CHAPTER 17: DIVING DEEPER INTO TEXT AND FONTS WITH WINAPI

Welcome to Chapter 17, where we'll delve into the fascinating world of text and fonts in the context of WinAPI! As you mentioned, displaying text was our initial foray into graphics programming, and now it's time to refine our skills by exploring:

Font Varieties in Windows: We'll delve into the diverse world of fonts available in Microsoft Windows, including the revolutionary TrueType technology that brought WYSIWYG to life.

Font Manipulation Magic: TrueType's power goes beyond simple display. We'll explore exciting techniques like font scaling, rotation, pattern fills, and even using fonts as clipping regions!

Justifying Text: Learn how to make your text visually appealing by aligning it to the left, right, or center of the window.

Remember, you're currently learning WinAPI, so we'll focus on using its functionalities to achieve these effects. Buckle up and get ready to unleash your inner typography guru!

TrueType: The Game Changer for Text in Windows

The introduction of TrueType in Windows 3.1 marked a significant milestone in text rendering. Unlike older bitmap fonts, TrueType uses mathematical outlines to define character shapes. This offers several advantages:

Scalability: TrueType fonts can be smoothly scaled to any size without losing quality, unlike pixelated bitmap fonts. This is crucial for WYSIWYG, ensuring what you see on screen is what gets printed.

Platform Independence: TrueType fonts work across different platforms like Windows and macOS, promoting compatibility and flexibility.

Advanced Features: TrueType's outline-based approach opens doors for exciting font manipulation techniques like:

- Rotation: You can rotate characters to create unique effects.
- Pattern Filling: Fill character interiors with custom patterns for visual flair.
- Clipping Regions: Use font outlines to define clipping regions for other graphical elements.

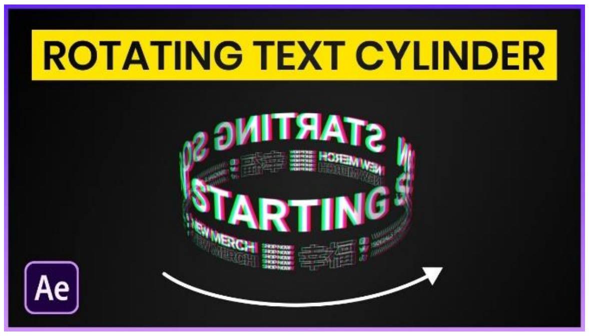
We'll explore these functionalities later in the chapter, so stay tuned!

Beyond Basic Display: Mastering Text Manipulation

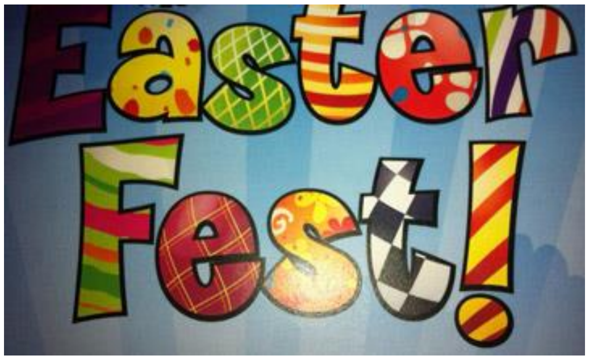
TrueType's capabilities extend far beyond simply displaying text on the screen. With WinAPI, you can unleash your creativity and manipulate fonts in various ways:

Scaling: Change the size of text without compromising quality using functions like CreateScalableFontResource and SetTextScale.

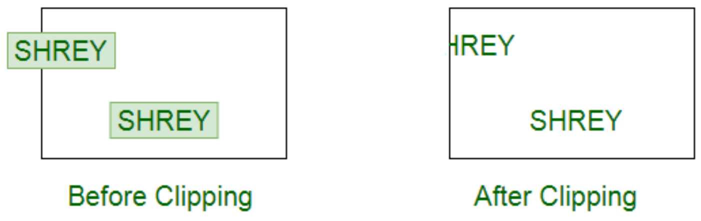
Rotation: Rotate individual characters or entire strings for a dynamic and eye-catching look. Utilize functions like GetGlyphOutline and PolyDraw to achieve this.



Pattern Filling: Fill the interiors of characters with custom patterns using WinAPI functions like FillPath. Imagine text shimmering with stripes or polka dots!



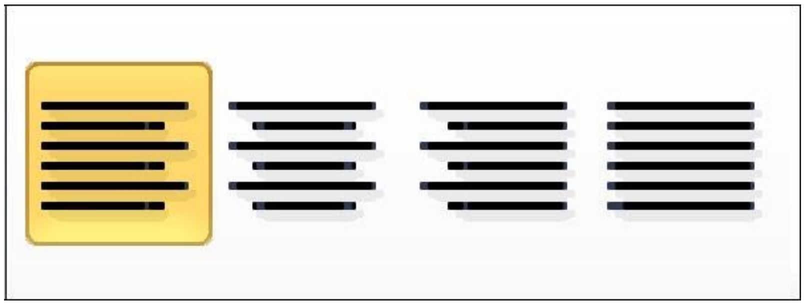
Clipping Regions: Define clipping regions based on font outlines using functions like SelectClipRgn. This allows you to mask other graphical elements behind the text, creating interesting effects.



Remember, these are just a few examples. As you delve deeper into WinAPI text manipulation, you'll discover a vast playground for experimentation and creative expression.

Justifying Text for Visual Balance

Justified text, where both margins are aligned, adds a touch of refinement to your applications. WinAPI provides functions like SetTextAlign and GetTextExtentPoint32 to achieve this effect. You can choose left, right, or center alignment based on your desired layout.



By mastering text justification, you can elevate the visual appeal of your WinAPI applications and create a more polished and professional user experience.

WINDOWS TEXT OUTPUT FUNCTIONS

In Windows programming, text output is facilitated by various functions, and one commonly used function is:

TextOut:

```
TextOut(hdc, xStart, yStart, pString, iCount).
```

Core function for displaying text.

Arguments:

- hdc: Handle to the device context.
- xStart: Horizontal starting position (logical coordinates).
- yStart: Vertical starting position (logical coordinates).
- pString: Pointer to the character string.
- iCount: Length of the string (not NULL-terminated).

Coordinates and Positioning

The xStart and yStart parameters determine the starting position of the text in logical coordinates.

Typically, Windows starts drawing at the upper left corner of the first character.

The function requires a pointer to the character string (pString) and the length of the string (iCount). Notably, it does not recognize NULL-terminated character strings.

The positioning of the text can be influenced by the SetTextAlign function. Flags like TA_LEFT, TA_RIGHT, TA_CENTER, TA_TOP, TA_BOTTOM, and TA_BASELINE affect how xStart and yStart are used for horizontal and vertical positioning.

SetTextAlign Function

By calling SetTextAlign with the TA_UPDATECP flag, the xStart and yStart arguments in TextOut are ignored.

Instead, Windows uses the current position set by functions like MoveToEx or LineTo.

The TA_UPDATECP flag also updates the current position after a TextOut call, which is useful for displaying multiline text.

Controls horizontal and vertical positioning of text.

Flags:

- Horizontal: TA_LEFT, TA_RIGHT, TA_CENTER
- Vertical: TA_TOP, TA_BOTTOM, TA_BASELINE

TA_UPDATECP flag:

- Ignores xStart/yStart in TextOut, uses current position.
- Updates current position after TextOut (except for TA_CENTER).

TabbedTextOut Function

An alternative to multiple TextOut calls for columnar text is the TabbedTextOut function:

```
TabbedTextOut(hdc, xStart, yStart, pString, iCount, iNumTabs, piTabStops, xTabOrigin);
```

If the text string contains tab characters (\t or 0x09), TabbedTextOut expands the tabs based on an array of tab stops.

Handles text with embedded tab characters.

Expands tabs based on specified tab stops.

Arguments:

- iNumTabs: Number of tab stops.
- piTabStops: Array of tab stop positions (pixels).
- xTabOrigin: Starting position for measuring tab stops.

Tab Stops and Customization

The sixth argument (iNumTabs) is the number of tab stops, and the seventh argument (piTabStops) is an array of tab stops in pixels.

You can customize tab stops by providing specific pixel values. If these arguments are set to 0 or NULL, tab stops are set at every eight average character widths.

ADVANCED TEXT OUTPUT FUNCTIONS IN WINDOWS

In Windows programming, the ExtTextOut function provides extended capabilities for text rendering:

```
ExtTextOut(hdc, xStart, yStart, iOptions, &rect, pString, iCount, pxDistance);
```

Arguments:

- hdc: Handle to the device context.
- xStart, yStart: Starting position (logical coordinates).
- iOptions: Flags for clipping and background:
- ETO_CLIPPED: Clips text to the specified rectangle.
- ETO_OPAQUE: Fills the rectangle with the background color before drawing text.
- &rect: Pointer to a rectangle structure for clipping or background.
- pString: Pointer to the character string.
- iCount: Length of the string.
- pxDistance: Optional array of integers for intercharacter spacing (NULL for default).

Clipping and Background Rectangles

The fifth argument (rect) in ExtTextOut is a pointer to a rectangle structure. If iOptions is set to ETO_CLIPPED, it serves as a clipping rectangle; if set to ETO_OPAQUE, it becomes a background rectangle filled with the current background color. Both options can be specified or omitted as needed.

Character Spacing

The last argument (pxDistance) is an array of integers specifying the spacing between consecutive characters. This feature allows fine-tuning of intercharacter spacing, which proves valuable for justifying text in narrow columns. Setting it to NULL defaults to the standard character spacing.

DrawText Function

A higher-level text rendering function in Windows is DrawText:

```
DrawText(hdc, pString, iCount, &rect, iFormat);
```

Arguments:

- hdc: Handle to the device context.
- pString: Pointer to the character string.
- iCount: Length of the string (-1 for NULL-terminated strings).
- &rect: Pointer to a rectangle structure defining the text area.
- iFormat: Flags controlling text formatting.

The iFormat above has flags that control various aspects of text formatting. Here's a breakdown of each flag and its purpose:

Alignment:

- DT_LEFT (default): Specifies left justification of the text.
- DT_RIGHT: Specifies right justification of the text.
- DT_CENTER: Specifies center alignment of the text.

Line Breaking:

- DT_SINGLELINE: Treats carriage returns and linefeeds as displayable characters, meaning they will be shown as they are.
- DT_TOP (default): Places the text at the top of the rectangle.
- DT_BOTTOM: Places the text at the bottom of the rectangle.
- DT_VCENTER: Vertically centers the text within the rectangle.
- DT_WORDBREAK: Breaks lines at the end of words if they don't fit within the rectangle. This ensures that words are not split in the middle.

Clipping:

- DT_NOCLIP: Disables clipping, allowing text to extend beyond the boundaries of the rectangle. This means the text may overflow outside the specified area.

Spacing:

- DT_EXTERNALLEADING: Includes external leading in line spacing. External leading refers to the extra space between lines of text.

Tabs:

- DT_EXPANDTABS: Expands tab characters (\t) to spaces, aligning the text based on the specified tab stops.
- DT_TABSTOP (use cautiously): Sets custom tab stops. The upper byte of iFormat specifies the positions of the tab stops. This flag should be used carefully, as incorrect tab stops can lead to inconsistent or unexpected text alignment.

These flags provide control over the alignment, line breaking, clipping, spacing, and tab behavior of the text. By combining different flags, you can achieve the desired formatting for displaying text within a given rectangle.

Key Points:

- ✓ Use ExtTextOut for granular control over clipping, background, and intercharacter spacing.
- ✓ Use DrawText for simplified text output within a rectangle, with various formatting options.
- ✓ Understand the available flags to tailor text output to your specific needs.
- ✓ Consider using DrawText for common text output tasks due to its convenience.
- ✓ Use ExtTextOut when you need precise control over text rendering.

Specifying Text within a Rectangle

Instead of specifying a coordinate starting position, DrawText uses a RECT structure defining a rectangle where the text should appear.

The function requires a pointer to the character string (pString) and its length (iCount). For NULL-terminated strings, setting iCount to -1 prompts Windows to calculate the length automatically.

Text Formatting Options

The iFormat argument in DrawText allows customization of the text's appearance within the specified rectangle.

Flags such as DT_LEFT (default for left-justified), DT_RIGHT (right-justified), and DT_CENTER (centered) control the horizontal alignment.

Including DT_SINGLELINE prevents newline characters, and DT_TOP, DT_BOTTOM, and DT_VCENTER dictate vertical alignment.

Handling Line Breaks and Word Wrapping

Windows interprets carriage return and linefeed characters as newline characters by default. The DT_SINGLELINE flag changes this behavior.

