications. The program achieves this by utilizing the GetSystemPaletteEntries function, which retrieves the RGB values associated with each color entry in the system palette.

Implementation Details:

The program begins by registering a window class and creating a window. It checks whether the display mode supports a 256-color palette, as this is a prerequisite for the program's operation. The window class and main window creation are standard steps in setting up the graphical user interface.

The heart of the program lies in its Window Procedure (WndProc), where it handles various messages such as WM\_CREATE, WM\_SIZE, WM\_PAINT, WM\_PALETTECHANGED, and WM\_DESTROY.

Initialization (WM\_CREATE):

The WM\_CREATE message initializes the program, checking if the display supports the required 256-color palette. It obtains the device context (hdc) and sets up the font and character dimensions to be used in displaying the RGB values later.

Palette Checking (CheckDisplay):

The CheckDisplay function ensures that the display mode has a 256-color palette. If not, an error message is displayed, and the program exits. This check is crucial for the proper functioning of the subsequent palette-related operations.

Painting the Palette (WM\_PAINT):

The WM\_PAINT message is triggered when the window needs to be redrawn, and it is here that the system palette entries are obtained and displayed. The program uses GetSystemPaletteEntries to fetch the RGB values of all colors in the system palette. It then formats and displays these RGB values as text strings in the client area of the window.

Palette Change Handling (WM\_PALETTECHANGED):

The WM\_PALETTECHANGED message is handled to invalidate the client area when a change in the system palette occurs. This ensures that the display is updated in response to palette modifications.

Cleanup and Exit (WM\_DESTROY):

When the user closes the window, the WM\_DESTROY message is processed, leading to the cleaning up of resources and the termination of the program.

Observations and Usage:

It's important to note that SYSPAL1 specifically caters to environments with a 256-color palette. The program's execution provides a visual representation of the system palette, presenting RGB values in the client area.

This information is valuable for developers working on applications that rely on precise color handling, allowing them to debug and optimize their palette-related functionalities. The program captures the essence of the Windows graphical environment, emphasizing the significance of the system palette in managing color across diverse applications.



Compile it first, then change the properties because it won’t run the first time.

Output:



SYSPAL2 PROGRAM



Core Functionality:

Visualizing a Custom Palette: The code's primary purpose is to create a window that visually demonstrates a custom 256-color palette, specifically composed of 256 shades of gray.

Window Creation and Palette Setup:

Class Registration: The WinMain function sets the foundation by registering a window class, defining its basic appearance and behavior.

Window Instance and Error Handling: An instance of the window is then created, bearing the title "System Palette #2." The code prioritizes error handling by using the CheckDisplay function to verify if the current display mode supports a 256-color palette, an essential requirement for the intended visualization. If the display mode is incompatible, a user-friendly error message is displayed, and the program gracefully terminates.

Palette Construction: Upon successful window creation, the WM\_CREATE message handler meticulously crafts a custom logical palette:

* An allocated LOGPALETTE structure is used to store 256 palette entries.
* Each entry is filled with a different shade of gray, covering the entire spectrum from pure black (0, 0, 0) to pure white (255, 255, 255).
* The CreatePalette function is then used to create the logical palette, providing a handle for further manipulation.

Orchestrating the Window's Visual Symphony:

* When the window needs to be repainted, the WM\_PAINT message handler takes action.
* It selects the custom palette for the device context (DC) using the SelectPalette function, similar to a painter choosing their preferred colors.
* It then uses the RealizePalette function to accurately map the logical palette to the system palette, ensuring high-quality display and color accuracy.
* A nested loop is used to traverse a 16x16 grid, mimicking an artist's brushstrokes.
* For each grid position, a unique solid brush is created using the custom palette's distinct shades of gray.
* A rectangular canvas is defined within the grid to serve as the area for the brushstroke.
* Finally, the FillRect function is used to apply the chosen gray brush to the canvas, simulating an artist painting on a canvas.

Adapting to Palette Dynamics and Window Closure:

* The WM\_DISPLAYCHANGE message handler keeps track of changes in the display mode.
* It uses the CheckDisplay function to confirm support for the 256-color palette, ensuring a consistent visual experience.
* If compatibility is lost, it gracefully closes the window to maintain visual integrity.
* The WM\_PALETTECHANGED message handler ensures a harmonious color display.
* If another window modifies the system palette, it promptly initiates a repaint by calling InvalidateRect, ensuring the window adapts smoothly to the new palette.
* The WM\_DESTROY message handler takes care of resource cleanup when the window is closed.
* It uses the DeleteObject function to responsibly dispose of the custom palette, demonstrating proper memory management and system hygiene.

Key Takeaways:

The code eloquently showcases palette manipulation in Windows, empowering developers to create custom color schemes that align with their artistic vision.

It underscores the importance of gracefully handling display changes and palette updates to deliver a consistent and visually pleasing experience, akin to an artist adapting to changing lighting conditions.

The code not only reacts to outside influences like changes in the display but also takes charge of its visual appearance.

By carefully creating its own palette of 256 colors, it ensures a consistent and predictable range of colors, regardless of potential changes in the system palette.

This deliberate palette crafting allows the program to act like a careful artist, mixing and choosing its own colors to achieve a desired look.

Each pixel, filled with a specific shade of gray, becomes a subtle note in the overall visual picture. The program orchestrates the interplay of light and dark, creating a sense of depth and texture, much like a skilled artist using shading and tonal variations to bring life to their artwork.

Even though the displayed palette remains the same, the user's interaction with the window adds a dynamic element.

Resizing the window stretches and compresses the grid of colored squares, prompting the viewer to see the same shades of gray differently in various spatial contexts.



This interactive aspect encourages the user to participate in the artistic interpretation, similar to how a museum visitor engages with a painting from different angles.

The core ideas of palette manipulation showcased in the code go beyond grayscale.

By changing the rules used to define the palette entries, artists and developers can unlock a wide range of creative possibilities.

Vibrant rainbow colors, earthy tones, or dreamlike pastel combinations can be brought to life, turning the code into a versatile tool for creating unique visual experiences.

This program can be a starting point for further exploration into color manipulation.

Understanding how pixels, palettes, and code work together offers valuable insights into creating visual stories.

By experimenting with different ways of constructing palettes and adding interactive elements, aspiring artists and programmers can push the boundaries of digital art and find their own path in the ever-changing world of visual expression.

In conclusion, the presented code is more than just a display of a system palette; it reveals a small digital art studio, carefully created and adaptable, ready to be explored and used to create a beautiful combination of colors and pixels. It invites both technical minds and artistic souls to engage in a captivating dance of creativity.

Why is my output red and black?

The code you provided, SYSPAL2, creates a window that displays a grid of colored squares using a custom palette. The colors range from black to red, and the intensity of red increases as you move down the grid. This behavior is intentional and defined in the code.

Here's the relevant section of the code that determines the colors:



In this loop, the program is creating a custom palette with 256 entries. Each entry represents a color, and the red component of each color is set based on the loop index (i). The green and blue components are set to zero, resulting in shades of red.

So, as i increases from 0 to 255, the red component of the color increases, creating a gradient from black to red. This gradient is then displayed in a grid in the window, where the intensity of red becomes more pronounced as you move down the grid.

If you want to modify the color range or experiment with different gradients, you can adjust the logic inside this loop. For example, you could create a gradient that goes from black to a different color or a more complex combination of colors. Experimenting with these values will allow you to create various visual effects based on your preferences.

In the notes, the author explains a technique used in the third version of the SYSPAL program. The technique involves using two GDI functions, BitBlt and StretchBlt, to specify colors directly in pixel bits.

Here are the key points to explain:

GDI functions typically work with color specified as RGB values. The RGB values are converted into pixel bits that correspond to the desired color.

In some video modes (such as 16-bit or 24-bit color modes), the conversion from RGB values to pixel bits is straightforward. However, in other video modes (such as 4-bit or 8-bit color modes), a process called nearest-color search is performed to find the closest matching color in the limited palette.

BitBlt and StretchBlt are two GDI functions that allow specifying color directly in pixel bits. When used in this way, these functions bypass the RGB-to-pixel-bits conversion process.

The behavior of BitBlt and StretchBlt when used to directly specify pixel bits is highly device-dependent. It depends on the capabilities of the video display adapter and the specific video mode in use.

By using BitBlt or StretchBlt to directly specify pixel bits, it becomes possible to display the actual palette lookup table on the video display adapter.

This means that the colors shown on the screen correspond directly to the colors in the palette, without any intermediate color approximation.

Here's a breakdown of the additional points mentioned in the paragraph, focusing on color conversion in GDI and the nearest-color search:

Color Conversion in GDI:

As discussed above, in most cases, GDI functions require color to be specified using RGB values. These RGB values are then converted into pixel bits that represent the corresponding color on the screen.

The conversion process varies depending on the video mode and color depth. In simpler terms, in higher color-depth modes (like 24-bit), the conversion is relatively straightforward, while in lower color-depth modes (like 8-bit or 4-bit), it can be more complex.

Nearest-Color Search:

In limited color modes, where the number of available colors in the display's palette is restricted (such as 4-bit or 8-bit color modes), GDI may need to approximate colors that are not directly available in the palette.

When a color specified in RGB values is not present in the palette, GDI performs a nearest-color search to find the closest matching color available in the palette.

The nearest-color search algorithm determines the best match by comparing the RGB components of the desired color with the RGB components of the colors in the palette. The goal is to find the palette entry that most closely resembles the desired color.

It's important to note that the direct pixel access technique using BitBlt and StretchBlt bypasses the color conversion and nearest-color search process altogether.

Instead of specifying colors using RGB values, these functions allow you to directly manipulate pixel bits, which provides more control over the exact colors displayed on the screen.

However, this technique is highly device-dependent and may not be applicable or produce the desired results in all video modes or display configurations.

SYSPAL3 PROGRAM



The image you sent indeed shows the output of the SYSPAL3 program. It displays all 256 colors available in the system palette, arranged in a neat 16x16 grid. Each pixel in the grid directly corresponds to an entry in the palette, showcasing the full spectrum of colors available for manipulation in your specific video mode.

It's truly fascinating to see how SYSPAL3 bypasses the usual RGB conversion and logical palette management by directly accessing pixel bits. This technique grants deeper insights into the inner workings of the system palette and its connection to the hardware display.

The image itself is visually captivating, with the vibrant colors arranged in a uniform pattern. It serves as a testament to the program's effectiveness in revealing the capabilities of the system palette and the intricate relationship between pixel values and color representation.

Context and Motivation:

Palette Management: In Windows graphics programming, managing color palettes is crucial, especially in limited-color modes (e.g., 256 colors).

RGB Conversion: Most GDI functions specify color in RGB format, which undergoes internal conversion to pixel bits for display.

SYSPAL3's Approach: Explores a direct pixel-based method for accessing and displaying the system palette, bypassing RGB and providing deeper insights into palette mechanics.

Key Features and Steps:

Window Creation:

Establishes a standard window for displaying the palette.

Verifies the presence of a 256-color display mode, as the technique relies on this specific configuration.

Bitmap Construction (WM\_CREATE):

Generates a compact 16x16 bitmap with 8 bits per pixel, serving as a canvas for representing all 256 possible palette entries.

Each pixel value in the bitmap directly maps to an index in the system palette.

Displaying the Palette (WM\_PAINT):

Creates a memory device context, a virtual canvas for graphical operations.

Associates the bitmap with the memory device context, preparing it for display.

Employs the StretchBlt function to enlarge the bitmap to fill the window's client area, effectively showcasing all 256 colors.

During this process, Windows transfers the raw pixel bits from the bitmap to the video hardware, seamlessly accessing the corresponding colors in the palette lookup table.

Palette Synchronization:

Any modifications to the system palette, either by SYSPAL3 itself or other programs, are instantly reflected in the displayed bitmap. This eliminates the need for manual repainting or invalidation of the window's client area, ensuring a dynamic and up-to-date representation of the palette.

Implications and Considerations:

Device Dependence: The technique's reliance on direct pixel manipulation makes it highly specific to 256-color display modes and hardware configurations. It might not function as expected on systems with different color depths or display technologies.

Performance: By circumventing RGB color conversion and logical palette handling, it can potentially offer performance advantages in scenarios where direct pixel access is crucial.

Palette Exploration: SYSPAL3 serves as a valuable tool for exploring the inner workings of the system palette and understanding the relationship between pixel values and color representation in device-dependent graphics programming.

Window Initialization:

Defines a window class and creates a window titled "System Palette #3."

Checks if the display mode supports a 256-color palette through the CheckDisplay function.

Displays an error message and exits if the display mode does not support the required palette.

Bitmap Creation:

During the WM\_CREATE message, creates a 16-by-16 bitmap with 8 bits per pixel.

The bitmap serves as a representation of the system palette, with each pixel corresponding to one of the 256 possible color indices.

Displaying the Palette:

Upon receiving the WM\_PAINT message, initiates the painting process.

Creates a compatible device context (hdcMem) and selects the previously created bitmap into this memory context.

Uses the StretchBlt function to transfer the pixel bits from the bitmap to the hardware display, filling the entire client area.

Dynamic Adaptation:

Unlike SYSPAL2, does not rely on handling WM\_PALETTECHANGED messages to update its display.

Any changes to the system palette are immediately reflected in SYSPAL3's display due to the direct use of the StretchBlt function.

Ensures that the program always represents the current state of the system palette without explicit invalidation.

Program Destruction:

Upon receiving the WM\_DESTROY message, deletes the created bitmap object, freeing up system resources.

Posts a quit message to exit the application.

StretchBlt and Palette Interaction:

The StretchBlt function efficiently transfers the pixel bits from the bitmap to the display, leveraging the system's hardware capabilities.

Allows SYSPAL3 to access the 256 entries in the palette lookup table directly.

Any modifications to the system palette are instantly reflected in the displayed colors without requiring additional handling.

In summary, SYSPAL3's streamlined approach utilizes a bitmap and the StretchBlt function for efficient and dynamic representation of the 256-color system palette, eliminating the need for explicit handling of palette change messages.

PALETTE ANIMATION: A DIFFERENT BREED OF MOTION

While the title might conjure up visions of cartoon characters zipping across your screen, Windows Palette Manager offers a distinct form of animation – much subtler, yet surprisingly effective.

Moving Objects Without Moving Pixels:

Instead of displaying a rapid sequence of bitmaps, palette animation focuses on manipulating the existing visual elements on screen. Imagine drawing your scene, then strategically shifting its colors through palette adjustments.

Certain objects blend into the background, creating the illusion of movement without ever redrawing pixels. This approach boasts impressive speed, ideal for situations requiring smooth and responsive animation.

Palette Preparation – Reserving the Stage:

Creating a palette for animation demands a slight twist compared to previous examples. Each RGB color you plan to adjust during the animation must have its peFlags field set to PC\_RESERVED within the PALETTEENTRY structure.

Remember, normally setting peFlags to 0 allows GDI to merge identical colors from various logical palettes into one system palette entry.

But for animation, we want exclusive control. Setting PC\_RESERVED prevents GDI from merging, ensuring your animation's color changes don't affect other programs or disturb the system palette organization. This isolation keeps the animation swift and predictable.

Bringing the Palette to Life:

During the WM\_PAINT message, standard SelectPalette and RealizePalette calls remain in play. You continue specifying colors using the PALETTEINDEX macro, relying on its logical palette index reference.

The real magic happens in response to WM\_TIMER messages. This is where you control the animation by modifying the palette.

The AnimatePalette function comes into play, taking an array of PALETTEENTRY structures as input. It efficiently updates the necessary entries in both the system palette and the video board hardware's palette table, ensuring a smooth and responsive animation experience.

In essence, palette animation offers a unique approach to moving visuals on screen.

By strategically manipulating the colors of existing elements, you achieve animation effects without the burden of constant redrawing.

This technique shines in scenarios where speed and efficiency are paramount, adding a dynamic touch to your graphical applications.

BOUNCE PROGRAM

The provided code outlines the structure of the BOUNCE program, a Windows application designed to display a bouncing ball.

The program incorporates palette animation, an approach to animation that involves manipulating the palette to alter colors dynamically without redrawing images extensively.

The code is organized into several components, each responsible for a specific aspect of the program's functionality.

*The first portion of the program PalAnim.c*

Header Inclusions and External Declarations:

The program begins with necessary header inclusions, notably the <windows.h> header for Windows-specific functions and types.

Additionally, there are external declarations for functions such as CreateRoutine, PaintRoutine, TimerRoutine, and DestroyRoutine.

These functions handle palette creation, painting, timer-based actions, and destruction, respectively.

Window Procedure (WndProc):

The WndProc function serves as the window procedure for message handling.

It responds to various messages, including WM\_CREATE, WM\_DISPLAYCHANGE, WM\_SIZE, WM\_PAINT, WM\_TIMER, WM\_QUERYNEWPALETTE, WM\_PALETTECHANGED, and WM\_DESTROY.

Each case within this function corresponds to a specific message type and dictates the corresponding actions.

Initialization (WM\_CREATE):

During the WM\_CREATE message, the program checks the display capabilities using the CheckDisplay function.

If the display does not support a 256-color palette, an error message is displayed, and the program exits.

Otherwise, the CreateRoutine function is called to create the initial palette, and the handle to the palette is stored in the hPalette variable.

Display Change Handling (WM\_DISPLAYCHANGE):

In the event of a display change, the program checks the display capabilities again.

If the new display does not support a 256-color palette, the window is destroyed.

This ensures that the program adapts to changes in the display environment.

Window Resizing (WM\_SIZE):

The WM\_SIZE message updates the client area dimensions, storing the width and height in cxClient and cyClient variables, respectively.

Painting (WM\_PAINT):

During the WM\_PAINT message, the program begins painting by obtaining a device context (hdc).

It selects the current palette, realizes the palette, and invokes the PaintRoutine to paint the bouncing ball within the client area. Finally, the painting is concluded with EndPaint.

The subsequent parts of the code handle timer events, palette queries, palette changes, and program destruction.

These components collectively create a framework for a palette-animated bouncing ball application in a Windows environment. Let’s discuss them.

Timer Event Handling (WM\_TIMER):

In response to the WM\_TIMER message, the program retrieves the device context (hdc) for the window.

It then selects the current palette (hPalette) into the device context.

The TimerRoutine function is called, which likely contains the logic for updating the palette or performing actions related to the timer event.

After the timer-related actions are completed, the device context is released using ReleaseDC.

Palette Query (WM\_QUERYNEWPALETTE):

When the program receives a WM\_QUERYNEWPALETTE message, it checks if the palette (hPalette) is valid.

If the palette is valid, the program retrieves the device context (hdc) for the window and selects the current palette into the device context.

The palette is realized in the device context using RealizePalette.

The window is invalidated using InvalidateRect to trigger a repaint, ensuring that the new palette is reflected in the display.

The device context is then released.

Palette Change Notification (WM\_PALETTECHANGED):

Upon receiving a WM\_PALETTECHANGED message, the program checks whether the palette (hPalette) is valid and if the sender of the message is not the current window ((HWND)wParam != hwnd).

If these conditions are met, the program retrieves the device context (hdc) for the window and selects the current palette into the device context.

The palette is realized in the device context using RealizePalette.

The UpdateColors function is called, which likely contains logic for updating colors or taking actions related to palette changes.

The device context is released.

Program Destruction (WM\_DESTROY):

In response to the WM\_DESTROY message, the program initiates cleanup procedures.

The DestroyRoutine function is called, passing the window handle (hwnd) and the palette handle (hPalette).

This function likely handles any necessary cleanup, such as freeing resources associated with the palette.

Finally, the program posts a quit message using PostQuitMessage(0) to terminate the application.

*Let’s now discuss Bounce.c*

CreateRoutine: Crafting the Palette's Canvas

Palette Allocation: Allocates memory for a LOGPALETTE structure, a blueprint for defining a custom palette.

Red Hues: The Palette's Palette: Fills the palette with 34 entries, all sharing a maximum red value (255). The first entry is pure black (0, 0, 0), while the remaining 33 are shades of red (255, G, B).

Green Lights, Red Alert: Sets the peFlags field to PC\_RESERVED for all entries except the last one (index 33), ensuring exclusive control for animation purposes.

Palette Creation and Timer Trigger: Translates the defined palette into a usable HPALETTE object and sets a timer with an ID of 1, initiating the animation process.

PaintRoutine: Bringing the Ball to Life

Background Brush: Uses palette index 33 (presumably a light gray or white shade) to paint the window background, creating a canvas for the ball's movement.

33 Shades of Red: A Ball Symphony: Draws 33 ellipses (balls) arranged in a visually appealing pattern, each colored with a different shade of red from the palette.

Coordinate Calculations: Meticulously positions the balls within the window, ensuring proper placement and spacing for a visually cohesive effect.

TimerRoutine: The Engine of Animation

Ball Tracking and Direction: Maintains two static variables: bLeftToRight (a boolean indicating the ball's movement direction) and iBall (an integer tracking the active ball index).

Whitewash the Past: Sets the green and blue components of the previous ball's palette entry to 255, effectively turning it white and removing its visual presence.

Step Forward (or Back): Updates the iBall index based on the current direction, either incrementing or decrementing it to move the animation forward.

Reverse Course: If the ball reaches the end of its path (index 33 or 0), it reverses direction, creating a continuous bouncing effect.

Red Alert: Sets the green and blue components of the new ball's palette entry to 0, making it stand out in a vibrant red hue.

Animating the Palette: Employs the AnimatePalette function to seamlessly update the colors in the palette, orchestrating the visual transition between balls and producing the smooth bouncing illusion.

DestroyRoutine: Cleaning Up the Playground

Halting the Timer: Stops the timer that drives the animation, preventing unnecessary resource usage when the program exits.

Palette Disposal: Deletes the custom palette, freeing up associated memory and ensuring proper program termination.

Memory Release: Frees the allocated memory block that housed the LOGPALETTE structure, preventing memory leaks.



*Let’s discuss this code a bit more*

PaintRoutine:

* The PaintRoutine function is responsible for painting the bouncing balls on the window.
* It starts by drawing the window background using a solid brush with the palette index 33. This index was likely reserved for the background color in the CreateRoutine function.
* Next, it draws 33 balls, each represented by an ellipse.
* The color of each ball is determined by its palette index, ranging from 0 to 32.
* The balls are drawn in different positions based on their indices, creating a visually appealing pattern.
* The function uses the Ellipse function to draw the individual balls.

TimerRoutine:

* The TimerRoutine function handles the logic for animating the bouncing balls.
* It maintains a static variable bLeftToRight to keep track of the direction in which the animation is moving.
* The variable iBall represents the index of the ball currently undergoing animation.
* It updates the color of the previously animated ball to white in the palette to make it disappear.
* The bLeftToRight flag and the value of iBall are adjusted to determine the direction and next ball to animate.
* The color of the new ball is set to red in the palette, making it visible.
* Finally, the AnimatePalette function is called to update the palette and create the animation effect.

DestroyRoutine:

* The DestroyRoutine function is responsible for cleanup operations when the window is being destroyed.
* It kills the timer associated with the window using KillTimer.
* The palette is deleted using DeleteObject.
* The memory allocated for the plp structure (LOGPALETTE) is freed using free.

In summary, the PaintRoutine handles the visual representation of bouncing balls, the TimerRoutine manages the animation logic, and the DestroyRoutine takes care of cleanup during program termination. Together, these functions create a palette-animated bouncing ball demonstration in a Windows environment.

PALETTE ANIMATION FUNDAMENTALS:

Palette-Driven Animation: A technique that creates the illusion of movement by strategically manipulating colors within a palette, rather than redrawing entire images.

Key Requirements:

256-Color Video Mode: Essential for palette animation to function effectively.

Logical Palette: A custom palette designed for the animation, defining the available colors.

AnimatePalette Function: The core function that modifies entries within the logical palette, triggering visual changes.

CreateRoutine function:



PaintRoutine function:



TimerRoutine function:



Optimization Considerations:

Targeted Palette Updates: Instead of modifying all 33 palette entries, focus on the specific entries that change (the previous and new ball positions).

Using a Single PALETTEENTRY Structure: Create a single structure to hold the color information for the current ball, reducing overhead.

Palette Animation Suitability:

While a bouncing ball might not be the most efficient demonstration of palette animation, this technique excels in scenarios that involve repetitive patterns of movement.

*To troubleshoot the issue with your bouncing ball program not displaying the expected animation, follow these steps:*

Palette Animation Support:

Confirm that your display mode supports palette animation. Palette animation relies on the use of a palette, and not all display modes support palettes. Ensure that you are running the program on a system that supports palette animation.

Logical Palette Initialization:

Check the initialization of the logical palette in the CreateRoutine function. Ensure that the logical palette is correctly created, and the entries are initialized with the appropriate colors for the bouncing ball.

Timer Interval:

Verify the timer interval in the SetTimer function call. The interval determines how frequently the WM\_TIMER message is triggered, affecting the animation speed. Adjust the interval to see if it has an impact on the animation.

Palette Animation Call:

Inspect the AnimatePalette function call in the TimerRoutine. Ensure that it is correctly updating the palette entries and that the parameters passed to AnimatePalette are accurate.

Painting Routine:

Examine the PaintRoutine function. Confirm that the background is drawn using the palette index corresponding to the background color, and the balls are drawn with the correct palette indices.

Debugging Output:

Insert debug output statements or use a debugger to print or inspect variable values during runtime. This can help identify any issues with variable values or logic flow.

Error Handling:

Check for error messages or return values from functions like CreatePalette, SetTimer, and AnimatePalette. Handle any errors that may be occurring during runtime.

Memory Allocation:

Ensure that memory allocation for the LOGPALETTE structure is successful. Check for any errors or warnings related to memory allocation.

Code Review:

Review the code for typos, syntax errors, or logical errors. Pay attention to details in variable names, comparisons, and calculations.

Run with Other Programs:

As mentioned in the explanation, run the bouncing ball program alongside SYSPAL2 or SYSPAL3 to observe how the palette animation interacts with other programs.

Testing Environment:

Test the program on different Windows environments to ensure compatibility.

Operating System Compatibility:

Verify that the program is compatible with the operating system you are using. Some functionalities may behave differently across Windows versions.

By systematically checking these aspects and providing more context about your implementation, we can get to the bottom of why your bouncing ball isn't animating as expected.

FADER PROGRAM



1. Program Overview:

The FADER program, demonstrated in Figure above, is another example of palette animation. Unlike the BOUNCE program, FADER focuses on a single palette entry to create a fade-in and fade-out effect. Similar to other palette animation demos, this program requires the PALANIM.C file.

2. Palette Initialization:

In the CreateRoutine function, the program initializes a logical palette (lp) with a single palette entry. The palette entry is initially set to white (RGB values of 255 for each component), and the PC\_RESERVED flag is applied. The CreatePalette function is then called to create the palette, and a timer is set with an interval of 50 milliseconds.

3. Painting Routine:

The PaintRoutine function is responsible for drawing the text "Fade In and Out" repeatedly on the window's client area. The text is drawn using the palette index 0, which corresponds to the single entry in the logical palette.

4. Timer Routine for Fade Effect:

The TimerRoutine function manages the palette animation by altering the RGB values of the single palette entry. The program implements a fade-in and fade-out effect by incrementing and decrementing the RGB values. The AnimatePalette function is then called to update the palette dynamically.

5. Fade-In and Fade-Out Logic:

*Fade-In:*

The program checks if it's currently in the fade-in phase. If true, it decreases the red and green components of the palette entry. Once the red component reaches 3 (near black), the fade-in phase concludes, and the program switches to the fade-out phase.

*Fade-Out:*

In the fade-out phase, the red and green components are incremented until the red component reaches 255 (white). At this point, the fade-out phase concludes, and the program switches back to the fade-in phase.

6. Palette Animation Call:

The AnimatePalette function is used to apply the changes to the palette. It takes the palette handle, a range of entries to update (in this case, just one entry at index 0), and the array of PALETTEENTRY structures containing the modified RGB values.

7. Cleanup on Program Termination:

The DestroyRoutine function handles the cleanup tasks. It kills the timer, deletes the palette object, and ensures proper resource management.

The FADER program showcases a creative use of palette animation, emphasizing the simplicity of manipulating a single palette entry to achieve dynamic visual effects. If you have any specific questions or if there's an aspect you'd like to explore further, feel free to ask!

*Here are a few additional aspects we can discuss about the FADER program:*

1. Palette Animation Techniques:

Color Blending: Explore different methods of color blending for smoother transitions during the fade-in and fade-out phases.

Multiple Palette Entries: Experiment with using more than one palette entry for varied effects, such as simultaneous fading of multiple colors.

2. Enhancements and Variations:

Text Effects: Modify the PaintRoutine to implement various text effects, such as rotation, scaling, or changing font styles.

Background Changes: Introduce background changes or patterns during the animation to create a more dynamic visual experience.

3. User Interaction:

Mouse or Keyboard Interaction: Implement user interaction to control the animation, allowing users to pause, resume, or manipulate the animation speed.

4. Optimizations and Performance:

Optimizing Timer Interval: Fine-tune the timer interval to achieve a balance between smooth animation and system performance.

Memory Management: Explore ways to optimize memory usage, especially if handling a large number of palette entries.

5. Integration with Other Programs:

Combine with SYSPAL Programs: Run the FADER program alongside other palette animation programs like SYSPAL2 or SYSPAL3 to observe interactions and visual effects.

6. Dynamic Text Content:

Text Content Changes: Dynamically change the text content during the animation to display messages or create visual storytelling.

7. Debugging and Troubleshooting:

Debug Output: Implement debug output statements or use a debugger to trace the program's execution and understand how variables change over time.

Error Handling: Enhance error handling mechanisms to gracefully handle unexpected scenarios.

8. Cross-Platform Considerations:

Compatibility: Consider adapting the program for cross-platform compatibility or different Windows versions.

9. Educational Insights:

Teaching Tool: Discuss how the FADER program can serve as an educational tool for understanding palette animation concepts, RGB color model, and Windows graphics programming.

ALLCOLORS PROGRAM







Colorful Prelude:

The ALLCOLOR program, showcased in Figure 16-10, is designed to exhibit a spectrum of colors sequentially, providing a visual representation of the palette's capabilities.

Palette Initialization:

The program begins by creating a logical palette in the CreateRoutine function. It dynamically determines the color resolution of the system, setting the variable iIncr to reflect the color depth. The logical palette consists of a single entry initialized to black (0, 0, 0) and marked as PC\_RESERVED.



Displaying RGB Information:

The DisplayRGB function prints the current RGB values of the palette entry at the top left corner of the window. It utilizes the TextOut function to display the hexadecimal representation of the RGB values.