For multi-line displays, using DT_WORDBREAK breaks lines at the end of words, ensuring more readable text.

Additionally, the DT_NOCLIP flag prevents text truncation outside the specified rectangle.

Tab Handling

For text containing tab characters (\t or 0x09), including the DT_EXPANDTABS flag in DrawText ensures proper rendering.

By default, tab stops are set every eighth character position.

While the DT_TABSTOP flag allows custom tab settings, caution is advised due to potential conflicts with other flags in the iFormat argument.

In programming and character encoding, the term "tab character" refers to a control character that is commonly represented as \t in escape sequences or as the hexadecimal value 0x09. It is a non-printable character used to advance the cursor to the next tab stop.

When you encounter a tab character (\t or 0x09) in a string, it serves as an instruction to move the cursor to the next predefined position, which is typically at regular intervals.

The default convention is to set tab stops every eight character positions, but this can be customized.

DrawTextEx: Enhanced Text Handling with Tab Stops

Purpose:

- ✓ Offers more control over text formatting compared to DrawText, particularly for handling tab stops.
- ✓ Introduced to address limitations of DT_TABSTOP flag in DrawText.

Syntax:

```
DrawTextEx(hdc, pString, iCount, &rect, iFormat, &drawtextparams);
```

Arguments:

- hdc: Handle to the device context.
- pString: Pointer to the character string.
- iCount: Length of the string.
- &rect: Pointer to a rectangle structure defining the text area.
- iFormat: Flags controlling text formatting (same as in DrawText).
- &drawtextparams: Pointer to a DRAWTEXTPARAMS structure for additional settings.

DRAWTEXTPARAMS Structure:

- cbSize: Size of the structure (set to sizeof(DRAWTEXTPARAMS)).
- iTabLength: Size of each tab stop, in units of average character width.
- iLeftMargin: Left margin, in units of average character width.
- iRightMargin: Right margin, in units of average character width.
- uiLengthDrawn: Receives the number of characters processed.

```
DRAWTEXTPARAMS dtp = { sizeof(DRAWTEXTPARAMS), 8, 5, 5 };  // Tabs every 8 characters, 5-character margins
DrawTextEx(hdc, "This is a\ttabbed\tstring.", -1, &rect, DT_EXPANDTABS, &dtp);
```

Key Points:

- ✓ Use DrawTextEx when you need precise control over tab stops.
- ✓ Set iTabLength to define the spacing between tab stops.
- ✓ Margins are optional for further text positioning.
- ✓ uiLengthDrawn provides feedback on the amount of text drawn.
- ✓ Avoids conflicts with other flags in iFormat.
- ✓ DrawTextEx might not be available on older Windows systems.
- ✓ For simple tab handling without margins, DT_TABSTOP in DrawText might still suffice.
- ✓ Choose the appropriate function based on your specific tab stop requirements and compatibility needs.

Utilizing Enhanced Settings

With DrawTextEx, developers gain greater flexibility in text layout and formatting. The DRAWTEXTPARAMS structure allows for fine-tuning tab stops, adjusting margins, and obtaining information about the processed text.

This enhanced functionality is particularly valuable in scenarios where precise text alignment, tabulation, and margin control are essential.

By leveraging the DrawTextEx function with the DRAWTEXTPARAMS structure, developers can create visually appealing and precisely formatted text displays in Windows applications. This improved feature set contributes to a more robust and customizable text rendering experience.

DEVICE CONTEXT ATTRIBUTES FOR TEXT RENDERING IN WINDOWS

In Windows programming, several device context attributes play a crucial role in determining how text is displayed. These attributes allow developers to customize aspects such as text color, background color, background mode, and intercharacter spacing.

Customizing Text Color

The default text color in the device context is black, but developers can alter it using the SetTextColor function: Text Color is controlled by:

```
SetTextColor(hdc, rgbColor);
```

The rgbColor parameter represents the desired color, and Windows converts this value into a pure color. To retrieve the current text color, developers can use the GetTextColor function.

Background Mode and Color

Windows displays text within a rectangular background area, which can be colored based on the background mode setting. Developers can change the background mode using SetBkMode:

```
SetBkMode(hdc, iMode);
```

The iMode parameter can be either OPAQUE or TRANSPARENT. The default is OPAQUE, where Windows fills the background with the specified color. The background color can be set using:

```
SetBkColor(hdc, rgbColor);
```

The rgbColor parameter is converted to a pure color. In TRANSPARENT mode, Windows ignores the background color, enhancing text visibility.

Handling Intercharacter Spacing

Intercharacter spacing can be adjusted using the SetTextCharacterExtra function:

```
SetTextCharacterExtra(hdc, iExtra);
```

The iExtra parameter, in logical units, determines the spacing between characters. A value of 0 means no additional space.

Negative values are converted to their absolute values, preventing spacing less than 0. Developers can retrieve the current intercharacter spacing with GetTextCharacterExtra.

Adapting to System Colors

For consistency with system color settings, developers can set text and background colors using system colors:

```
SetTextColor(hdc, GetSysColor(COLOR_WINDOWTEXT));
SetBkColor(hdc, GetSysColor(COLOR_WINDOW));
```

In case of system color changes, developers should handle the WM_SYSCOLORCHANGE message and update the display accordingly:

```
case WM_SYSCOLORCHANGE:
    InvalidateRect(hwnd, NULL, TRUE);
    break;
```

Understanding and manipulating these device context attributes provide developers with the tools needed to create visually appealing and adaptable text displays in Windows applications. Fine-tuning text color, background settings, and intercharacter spacing contribute to a more polished and user-friendly graphical interface.

Key Points:

- ✓ These attributes influence text appearance and background rendering.
- ✓ Choose appropriate settings for text readability and visual appeal.
- ✓ Consider system-consistent colors for seamless user experience.
- ✓ Adjust intercharacter spacing for fine-tuning text layout.

LEVERAGING STOCK FONTS FOR TEXT RENDERING IN WINDOWS

In Windows programming, the rendering of text is closely tied to the font selected in the device context.

While developers often need diverse fonts for displaying text, Windows provides a convenient solution through stock fonts.

Stock fonts are predefined fonts that offer a straightforward way to handle various typefaces and sizes.

Obtaining a Stock Font

To obtain a handle to a stock font, developers can use the GetStockObject function:

```
hFont = GetStockObject(iFont);
```

Here, iFont is one of several identifiers representing different stock fonts. The obtained font handle can then be selected into the device context using SelectObject:

```
SelectObject(hdc, hFont);
```

Alternatively, these two steps can be combined into a single call:

```
SelectObject(hdc, GetStockObject(iFont));
```

Understanding Font Roles and Selection:

Font Importance: Fonts play a vital role in text readability, visual appeal, and brand identity.

Stock Fonts: Quick Access: Windows offers a set of predefined stock fonts for immediate use, simplifying font selection for common scenarios.

Accessing Stock Fonts: The GetStockObject function retrieves a handle to a desired stock font, which is then selected into a device context using SelectObject.

Common Stock Fonts and Applications:

SYSTEM_FONT: The default proportional font, suitable for general text display.

SYSTEM_FIXED_FONT: A fixed-pitch font ensuring consistent character widths, often used for code listings or tabular data.

OEM_FIXED_FONT: The Terminal font, designed for compatibility with MS-DOS environments and legacy character sets.

DEFAULT_GUI_FONT: The font used in Windows UI elements like title bars, menus, and dialog boxes, ensuring visual consistency.

Font Metrics and Text Layout:

Measuring Fonts: The GetTextMetrics function provides essential information about a font's character height and average width, crucial for accurate text positioning and layout calculations.

Proportional Font Considerations: Proportional fonts have characters with variable widths, requiring careful consideration when determining text dimensions. Techniques for handling variable-width fonts will be discussed later.

Beyond Stock Fonts: Customization and Fine-Tuning:

Limitations: Stock fonts offer a limited range of choices and less control over specific typefaces and sizes.

Greater Flexibility: Windows provides font creation functions, discussed later, that enable you to precisely specify the desired typeface and size, unlocking a wider range of font possibilities.

Key Takeaways:

- ✓ Stock fonts offer a convenient starting point for basic font usage in Windows programming.
- ✓ Understanding their characteristics, limitations, and appropriate usage is essential for effective text rendering.
- ✓ For more precise control over font selection and customization, explore font creation functions to achieve the desired visual effects and text presentation.

UNDERSTANDING FONT BASICS IN WINDOWS

Before delving into specific code, it's crucial to establish a solid understanding of fonts in the Windows environment. Fonts play a pivotal role in text rendering, and Windows supports two main categories: "GDI fonts" and "device fonts."

Categories of Fonts

GDI Fonts:

Raster Fonts: Also known as bitmap fonts, these fonts store each character as a bitmap pixel pattern. They are designed for specific aspect ratios and character sizes. Raster fonts are "nonscaleable," meaning they cannot be expanded or compressed arbitrarily. They are fast to display and legible.

- ✓ Characters stored as bitmaps.
- ✓ Limited scalability.
- ✓ Design-specific sizes and aspect ratios.
- ✓ Fast display and high readability.
- ✓ Common typefaces: System, FixedSys, Terminal, Courier, MS Serif, MS Sans Serif, Small Fonts.

Stroke Fonts: Defined as a series of line segments in a "connect-the-dots" format, stroke fonts are continuously scalable. However, their performance is poorer, legibility suffers at small sizes, and characters may appear weak at large sizes. Stroke fonts include Modern, Roman, and Script.

- ✓ Defined by line segments.
- ✓ Continuously scalable.
- ✓ Lower performance and legibility concerns.
- ✓ Suitable for plotters.
- ✓ Common typefaces: Modern, Roman, Script.

TrueType Fonts:

- ✓ Outline-based, scalable to any size.
- ✓ High quality and visual appeal.
- ✓ Widely used in modern Windows systems.

Device Fonts:

Native to output devices like printers, these fonts are built into the hardware. Printers often have a collection of device fonts.

Typeface Names:

- System Font: Typeface name is "System," used for SYSTEM_FONT.
- System Fixed Font: Typeface name is "FixedSys," used for SYSTEM_FIXED_FONT.
- OEM Fixed Font (Terminal): Typeface name is "Terminal," used for OEM_FIXED_FONT.
- Courier: A fixed-pitch font resembling typewriter text.
- MS Serif and MS Sans Serif: Used for DEFAULT_GUI_FONT. "Serif" fonts have small turns finishing strokes, while "sans serif" fonts lack serifs.
- Small Fonts: Specifically designed for displaying text in small sizes.

Font Attributes/Characteristics:

- Typeface: The design of the characters (e.g., Times New Roman, Arial).
- Size: Measured in points (e.g., 12pt).
- Weight: Regular, bold, or light.
- Style: Normal, italic, or oblique.
- Synthesized Attributes: Windows can create bold, italic, underlined, and strikethrough effects without separate fonts.

Key Considerations for Font Selection:

- Readability: Raster fonts often excel in clarity and legibility.
- Scalability: TrueType fonts offer flexibility for various sizes and resolutions.
- Performance: Raster fonts display faster, while TrueType fonts may require more processing.
- Visual Appeal: TrueType fonts generally provide superior aesthetics and design variety.
- Device Compatibility: Device fonts are specific to output devices, while GDI fonts offer broader compatibility.

TrueType Fonts

TrueType fonts constitute a significant part of Windows font capabilities. Unlike raster fonts and stroke fonts, TrueType fonts offer scalability without compromising legibility.

TrueType fonts will be explored in greater detail in subsequent sections. Combine advantages of scalability and visual quality.

OpenType fonts, a later extension of TrueType, offer additional features and cross-platform compatibility.

Font Attributes Synthesis

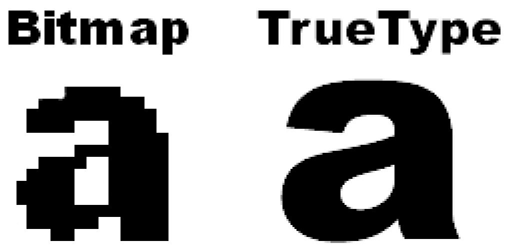
For both GDI raster fonts and GDI stroke fonts, Windows can synthesize attributes like boldface, italics, underlining, and strikethrough without storing separate fonts for each variant. For example, to achieve italics, Windows shifts the upper part of the character to the right.

Essential Takeaways:

- ✓ Windows supports a diverse range of font technologies to meet different needs.
- ✓ Understanding the characteristics and trade-offs of each font type is crucial for effective font selection and usage.
- ✓ TrueType fonts have become the dominant choice in contemporary Windows environments due to their versatility and visual appeal.