Painting Routine:

In the PaintRoutine function, the program fills the entire window with the color represented by palette index 0. This serves as the background against which the changing colors will be displayed. The DisplayRGB function is then called to show the RGB values at the top left corner.



Color Transition - TimerRoutine:

The TimerRoutine function handles the color transition. It incrementally adjusts the RGB values of the palette entry, creating a smooth transition from one color to another. The AnimatePalette function is then called to update the logical palette with the new color.



Continuous Animation:

The program sets up a timer with a 10-millisecond interval to trigger the TimerRoutine function periodically. This continuous execution results in a seamless animation of color changes.



The number:

The number displayed at the top left corner of the window in the ALLCOLOR program serves as a real-time representation of the RGB (Red, Green, Blue) values of the current color being displayed. This dynamic display is achieved through a series of intricate processes that unfold as the program executes.

The journey begins with the initialization of the logical palette in the CreateRoutine function. Here, the program gauges the color resolution of the system, determining the variable iIncr accordingly.

Subsequently, a logical palette with a single entry is created, initially set to black (0, 0, 0), and marked as PC\_RESERVED.

The responsibility of showcasing the RGB values falls on the DisplayRGB function, which employs the TextOut function to print the RGB values in hexadecimal format at the window's top left corner.

These RGB values are obtained from the PALETTEENTRY structure (pe), representing the current color in the palette.

The crux of the color transition lies in the TimerRoutine function, triggered at regular intervals by the WM\_TIMER message.

This function orchestrates a dynamic change in the RGB values, creating a fluid transition from one color to another.

The process involves incrementing or decrementing the blue, green, and red components based on boolean flags and reversing directions when reaching boundaries.

The AnimatePalette function is then summoned to update the logical palette with the recalibrated color, ensuring a smooth and continuous animation.

Simultaneously, the DisplayRGB function is called within TimerRoutine to refresh and exhibit the updated RGB values, providing a visual insight into the changing color.

The numbers displayed at the top left corner constantly change as a result of a fascinating interplay between RGB values. This mesmerizing dance is achieved through a well-planned process that involves setting up the color palette, smoothly transitioning between colors, and updating the display in real-time.

The program's intricate design ensures that the colors keep shifting, creating a captivating visual experience. This captivating display is made possible by a timer that controls the rhythm of color changes and the DisplayRGB function, which presents the evolving colors on the screen.

AllColor and Fader Routines:

The ALLCOLOR routine is structurally similar to the FADER routine.

In the CreateRoutine function, ALLCOLOR initializes a palette with a single entry that represents the color black (all RGB fields set to 0).

In the PaintRoutine function, ALLCOLOR creates a solid brush using PALETTEINDEX(0) and fills the entire client area with that brush, effectively coloring it black.

The TimerRoutine in ALLCOLOR brings animation to the palette by smoothly changing the color of the PALETTEENTRY and invoking AnimatePalette.

The color transition is achieved in a gradual manner. Initially, the blue value is incremented until it reaches its maximum value.

Then, the green value is incremented while progressively decrementing the blue value.

The increments and decrements of the red, green, and blue color values are determined by the iIncr variable, which is calculated during CreateRoutine based on the COLORRES value returned by GetDeviceCaps.

For example, if GetDeviceCaps returns 18, iIncr is set to 4, which is the lowest value necessary to achieve the full range of colors.

Additionally, ALLCOLOR displays the current RGB color value in the upper left corner of the client area. Initially added for testing purposes, this feature proved to be useful and was kept in the final implementation.

UNVEILING THE DYNAMICS OF PIPES: A PALETTE ANIMATION ODYSSEY

In the realm of engineering applications, the PIPES program stands as a testament to the power of animation in elucidating complex processes.

This palette animation demonstration, created by Charles Petzold in 1998, ingeniously visualizes fluid flow through pipes, transforming a static representation into a dynamic, easily comprehensible showcase.



1. Palette Initialization:

The heart of PIPES lies in its palette initialization process, a symphony of colors orchestrated to represent fluid movement. The CreateRoutine function dynamically allocates memory for a LOGPALETTE structure, housing 16 palette entries. These entries are strategically designed to convey the fluid's progression through the pipes. The color variations, ranging from dark to light, create a visually intuitive representation.

2. Painting the Canvas:

The PaintRoutine function serves as the artistic brushstroke, painting the canvas of the client area. It meticulously draws the background and interiors of two horizontal pipes. The color gradient within each pipe, determined by the palette entries, gives the illusion of fluid moving in opposite directions.

3. Crafting the Animation:

As the PIPES program comes to life, the TimerRoutine function takes center stage. Animated palette manipulation becomes the maestro, smoothly transitioning between palette entries. This dynamic shift in colors simulates the continuous flow of fluid through the pipes. The rhythmic updating of the palette captures the essence of motion, making the demonstration both informative and engaging.

4. Palette Dance Continues:

With each tick of the timer, AnimatePalette elegantly animates the color transitions. The fluid's movement is not just visualized but felt through the carefully selected palette entries. The cyclical nature of the palette updates ensures a seamless and captivating animation, making the program a compelling tool for conveying dynamic engineering processes.

5. A Palette Symphony:

In essence, PIPES is a symphony of palette entries, where each note represents a moment in the fluid's journey. The combination of color, motion, and symbolism creates a powerful educational tool. Unlike a static representation, this palette animation provides clarity and insight, turning the complexities of fluid dynamics into a visually digestible masterpiece.

6. The Grand Finale:

As the program gracefully concludes, the DestroyRoutine function takes its bow. Timers are halted, palettes are bid farewell, and memory is gracefully released. PIPES, in its elegant simplicity, leaves behind a legacy as a testament to how animation can elevate the understanding of intricate engineering processes.

The PIPES program showcases the versatility of palette animation for illustrating dynamic processes, specifically the flow of fluids in this case. This program employs a two-pronged approach: creating a visually appealing pipe layout and simulating fluid movement through animation.

Building the Pipes:

* Palette Palette: PIPES utilizes 16 palette entries, though fewer might suffice. These entries form a gradient, transitioning from red-green at the center to pure red and pure green at the edges. This color scheme symbolizes the direction of the fluid flow in each pipe.
* Dual Pipes, Different Directions: The client area features two horizontal pipes positioned above and below each other. Fluid in the top pipe flows from left to right (red to green), while the bottom pipe exhibits the opposite flow (green to red).

Animating the Flow:

* Palette Manipulation: The animation hinges on modifying the 16 palette entries with each timer tick. The TimerRoutine function increments an index variable that specifies the starting point for modifying the palette. This essentially shifts the gradient, creating the illusion of fluid movement.
* Visual Cues: To further depict the flow, the program draws rectangle segments within each pipe using specific palette indices based on the animation stage. This creates a "wave-like" effect, reinforcing the perception of fluid motion.

Beyond PIPES:

While PIPES demonstrates palette animation for simulating fluid flow, the technique's potential extends beyond this specific application. Engineering can benefit from palette animation in numerous ways, such as:

* Visualizing heat transfer: Gradual palette transitions can portray the spread of heat across a surface.
* Depicting stress distribution: Shifting color palettes can indicate areas of high and low stress in a mechanical structure.
* Illustrating electrical currents: Dynamically changing colors can represent the flow of electricity through a circuit.

By leveraging palette animation in conjunction with carefully chosen color schemes and visual elements, engineers can effectively communicate complex processes in a clear and intuitive manner.

TUNNEL PROGRAM



The TUNNEL program is a palette animation demo developed by Charles Petzold in 1998. The purpose of this program is to create a visually appealing effect that simulates traveling through a tunnel using 64 moving gray shades within the 128 palette entries. The program utilizes the Windows API and C programming language to achieve this animation.

Palette Initialization:

The program begins by initializing a palette to be used for animation. It allocates memory for a LOGPALETTE structure and sets its version and the number of palette entries. In this case, 128 palette entries are used. The program then iterates through these entries, setting the red, green, and blue components of each entry to create a gradient of gray shades. The peFlags field is set to PC\_RESERVED, indicating that the color is reserved and should not be modified.

Animation Logic:

The heart of the animation lies in the TimerRoutine function, which is called periodically using a timer. This function increments the color level (iLevel) in a cyclic manner, creating the effect of smoothly transitioning through the 128 palette entries. The AnimatePalette function is then called to update the display with the new palette entries, creating the illusion of movement through the tunnel.

Painting the Tunnel:

The PaintRoutine function is responsible for painting the tunnel on the window. It uses a series of rectangles, each filled with a solid brush of a specific palette index. The palette index determines the color of the rectangle based on the current state of the palette entries. By varying the position and size of these rectangles, the program achieves the visual representation of a tunnel.

Cleanup:

Finally, the DestroyRoutine function is called when the program is exiting. It deallocates resources, including stopping the timer, deleting the palette, and freeing the memory allocated for the LOGPALETTE structure.

In summary, the TUNNEL program is a creative demonstration of palette animation, utilizing a carefully crafted palette and timer-based color transitions to create the immersive illusion of traveling through a tunnel. The use of gray shades and the cyclic animation give the program a visually captivating effect.

PACKED DIBS AND PALETTE POWER: DISPLAYING REAL-WORLD IMAGES

The next chapter of our palette exploration dives into using them with "real-world" images under 8-bit video modes. This involves delving into the world of packed DIBs, a special type of memory block holding both image data and color information. To assist us, we'll utilize functions defined in the PACKEDIB files, specifically focusing on three key aspects:

On page 685 of the book, there is the packedDIB.c and .h files which are going to be custom creations to be used to customize the upcoming programs.

1. Extracting Palette Information:

These functions help us navigate the packed DIB structure and extract crucial details about the palette it uses. This includes:

* PackedDibGetColorsUsed: Retrieves the number of colors actually used in the image, which might be less than the total entries in the color table.
* PackedDibGetNumColors: Determines the actual number of entries in the color table.
* PackedDibGetColorTablePtr: Locates the memory address of the color table within the packed DIB.
* PackedDibGetColorTableEntry: Fetches a specific color entry from the table using its index.

2. Building a Logical Palette:

With the gathered information, we can construct a logical palette that Windows understands. The PackedDibCreatePalette function performs this magic, taking the packed DIB's color table and generating a corresponding logical palette object. This object encapsulates the color mapping information needed to properly display the image.

3. Leveraging the Palette for Display:

The created logical palette can then be used in conjunction with GDI functions to render the image on the screen. This allows our program to display real-world images stored as packed DIBs with their intended colors, despite the limitations of an 8-bit video mode.

Beyond the Code:

The PACKEDIB functions and palette manipulation techniques showcased here offer a powerful toolkit for working with image data in 8-bit environments. This opens doors to various applications, such as:

* Displaying resource images embedded within programs.
* Loading and showing external image files in custom formats.
* Implementing basic image viewers or editors with limited color capabilities.

In summary, these code snippets demonstrate a comprehensive set of functions to handle packed DIBs and palettes. The provided functionalities are essential for managing color information and creating logical palettes when dealing with real-world images in scenarios with limited color depth, such as 8-bit video modes. This groundwork is crucial for subsequent programs that involve displaying real-world images under such constraints.

*KeyNotes:*

1. Function Ordering:

Petzold organizes the functions in a "bottom-up" order, where each function relies on information obtained from earlier functions. This approach ensures that the functions are used in a logical sequence, with dependencies handled appropriately. For example, information obtained about the packed DIB's size and structure in earlier functions is crucial for subsequent functions to operate correctly.

2. Limitations of the Provided Functions:

Petzold acknowledges that the set of functions provided is not intended to be exhaustive for working with packed DIBs. He highlights that he hasn't included functions for certain operations, such as retrieving individual pixels (PackedDibGetPixel), due to their potential inefficiency and slowness resulting from nested function calls.

3. Concerns about Function Efficiency:

Petzold raises concerns about the efficiency of certain operations, indicating that functions like PackedDibGetPixel would involve numerous nested calls and could be inefficient and slow. This hints at the potential performance drawbacks of certain approaches to working with packed DIBs.

4. Alternative Approaches:

Petzold hints at a better approach to working with packed DIBs, which he promises to describe later in the chapter. This suggests that there might be alternative and more efficient methods for certain operations, and the reader can expect further insights or recommendations as they progress through the chapter.

5. Handling OS/2-Compatible DIBs:

The note mentions that many functions require different processing for OS/2-compatible DIBs. The functions frequently check the size of the first field in the BITMAPINFO structure to determine whether it corresponds to the size of the BITMAPCOREHEADER structure. This highlights the need for conditional processing based on the type of DIB being handled.

6. PackedDibCreatePalette Function:

The final function, PackedDibCreatePalette, is highlighted as particularly interesting. This function creates a palette using the color table in the DIB. Importantly, if the DIB lacks a color table (as is the case for DIBs with 16, 24, or 32 bits per pixel), no palette is created. The palette created from the DIB color table is referred to as the DIB's native palette.

In summary, these notes provide context and guidance for using the provided functions to work with packed DIBs. On page 685 of the book, the PACKEDIB.C and .H files have been introduced, containing custom code written by Charles Petzold. These custom functions are designed to handle packed DIBs (Device Independent Bitmaps) and are intended for customization in upcoming programs.

Therefore, what we have created are custom functions for managing packed DIBs and generating logical palettes to be utilized in subsequent programs. They address limitations, efficiency concerns, and foreshadow alternative approaches to enhance the handling of packed DIBs in the subsequent parts of the chapter.

*Let’s move on to the first program showdib3 which will use these function and header file packedDIB.c and .h … I downloaded .bmp files and then loaded them and also must have is the afresx.h header file.*

SHOWDIB3 PROGRAM

**

The SHOWDIB3 program, as outlined in the provided code snippets, is designed to display a Device Independent Bitmap (DIB) with its native palette in a Windows graphical user interface.

The program begins by defining a window procedure (WndProc) and necessary variables, including a pointer to a packed DIB (pPackedDib), a handle to the palette (hPalette), and information about the client area (cxClient and cyClient). Additionally, the program uses the OPENFILENAME structure to handle file opening dialogues.

Upon initialization, the window class is registered, and a window is created with the title "Show DIB #3: Native Palette." The main functionality of the program involves loading a DIB from a file, creating a logical palette from its color table, and displaying the DIB using the native palette.

When the user selects "Open" from the File menu, the program triggers the File Open dialog, allowing the user to choose a bitmap file. The selected file is then loaded as a packed DIB into memory using the custom PackedDibLoad function. Subsequently, the program creates a logical palette from the DIB's color table using the PackedDibCreatePalette function.

The painting process involves setting up the device context, selecting the palette, and using SetDIBitsToDevice to display the DIB. If a palette is associated with the DIB, the program ensures proper palette realization and updates the colors. The program handles palette changes using messages like WM\_QUERYNEWPALETTE and WM\_PALETTECHANGED.

In the SHOWDIB3 program, the main thing to watch out for is the handling of logical palettes associated with the loaded DIB. The program goes beyond simply loading and displaying a DIB; it takes into account the creation and utilization of a logical palette. Here are key points to note:

Palette Handling:

The program creates a logical palette (hPalette) from the color table of the loaded DIB using the PackedDibCreatePalette function. This step is crucial for accurate color representation, especially when dealing with images that have a limited color palette.

Palette realization is managed through messages such as WM\_QUERYNEWPALETTE and WM\_PALETTECHANGED. This ensures that the program responds appropriately when the system palette changes.

Palette-Dependent Display:

During painting (WM\_PAINT), the program selects the logical palette into the device context and uses RealizePalette to ensure proper color mapping. This step is essential for displaying the DIB with its native palette, preserving the intended colors of the image.

Dynamic Palette Update:

The program dynamically updates the palette in response to changes using the WM\_PALETTECHANGED message. When the program loses focus and regains it, the palette is re-realized to reflect any modifications in the system palette.

Difference from Normal DIB Handling:

Unlike a normal program that loads DIBs without considering color palettes, SHOWDIB3 pays special attention to palette creation and utilization. This is particularly important for images with a color table, as it ensures that the displayed colors closely match the original ones.

In summary, the unique aspect of SHOWDIB3 is its focus on handling logical palettes associated with DIBs. It goes beyond the basics of loading and displaying images, incorporating palette management to provide accurate and vibrant color representation. This is especially relevant when dealing with images that rely on a limited set of colors.

*Let’s explain this a bit more…*

1. Native Palette Focus:

*SHOWDIB3 prioritizes displaying the image using its original palette embedded in the DIB file. This ensures accurate color representation, unlike a standard program that might rely on the default system palette.*

1. Logical Palette Creation:

*It actively creates a logical palette based on the DIB's color table. This mapping allows the program to translate the image's colors to the available hardware palette on the system, minimizing color errors.*

1. Palette Management:

*SHOWDIB3 properly handles the logical palette throughout its execution. It selects and realizes the palette when necessary, responds to system palette changes, and updates colors accordingly. This ensures color consistency across different events.*

1. Packed DIBs:

*It likely utilizes functions specific to "packed DIBs," a special format for storing both image data and color information efficiently. This might differ from programs dealing with standard uncompressed DIBs.*

1. Memory Management:

*SHOWDIB3 carefully frees memory allocated for the DIB data and the logical palette when they're no longer needed. This helps prevent memory leaks, a common issue in image-handling programs.*

In essence, SHOWDIB3 goes beyond simply displaying a DIB. It focuses on accurately representing the image's colors by actively managing its native palette and ensuring a smooth integration with the system's color capabilities.

The SHOWDIB3 program pays close attention to memory management to ensure efficient resource usage. The window procedure carefully handles memory allocation and deallocation, preventing memory leaks and conflicts. It stores the packed DIB pointer and palette as static variables, ensuring their accessibility throughout the program's execution.

To accurately represent colors, SHOWDIB3 gives priority to using the DIB's native palette. When opening a file, it creates a logical palette from the DIB's color table using the PackedDibCreatePalette function. During the WM\_PAINT message, it selects and realizes the palette before drawing the image using SetDIBitsToDevice. This ensures proper color mapping and faithful color reproduction.

For DIBs without color tables (such as 16-bit, 24-bit, and 32-bit DIBs), SHOWDIB3 recognizes that displaying them without a palette in 8-bit video modes would result in a limited color range, typically limited to the standard 20 colors.

To address this limitation, the program offers two potential solutions. One approach is to create an all-purpose palette that can accommodate a wide variety of images. Although relatively simple, this method may not provide the highest level of color accuracy for specific images.

The other approach involves dynamic color analysis, where the pixel data within the DIB is analyzed to determine the most suitable colors for display. This technique requires more computational effort but can potentially offer superior color fidelity.

The choice between the two approaches depends on the specific requirements of the application and the desired level of color accuracy.

For general-purpose image viewing, an all-purpose palette might be sufficient.

However, for professional image editing or tasks that demand precise color representation, dynamic color analysis could be the preferred option.

The author hints at the possibility of exploring the dynamic color analysis technique in a later chapter, suggesting a more advanced approach to handling DIBs without color tables.

SHOWDIB4 PROGRAM

The SHOWDIB4 program introduces an innovative approach to palette management compared to its predecessor, SHOWDIB3. The primary focus is on constructing an "all-purpose" palette that it utilizes for displaying all DIBs loaded into the program. Let's delve into the notable features and changes in depth:



All-Purpose Palette Creation:

SHOWDIB4 introduces the function CreateAllPurposePalette that dynamically generates an all-purpose palette suitable for a wide variety of images. This palette consists of 247 entries, carefully chosen to cover a range of colors. The palette creation process is meticulously designed to avoid duplicating or conflicting with the standard 20 colors.

Palette Handling Enhancement:

Unlike SHOWDIB3, which relies on creating a palette specific to each DIB, SHOWDIB4 takes a more generic approach by using a single, versatile palette for all images. This is a departure from the native palette focus in SHOWDIB3. The palette created in CreateAllPurposePalette covers a diverse spectrum of colors, ensuring compatibility with various image types.

Simplified Palette Management:

With the adoption of an all-purpose palette, the program simplifies palette management. There's no need to create and update individual palettes for each loaded DIB. This streamlining enhances efficiency and reduces the complexity associated with palette realization and handling.

Unified Color Representation:

SHOWDIB4 aims for a consistent and unified color representation across different images. By using the all-purpose palette, it ensures that the displayed colors maintain a certain level of consistency, regardless of the specific color characteristics of each loaded DIB.

Palette Initialization:

The creation of the all-purpose palette occurs during the WM\_CREATE message handling. This ensures that the palette is ready for use as soon as the program starts, and it remains constant throughout its execution.

Menu and UI Similarities:

The structure of the program, including menu options and the user interface, remains largely similar to SHOWDIB3. This continuity allows users familiar with the previous version to adapt seamlessly to the new palette management strategy.

In essence, SHOWDIB4 distinguishes itself by adopting a unified approach to palette management with the creation of an all-purpose palette. This shift simplifies the handling of color information, making the program more versatile and suitable for displaying a diverse range of images while maintaining a consistent and well-defined color palette.

In the process of handling the WM\_CREATE message, SHOWDIB4 efficiently manages its palette by calling the CreateAllPurposePalette function. This palette remains persistently available throughout the program's execution and is appropriately destroyed when handling the WM\_DESTROY message.

The assurance of the palette's continuous existence eliminates the need for constant checks during the processing of WM\_PAINT, WM\_QUERYNEWPALETTE, or WM\_PALETTECHANGED messages.

The CreateAllPurposePalette function is noteworthy for generating a logical palette with 247 entries, surpassing the typical 236 entries in the system palette. Despite this excess, 15 entries are intentionally duplicated or mapped to colors within the standard 20 reserved colors, providing a buffer for flexibility and convenience.

The palette creation process begins by establishing 31 gray shades, strategically aligning with certain entries in the standard 20 reserved colors. Subsequently, the function generates colors with various combinations of red, green, and blue values, resulting in a total of 216 colors.

While eight of these duplicate standard 20 colors and another four replicate previously calculated gray shades, Windows intelligently avoids placing duplicate entries in the system palette if the peFlags field of the PALETTEENTRY structure is set to 0.

It's important to note that, for practical applications not concerned with optimizing palettes for 16-bit, 24-bit, or 32-bit DIBs, utilizing the DIB color table for displaying 8-bit DIBs might be a preferred approach.

However, SHOWDIB4 deviates from this convention, using its all-purpose palette for all image types. This deliberate choice aligns with SHOWDIB4's role as a demonstration program, allowing users to compare its display of 8-bit DIBs with that of SHOWDIB3.

As an interesting experiment, modifying the CreateAllPurposePalette function in SHOWDIB4, such as by reducing the logical palette size, reveals that, when a palette is selected into a device context, Windows exclusively utilizes the colors from the selected palette, excluding any colors from the standard 20-color palette. This behavior underscores the level of control and precision achievable in color rendering when manipulating logical palettes in Windows programming.

SHOWDIB5 PROGRAM

SHOWDIB5 introduces the utilization of the halftone palette provided by the Windows API, showcasing its application in conjunction with the bitmap stretching mode known as HALFTONE. Let's delve into the notable aspects of this program:

Halftone Palette Integration:

In the WM\_CREATE message processing, SHOWDIB5 stands out by creating the halftone palette using the CreateHalftonePalette function. This palette is designed to be versatile and adaptable for various images. The program takes advantage of this palette throughout its execution.

Dynamic Stretching with HALFTONE Mode:

One of the key differences in SHOWDIB5 lies in its approach to displaying images. Instead of using the traditional SetDIBitsToDevice method seen in previous examples, SHOWDIB5 employs the StretchDIBits function. This allows for dynamic stretching of the image, offering better control over the scaling process.

Halftone Stretch Mode Setup:

Prior to performing the stretching operation, SHOWDIB5 sets the stretching mode to HALFTONE using SetStretchBltMode. This mode enhances the quality of the stretched image by employing a halftone pattern. The SetBrushOrgEx function is also used to set the brush origin to (0,0), ensuring proper alignment during the stretching process.

Palette Handling in Stretching:

During the WM\_PAINT message processing, the program selects and realizes the halftone palette before executing the StretchDIBits function. This palette management ensures that the stretched image maintains a high-quality appearance, with colors accurately represented.

Comparison with SHOWDIB4:

SHOWDIB5's use of the halftone palette showcases an alternative approach compared to SHOWDIB4, which relies on a custom all-purpose palette. The choice between these palettes can impact how images are displayed, especially in scenarios where dynamic stretching is involved.

Palette Update on Events:

Similar to previous examples, SHOWDIB5 responds to palette-related messages, such as WM\_QUERYNEWPALETTE and WM\_PALETTECHANGED. These messages trigger palette updates, ensuring consistent and accurate color representation, especially when the program loses or gains focus.

The SHOWDIB5 program, akin to SHOWDIB4, diverges from utilizing the color table within the DIB, opting instead for a palette suitable for a broad spectrum of images.

However, in contrast to the custom all-purpose palette of SHOWDIB4, SHOWDIB5 leverages the logical palette provided by Windows, acquired through the CreateHalftonePalette function.

The halftone palette generated by CreateHalftonePalette, while not significantly more intricate than the one crafted by CreateAllPurposePalette in SHOWDIB4, proves to be a pivotal component when coupled with specific operations.

Notably, when the program sets the stretching mode using SetStretchBltMode to HALFTONE and adjusts the brush origin with SetBrushOrgEx before displaying the DIB through StretchDIBits, the outcomes surpass expectations.

This strategic combination of operations, particularly the application of the HALFTONE stretching mode, yields remarkable improvements in color accuracy, especially for flesh tones.

The halftone palette, when employed in conjunction with the specified stretching mode, introduces a form of dithering pattern. This pattern, derived from the colors of the halftone palette, enhances the approximation of the original image's colors on 8-bit video boards. The result is a more faithful representation of subtle color variations, particularly crucial in capturing the nuances of skin tones.

However, it's crucial to acknowledge that this enhanced color fidelity comes at a cost—increased processing time. The utilization of a dithering pattern for improved color approximation involves additional computational overhead.

This trade-off highlights the balance between achieving optimal visual results and considering the efficiency of image processing, a common consideration in graphics programming.

In essence, SHOWDIB5 showcases the significance of thoughtful palette management and strategic utilization of Windows-provided palettes in conjunction with advanced stretching modes.

The program's ability to enhance color accuracy, particularly for challenging elements like flesh tones, exemplifies the nuanced decisions involved in achieving optimal visual rendering in graphics programming.

It demonstrates the integration of the halftone palette for enhanced image stretching, providing a different perspective on color rendering and image quality compared to previous examples.

The dynamic nature of stretching, coupled with thoughtful palette management, makes this program a valuable exploration of advanced techniques in DIB handling.

SHOWDIB6 PROGRAM

SHOWDIB6 is a Windows program that exemplifies the utilization of palette indices when displaying an 8-bit Device-Independent Bitmap (DIB). It employs the DIB\_PAL\_COLORS flag in functions like SetDIBitsToDevice and takes advantage of palette indices, offering a performance boost by sidestepping the need for a nearest-color search.

Palette Indices and DIB\_PAL\_COLORS:

The DIB\_PAL\_COLORS flag allows the color table in the BITMAPINFO structure to be interpreted as 16-bit indices into a logical palette, rather than RGB color values.

This logical palette is the one currently selected in the device context given as the first argument to the function. In the case of SHOWDIB6, this logical palette is created from the DIB's color table.

When DIB\_PAL\_COLORS is used, the color table in the DIB is assumed to contain indices pointing to entries in the logical palette.

This eliminates the need for Windows to perform a nearest-color search, which is typically required when displaying images in an 8-bit video mode. Instead, the program can directly map DIB pixel values to device pixels using the palette indices.

Program Workflow:

Loading and Displaying DIB:

* The program prompts the user to open a bitmap file (BMP).
* It loads the DIB into memory and creates a logical palette from the DIB color table.
* The program then replaces the RGB color values in the DIB color table with WORD indices starting from 0.

Displaying DIB with Palette Indices:

* During painting (WM\_PAINT), the logical palette is selected into the device context.
* The DIB is then displayed using SetDIBitsToDevice with the DIB\_PAL\_COLORS flag.
* This process takes advantage of the palette indices, avoiding the need for a nearest-color search.

Palette Management:

* The program handles palette-related messages such as WM\_QUERYNEWPALETTE and WM\_PALETTECHANGED to ensure proper palette realization and updates.

Optimization and Considerations:

* Performance Improvement: The use of DIB\_PAL\_COLORS provides a performance improvement, especially when displaying 8-bit DIBs in an 8-bit video mode. By utilizing palette indices, the program bypasses the nearest-color search, potentially resulting in faster image rendering.
* Limitation with Palette Creation: This technique is feasible when creating a palette directly from the color table of the DIB. If an all-purpose palette were used, a manual nearest-color search would be necessary, making the approach less practical.
* Caution on Saving and Clipboard Usage: If palette indices are employed, it's crucial to replace the color table in the DIB before saving it to disk. Additionally, DIBs containing palette indices should be handled carefully, and it's recommended to restore RGB color values before saving to ensure compatibility.

Importance of DIB\_PAL\_COLORS:

* It signifies to Windows that the color table contains palette indices, not RGB values.
* This optimizes performance by eliminating nearest-color searches, as Windows directly maps indices to device colors.

Key Considerations and Best Practices:

* Use palette indices primarily when working with DIBs that have limited color palettes (e.g., 8-bit DIBs).
* Create the palette directly from the DIB's color table to avoid manual nearest-color searches.
* Revert to RGB color values before saving DIBs to disk or placing them on the clipboard to ensure compatibility.
* Consider using palette indices only for display purposes and reverting to RGB colors afterward for consistency.

Additional Insights from Feedback:

* Provide a visual representation of the DIB structure and palette concept for better understanding.
* Discuss potential drawbacks of using palette indices, such as reduced color accuracy in certain scenarios.
* Explore alternative color management techniques, such as halftoning, for handling images with large color palettes.
* Offer guidance on choosing between RGB values and palette indices based on image characteristics and display requirements.

SHOWDIB7 PROGRAM

SHOWDIB7, much like SHOWDIB6, is a Windows program that focuses on the utilization of palettes in conjunction with Device-Independent Bitmaps (DIBs). However, SHOWDIB7 introduces a new approach by demonstrating how to convert a DIB into a Device-Dependent Bitmap (DDB) using the CreateDIBitmap function. This alteration results in a change in the method of displaying the image.

Conversion to DDB:

In SHOWDIB7, the primary deviation from SHOWDIB6 lies in the conversion of the DIB to a DDB. Instead of directly using SetDIBitsToDevice, SHOWDIB7 employs CreateDIBitmap to create a DDB from the DIB. This transition allows for more flexibility in handling and displaying the bitmap.