OffBit Fonts
OffBit Contain 6 Styles
Bitmap & Display
<Regular, Bold, Dot, Dot Bold, 101, 101 Bold/>
Power Type Bitmap & Display Font Collection
BitOFF 05
Display Fonts
©022

Raster font.



True type font.

EXPLORING TRUETYPE FONTS IN WINDOWS

TrueType fonts represent a significant advancement in font technology, offering scalable and detailed characters defined by filled outlines of straight lines and curves. Understanding how TrueType fonts work is crucial for effective utilization in Windows programming.

Characteristics of TrueType Fonts

Character Definition:

TrueType fonts define individual characters through filled outlines of straight lines and curves.

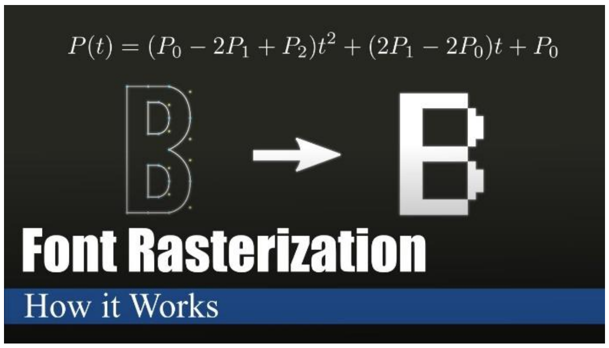
These fonts can be scaled by altering the coordinates that define the outlines.

Rasterization Process:

When a program begins using a TrueType font at a specific size, Windows engages in a process called "rasterization."

Rasterization involves scaling the coordinates of lines and curves in each character based on hints embedded in the TrueType font file.

These hints compensate for rounding errors, ensuring characters maintain their intended appearance.



Bitmap Creation:

The scaled outlines of characters are used to create bitmaps for each character.

These bitmaps are cached in memory for future use, contributing to improved performance.

Key Characteristics in summary:

Outline-Based: Characters defined by mathematical outlines of lines and curves, ensuring smooth scaling and high visual quality.

Scalability: Can be expanded or reduced to any size without losing clarity, offering versatility in design and output.

Hints: Embedded instructions within font files guide scaling and rendering for optimal appearance, preventing distortions and preserving design integrity.

Rasterization: Windows converts TrueType outlines to bitmaps for display and printing, balancing visual appeal with performance.

Caching: Frequently used bitmaps are stored in memory for faster retrieval, reducing processing overhead.

Common TrueType Fonts in Windows:

Courier New: Fixed-pitch font resembling typewriter output.

Times New Roman: Highly readable serif font, popular for printed material.

Arial: Versatile sans-serif font, suitable for screens and print.

Symbol: Contains a collection of symbols and characters for specialized use.

Lucida Sans Unicode: Includes a wider range of global alphabets for multilingual support.

Symbol

In recent Windows versions, the list has expanded, with additional fonts like Lucida Sans Unicode, catering to diverse alphabets used globally.

Advantages:

- ✓ Versatility: Adaptable to various sizes and output devices.
- ✓ Visual Quality: Smooth outlines and careful rendering produce crisp, professional-looking text.
- ✓ Readability: Well-designed TrueType fonts enhance text clarity and ease of reading.
- ✓ Aesthetics: Offer a wide range of creative font styles and designs for visual appeal.

Considerations:

- ✓ Processing Overhead: Rasterization can require more processing compared to raster fonts.
- ✓ Font Availability: Not all fonts are available in TrueType format.

Overall, TrueType fonts have become the dominant font technology in modern Windows environments due to their flexibility, visual appeal, and widespread support across devices and applications.

RECONCILING TRADITIONAL AND COMPUTER TYPOGRAPHY:s

Windows' Adaptive Approach: Navigates the subtle distinctions between traditional typography's emphasis on distinct font styles and computer typography's attribute-based approach by supporting both methods of font selection. This flexibility empowers developers to align with specific needs or preferences for font management.

Point Size: A Design Guideline, Not a Precise Ruler:

While point size serves as a foundation for font sizing, it's crucial to recognize its role as a typographic concept rather than a rigid measurement.

Actual character dimensions within a font can deviate from expectations based on point size alone.

This underscores the importance of utilizing GetTextMetrics for accurate character measurements in computer typography.

Historical Context: The point, approximately 1/72 of an inch, originates from traditional print typography and is now a digital fixture for font sizing. However, its role as a guide rather than a strict measure highlights the nuances of font design and rendering across different technologies.

Leading: Orchestrating Vertical Harmony:

Internal Leading: Space for Diacritics: This design element within a font accommodates diacritics (accents) that often extend above or below the standard character height, ensuring proper visual balance.

External Leading: Guiding Line Spacing: The TEXTMETRIC structure's tmExternalLeading value offers a suggestion for spacing between lines of text, promoting readability and visual rhythm. Developers can directly utilize this value or adjust it based on specific design requirements.

TextMetrics for Accurate Metrics: This essential function provides a detailed look at font metrics, including tmHeight (representing line spacing rather than precise font size), tmInternalLeading, and tmExternalLeading. Leveraging this information is crucial for achieving accurate text layout and spacing in digital environments.

Beyond the Basics: Embracing the Art and Science of Typography:

Font Design Nuances: Font designers carefully consider character proportions and leading values to achieve a balance between readability, visual appeal, and the intended tone of the text. These considerations are important when selecting fonts and presenting text effectively.

Balancing Conventions and Technical Metrics: Successful text rendering requires a blend of typographic conventions and the technical aspects of font metrics in digital systems. Understanding this interplay is crucial for creating visually pleasing and professional-looking documents.

Experimentation and Exploration: The world of fonts offers a wide range of diversity and expressive possibilities. By exploring different font families, styles, sizes, leading, and spacing, you can discover combinations that align with your design goals and enhance the overall user experience.

ADDRESSING THE LOGICAL INCH CONUNDRUM IN WINDOWS DISPLAY

In a previous discussion in Chapter 5, we delved into the intricacies of the system font in Windows 98, highlighting its definition as a 10-point font with 12-point line spacing.

The choice between Small Fonts and Large Fonts in Display Properties dictated the tmHeight value, impacting the pixel resolution and resulting in an implied resolution of either 96 dpi or 120 dpi.

This implied resolution is ascertainable through GetDeviceCaps with LOGPIXELSX or LOGPIXELSY arguments, denoting the dots per inch.

The Logical Inch Phenomenon

The concept of a "logical inch" emerges, representing the metrical distance occupied by 96 or 120 pixels on the screen.

When measured with a ruler, a logical inch often appears larger than an actual inch.

This discrepancy stems from the need to display legible 8-point type on the screen, considering factors like readability and typical viewing distances.

In typography, 8-point type on paper, with approximately 14 characters per inch, is easily readable.

Translating this to a video display directly might result in insufficient pixel density for legible characters.

The logical inch acts as a magnification factor, facilitating the display of legible fonts, even as small as 8 points.

Additionally, the logical inch takes advantage of the screen width, aligning with standard paper margins and optimizing text display.

Concept: A unit of measurement introduced to ensure legible font display on screens, even when physical pixel density might be insufficient for small type sizes.

Purpose:

- Enables readable text sizes (e.g., 8-point) on various display resolutions.
- Accounts for typical viewing distances from screens (usually further than reading print on paper).
- Optimizes display width for text layout, aligning with standard paper margins.

Windows NT Distinction

Windows NT introduces some variations in its approach.

Unlike Windows 98, the LOGPIXELSX and LOGPIXELSY values in Windows NT are not equivalent to pixel count divided by size in millimeters, multiplied by 25.4.

While Windows uses HORZRES, HORZSIZE, VERTRES, and VERTSIZE values for mapping modes, programs displaying text are better off assuming a display resolution based on LOGPIXELSX and LOGPIXELSY.

Windows 98:

- System font defined as 10-point with 12-point line spacing.
- "Small Fonts" setting implies 96 dots per logical inch (dpi).
- "Large Fonts" setting implies 120 dpi.
- Obtain logical dpi using GetDeviceCaps(hdc, LOGPIXELSX) or GetDeviceCaps(hdc, LOGPIXELSY).

Windows NT:

- Inconsistency between LOGPIXELS values and actual display dimensions.
- Recommended to use logical dpi for text display, aligning with Windows 98 behavior.
- Create custom mapping mode (e.g., "Logical Twips") for consistent text handling.

Introducing the "Logical Twips" Mapping Mode

Given the disparities, a program under Windows NT might opt for a custom mapping mode aligned with logical pixels per inch, similar to Windows 98.

One such mapping mode, termed "Logical Twips," involves the following setup:

```
SetMapMode(hdc, MM_ANISOTROPIC);
SetWindowExtEx(hdc, 1440, 1440, NULL);
SetViewportExt(hdc, GetDeviceCaps(hdc, LOGPIXELSX), GetDeviceCaps(hdc, LOGPIXELSY), NULL);
```

In this mode, font dimensions can be specified in 20 times the point size (e.g., 240 for 12 points). Notably, the y-values increase downward, enhancing the display of successive lines of text.

Mapping Modes and Text Consistency:

Windows 98: Predefined mapping modes generally work well for text display.

Windows NT:

- Avoid default mapping modes for text due to inconsistencies.
- Define a custom mapping mode like "Logical Twips" to ensure consistent text sizing and positioning across different Windows versions.

Key Points:

- ✓ Logical inch often larger than an actual inch, intentionally magnifying text for better readability on screens.
- ✓ 640-pixel minimum display width at 96 dpi closely aligns with 8.5-inch paper width with standard margins.
- ✓ Discrepancy between logical and real inches only applies to displays; printers maintain consistency between GDI measurements and physical dimensions.

Additional Considerations:

User Feedback: Consider incorporating user preferences for font size and display settings to enhance readability and accessibility.

Modern Systems: While logical inch concept remains relevant, newer Windows versions and high-resolution displays may offer alternative approaches to font scaling and display optimization.

It's crucial to recognize that the logical inch vs. real inch inconsistency is specific to the display, and on printer devices, consistency prevails with GDI and rulers.

This nuanced understanding is imperative for accurate and visually appealing text rendering in Windows programming.

UNDERSTANDING LOGICAL FONTS:

Abstract Descriptions: Logical fonts act as blueprints for text appearance, defining characteristics like typeface, size, weight, and style.

GDI Objects: They are handles of type HFONT, created and managed by the Windows Graphics Device Interface (GDI).

Device Independence: Logical fonts bridge the gap between desired text styles and device-specific font capabilities, ensuring consistent text rendering across different displays and printers.

- ✓ A logical font, encapsulated in a GDI object with a handle of type HFONT, serves as a descriptor for a font. Much like logical pens and logical brushes, it remains an abstract entity until selected into a device context using SelectObject. This selection process solidifies its existence, and only then does Windows gain awareness of the device's font capabilities.

Creating and Selecting Logical Fonts

Logical fonts come into existence through the invocation of either CreateFont or CreateFontIndirect.

While CreateFont takes 14 individual arguments, mirroring the fields of a LOGFONT structure, CreateFontIndirect receives a pointer to a LOGFONT structure with these 14 fields.

Creating a LOGFONT structure for CreateFontIndirect involves three primary approaches:

Direct Specification: Set the fields of the LOGFONT structure to the desired font characteristics. However, Windows employs a "font mapping" algorithm during SelectObject, attempting to provide the closest match available on the device.

Enumeration: Enumerate all fonts on the device, allowing you to choose from or present them to the user. Font enumeration functions exist for this purpose, although they are less common nowadays due to a more automated alternative.

ChooseFont Function: Utilize the ChooseFont function, which returns a LOGFONT structure, streamlining the font selection process.

In this discussion, the focus will be on the first and third approaches.

The lifecycle of a logical font involves three key steps:

Steps for Handling Logical Fonts:

Create: Use CreateFontIndirect (or CreateFont) to generate a logical font based on desired attributes.

Select: Employ SelectObject to activate the logical font in a device context. Windows maps it to a suitable real font.

Query: Obtain detailed information about the selected real font using GetTextMetrics and GetTextFace functions.

Utilize: Draw text with the selected font.

Delete: Call DeleteObject to remove the logical font when finished (not applicable to stock fonts or those still selected in a DC).

Creation:

Initiate a logical font using CreateFont or CreateFontIndirect. These functions yield a handle to an HFONT.

CreateFontIndirect:

- ✓ Preferred function for creating logical fonts.
- ✓ Takes a pointer to a LOGFONT structure specifying font attributes.

ChooseFont:

- ✓ Simplifies font selection by presenting a user-friendly dialog.
- ✓ Returns a LOGFONT structure with user-specified attributes.

Selection:

Select the logical font into the device context with SelectObject. Windows, in turn, chooses a real font that closely matches the logical font.

Deletion:

After utilizing the font, delete it by invoking DeleteObject. It's crucial to avoid deleting the font while it is selected in a valid device context and refrain from deleting stock fonts.

Font Mapping:

Key Process: Windows matches a logical font to the closest available real font on the device when selected into a device context.

Possible Variance: The actual font displayed or printed might differ slightly from the requested logical font, depending on device-specific font availability.

Structures for Font Information:

LOGFONT: Defines font attributes during creation (14 fields, including typeface, size, weight, style, etc.).

```
// LOGFONT structure definition
typedef struct tagLOGFONT {
  LONG lfHeight;
  LONG lfWidth;
  LONG lfEscapement;
  LONG lfOrientation;
  LONG lfWeight;
  BYTE lfItalic;
  BYTE lfUnderline;
  BYTE lfStrikeOut;
  BYTE lfCharSet;
  BYTE lfOutPrecision;
  BYTE lfClipPrecision;
  BYTE lfQuality;
  BYTE lfPitchAndFamily;
  TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT;
```

Key Considerations:

- ✓ Font Availability: Be mindful of device-specific font limitations and potential variations in rendering.
- ✓ User Preferences: Consider incorporating user choices for font styles and sizes to enhance readability and accessibility.
- ✓ Font Families: Explore font families with broad availability to increase the likelihood of consistent rendering across devices.
- ✓ Modern Font Technologies: Stay updated on advancements in font rendering and management for optimal text presentation in contemporary Windows environments.

TEXTMETRIC: Retrieves information about the currently selected font in a device context (20 fields, including size metrics, character spacing, etc.).

```
// TEXTMETRIC structure definition
typedef struct tagTEXTMETRIC {
  LONG tmHeight;
  LONG tmAscent;
  LONG tmDescent;
  LONG tmInternalLeading;
  LONG tmExternalLeading;
  LONG tmAveCharWidth;
  LONG tmMaxCharWidth;
  LONG tmWeight;
  LONG tmOverhang;
  LONG tmDigitizedAspectX;
  LONG tmDigitizedAspectY;
  TCHAR tmFirstChar;
  TCHAR tmLastChar;
  TCHAR tmDefaultChar;
  TCHAR tmBreakChar;
  BYTE tmItalic;
  BYTE tmUnderlined;
  BYTE tmStruckOut;
  BYTE tmPitchAndFamily;
  BYTE tmCharSet;
} TEXTMETRIC;
```

Extracting Font Information

To discern the face name of the font currently in the device context, GetTextFace proves useful:

```
GetTextFace(hdc, sizeof(szFaceName) / sizeof(TCHAR), szFaceName);
```

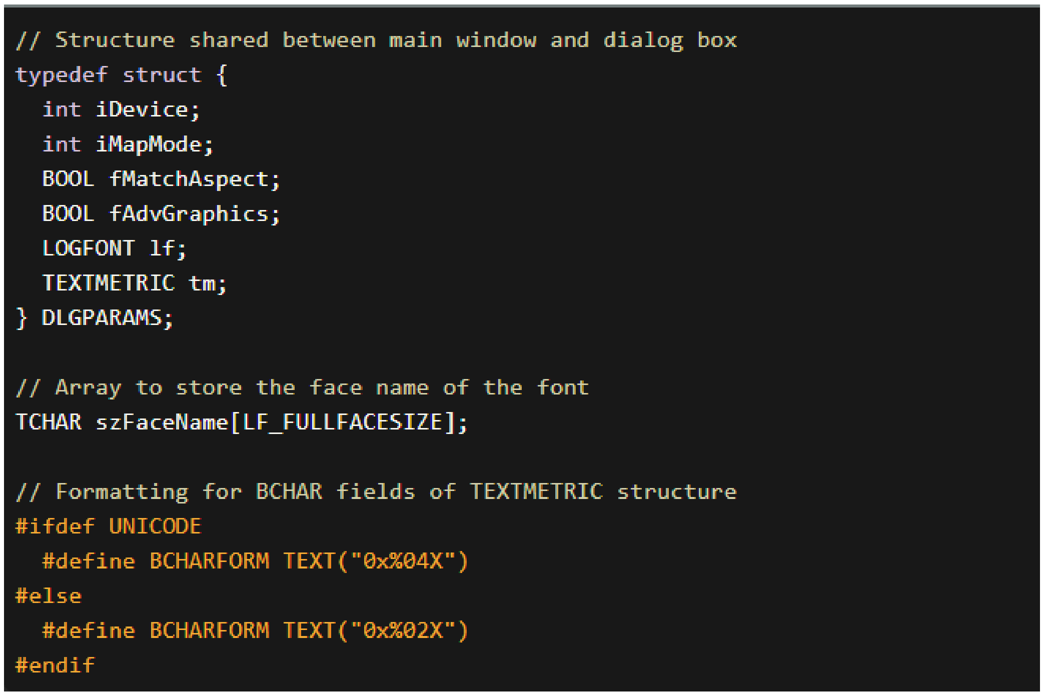
For detailed font information, the GetTextMetrics function comes into play:

```
GetTextMetrics(hdc, &textmetric);
```

PICKFONT PROGRAM

Overview of "PICKFONT.C" Program Sections:

Structure Definition:



```
// Structure shared between main window and dialog box
typedef struct {
  int iDevice;
  int iMapMode;
  BOOL fMatchAspect;
  BOOL fAdvGraphics;
  LOGFONT lf;
  TEXTMETRIC tm;
} DLGPARAMS;

// Array to store the face name of the font
TCHAR szFaceName[LF_FULLFACESIZE];

// Formatting for BCHAR fields of TEXTMETRIC structure
#ifdef UNICODE
  #define BCHARFORM TEXT("0x%04X")
#else
  #define BCHARFORM TEXT("0x%02X")
#endif
```

In the "PICKFONT.C" program, the "Structure Definition" section plays a crucial role in organizing and encapsulating the parameters necessary for handling device information, font characteristics, and flags.

The structure defined, named DLGPARAMS, serves as a container to hold diverse data elements that are essential for the program's functionality.

The DLGPARAMS structure includes several fields that collectively store information related to the font creation process.

These fields cover aspects such as the selected device (screen or printer), font attributes (like height, width, escapement, etc.), and flags indicating whether certain advanced graphics features are enabled.

Additionally, a boolean field, fMatchAspect, is present to signify whether the aspect ratio should be matched.

Furthermore, the structure features an array named szFaceName, designed to store the face name of the font.

This array is crucial for preserving the user's choice of font style and ensuring that the selected font is accurately represented in the program.

The inclusion of such a structure enhances the program's modularity and readability by consolidating related pieces of information into a single, well-defined entity.

In essence, the "Structure Definition" section establishes the blueprint for organizing and managing the program's data, promoting a clean and structured approach to handling the diverse parameters associated with font creation and display.

The use of a structured format not only facilitates ease of access and modification but also contributes to the overall clarity and maintainability of the code.

Global Variables

In the "Global Variables" section of the "PICKFONT.C" program, various global variables are declared. Let's break down each element:

hwnd (Main Window Handle):

The hwnd variable is of type HWND, which stands for "window handle." In the Windows API, a window handle is a unique identifier for a graphical window. This variable is used to store the handle of the main window created by the program.

```
HWND hwnd;
```

hdlg (Dialog Box Handle):

The hdlg variable, also of type HWND, is used to store the handle of the dialog box created in the program. This handle allows the program to interact with and control the dialog box.

```
HWND hdlg;
```

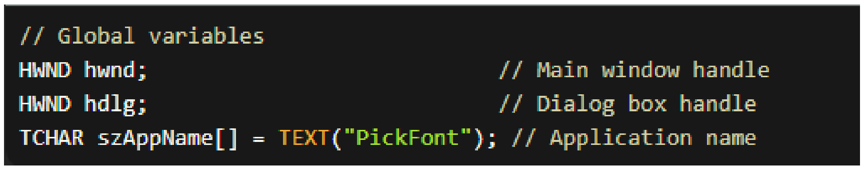
szAppName (Application Name):

The szAppName variable is an array of characters (of type TCHAR) representing the name of the application. It is initialized with the string "PickFont." The TEXT macro is used to make the string compatible with both Unicode and ANSI character sets.

```
TCHAR szAppName[] = TEXT("PickFont");
```

These global variables play essential roles in the program:

- ✓ Window Handles (hwnd and hdlg): Window handles are crucial for interacting with different parts of the graphical user interface (GUI). The main window handle (hwnd) is used to manage the main window, while the dialog box handle (hdlg) is employed for interacting with the dialog box and its components.
- ✓ Application Name (szAppName): The application name is a human-readable identifier for the program. It can be used in various places, such as window class registration, message boxes, and menu items.



```
// Global variables
HWND hwnd;                           // Main window handle
HWND hdlg;                           // Dialog box handle
TCHAR szAppName[] = TEXT("PickFont"); // Application name
```

By declaring these variables globally, they can be accessed and modified throughout the program, allowing for communication between different parts of the code, such as the main window procedure (WndProc) and the dialog box procedure (DlgProc).

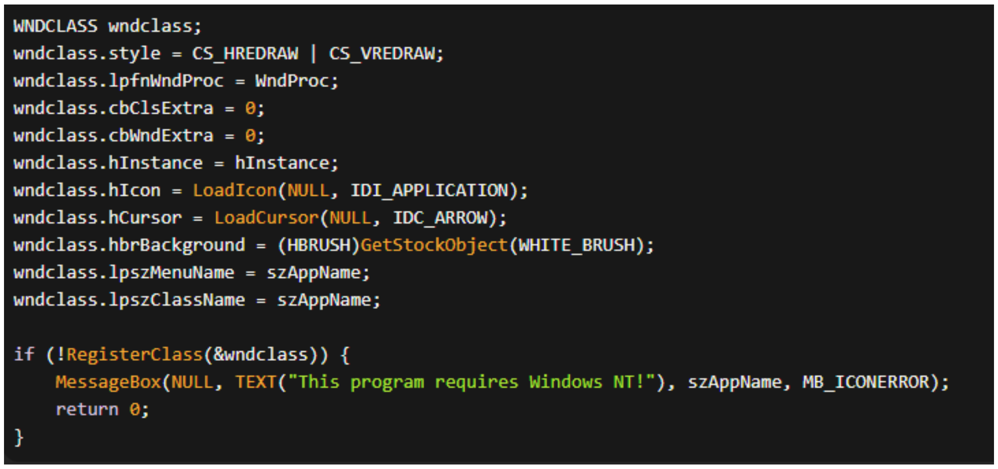
Global variables are often used in Windows programming to maintain state and share information between different components of a graphical application.

Forward Declarations: Declares forward functions, WndProc, DlgProc, SetLogFontFromFields, SetFieldsFromTextMetric, and MySetMapMode.