Bitmap Conversion Process:

Loading and Displaying:

* The user opens a bitmap file, similar to SHOWDIB6.
* The program loads the DIB into memory and creates a logical palette from the DIB's color table.
* It then creates a DDB using CreateDIBitmap, initializing it with information from the packed DIB.

Palette Management:

* The logical palette is created and selected into the device context, as in SHOWDIB6.
* Palette-related messages are handled for proper realization and updates.

Displaying the DDB:

* During painting (WM\_PAINT), the DDB is displayed using BitBlt.
* The DDB is compatible with the device context, allowing efficient copying to the screen.

Advantages of DDB Conversion:

* Flexibility in Displaying: Creating a DDB provides greater flexibility in displaying the image. BitBlt allows for efficient copying of the DDB to the screen, and it opens up possibilities for additional manipulations.
* Compatibility: DDBs are compatible with standard GDI functions, offering a seamless integration into the Windows graphical environment.
* Resource Management: The DIB memory can be freed after creating the DDB, potentially optimizing resource usage.

Changes from SHOWDIB6:

The transition from SetDIBitsToDevice to CreateDIBitmap represents a shift in strategy. Instead of directly rendering the DIB on the screen, SHOWDIB7 takes an intermediary step by creating a DDB. This not only provides more options for handling and displaying the image but also aligns with the standard GDI practices for working with bitmaps.

Nearest-Color Searches:

SHOWDIB6 replaces DIB colors with palette indices to optimize nearest-color searches.

SHOWDIB7 uses DIB\_RGB\_COLORS when creating the DDB, indicating that colors are already in RGB format, eliminating the need for index-based optimization.

Key Considerations:

* Converting to a DDB can improve performance for certain operations but might not be necessary in all cases.
* Handling palettes appropriately is crucial for accurate color display, especially when working with DDBs.
* Memory management is important to avoid leaks and ensure efficient resource usage.
* The choice between using palette indices or RGB colors depends on the specific requirements of the application and the target display device.

Key Points Not Yet Discussed:

Importance of Palette Selection and Realization:

* Selecting and realizing the palette before creating the DDB ensures accurate color mapping.
* This is crucial because DDBs directly use device colors, and the palette dictates how DIB colors are translated.
* Failing to do so could result in incorrect or limited colors in the resulting DDB.

Memory Management Considerations:

* DIBs can be freed after DDB creation, while DDBs must be retained for later display.
* This difference stems from DIBs being standalone data structures, while DDBs are GDI objects managed by Windows.

Clipboard Handling Best Practices:

* Copying DIBs to the clipboard is generally preferred for compatibility and flexibility.
* If DDB copying is necessary, selecting and realizing the palette ensures proper conversion to a DIB based on the current system palette.

Understanding GetObject:

* This function retrieves information about a GDI object, including its width and height in this case.
* It's essential for determining the size of the bitmap when performing operations like BitBlt.

Additional Insights:

*Trade-offs Between DIBs and DDBs:*

* DIBs offer portability and flexibility, while DDBs can be faster for certain operations and compatible with DDB-specific functions.
* The choice depends on application requirements and performance needs.

*Palette Implications for Color Accuracy:*

* Using the correct palette throughout DIB loading, DDB creation, and display is critical for ensuring accurate color representation.
* This is especially important when working with limited-color devices or images with custom palettes.

Importance of BitBlt for DDB Display:

This function efficiently copies bitmaps between device contexts, essential for displaying DDBs on the screen or other devices.

Summary:

SHOWDIB7 demonstrates DIB-to-DDB conversion, palette handling for accurate color mapping, memory management considerations for different bitmap types, and best practices for clipboard interactions.

Understanding these concepts is vital for working effectively with bitmaps in Windows programming contexts.

SHOWDIB8 PROGRAM

DIB Sections: Unlocking Performance and Direct Pixel Access:

Memory-Mapped Bitmaps: DIB sections are distinctive in their memory management. They're memory-mapped GDI bitmap objects, meaning their pixel data resides in a specified memory region, providing direct access for efficient manipulations. This eliminates the need for intermediate buffers or data copying, potentially boosting performance.

Creation and Handling: SHOWDIB8 employs CreateDIBSection to generate a DIB section from the loaded DIB. It obtains a pointer to the pixel data using pBits and directly accesses it for operations like copying. This contrasts with packed DIBs, where memory management is manual, and DDBs, where pixel data is managed internally by Windows.

Palette Management: While DIB sections offer direct pixel access, they still require appropriate palette handling to ensure accurate color representation. SHOWDIB8 demonstrates this by selecting and realizing the palette associated with the DIB, similar to previous programs.

Key Considerations When Choosing Bitmap Representations:

Performance vs. Compatibility: DIB sections often excel in performance for operations involving frequent pixel access, such as image processing tasks. However, packed DIBs generally offer wider compatibility across different platforms and file formats. DDBs, while not directly addressed in SHOWDIB8, can be advantageous for certain device-specific operations.

Clipboard Operations: When working with the clipboard, packed DIBs are generally favored due to their broader compatibility with other applications.

Unlocking the Potential of DIB Sections:

Image Processing Optimizations: DIB sections enable direct pixel manipulation without intermediate copies, potentially accelerating image processing tasks like filtering, transformations, and adjustments.

Custom Memory Management: For specialized memory needs or optimization strategies, DIB sections offer greater flexibility in memory management compared to packed DIBs or DDBs.

*Let’s discuss this program in-depth:*

SHOWDIB8 is a Windows program that demonstrates the usage of palettes with Device-Independent Bitmap (DIB) sections. It explores the conversion of a DIB to a DIB section, highlighting the benefits of this approach in handling bitmap data. Let's delve into the key aspects of SHOWDIB8:

DIB to DIB Section Conversion:

File Loading:

* Similar to previous programs, the user can open a bitmap file using the File Open dialog.
* The packed DIB is loaded into memory, and the logical palette is created from the DIB's color table.

DIB Section Creation:

* A DIB section is created using the CreateDIBSection function.
* The function returns a handle to the newly created DIB section, and a pointer (pBits) to the memory location where the DIB bits are stored.

Bit Copying:

* The bits from the packed DIB are copied to the DIB section using CopyMemory.
* This step ensures that the DIB section contains the pixel data from the original DIB.

Palette Management:

* A logical palette is created from the original DIB.
* The palette is associated with the device context, allowing for proper color realization during painting.

Displaying the DIB Section:

* During the painting process (WM\_PAINT), the DIB section is displayed on the window using BitBlt.
* Palette management ensures that colors are accurately represented.

Advantages of DIB Section:

* Direct Memory Access: DIB sections provide direct access to the bitmap's pixel data through a pointer (pBits), facilitating efficient manipulation and updates.
* Palette Compatibility: The logical palette associated with the DIB section ensures proper color mapping and representation.

Changes from Previous Versions:

DIB Section Usage: Unlike SHOWDIB6 and SHOWDIB7, which focused on DDBs, SHOWDIB8 specifically emphasizes DIB sections. This choice is driven by the benefits offered by DIB sections, especially in scenarios where direct access to pixel data is crucial.

Memory Management: SHOWDIB8 manages memory more directly, leveraging the capabilities of DIB sections for pixel data storage and manipulation.

Overall Significance:

SHOWDIB8 showcases an advanced approach to handling bitmaps by utilizing DIB sections.

The program emphasizes the benefits of direct memory access and efficient color representation, offering insights into an alternative method of working with bitmap data in Windows graphics programming.

This versatility is particularly valuable in scenarios where low-level control over pixel data is essential.

SHOWDIB8 showcases DIB sections as a powerful bitmap representation for Windows programming, balancing performance with flexibility.

Understanding the strengths and trade-offs of DIB sections, packed DIBs, and DDBs empowers developers to make informed choices based on specific application requirements and performance goals.

By effectively leveraging DIB sections, developers can optimize image handling and processing tasks in their Windows applications.

FINAL WORDS ABOUT THE PROGRAMS 7&8 ABOVE:

The WM\_PAINT processing in both SHOWDIB7 and SHOWDIB8 serves the common purpose of displaying the bitmap stored in the static variables hBitmap and hPalette onto the client area. While the overall structure is similar, the crucial distinction lies in how these programs handle the File Open menu command and the subsequent processing of the loaded bitmap.

Common WM\_PAINT Processing:

Retaining Bitmap and Palette Handles: Both programs maintain static variables for the bitmap handle (hBitmap) and the logical palette handle (hPalette).

Palette Management: The logical palette is selected into the device context and realized, ensuring accurate color representation during painting.

Bitmap Display:

* The dimensions of the bitmap are obtained using the GetObject function.
* A memory device context is created, and the bitmap is selected into it.
* The BitBlt function is used to display the bitmap on the client area.

Distinctions in File Open Processing:

SHOWDIB7:

Palette Handling: After obtaining the packed DIB pointer, SHOWDIB7 selects the palette into a video device context and realizes it.

The CreateDIBitmap function is then called, requiring the palette to be associated with the device context.

SHOWDIB8:

Palette Independence:

* SHOWDIB8, after obtaining the packed DIB pointer, directly calls CreateDIBSection without the need to select the palette into a device context.
* The first argument to CreateDIBSection (device context handle) is used only if the DIB\_PAL\_COLORS flag is employed.

Pixel Copying:

* The pixel bits are copied from the packed DIB to the memory location returned by CreateDIBSection.
* PackedDibCreatePalette is called to create a palette, leveraging information from the packed DIB.

Significance of SHOWDIB8 Approach:

Direct Bitmap Section Creation:

* SHOWDIB8 directly utilizes CreateDIBSection, bypassing the need to convert the DIB to a device-dependent format.
* The program gains efficiency by avoiding unnecessary conversions.

Palette Independence:

* The absence of palette selection in SHOWDIB8 signifies that CreateDIBSection inherently deals with DIBs without relying on device-dependent palettes.

Efficient Memory Handling:

* Pixel bits are directly copied to the memory location created by CreateDIBSection, streamlining the process.

Dynamic Palette Creation:

* Palette creation in SHOWDIB8 is done through PackedDibCreatePalette, allowing flexibility in handling palette information.

Conclusion:

SHOWDIB7 and SHOWDIB8 share commonalities in displaying bitmaps but diverge in their approaches to handling palettes during the loading of a new bitmap.

SHOWDIB8's strategy, centered around CreateDIBSection and dynamic palette creation, showcases a more direct and efficient method for dealing with DIBs, particularly in scenarios where palette independence and direct memory access are prioritized.

BUILDING A ROBUST DIB LIBRARY IN C: BEYOND PACKED DIBS

The journey through GDI bitmaps, DIBs, and the Windows Palette Manager has equipped us with the knowledge to construct a robust library for working with bitmaps in C. While the previous approach using packed DIBs offered simplicity, its performance bottlenecks, particularly for "get pixel" and "set pixel" operations, necessitate a more efficient solution.

This is where the DIB section shines. By abandoning the packed DIB model and leveraging DIB sections, we can achieve:

* Performance Boost: Direct pixel access eliminates intermediate data copies, significantly speeding up image processing tasks.
* Flexibility: Similar to packed DIBs, DIB sections allow manipulation of bitmap pixels in a device-independent manner.
* Windows NT Optimization: DIB sections are particularly efficient under Windows NT, making them a preferable choice for modern applications.

Here's how we can build a C library around this approach:

1. *Defining the HDIB Handle:*

**

This simple declaration defines an opaque handle of type HDIB that acts as a reference to a DIB section object. The specifics of its internal structure remain hidden, adhering to the "none of your business!" principle.

1. *The DIBInfo Structure:*

**

This structure encapsulates relevant information about the DIB section, including the associated device context, bitmap handle, dimensions, color depth, bytes per row, and most importantly, a direct pointer (pBits) to the raw bitmap data. This enables efficient pixel access through functions like DibGetPixel and DibSetPixel.

1. *Library Functions:*

Here are some example functions for manipulating DIB sections:

*DibCreate(int cx, int cy, int cBitsPerPixel):* Creates a new DIB section with the specified width, height, and color depth. This function allocates memory for the bitmap data and returns an HDIB handle referencing the DIBInfo structure.

*DibGetPixel(HDIB hdib, int x, int y):* Uses the hdib handle to access the DIBInfo structure and retrieves the pixel value at the specified coordinates (x, y) within the bitmap data (pointed to by pBits). The function can then return the color value based on the color depth format.

*DibSetPixel(HDIB hdib, int x, int y, DWORD color):* Similar to DibGetPixel, this function sets the pixel at the specified coordinates to the provided color value, manipulating the raw bitmap data directly through the pBits pointer.

*DibDestroy(HDIB hdib):*Release resources associated with the DIB section, freeing memory and deleting the bitmap object. This should be called when the DIB is no longer needed.

1. *Advantages of this Approach:*

*Faster Pixel Access:* Direct manipulation of raw bitmap data through pBits eliminates expensive data copies and context switching, significantly speeding up pixel operations compared to packed DIBs.

*Simplified Memory Management:* The library functions handle memory allocation and deallocation for the DIB section, reducing the burden on the programmer.

*Flexibility and Control:* The internal structure of the DIBInfo can be adapted to include additional information or functionalities as needed.

1. *Conclusion:*

By leveraging DIB sections and encapsulating them within a well-defined C library, we can achieve efficient and flexible bitmap manipulation in Windows applications. This approach balances performance advantages with clear and structured function interfaces, empowering developers to write robust image processing and graphics applications.

*Note: While the code examples are presented in code boxes, remember that this is a conceptual explanation. The actual implementation of the library functions would involve detailed error checking, memory management considerations, and platform-specific adaptations.*

Additional Considerations for Effective DIB Handling:

File-Mapping Fields: Out of Scope: The dshSection and dsOffset fields within DIBSECTION, primarily used for file-mapping objects, lie beyond the realm of DIBHELP and can be safely disregarded in this context.

Structure Ordering: Macro-Friendly: The strategic placement of ppRow at the structure's forefront simplifies macro definitions, enabling convenient access to row pointers. It's like positioning the most frequently used tool within easy reach.

Error Prevention with iSignature: By diligently checking the iSignature field, DIBHELP functions can thwart potential errors stemming from invalid structure pointers. This vigilance serves as a watchful guard against mishaps.

In essence, DIBSTRUCT serves as a meticulously crafted blueprint, guiding developers through the intricate world of DIB manipulation within the DIBHELP library. By mastering its intricacies, you'll unlock the power to create, modify, and unleash the full potential of device-independent bitmaps in your Windows applications.

DIBSTRUCT: THE CORNERSTONE OF DIBHELP



Accelerated Pixel Access with ppRow:

This array of pointers acts as a fast lane to individual pixel rows, enabling direct manipulation without complex calculations. It's like having an address book for every pixel row, streamlining image processing operations.

An array of pointers, each pointing to a row of pixels in the DIB. These pointers facilitate a faster method for accessing DIB pixel bits, with the rows arranged from the top row first.

Integrity Check with iSignature:

This field safeguards against erroneous pointer usage by confirming the validity of DIBSTRUCT instances. Think of it as a security guard ensuring you're working with the correct structure.

A signature check field set to the binary equivalent of the text string "Dib." This serves as a validity check for pointers to the structure.

Handle to a DIB's Potential:

hBitmap This handle, bestowed by CreateDIBSection, unlocks a DIB's capabilities for diverse bitmap operations. It's like the key to a treasure chest filled with graphical possibilities.