WinMain function:

The WinMain function serves as the entry point for the program, orchestrating the initialization, execution, and termination phases. Below is an in-depth explanation of each part of the WinMain function:

Registering Window Class:

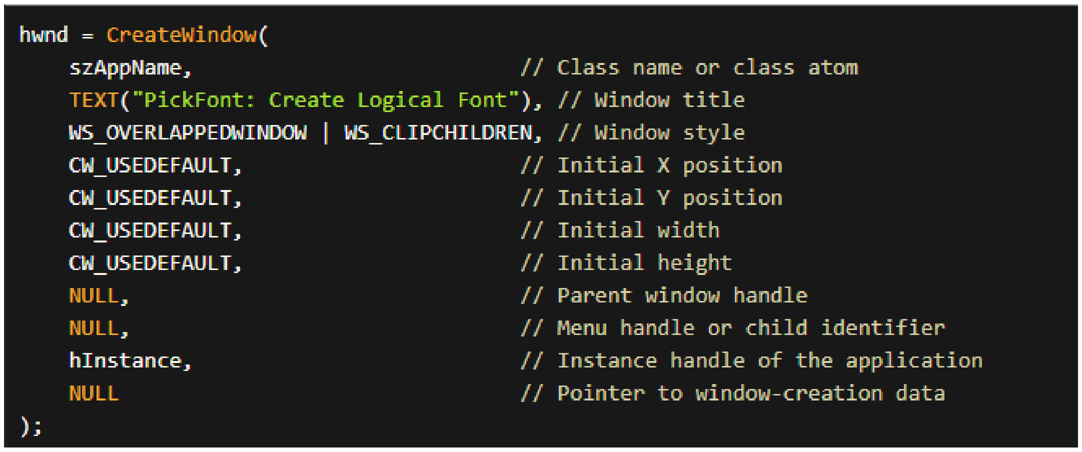


```
WNDCLASS wndclass;
wndclass.style = CS_HREDRAW | CS_VREDRAW;
wndclass.lpfnWndProc = WndProc;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = 0;
wndclass.hInstance = hInstance;
wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
wndclass.lpszMenuName = szAppName;
wndclass.lpszClassName = szAppName;

if (!RegisterClass(&wndclass)) {
    MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
    return 0;
}
```

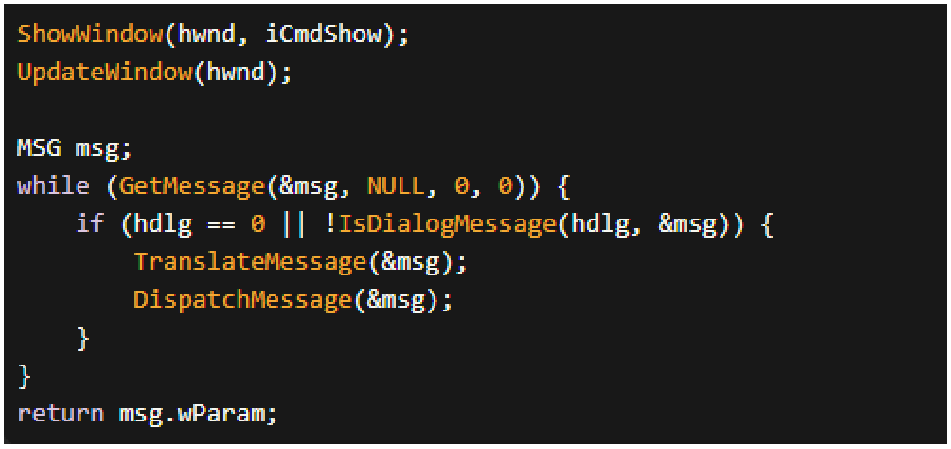
In this code snippet, a WNDCLASS structure named wndclass is defined. Its members are set as follows:

- ✓ style is set to CS_HREDRAW | CS_VREDRAW, which enables the window to be redrawn when its width or height changes.
- ✓ lpfnWndProc is set to WndProc, which is the callback function that handles window messages.
- ✓ cbClsExtra and cbWndExtra are both set to 0, indicating no extra bytes are allocated for the class and window instances.
- ✓ hInstance is set to the value of the hInstance parameter, which represents the instance handle of the application.
- ✓ hIcon is set to the icon associated with the application, loaded using LoadIcon() with NULL as the first parameter and IDI_APPLICATION as the icon identifier.
- ✓ hCursor is set to the cursor associated with the application, loaded using LoadCursor() with NULL as the first parameter and IDC_ARROW as the cursor identifier.
- ✓ hbrBackground is set to the white brush obtained from GetStockObject(WHITE_BRUSH).
- ✓ lpszMenuName is set to szAppName, which is the application name stored as a TCHAR array.
- ✓ lpszClassName is also set to szAppName.
- ✓ Finally, the code checks if the class registration was successful using RegisterClass(). If not, a message box is displayed indicating that the program requires Windows NT, and the program returns 0.



```
hwnd = CreateWindow(
    szAppName,                          // Class name or class atom
    TEXT("PickFont: Create Logical Font"), // Window title
    WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN, // Window style
    CW_USEDEFAULT,                      // Initial X position
    CW_USEDEFAULT,                      // Initial Y position
    CW_USEDEFAULT,                      // Initial width
    CW_USEDEFAULT,                      // Initial height
    NULL,                               // Parent window handle
    NULL,                               // Menu handle or child identifier
    hInstance,                          // Instance handle of the application
    NULL                                // Pointer to window-creation data
);
```

- ✓ szAppName is the name of the window class previously registered with RegisterClass.
- ✓ "PickFont: Create Logical Font" is the window title.
- ✓ WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN specifies the window style, which includes features such as a title bar, sizing border, system menu, and minimize/maximize buttons. The WS_CLIPCHILDREN style is used to ensure that child windows are not drawn outside the boundaries of the parent window.
- ✓ CW_USEDEFAULT is used for the initial position and size of the window, which allows the system to choose the default values.
- ✓ NULL specifies the parent window handle since this is a top-level window.
- ✓ NULL specifies the menu handle, as no menu is associated with the window.
- ✓ hInstance is the instance handle of the application.
- ✓ NULL is passed as the last parameter, which is reserved for future use.



```
ShowWindow(hwnd, iCmdShow);
UpdateWindow(hwnd);

MSG msg;
while (GetMessage(&msg, NULL, 0, 0)) {
    if (hdlg == 0 || !IsDialogMessage(hdlg, &msg)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
return msg.wParam;
```

- ✓ ShowWindow(hwnd, iCmdShow): Displays the main window based on the specified iCmdShow parameter (e.g., SW_SHOWNORMAL).
- ✓ UpdateWindow(hwnd): Sends a WM_PAINT message to the main window, prompting it to paint its client area.
- ✓ The subsequent while loop runs the message loop, processing messages until a WM_QUIT message is received.
- ✓ GetMessage(&msg, NULL, 0, 0): Retrieves a message from the message queue.
- ✓ If there is a dialog box (hdlg) and the message is a dialog message, it is processed by IsDialogMessage to ensure proper handling.
- ✓ TranslateMessage(&msg): Translates virtual-key messages into character messages.
- ✓ DispatchMessage(&msg): Sends the message to the window procedure (WndProc) for processing.
- ✓ The function returns when a WM_QUIT message is received, and the program exits with the wParam from that message.

WndProc function:

The WndProc function serves as the window procedure for the main window, managing various messages that the window receives during its lifetime. Let's break down the key aspects of this function in depth:

Initialization and Dialog Box Creation:

```
static DLGPARAMS dp;
static TCHAR szText[] = TEXT("\x41\x42\x43\x44\x45 ") TEXT("\x61\x62\x63\x64\x65 ")
                        TEXT("\xC0\xC1\xC2\xC3\xC4\xC5 ") TEXT("\xE0\xE1\xE2\xE3\xE4\xE5 ")
#ifdef UNICODE
                        TEXT("\x0390\x0391\x0392\x0393\x0394\x0395 ") ...
#endif
```

static DLGPARAMS dp; Declares a static variable dp of type DLGPARAMS to store various parameters, including device information, font characteristics, and flags.

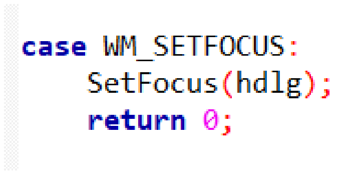
static TCHAR szText[] = TEXT("..."); Initializes an array szText with Unicode characters. The text includes both uppercase and lowercase letters, as well as characters with accent marks (for Unicode builds).

```
case WM_CREATE:
     dp.iDevice = IDM_DEVICE_SCREEN;
     hdlg = CreateDialogParam(((LPCREATESTRUCT)lParam)->hInstance, szAppName, hwnd, DlgProc, (LPARAM)&dp);
     return 0;
```

In this code snippet, the WM_CREATE message is handled. When a window receives this message, it indicates that it is being created.

Within the WM_CREATE case, the following actions are performed:

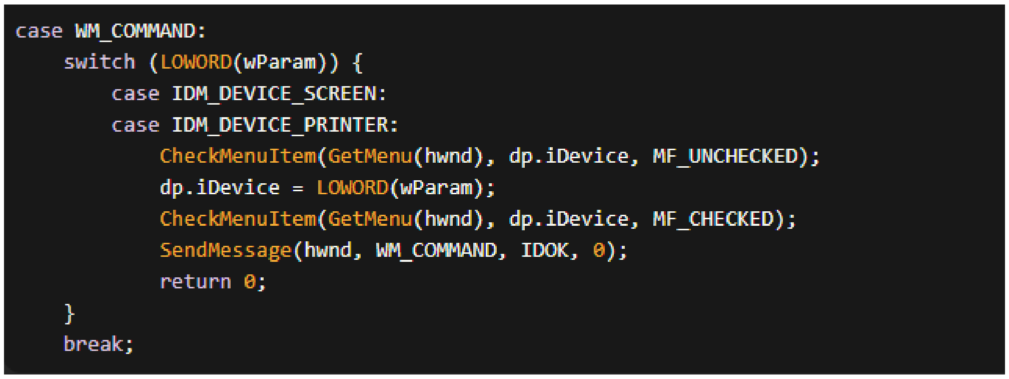
- dp.iDevice is set to IDM_DEVICE_SCREEN. It assigns a value to the iDevice member of a dp variable. The specific value is not provided in the code snippet, but it appears to be related to the device being used (possibly the screen).
- hdlg is assigned the result of the CreateDialogParam function. This function creates a modal dialog box from a dialog box template resource. The parameters passed to CreateDialogParam are:
  - ✓ ((LPCREATESTRUCT)lParam)->hInstance is the instance handle of the application, extracted from the lParam parameter of the WM_CREATE message.
  - ✓ szAppName is the name of the dialog box template resource to be loaded.
  - ✓ hwnd is the handle to the owner window of the dialog box.
  - ✓ DlgProc is a pointer to the dialog box procedure that will handle messages for the dialog box.
  - ✓ (LPARAM)&dp is a pointer to additional data that can be passed to the dialog box procedure, in this case, the address of the dp variable.
  - ✓ Finally, the function returns 0 to indicate that the WM_CREATE message has been processed.



```
case WM_SETFOCUS:
    SetFocus(hdlg);
    return 0;
```

When the window receives focus (WM_SETFOCUS), the focus is set to the dialog box (hdlg) to ensure user interaction with the controls inside the dialog.

WM_COMMAND handling:



```
case WM_COMMAND:
    switch (LOWORD(wParam)) {
        case IDM_DEVICE_SCREEN:
        case IDM_DEVICE_PRINTER:
            CheckMenuItem(GetMenu(hwnd), dp.iDevice, MF_UNCHECKED);
            dp.iDevice = LOWORD(wParam);
            CheckMenuItem(GetMenu(hwnd), dp.iDevice, MF_CHECKED);
            SendMessage(hwnd, WM_COMMAND, IDOK, 0);
            return 0;
    }
    break;
```

In this code snippet, the WM_COMMAND message is handled. This message is sent to the window procedure when the user selects a command item from a menu, presses an accelerator key, or uses a control.

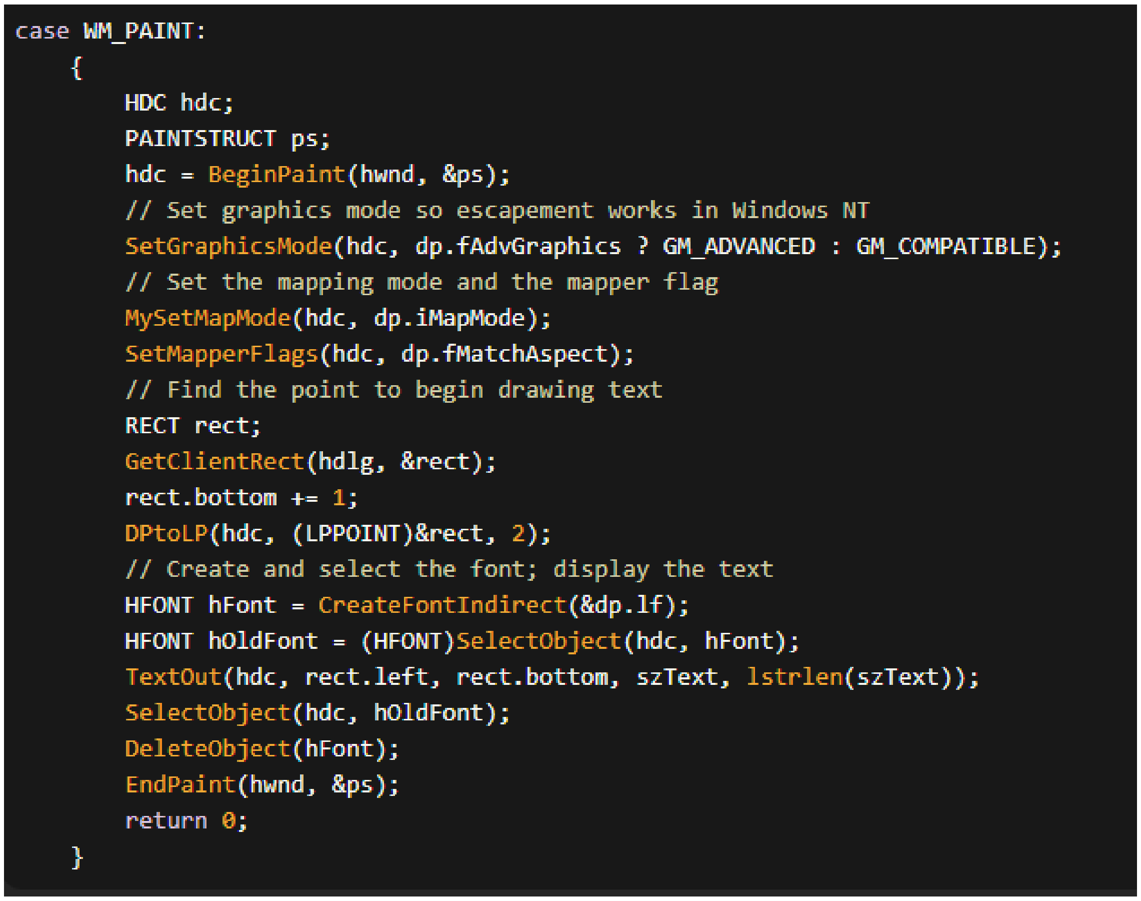
Within the WM_COMMAND case, a switch statement is used to handle different command identifiers (LOWORD(wParam)). The code snippet shows two cases: IDM_DEVICE_SCREEN and IDM_DEVICE_PRINTER, which are likely command identifiers for different device options.