The bitmap handle returned from the CreateDIBSection function. It behaves similarly to handles of GDI bitmap objects and allows for operations on the DIB.

Unlocking Pixel Data with pBits:

This pointer grants unfettered access to the raw pixel data within the DIB section, empowering efficient manipulations. It's like holding a direct line to the heart of a DIB's visual information.

A pointer to the bitmap bits. This memory block is controlled by the operating system, and applications have access to it. It is automatically freed when the bitmap handle is deleted.

DIBSECTION:

A Wealth of Information: This embedded structure holds crucial details about the DIB's anatomy, including its dimensions, color depth, and compression scheme. It's like a detailed blueprint revealing a DIB's inner workings.

A DIBSECTION structure obtained by passing the bitmap handle to the GetObject function. It contains information about the DIB, such as dimensions, color masks, and compression. This provides a convenient way to retrieve detailed information about the DIB.

Mastering Color Masks with iRShift and iLShift:

These arrays hold the keys to unlocking color information in 16-bit and 32-bit DIBs with BI\_BITFIELDS compression. They're like skilled decoders, translating pixel values into their true RGB colors.

Arrays storing right and left shift values for color masks in 16-bit and 32-bit DIBs. These values are used to manipulate and extract color components from the DIB pixels.

DIBHELP.C

The DIBHELP.C file presented in Figure 16−20 provides a set of functions for handling Device-Independent Bitmaps (DIBs) in a Windows environment. Let's delve into the details of these information functions and understand their workings.

1. DIBSTRUCT Structure:

The file begins by defining a structure called DIBSTRUCT, which encapsulates essential information about a DIB. It includes fields such as pointers to rows of pixels, a signature check, a bitmap handle, a pointer to bitmap bits, a DIBSECTION structure, and arrays for right and left shift values for color masks.

2. DibIsValid Function:

The DibIsValid function checks the validity of a given HDIB (Handle to a DIB) by ensuring that it is not NULL, the memory it points to is readable, and the signature matches an expected value.

3. DibBitmapHandle Function:

The DibBitmapHandle function returns the handle to the DIB section bitmap object from the provided HDIB. It first checks the validity of the HDIB using the DibIsValid function.

4. DibWidth, DibHeight, DibBitCount Functions:

These functions, namely DibWidth, DibHeight, and DibBitCount, provide information about the dimensions and bit count of the DIB, respectively. They retrieve values from the DIBSECTION structure within the DIBSTRUCT.

5. DibRowLength Function:

The DibRowLength function calculates and returns the number of bytes per row of pixels in the DIB, considering padding for alignment.

6. DibNumColors, DibMask, DibRShift, DibLShift Functions:

These functions provide information about the color-related aspects of the DIB, such as the number of colors, color masks, and shift values. They access the relevant fields in the DIBSECTION structure.

7. DibCompression Function:

The DibCompression function retrieves the value of the biCompression field from the BITMAPINFOHEADER structure within the DIBSECTION.

8. DibIsAddressable Function:

The DibIsAddressable function checks whether the DIB is compressed or not by examining the compression type in the DIBSECTION.

9. DibInfoHeaderSize, DibMaskSize, DibColorSize, DibInfoSize, DibBitsSize, DibTotalSize Functions:

These functions calculate the sizes of various components of the DIB section as they would appear in a packed DIB. They aid in converting the DIB section to a packed DIB and saving DIB files.

10. DibInfoHeaderPtr, DibMaskPtr, DibBitsPtr Functions:

These functions return pointers to different components of the DIB section, facilitating easy access to header information, color masks, and bitmap bits.

11. DibGetColor, DibSetColor Functions:

The DibGetColor and DibSetColor functions retrieve and set entries in the DIB color table, respectively. They leverage device context operations for interacting with the color table.

*Here's a comprehensive explanation of the information functions in DIBHELP.C:*

Safeguarding Data Integrity:

DibIsValid: This function acts as a gatekeeper, ensuring that only valid DIBSTRUCT pointers are used for accessing DIB information. This prevents potential errors and crashes, enhancing the robustness of the library.

Streamlined Access to Essential Information:

Functions for Basic Attributes: DibWidth, DibHeight, DibBitCount, and DibRowLength provide direct access to fundamental DIB properties, including dimensions, color depth, and pixel row lengths. These are crucial for understanding and manipulating DIB content.

DibNumColors: Determines the number of colors in the DIB's color table, essential for various color-related operations.

Handling Compression and Pixel Accessibility:

DibCompression: Reveals the compression technique used in the DIB, guiding appropriate handling methods.

DibIsAddressable: Determines whether direct pixel access is possible based on the compression type. This is crucial for efficient pixel manipulation.

Memory Allocation and File Operations:

Size-Related Functions: DibInfoHeaderSize, DibMaskSize, DibColorSize, DibInfoSize, DibBitsSize, and DibTotalSize calculate the sizes of various DIB components, aiding in memory allocation and file management.

Direct Pointers for Efficient Manipulation:

Pointer-Providing Functions: DibInfoHeaderPtr, DibMaskPtr, and DibBitsPtr offer direct access to the BITMAPINFOHEADER, color masks, and pixel data, respectively, enabling efficient editing and processing of these structural elements.

Color Table Management:

DibGetColor and DibSetColor: Retrieve and modify individual color entries within the DIB's color table, facilitating color customization. These functions act as wrappers around the more fundamental GetDIBColorTable and SetDIBColorTable functions.

Indirect Access to BITMAPINFO:

No Direct Pointer: While most DIB components are accessible through pointers, the BITMAPINFO structure (information header plus color table) is not directly available in DIB sections. This is due to the underlying memory management of DIB sections.

Indirect Access: The color table can be accessed indirectly through DibGetColor and DibSetColor, and the BITMAPINFOHEADER is available through DibInfoHeaderPtr.

DibCopyToInfo: This function, explained later, allocates a new BITMAPINFO structure and fills it with DIB information, but it doesn't provide a pointer to the existing structure in memory.

Key Takeaways:

* The information functions in DIBHELP.C offer a comprehensive suite of tools for understanding, navigating, and interacting with DIB sections.
* They prioritize data integrity through validation and handle various compression formats effectively.
* They provide direct access to most DIB components, with indirect access for the BITMAPINFO structure.
* These functions serve as a foundation for efficient DIB manipulation within the DIBHELP library.

In summary, these functions collectively provide a comprehensive set of tools for extracting various attributes of a DIB and manipulating its components. The DIBSTRUCT structure serves as a central container for organizing DIB-related information, and the functions utilize this structure to offer a convenient interface for DIB handling in Windows programming.

*Let’s now move on to the next code portion for dibhelp.c part 2… pg 725*

DIBHELP.C PART 2

12. DibPixelPtr Function:

The DibPixelPtr function is designed to return a pointer to the pixel at a specified position (x, y) in the DIB. Before providing the pointer, it performs checks to ensure that the DIB is addressable and that the specified coordinates are within the valid range. The returned pointer allows direct manipulation of the pixel bits.

13. DibGetPixel Function:

The DibGetPixel function retrieves the pixel value at a given position (x, y) within the DIB. It utilizes the previously defined DibPixelPtr function to obtain the pointer to the pixel. The pixel value is then extracted based on the bit count of the DIB, employing different bit manipulation operations for various bit depths (1, 4, 8, 16, 24, and 32 bits). This function is essential for reading pixel values.

14. DibSetPixel Function:

In contrast to DibGetPixel, the DibSetPixel function is responsible for setting a pixel value at a specified position (x, y) within the DIB. Similar to DibGetPixel, it uses DibPixelPtr to get the pointer to the target pixel. The function then modifies the pixel bits based on the bit count and the provided pixel value, facilitating the manipulation of individual pixels.

15. DibGetPixelColor and DibSetPixelColor Functions:

These functions, DibGetPixelColor and DibSetPixelColor, extend the pixel manipulation capabilities by allowing the retrieval and setting of pixel colors, respectively. The DibGetPixelColor function obtains the color information at a specified position (x, y) within the DIB. It considers various scenarios, such as different bit depths and color table indexing. On the other hand, the DibSetPixelColor function sets the pixel color at a specified position, again taking into account the bit depth and adjusting the pixel bits accordingly. These functions provide a convenient way to work with pixel colors directly.

16. Color Handling in DibGetPixelColor and DibSetPixelColor:

The color handling in these functions is adaptable to the DIB's characteristics. For instance, in the case of 24-bit bitmaps, it directly uses the pixel bits as the RGB color. For 32-bit bitmaps with BI\_RGB compression, it interprets the pixel bits as an RGBQUAD structure. In scenarios involving masks and shifts, the functions perform the necessary bit manipulations to extract or set the RGB components of the pixel color.

In summary, this section of DIBHELP.C provides functions for direct manipulation of pixel bits and colors within a DIB. These functions enhance the flexibility of DIB handling in applications, allowing for efficient reading and writing of pixel values and colors at specific positions in the bitmap. The modular design of these functions contributes to the overall ease of use and maintainability of DIB-related operations.

DibPixelPtr Function Explanation:

The DibPixelPtr function serves as the foundation for pixel manipulation within the DIB. Its primary purpose is to retrieve a pointer to the byte where a specific pixel is stored. By utilizing the ppRow field of the DIBSTRUCT structure, which points to the rows of pixels in the DIB, this function calculates the address of the desired pixel. The returned pointer points to the leftmost pixel of the top row for (0, 0), and it can be adjusted for any given (x, y) position. The function ensures the DIB's addressability and validates the coordinates, returning NULL if the DIB is compressed or if the coordinates reference an area outside the DIB.



DibGetPixel and DibSetPixel Functions:

Following DibPixelPtr, the functions DibGetPixel and DibSetPixel leverage the pointer obtained to either retrieve or set the pixel value at a specified position. These functions seamlessly handle different bit depths, casting the pointer to the appropriate data size for 8-bit, 16-bit, 24-bit, and 32-bit DIBs. However, for 1-bit and 4-bit DIBs, additional masking and shifting operations are necessary to manipulate individual pixels.



DibGetColor Function:

The DibGetColor function complements the pixel manipulation by obtaining the pixel color as an RGBQUAD structure. It handles 1-bit, 4-bit, and 8-bit DIBs by retrieving the color from the DIB color table. For 16-bit, 24-bit, and 32-bit DIBs, it involves masking and shifting the pixel value to derive an RGB color.



DibSetPixelColor Function:

Conversely, the DibSetPixelColor function enables setting a pixel value from an RGBQUAD structure. This function is specifically defined for 16-bit, 24-bit, and 32-bit DIBs.



These functions collectively facilitate the direct manipulation of pixels and colors within a DIB, offering flexibility and efficiency in handling various bit depths and compression formats.

DIBHELP.C PART 3

*Now we’re going in-depth to more advanced levels for dibhelp.c…Part 3 pg. 730*

Let's break down the code into sections for discussion:

Shift Calculation Functions:

The MaskToRShift and MaskToLShift functions play a critical role in handling color masks within the DIBHELP.C code.

These static functions are responsible for calculating the shift values necessary for proper bit manipulation.

Specifically, MaskToRShift determines the number of bits a color mask needs to be shifted to the right, while MaskToLShift calculates the leftward shift.

These shift values are essential for correctly extracting color information from the DIB and are particularly relevant in scenarios where color masks are used, such as in the case of BI\_BITFIELDS compression.

DibCreateFromInfo Function:

The DibCreateFromInfo function stands as a cornerstone in the creation of DIB sections.

It employs the CreateDIBSection API to establish a DIB, allocates memory for the associated DIBSTRUCT, and sets up the row pointer.

Beyond these fundamental tasks, it also undertakes the responsibility of handling color masks and bitfields.

This includes calculating shift values for color masks, a crucial aspect when dealing with DIBs that utilize BI\_BITFIELDS compression.

The function provides a robust foundation for creating and initializing DIB structures, ensuring their integrity and usability.

DibDelete Function:

The DibDelete function serves a pivotal role in memory management within the DIBHELP.C code.

Its primary function is to release all memory associated with a given DIB section.

Before initiating the deletion process, the function performs a validity check to ensure that the DIB is, indeed, valid.

This validation step is crucial in preventing potential issues that could arise from attempting to delete an invalid or corrupted DIB.

By encapsulating memory deallocation and validation logic, DibDelete contributes to the overall robustness and reliability of the DIB handling mechanism.

DibCreate Function:

The DibCreate function is tasked with the creation of an HDIB (Handle to DIB) based on explicit arguments provided, such as width, height, bit count, and color count.

This function encapsulates the process of allocating memory for the BITMAPINFO structure, setting its parameters based on the provided arguments, and invoking the DibCreateFromInfo function.

The use of DibCreateFromInfo ensures a consistent and standardized approach to DIB creation, leveraging a well-established mechanism for initializing DIB structures.

This function provides a convenient and flexible interface for developers to generate DIBs tailored to specific requirements.

DibCopyToInfo Function:

The DibCopyToInfo function plays a crucial role in constructing a BITMAPINFO structure based on an existing Device-Independent Bitmap (DIB).

This function is integral in scenarios where a developer needs to obtain a comprehensive representation of the DIB's attributes.

It adeptly handles various aspects of DIB information, including color masks and color table copying.

The primary purpose is to encapsulate the complexities associated with extracting pertinent details from the DIB, ultimately returning a well-formed BITMAPINFO structure.

This function serves as a valuable tool for developers seeking to interact with DIBs at a higher level, providing a standardized interface for obtaining essential information.

DibCopy Function:

The DibCopy function facilitates the creation of a new DIB section derived from an existing one.

One notable feature is its flexibility in allowing the potential swapping of the DIB's width and height.

This function relies on the DibCopyToInfo function to generate the BITMAPINFO structure, ensuring a consistent and standardized approach to the creation of the new DIB section.

By leveraging DibCopyToInfo, the function benefits from a well-established mechanism for capturing the necessary information from the source DIB, providing a convenient way to duplicate DIBs while offering an option for adjusting their dimensions.

This flexibility enhances the utility of the function in various scenarios, from straightforward duplication to more intricate transformations.

DibCopyToPackedDib Function:

The DibCopyToPackedDib function serves a dual purpose, primarily used for both saving DIBs to a file and transferring DIBs to the clipboard.

The function excels in copying DIB information to a packed DIB format, encapsulating color masks, color table, and pixel bits.

Notably, it provides the option to allocate memory with the GMEM\_SHARE flag, enhancing compatibility and sharing capabilities.

This function is particularly valuable in scenarios where the goal is to store or transmit DIBs in a compact and standardized format.

By encapsulating the complexities of packing DIB information, the function streamlines the process of handling DIBs in various contexts, from storage to clipboard operations.

DibCopyFromPackedDib Function:

The DibCopyFromPackedDib function is a pivotal component when dealing with clipboard operations involving Device-Independent Bitmaps (DIBs).

Its primary function is to reconstruct an HDIB (Handle to DIB) from a packed DIB format.

This involves copying the pixel bits and utilizing the information encapsulated in the packed DIB.

The function acts as a bridge between different representations of DIBs, facilitating seamless integration and manipulation of image data.

By intelligently handling the reconstruction process, it simplifies the task of pasting DIBs from the clipboard, ensuring that the HDIB retains all necessary attributes for further processing.

DibFileLoad and DibFileSave Functions:

The tandem of DibFileLoad and DibFileSave functions introduces file I/O operations to the program, enabling the reading and writing of DIBs to and from files.

DibFileLoad reads a DIB from a specified file, employing file access operations to retrieve both the header and pixel data.

This function is crucial for scenarios where pre-existing DIBs need to be loaded into the program.

On the other hand, DibFileSave performs the reverse operation by saving a DIB to a file.

It utilizes file-writing mechanisms to store the DIB's structure and pixel information persistently.

These functions extend the program's utility, allowing developers to seamlessly integrate DIBs with external storage, retrieval, and sharing mechanisms.

DibCopyToDdb Function:

The DibCopyToDdb function is tailored for efficient screen displays, particularly when working with device-dependent bitmaps (DDBs).

It orchestrates the copying of a DIB to a DDB for the purpose of display, leveraging a provided palette for color accuracy.

This function becomes invaluable when developers need to optimize the rendering of DIBs on the screen, as DDBs are typically more efficient for direct rendering on display devices.

By seamlessly integrating with the Windows Graphics Device Interface (GDI), DibCopyToDdb enhances the program's capabilities in visualizing DIBs in a manner that is both efficient and visually accurate.

Let’s keep going…

The initial segment of the DIBHELP.C file introduces two essential functions for deriving left-shift and right-shift values from color masks, specifically tailored for 16-bit and 32-bit DIBs.

These functions were previously detailed in the Color Masking section of Chapter 15.

Moving forward, the focal point becomes the DibCreateFromInfo function, the sole function in DIBHELP responsible for invoking CreateDIBSection and allocating memory for the DIBSTRUCT structure.

All other creation and copy functions within the codebase route through this central function. The function takes a single parameter—a pointer to a BITMAPINFO structure.

While the color table must exist, it doesn't necessarily need to be filled with valid values.

Following the CreateDIBSection call, the function meticulously initializes all fields of the DIBSTRUCT structure.

It's noteworthy that distinct logic is implemented for bottom-up and top-down DIBs when setting the values of the ppRow field, signifying the pointers to the DIB row addresses. The first element of ppRow invariably represents the top row of the DIB.

The subsequent function, DibDelete, serves a dual purpose. It not only deletes the bitmap created in DibCreateFromInfo but also releases the memory allocated during that function's execution.

In contrast to DibCreateFromInfo, the DibCreate function is more likely to be invoked by application programs. Its initial three arguments define the pixel width, height, and the number of bits per pixel, while the last argument allows flexibility for specifying a custom color table size or opting for the default size by setting it to 0.

DibCopy, a function responsible for creating a new DIB section from an existing one, relies on the DibCreateInfo function for memory allocation of a BITMAPINFO structure and populating it with the necessary data.

A BOOL argument in DibCopy determines whether the DIB width and height should be swapped when crafting the new DIB section, providing a level of flexibility for certain use cases.

The tandem of DibCopyToPackedDib and DibCopyFromPackedDib functions finds its purpose in facilitating the transfer of DIBs, particularly for operations involving the clipboard.

The DibFileLoad function reads a DIB section from a DIB file, while DibFileSave performs the complementary operation of saving a DIB to a file.

Concluding the functionalities is the DibCopyToDdb function, designed for efficient screen displays.

It creates a GDI bitmap object from a DIB, requiring handles to the current palette and the program's window.

The window handle serves to obtain a device context into which the palette is selected and realized, allowing the function to subsequently invoke CreateDIBitmap. This functionality was demonstrated in the SHOWDIB7 program earlier in the chapter.

*For those who don’t know english*😭 *let me educate you*😅

Meticulously:

The word "meticulously" means to do something with great attention to detail, precision, and thoroughness. When something is done meticulously, it implies that care and precision have been taken to ensure every aspect is handled with great accuracy and thoroughness.

Tandem:

The word "tandem" refers to two things or elements that work closely together or in coordination with each other. It suggests a partnership or collaboration between the two entities, often emphasizing their interdependence or joint functioning. In the given context, "DibCopyToPackedDib" and "DibCopyFromPackedDib" are described as a tandem, indicating that these two functions work together in a complementary manner to facilitate the transfer of DIBs.

DIBHELP HEADER FILES AND MACROS

DIBHELP.H is a header file that accompanies the DIBHELP.C file and provides various functions and macros related to handling Device Independent Bitmaps (DIBs).

The header file defines the HDIB handle as a void pointer, shielding the internal structure of the DIB from direct access by applications. It also includes declarations for all the functions present in the DIBHELP.C file.

However, the header file also introduces a set of macros that are designed to optimize the performance of certain operations involving DIB pixels. These macros are specific to different bit counts, and they provide a way to access and manipulate pixels in a more efficient manner.

For instance, the header file introduces macros such as DibPixelPtr1, DibPixelPtr4, DibPixelPtr8, and so on.

These macros allow direct access to the memory location of a pixel based on its coordinates (x, y) in the DIB, depending on the bit count of the DIB.

By using these macros, one can bypass function calls and directly manipulate the pixel values, potentially improving performance.

Similarly, the header file provides macros like DibGetPixel1, DibGetPixel4, DibGetPixel8, and their corresponding set functions (e.g., DibSetPixel1, DibSetPixel4, DibSetPixel8).

These macros enable quick retrieval and modification of pixel values without the need for function calls. Again, these macros are specific to different bit counts and are designed to enhance performance by eliminating function call overhead.

In summary, the DIBHELP.H header file augments the DIBHELP.C file by introducing macros that optimize pixel access and manipulation operations for different bit counts in Device Independent Bitmaps. These macros allow for faster pixel handling by eliminating function calls and directly accessing the relevant memory locations.

THE HEADER FILE

The header file provides several sets of macros for different bit counts, allowing efficient access and manipulation of pixels in Device Independent Bitmaps (DIBs). Here are the key sets of macros and their purposes:

DibPixelPtr Macros:

DibPixelPtr1, DibPixelPtr4, DibPixelPtr8, DibPixelPtr16, DibPixelPtr24, DibPixelPtr32: These macros provide a way to obtain a pointer to the memory location of a specific pixel in the DIB based on its coordinates (x, y) and the bit count of the DIB.

For example, DibPixelPtr8(hdib, x, y) returns a pointer to the memory location of the pixel at coordinates (x, y) in an 8-bit DIB.

DibGetPixel Macros:

DibGetPixel1, DibGetPixel4, DibGetPixel8, DibGetPixel16, DibGetPixel24, DibGetPixel32: These macros allow quick retrieval of the pixel value at a specific location in the DIB, based on its coordinates (x, y) and the bit count of the DIB.

For instance, DibGetPixel24(hdib, x, y) returns the RGB value of the pixel at coordinates (x, y) in a 24-bit DIB.

DibSetPixel Macros:

DibSetPixel1, DibSetPixel4, DibSetPixel8, DibSetPixel16, DibSetPixel24, DibSetPixel32: These macros enable the modification of the pixel value at a specific location in the DIB, based on its coordinates (x, y) and the bit count of the DIB.

For example, DibSetPixel8(hdib, x, y, p) sets the pixel at coordinates (x, y) in an 8-bit DIB to the value 'p'.

By utilizing these macros, developers can directly access and manipulate pixels in the DIB without the need for function calls. This can lead to improved performance by reducing the overhead associated with function calls and enabling more efficient pixel handling.

It's worth noting that the macros assume certain conditions and have limited error checking. They are designed to be used carefully, ensuring that the provided coordinates and bit count match the characteristics of the DIB. Hence, it's important to use these macros in a controlled and appropriate manner to avoid unexpected behavior or memory access issues.

*Here are some examples of the macros provided by the DIBHELP.H header file:*

DibPixelPtr Macros:

* DibPixelPtr1(hdib, x, y): Returns a pointer to the memory location of the pixel at coordinates (x, y) in a 1-bit DIB.
* DibPixelPtr8(hdib, x, y): Returns a pointer to the memory location of the pixel at coordinates (x, y) in an 8-bit DIB.
* DibPixelPtr24(hdib, x, y): Returns a pointer to the memory location of the pixel at coordinates (x, y) in a 24-bit DIB.

DibGetPixel Macros:

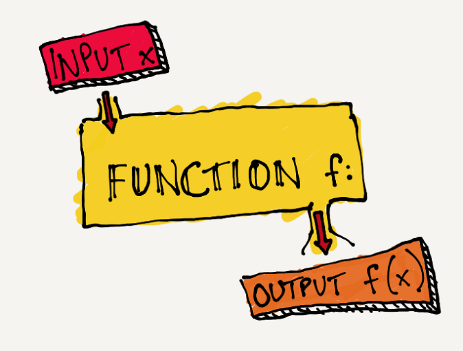
* DibGetPixel1(hdib, x, y): Retrieves the pixel value at coordinates (x, y) in a 1-bit DIB.
* DibGetPixel8(hdib, x, y): Retrieves the pixel value at coordinates (x, y) in an 8-bit DIB.
* DibGetPixel24(hdib, x, y): Retrieves the RGB value of the pixel at coordinates (x, y) in a 24-bit DIB.

DibSetPixel Macros:

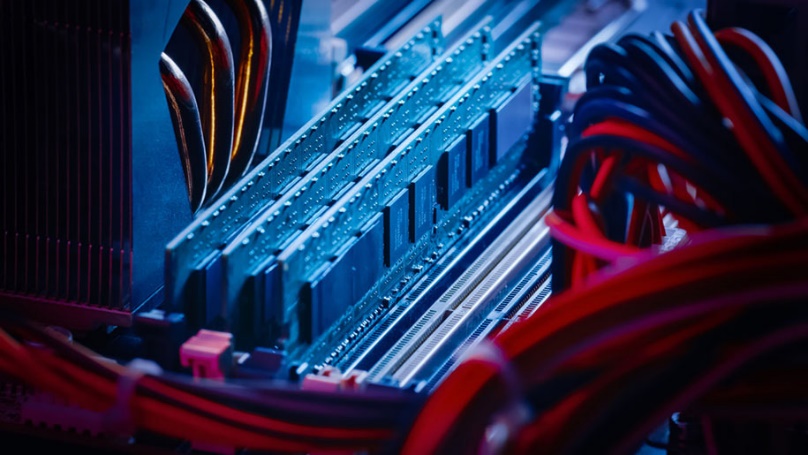
* DibSetPixel1(hdib, x, y, p): Sets the pixel at coordinates (x, y) in a 1-bit DIB to the value 'p'.
* DibSetPixel8(hdib, x, y, p): Sets the pixel at coordinates (x, y) in an 8-bit DIB to the value 'p'.
* DibSetPixel24(hdib, x, y, r, g, b): Sets the RGB value of the pixel at coordinates (x, y) in a 24-bit DIB to the specified (r, g, b) values.

*Now, let's discuss how the macros in the DIBHELP.H header file improve the performance of pixel operations:*

Elimination of function call overhead: The macros allow direct access to pixel values and memory locations, bypassing function calls. This eliminates the overhead associated with function calls, resulting in faster pixel operations.



Efficient memory access: The macros provide optimized memory access by directly calculating the memory location of a pixel based on its coordinates and the bit count of the DIB. This avoids unnecessary memory lookups and improves performance.



Reduced code size: By using macros instead of function calls, the resulting code can be more concise and compact. This can lead to improved cache utilization and instruction pipelining, enhancing overall performance.



*Now, let's discuss the purpose of the HDIB handle and why it is defined as a void pointer:*

The HDIB handle is a type definition used to represent a handle to a Device Independent Bitmap (DIB). It is defined as a void pointer (void\*) to provide flexibility and avoid exposing the internal structure of the DIB to applications.

By defining HDIB as a void pointer, the specific structure of the DIB is hidden from the application code. This abstraction allows the implementation of the DIB handling functions to change and evolve without impacting the application code that uses the HDIB handle.

The void pointer allows the handle to be used as a generic pointer that can be cast to the appropriate DIB structure type when needed. This enables compatibility with different DIB formats and simplifies the interface for working with DIBs.

In summary, the HDIB handle is defined as a void pointer to provide flexibility, abstraction, and compatibility in working with Device Independent Bitmaps. It allows the DIB handling functions to operate on different DIB formats without exposing the internal structure to the application code.

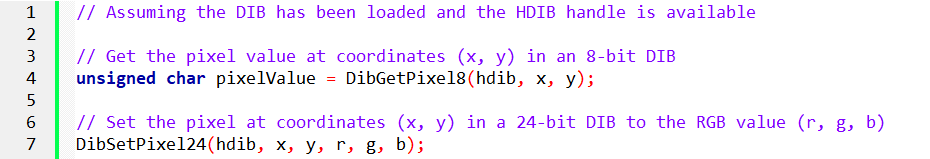
*How can I determine the bit count of a given DIB? Are there any limitations or considerations when using these macros? Can you provide an example of how these macros can be used in practice? What are some other benefits of using the DIBPixelPtr macros for pixel operations? Can you provide an example of how the HDIB handle is used in practice? How does the HDIB handle differ from other types of handles used in programming?*

To determine the bit count of a given Device Independent Bitmap (DIB), you can use the biBitCount member of the BITMAPINFOHEADER structure. The biBitCount member specifies the number of bits per pixel in the DIB. For example, if the biBitCount is 8, it indicates an 8-bit DIB.

When using the macros provided in the DIBHELP.H header file, there are a few limitations and considerations to keep in mind:

* Ensure the correct bit count: The macros are specific to different bit counts (e.g., DibPixelPtr8 for an 8-bit DIB). It's crucial to use the appropriate macros that match the bit count of the DIB you are working with. Using the wrong macro may lead to incorrect results or memory access violations.
* Coordinate validity: The macros assume that the provided coordinates (x, y) are within the valid range of the DIB's dimensions. It's important to ensure that the coordinates fall within the DIB's width and height to avoid accessing out-of-bounds memory locations.

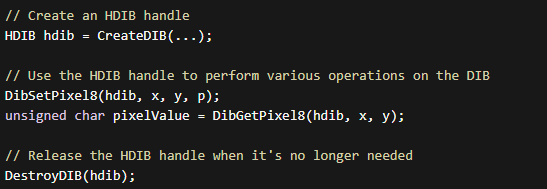
Here's an example of how these macros can be used in practice:



Some benefits of using the DibPixelPtr macros for pixel operations include:

* Direct memory access: The macros provide a pointer directly to the memory location of a pixel, allowing efficient and direct manipulation of pixel values without the need for intermediate steps.
* Performance optimization: By bypassing function calls and accessing pixels directly, the macros can improve the performance of pixel operations, especially in scenarios where a large number of pixels need to be processed.