For both cases, the following actions are performed:

- CheckMenuItem is called to uncheck the previously selected device option (dp.iDevice) in the menu using MF_UNCHECKED flag.
- dp.iDevice is updated with the current selected device option (LOWORD(wParam)).
- CheckMenuItem is called again to check the newly selected device option in the menu using MF_CHECKED flag.
- SendMessage is used to send a WM_COMMAND message with IDOK command identifier to the hwnd window. This is likely done to trigger further processing or actions related to the selected device option.
- Finally, the function returns 0 to indicate that the WM_COMMAND message has been processed.

Note: The code snippet provided assumes that GetMenu() and SendMessage() functions are correctly defined and accessible in the surrounding code.

WM_PAINT:



```
case WM_PAINT:
    {
        HDC hdc;
        PAINTSTRUCT ps;
        hdc = BeginPaint(hwnd, &ps);
        // Set graphics mode so escapement works in Windows NT
        SetGraphicsMode(hdc, dp.fAdvGraphics ? GM_ADVANCED : GM_COMPATIBLE);
        // Set the mapping mode and the mapper flag
        MySetMapMode(hdc, dp.iMapMode);
        SetMapperFlags(hdc, dp.fMatchAspect);
        // Find the point to begin drawing text
        RECT rect;
        GetClientRect(hdlg, &rect);
        rect.bottom += 1;
        DPtoLP(hdc, (LPPOINT)&rect, 2);
        // Create and select the font; display the text
        HFONT hFont = CreateFontIndirect(&dp.lf);
        HFONT hOldFont = (HFONT)SelectObject(hdc, hFont);
        TextOut(hdc, rect.left, rect.bottom, szText, lstrlen(szText));
        SelectObject(hdc, hOldFont);
        DeleteObject(hFont);
        EndPaint(hwnd, &ps);
        return 0;
    }
```

In this code snippet, the WM_PAINT message is handled. This message is sent to the window procedure when the window's client area needs to be repainted.

Within the WM_PAINT case, the following actions are performed:

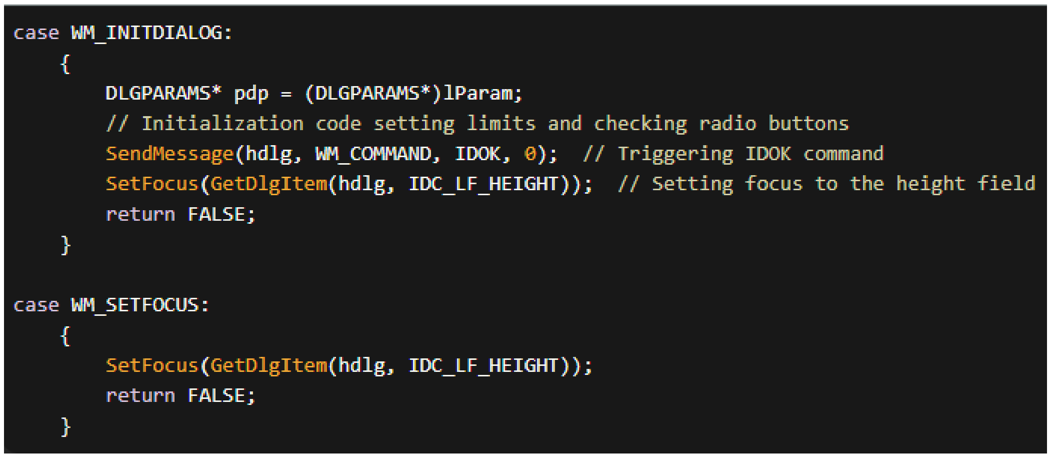
- HDC hdc and PAINTSTRUCT ps are declared to hold the device context and paint information.
- hdc is obtained by calling BeginPaint(hwnd, &ps). This prepares the device context for painting and provides the handle to the device context for subsequent drawing operations.
- The graphics mode is set using SetGraphicsMode based on the value of dp.fAdvGraphics. If dp.fAdvGraphics is true, the advanced graphics mode (GM_ADVANCED) is set; otherwise, the compatible graphics mode (GM_COMPATIBLE) is set.
- The mapping mode and the mapper flags are set using MySetMapMode and SetMapperFlags, respectively, based on the values of dp.iMapMode and dp.fMatchAspect.
- The starting point for drawing text is determined by getting the client rectangle of hdlg (the dialog box) using GetClientRect, and then converting the coordinates from device units to logical units using DPtoLP.
- A font is created using CreateFontIndirect with dp.lf (a LOGFONT structure) as the parameter. The font is then selected into the device context using SelectObject.
- The text is drawn using TextOut, specifying the starting point and the text string (szText).
- The original font is restored by selecting it back into the device context using SelectObject, and the created font is deleted using DeleteObject.
- The painting is finalized by calling EndPaint to release the device context and indicate that the painting is complete.
- Finally, the function returns 0 to indicate that the WM_PAINT message has been processed.

Note: The code snippet assumes that MySetMapMode is a user-defined function for setting the mapping mode and szText is a null-terminated string containing the text to be displayed.

DlgProc function:

The DlgProc function serves as the window procedure for the dialog box, responsible for handling messages specific to the dialog's operation. This function plays a crucial role in managing the initialization, user input, and updating font information based on the user's selections. Let's delve into the various aspects of the DlgProc function in depth:

Initialization and User Input Handling:



```
case WM_INITDIALOG:
    {
        DLGPARAMS* pdp = (DLGPARAMS*)lParam;
        // Initialization code setting limits and checking radio buttons
        SendMessage(hdlg, WM_COMMAND, IDOK, 0);  // Triggering IDOK command
        SetFocus(GetDlgItem(hdlg, IDC_LF_HEIGHT));  // Setting focus to the height field
        return FALSE;
    }

case WM_SETFOCUS:
    {
        SetFocus(GetDlgItem(hdlg, IDC_LF_HEIGHT));
        return FALSE;
    }
```

In the WM_INITDIALOG case, the following actions are performed:

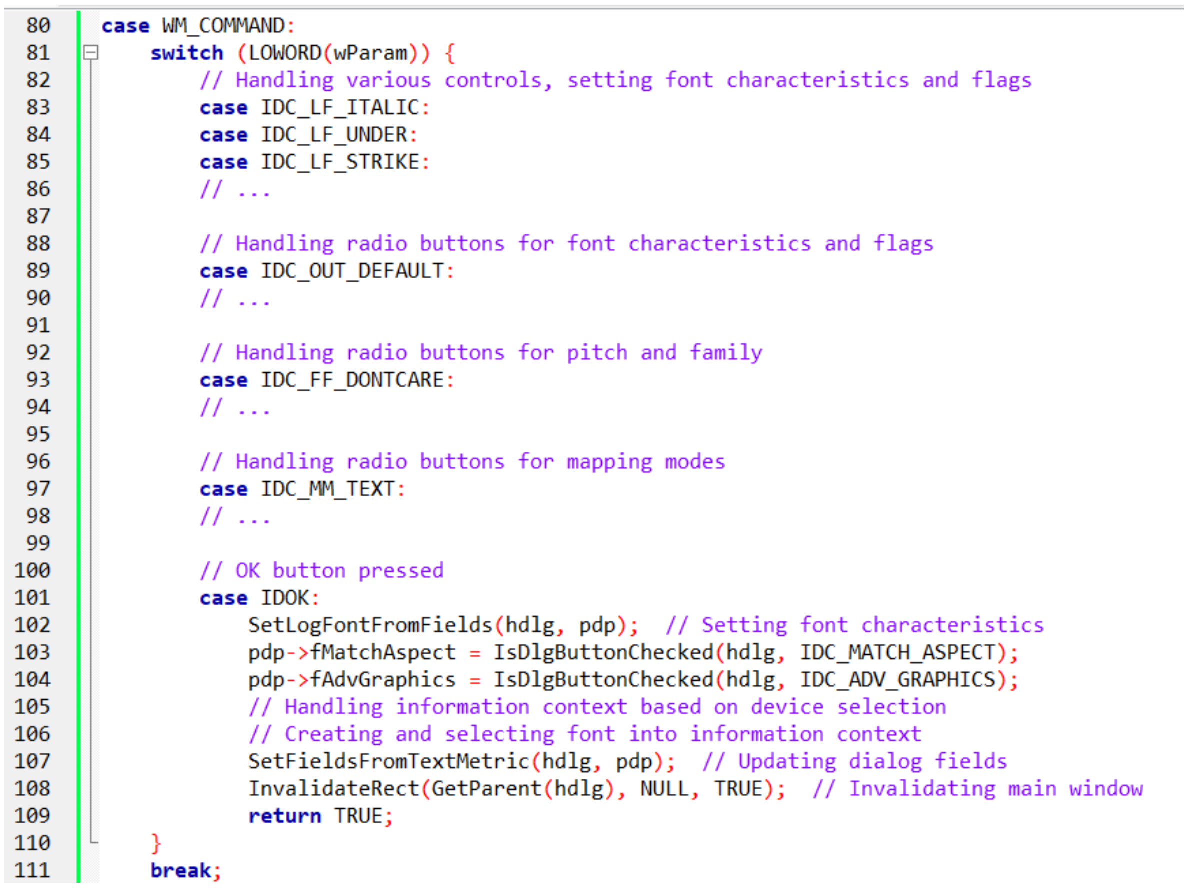
- The lParam parameter is cast to a pointer of type DLGPARAMS* and assigned to the variable pdp.
- Initialization code is executed, which likely includes setting limits and checking radio buttons.
- SendMessage is used to trigger a WM_COMMAND message with IDOK command identifier to the hdlg dialog box window. This is done to simulate the user clicking the OK button programmatically.
- SetFocus is called to set the input focus to the control with the ID IDC_LF_HEIGHT. It retrieves the handle of the control using GetDlgItem and passes it as the parameter to SetFocus.
- Finally, FALSE is returned to allow the system to set the focus to the control specified by SetFocus.

In the WM_SETFOCUS case, the following actions are performed:

- SetFocus is called to set the input focus to the control with the ID IDC_LF_HEIGHT. It retrieves the handle of the control using GetDlgItem and passes it as the parameter to SetFocus.
- Finally, FALSE is returned to allow the system to set the focus to the control specified by SetFocus.

Note: The code snippet assumes that IDC_LF_HEIGHT is the identifier of a control, such as an edit control, within the dialog box.

Handling WM_COMMAND Message - Font Characteristics and Flags:



```
case WM_COMMAND:
    switch (LOWORD(wParam)) {
        // Handling various controls, setting font characteristics and flags
        case IDC_LF_ITALIC:
        case IDC_LF_UNDER:
        case IDC_LF_STRIKE:
        // ...

        // Handling radio buttons for font characteristics and flags
        case IDC_OUT_DEFAULT:
        // ...

        // Handling radio buttons for pitch and family
        case IDC_FF_DONTCARE:
        // ...

        // Handling radio buttons for mapping modes
        case IDC_MM_TEXT:
        // ...

        // OK button pressed
        case IDOK:
            SetLogFontFromFields(hdlg, pdp);  // Setting font characteristics
            pdp->fMatchAspect = IsDlgButtonChecked(hdlg, IDC_MATCH_ASPECT);
            pdp->fAdvGraphics = IsDlgButtonChecked(hdlg, IDC_ADV_GRAPHICS);
            // Handling information context based on device selection
            // Creating and selecting font into information context
            SetFieldsFromTextMetric(hdlg, pdp);  // Updating dialog fields
            InvalidateRect(GetParent(hdlg), NULL, TRUE);  // Invalidating main window
            return TRUE;
    }
    break;
```

The WM_COMMAND message is extensively handled for various controls, including checkboxes and radio buttons, allowing users to specify font characteristics and flags.