Regarding the usage of HDIB handles in practice, here's an example:

**

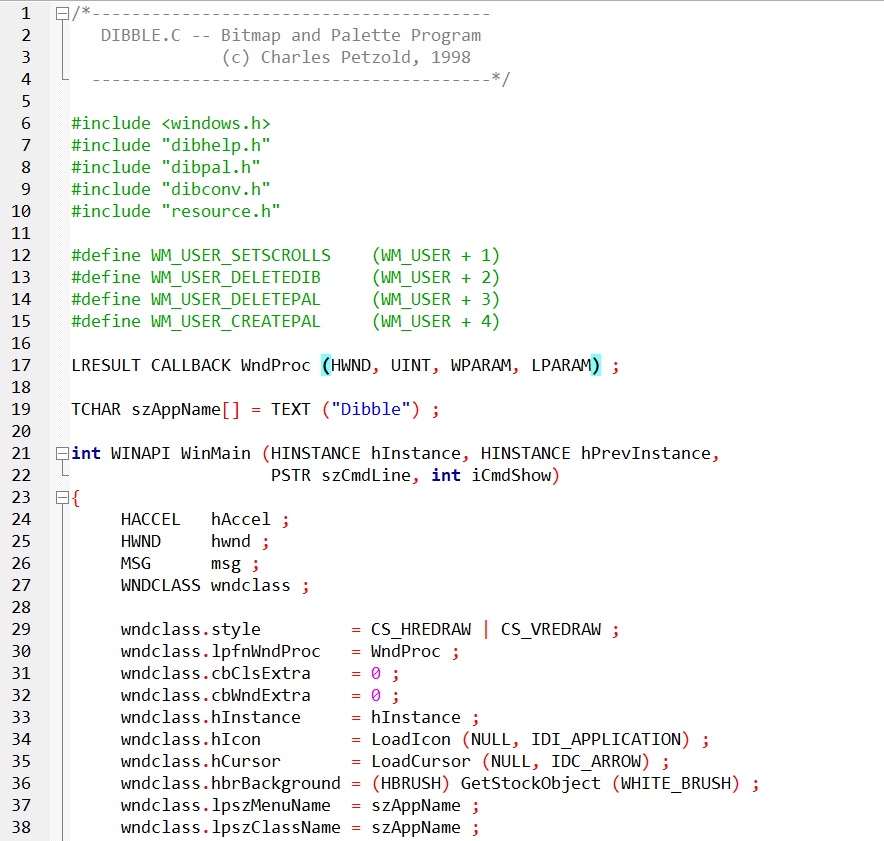
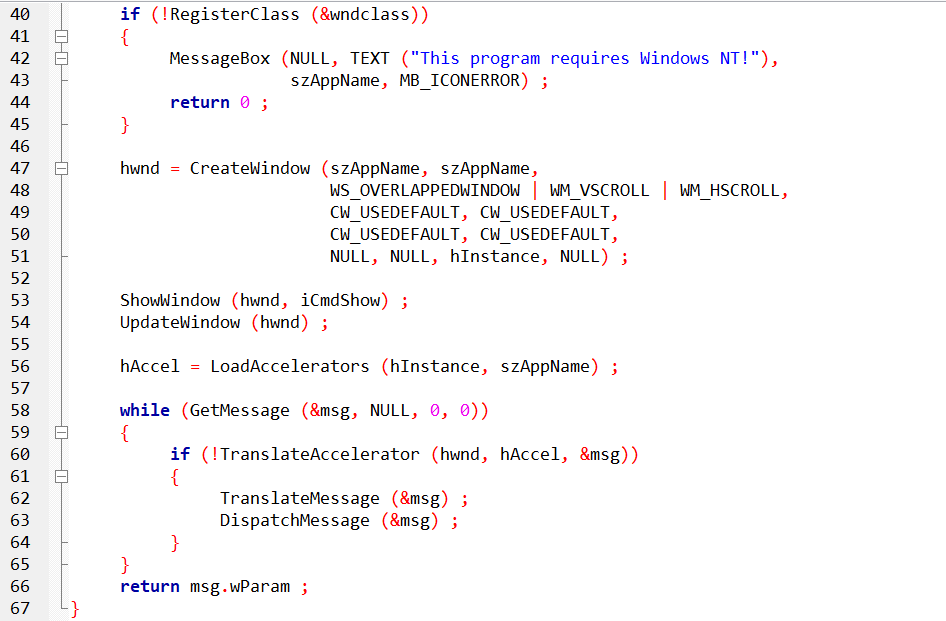
The *HDIB handle* is used to represent a handle to a DIB. It provides an abstraction that encapsulates the internal structure of the DIB, allowing operations to be performed on the DIB without exposing its implementation details. The HDIB handle can be used to pass DIB information between functions and manage the lifetime of the DIB.

The *HDIB handle differs from other types of handles* used in programming in that it is specific to DIBs. Other types of handles, such as file handles or window handles, are typically used to represent different resources or objects in a system. Each handle type has its own purpose and may have specific functions and operations associated with it. The HDIB handle specifically represents a handle to a DIB and provides functions and macros tailored for DIB manipulation.

DIBBLE PROGRAM

This is going to be long, let’s break it down into sections, so that we can cover it in-depth.



The program defines a window procedure (WndProc) and several constants, including window messages (WM\_USER\_SETSCROLLS, WM\_USER\_DELETEDIB, WM\_USER\_DELETEPAL, WM\_USER\_CREATEPAL).

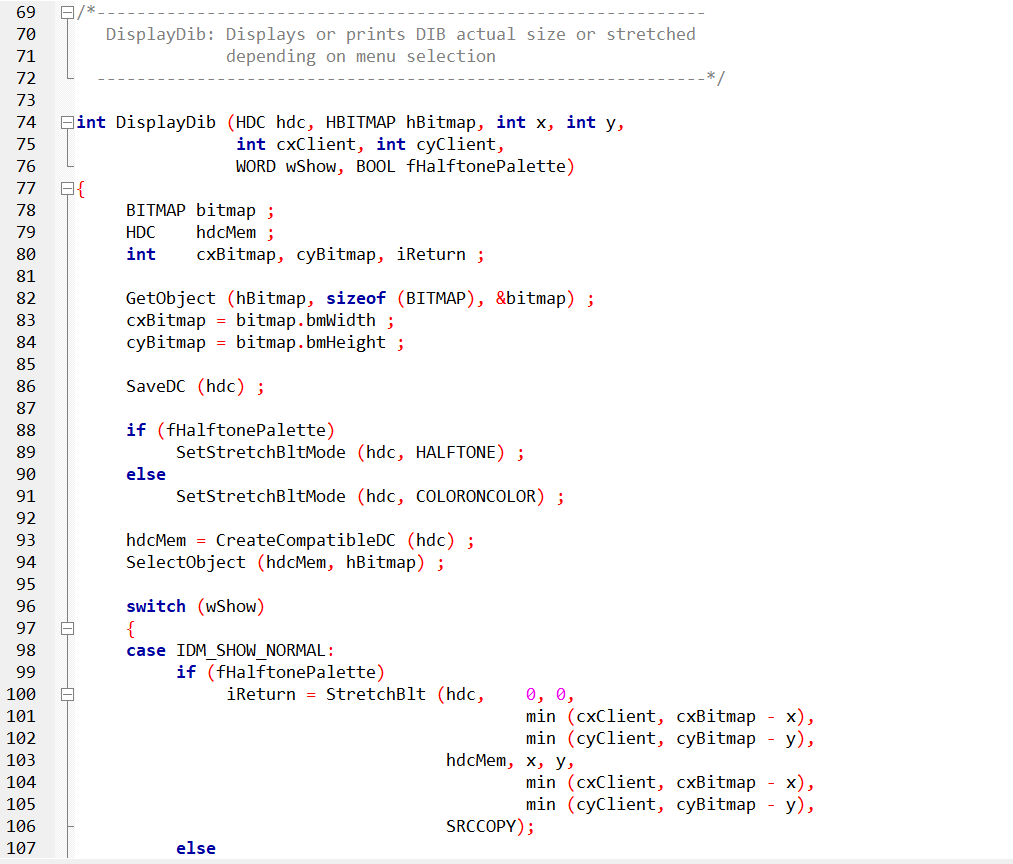
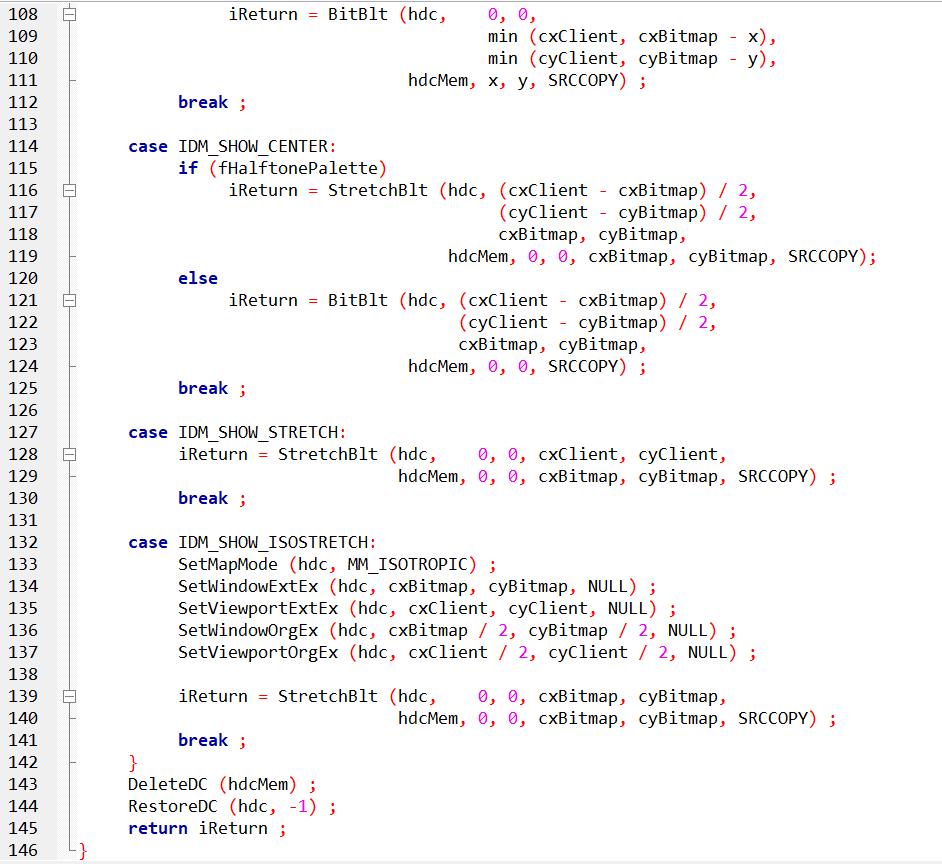
It also defines the application name (szAppName) and the entry point (WinMain).

The WndProc function handles messages for the main window. The window class is registered, and a window is created with scrollbars. The main loop processes messages, and accelerator keys are translated using LoadAccelerators and TranslateAccelerator.

The program involves working with bitmaps and palettes, as suggested by the header files (dibhelp.h, dibpal.h, dibconv.h) and resource file (resource.h).

The code mentions a requirement for Windows NT, indicating that it may have been developed for an older version of Windows. It uses standard Windows API functions for window creation, message handling, and resource loading.

*Let’s move to the second part of dibble.c file…*

The DisplayDib function is designed to display or print a Device Independent Bitmap (DIB) in a specified manner based on the menu selection.

It takes various parameters, including the device context (hdc), the bitmap handle (hBitmap), position coordinates (x and y), client area dimensions (cxClient and cyClient), a flag indicating the display mode (wShow), and a boolean flag for halftone palette usage (fHalftonePalette).

The function begins by retrieving information about the bitmap using the GetObject function, storing the width and height in cxBitmap and cyBitmap, respectively.

A device context (hdcMem) is then created as a memory-compatible DC, and the bitmap is selected into this DC using SelectObject. The subsequent operations are determined by the value of wShow, which corresponds to different display modes.

IDM\_SHOW\_NORMAL:

This mode displays the bitmap either stretched or normally, depending on whether the halftone palette is used or not. It uses either StretchBlt or BitBlt to copy the bitmap from the memory DC to the destination DC, considering the specified coordinates and dimensions.

IDM\_SHOW\_CENTER:

In this mode, the bitmap is displayed at the center of the client area. Similar to the normal mode, it uses either StretchBlt or BitBlt to copy the bitmap with appropriate calculations for centering.

IDM\_SHOW\_STRETCH:

The bitmap is stretched to fit the entire client area in this mode, using StretchBlt.

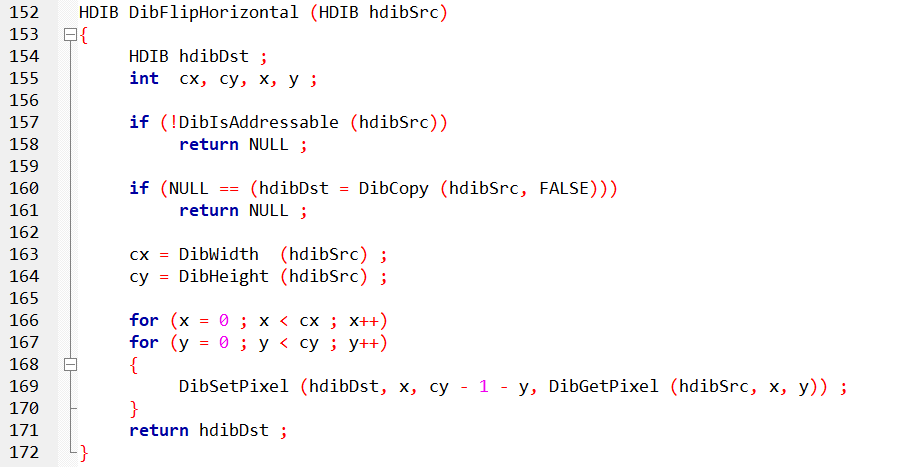
IDM\_SHOW\_ISOSTRETCH:

This mode employs isotropic stretching, setting the mapping mode to isotropic and adjusting the window and viewport extents and origins. The bitmap is then stretched using StretchBlt.

Finally, the memory DC is deleted, and the device context is restored to its original state. The function returns the result of the last GDI operation (iReturn), which can be used to check for success or failure.

The overall purpose of this function is to provide flexibility in displaying a bitmap in different ways, such as normal, centered, stretched, or isotropically stretched, based on user preferences and menu selections. The choice between halftone and color-on-color modes influences the quality of the displayed image, with halftone generally providing smoother results in stretched scenarios.

*The third part of dibble.c…*



The DibFlipHorizontal function is responsible for creating a horizontally flipped version of a given DIB (Device Independent Bitmap) represented by the hdibSrc parameter. The resulting flipped DIB is returned as a new DIB (hdibDst).

The function begins by checking if the source DIB is addressable using the DibIsAddressable function. If the source DIB is not addressable (e.g., due to memory issues or an invalid DIB structure), the function returns NULL.

Next, it allocates memory for the destination DIB (hdibDst) using the DibCopy function. This function creates a copy of the source DIB, and the FALSE parameter indicates that a deep copy (including pixel data) is desired.

The dimensions of the DIB (width and height) are then obtained using the DibWidth and DibHeight functions.

The core operation of the function involves iterating through each pixel of the source DIB and copying it to the corresponding position in the destination DIB, but with a horizontal flip. The nested for loops iterate over each column (x) and row (y) of the source DIB.

For each pixel at coordinates (x, y) in the source DIB, the function uses the DibGetPixel function to retrieve the pixel's color information.

It then employs the DibSetPixel function to set the color information at the mirrored position (x, cy - 1 - y) in the destination DIB.

The mirroring is achieved by subtracting the current y coordinate from the total height (cy - 1 - y), effectively flipping the pixel horizontally.

After iterating through all pixels in the source DIB, the function returns the resulting horizontally flipped DIB (hdibDst).

This function essentially performs a simple image transformation by flipping the pixels of the source DIB horizontally to create a mirrored version. It demonstrates basic pixel manipulation techniques using functions like DibGetPixel and DibSetPixel. Keep in mind that this implementation might not be optimized for performance, and more efficient algorithms could be employed for large images.