The state of these controls is checked, and the corresponding members of the DLGPARAMS structure are updated.

When the user presses the "OK" button, the SetLogFontFromFields function is invoked to set font characteristics, and flags are updated based on user input.

The information context is handled based on the selected device, and a font is created and selected into the information context.

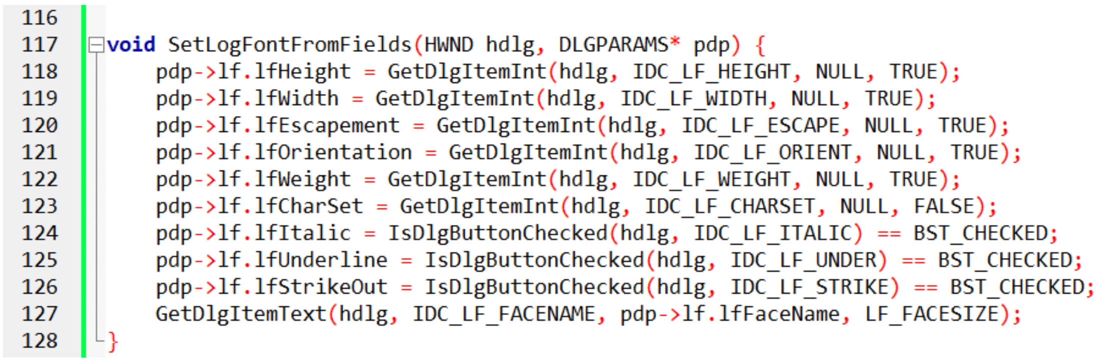
The SetFieldsFromTextMetric function is called to update the dialog fields with information obtained from the text metric.

Finally, the main window is invalidated to ensure a redraw and display of the updated font information.

In summary, the DlgProc function is pivotal in managing the initialization, user input, and updating of font information in the dialog box. It responds to user actions, sets font characteristics and flags, and ensures proper interaction with the main window through the invocation of relevant functions.

SetLogFontFromFields Function:

- ✓ Extracts font attributes from dialog box fields and updates the LOGFONT structure in DLGPARAMS.



```

void SetLogFontFromFields(HWND hdlg, DLGPARAMS* pdp) {
    pdp->lf.lfHeight = GetDlgItemInt(hdlg, IDC_LF_HEIGHT, NULL, TRUE);
    pdp->lf.lfWidth = GetDlgItemInt(hdlg, IDC_LF_WIDTH, NULL, TRUE);
    pdp->lf.lfEscapement = GetDlgItemInt(hdlg, IDC_LF_ESCAPE, NULL, TRUE);
    pdp->lf.lfOrientation = GetDlgItemInt(hdlg, IDC_LF_ORIENT, NULL, TRUE);
    pdp->lf.lfWeight = GetDlgItemInt(hdlg, IDC_LF_WEIGHT, NULL, TRUE);
    pdp->lf.lfCharSet = GetDlgItemInt(hdlg, IDC_LF_CHARSET, NULL, FALSE);
    pdp->lf.lfItalic = IsDlgButtonChecked(hdlg, IDC_LF_ITALIC) == BST_CHECKED;
    pdp->lf.lfUnderline = IsDlgButtonChecked(hdlg, IDC_LF_UNDER) == BST_CHECKED;
    pdp->lf.lfStrikeOut = IsDlgButtonChecked(hdlg, IDC_LF_STRIKE) == BST_CHECKED;
    GetDlgItemText(hdlg, IDC_LF_FACENAME, pdp->lf.lfFaceName, LF_FACESIZE);
}
```

SetLogFontFromFields takes two parameters: the handle to the dialog box (hdlg) and a pointer to the DLGPARAMS structure (pdp).

The function utilizes various Windows API functions to retrieve information from the dialog box controls and update the corresponding members of the LOGFONT structure within the DLGPARAMS.

GetDlgItemInt is used to retrieve integer values from specified controls such as height, width, escapement, orientation, weight, and character set.

IsDlgButtonChecked is employed to check the state of checkboxes for italic, underline, and strikeout attributes.

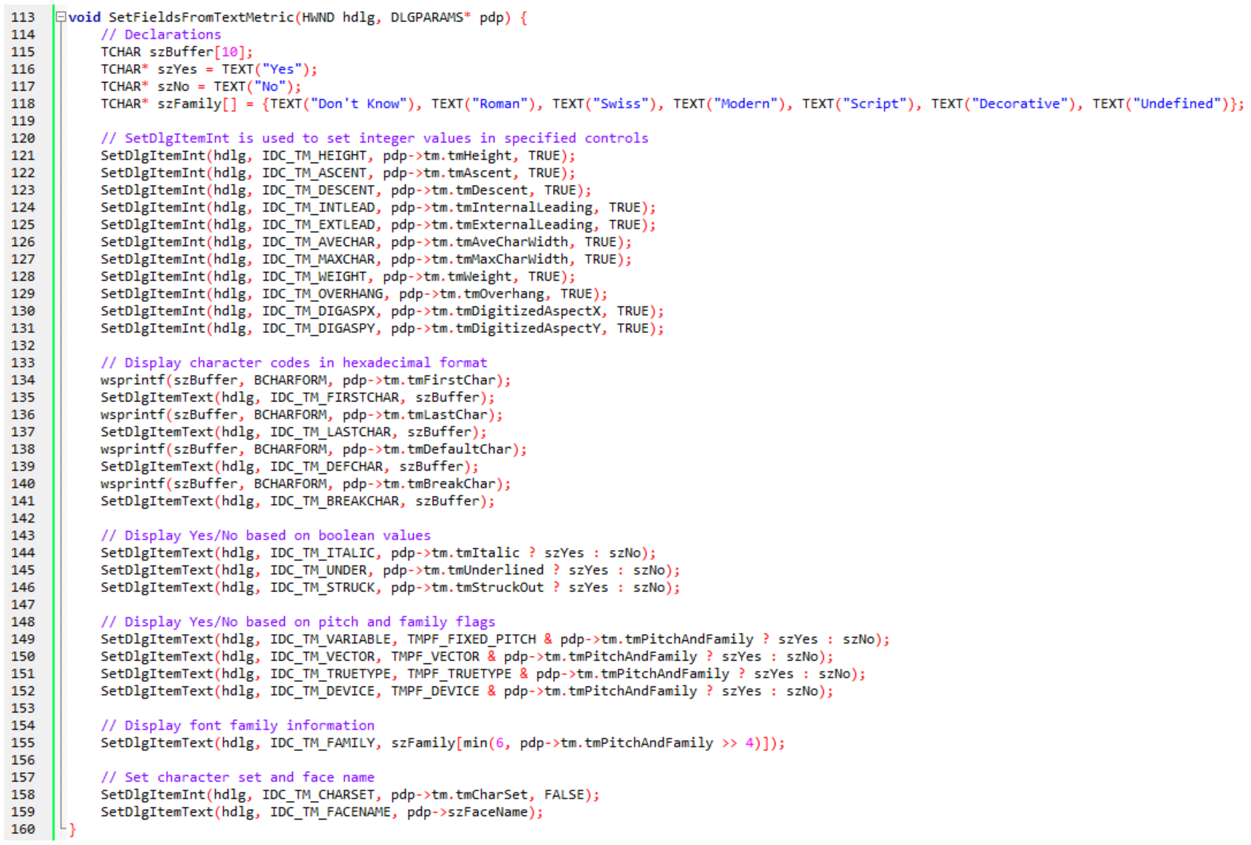
GetDlgItemText is used to obtain the face name of the font from the corresponding edit control.

The obtained values are then assigned to the respective members of the LOGFONT structure (pdp->lf), updating it with the user's input from the dialog box fields.

This function essentially bridges the user interface with the internal representation of font attributes. It allows the program to dynamically capture the user's preferences from the dialog box and reflect those preferences in the LOGFONT structure, facilitating the subsequent creation and manipulation of fonts in the application

SetFieldsFromTextMetric Function:

- ✓ Updates dialog box fields with information obtained from the TEXTMETRIC structure in DLGPARAMS.



```
void SetFieldsFromTextMetric(HWND hdlg, DLGPARAMS* pdp) {
    // Declarations
    TCHAR szBuffer[10];
    TCHAR* szYes = TEXT("Yes");
    TCHAR* szNo = TEXT("No");
    TCHAR* szFamily[] = {TEXT("Don't Know"), TEXT("Roman"), TEXT("Swiss"), TEXT("Modern"), TEXT("Script"), TEXT("Decorative"), TEXT("Undefined")};

    // SetDlgItemInt is used to set integer values in specified controls
    SetDlgItemInt(hdlg, IDC_TM_HEIGHT, pdp->tm.tmHeight, TRUE);
    SetDlgItemInt(hdlg, IDC_TM_ASCENT, pdp->tm.tmAscent, TRUE);
    SetDlgItemInt(hdlg, IDC_TM_DESCENT, pdp->tm.tmDescent, TRUE);
    SetDlgItemInt(hdlg, IDC_TM_INTLEAD, pdp->tm.tmInternalLeading, TRUE);
    SetDlgItemInt(hdlg, IDC_TM_EXTLEAD, pdp->tm.tmExternalLeading, TRUE);
    SetDlgItemInt(hdlg, IDC_TM_AVECHAR, pdp->tm.tmAveCharWidth, TRUE);
    SetDlgItemInt(hdlg, IDC_TM_MAXCHAR, pdp->tm.tmMaxCharWidth, TRUE);
    SetDlgItemInt(hdlg, IDC_TM_WEIGHT, pdp->tm.tmWeight, TRUE);
    SetDlgItemInt(hdlg, IDC_TM_OVERHANG, pdp->tm.tmOverhang, TRUE);
    SetDlgItemInt(hdlg, IDC_TM_DIGASPX, pdp->tm.tmDigitizedAspectX, TRUE);
    SetDlgItemInt(hdlg, IDC_TM_DIGASPY, pdp->tm.tmDigitizedAspectY, TRUE);

    // Display character codes in hexadecimal format
    wsprintf(szBuffer, BCHARFORM, pdp->tm.tmFirstChar);
    SetDlgItemText(hdlg, IDC_TM_FIRSTCHAR, szBuffer);
    wsprintf(szBuffer, BCHARFORM, pdp->tm.tmLastChar);
    SetDlgItemText(hdlg, IDC_TM_LASTCHAR, szBuffer);
    wsprintf(szBuffer, BCHARFORM, pdp->tm.tmDefaultChar);
    SetDlgItemText(hdlg, IDC_TM_DEFCHAR, szBuffer);
    wsprintf(szBuffer, BCHARFORM, pdp->tm.tmBreakChar);
    SetDlgItemText(hdlg, IDC_TM_BREAKCHAR, szBuffer);

    // Display Yes/No based on boolean values
    SetDlgItemText(hdlg, IDC_TM_ITALIC, pdp->tm.tmItalic ? szYes : szNo);
    SetDlgItemText(hdlg, IDC_TM_UNDER, pdp->tm.tmUnderlined ? szYes : szNo);
    SetDlgItemText(hdlg, IDC_TM_STRUCK, pdp->tm.tmStruckOut ? szYes : szNo);

    // Display Yes/No based on pitch and family flags
    SetDlgItemText(hdlg, IDC_TM_VARIABLE, TMPF_FIXED_PITCH & pdp->tm.tmPitchAndFamily ? szYes : szNo);
    SetDlgItemText(hdlg, IDC_TM_VECTOR, TMPF_VECTOR & pdp->tm.tmPitchAndFamily ? szYes : szNo);
    SetDlgItemText(hdlg, IDC_TM_TRUETYPE, TMPF_TRUETYPE & pdp->tm.tmPitchAndFamily ? szYes : szNo);
    SetDlgItemText(hdlg, IDC_TM_DEVICE, TMPF_DEVICE & pdp->tm.tmPitchAndFamily ? szYes : szNo);

    // Display font family information
    SetDlgItemText(hdlg, IDC_TM_FAMILY, szFamily[min(6, pdp->tm.tmPitchAndFamily >> 4)]);

    // Set character set and face name
    SetDlgItemInt(hdlg, IDC_TM_CHARSET, pdp->tm.tmCharSet, FALSE);
    SetDlgItemText(hdlg, IDC_TM_FACENAME, pdp->szFaceName);
}
```

Comprehensive Font Information Display: This function meticulously extracts and presents a wide range of font metrics within a designated dialog box, empowering users with granular insights into font characteristics.

Parameter Reception:

- Accepts a dialog box handle (hdlg) to specify the target interface for information display.
- Receives a pointer (pdp) to a DLGPARAMS structure, housing essential font data and configuration settings.

Data Preparation:

- Initializes variables for temporary storage and text formatting, ensuring efficient data manipulation.
- Constructs arrays to facilitate clear presentation of Boolean values and font family names, enhancing readability and comprehension.

Integer Field Population:

Leverages the SetDlgItemInt function to meticulously populate dialog box controls with integer values extracted from the TEXTMETRIC structure. This encompasses a multitude of font attributes, including:

- Height
- Ascent
- Descent
- Leading
- Average and maximum character widths
- Weight
- Overhang
- Aspect ratio
- Character Code Visualization:

Employs the wsprintf function to meticulously format character codes (first, last, default, and break) as hexadecimal strings, fostering clarity and precision in their representation.

Strategically utilizes SetDlgItemText to seamlessly integrate these formatted strings into designated dialog box controls, fostering effortless interpretation.

Boolean Property Indication:

- ❖ Deftly employs conditional logic to accurately display "Yes" or "No" values for italic, underlined, and struck-out font attributes, drawing upon boolean values within the TEXTMETRIC structure. This visual representation empowers users to readily discern these essential font characteristics.

Pitch and Family Illumination:

- ❖ The code examines certain flags within the tmPitchAndFamily field to determine the type of font. It distinguishes between fixed pitch, vector, TrueType, and device fonts based on these flags. By doing so, it can identify and categorize different font types accurately.
- ❖ The code displays "Yes" or "No" in the dialog box to indicate certain characteristics and capabilities of the font. This provides easy-to-understand information about the font's classification. Users can quickly see whether the font is fixed pitch, vector, TrueType, or a device font by looking at the displayed "Yes" or "No" values.

Font Family Exhibition:

- ❖ Meticulously extracts font family information from the tmPitchAndFamily field, ensuring accurate identification.
- ❖ Selects and displays the corresponding string from a predefined array, enhancing user comprehension and facilitating informed decision-making.

Character Set and Face Name Specification:

- ❖ Populates dialog box controls with the character set value, as obtained from the tmCharSet field, providing essential context for text rendering and interpretation.
- ❖ Presents the font face name, retrieved from the szFaceName array, enabling users to effortlessly recognize and manage font choices.

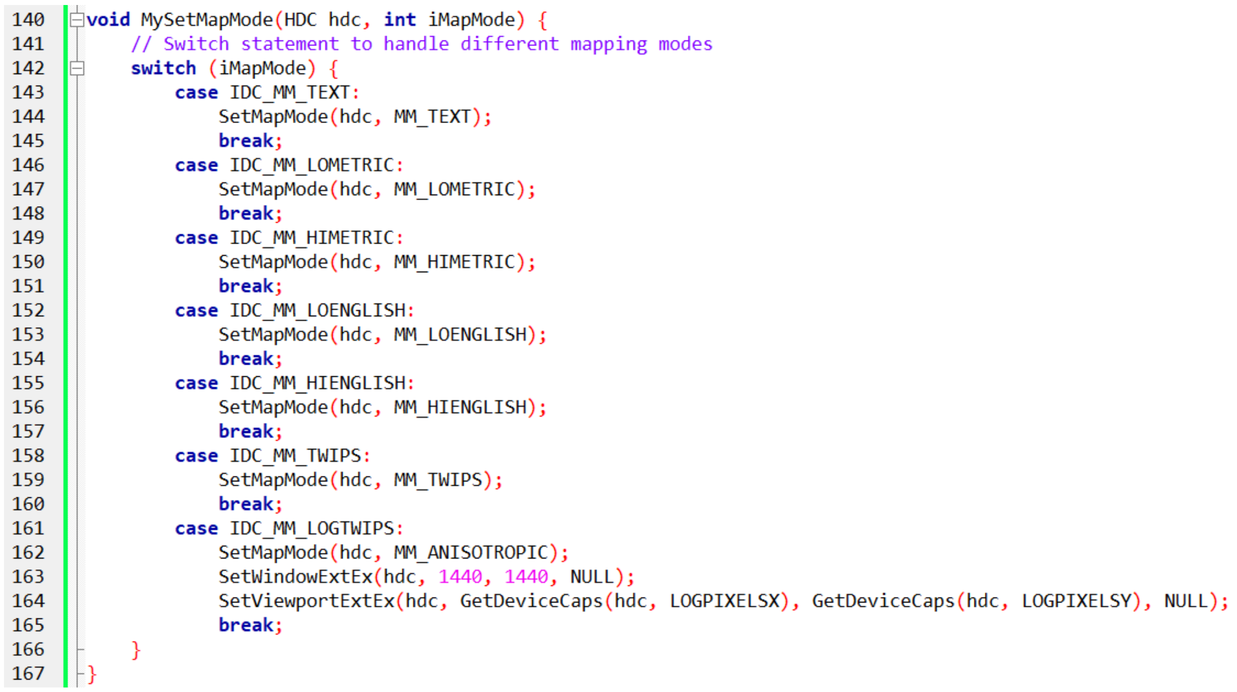
Overall Significance:

This function acts as an important link between the technical font metrics stored in the TEXTMETRIC structure and their user-friendly presentation in a dialog box. It helps bridge the gap between complex font data and a visually understandable format that users can easily interact with.

By providing a visual representation of various font characteristics, this function empowers users to make informed decisions and adjustments related to the appearance of text. Users can better understand and choose fonts that optimize readability and visual appeal, leading to enhanced text presentation.

MySetMapMode Function:

- ✓ Sets the mapping mode for the device context based on the user's selection in the dialog box.



```
void MySetMapMode(HDC hdc, int iMapMode) {
    // Switch statement to handle different mapping modes
    switch (iMapMode) {
        case IDC_MM_TEXT:
            SetMapMode(hdc, MM_TEXT);
            break;
        case IDC_MM_LOMETRIC:
            SetMapMode(hdc, MM_LOMETRIC);
            break;
        case IDC_MM_HIMETRIC:
            SetMapMode(hdc, MM_HIMETRIC);
            break;
        case IDC_MM_LOENGLISH:
            SetMapMode(hdc, MM_LOENGLISH);
            break;
        case IDC_MM_HIENGLISH:
            SetMapMode(hdc, MM_HIENGLISH);
            break;
        case IDC_MM_TWIPS:
            SetMapMode(hdc, MM_TWIPS);
            break;
        case IDC_MM_LOGTWIPS:
            SetMapMode(hdc, MM_ANISOTROPIC);
            SetWindowExtEx(hdc, 1440, 1440, NULL);
            SetViewportExtEx(hdc, GetDeviceCaps(hdc, LOGPIXELSX), GetDeviceCaps(hdc, LOGPIXELSY), NULL);
            break;
    }
}
```

MySetMapMode takes two parameters: the handle to the device context (hdc) and the mapping mode selected by the user (iMapMode).

It uses a switch statement to handle different mapping modes based on the value of iMapMode.

For each case, the function calls the SetMapMode function with the appropriate mapping mode constant.

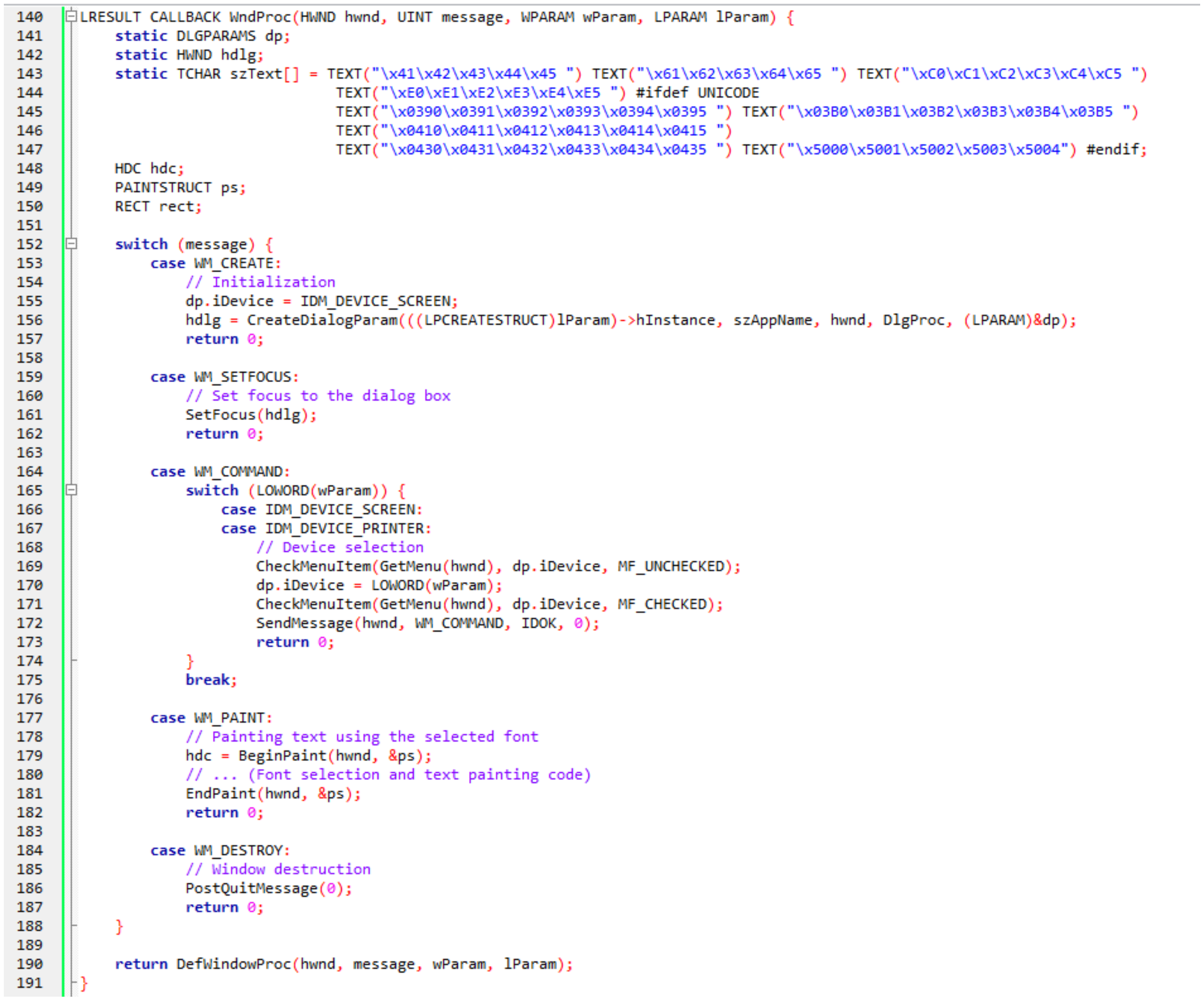
If the mapping mode is set to IDC_MM_LOGTWIPS, an additional configuration is done using the SetWindowExtEx and SetViewportExtEx functions. In this case, the mapping is set to anisotropic, and the window and viewport extents are adjusted to represent twips.

The SetMapMode function establishes the mapping mode for the device context, influencing how graphical elements are scaled and positioned.

This function plays a crucial role in ensuring that the graphical elements, especially text, are displayed in the desired manner on the device context. The mapping mode is a fundamental concept in graphics programming, allowing applications to control the scaling and units used for drawing operations. In this context, the function facilitates the adaptation of text rendering based on the user's preferences, enhancing the overall flexibility and user experience of the application.

Message Handling:

Handles various messages, such as initialization, focus, command input, painting, and destruction of windows and dialog boxes.



```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    static DLGPARAMS dp;
    static HWND hdlg;
    static TCHAR szText[] = TEXT("\x41\x42\x43\x44\x45 ") TEXT("\x61\x62\x63\x64\x65 ") TEXT("\xC0\xC1\xC2\xC3\xC4\xC5 ")
                             TEXT("\xE0\xE1\xE2\xE3\xE4\xE5 ") #ifdef UNICODE
                             TEXT("\x0390\x0391\x0392\x0393\x0394\x0395 ") TEXT("\x03B0\x03B1\x03B2\x03B3\x03B4\x03B5 ")
                             TEXT("\x0410\x0411\x0412\x0413\x0414\x0415 ")
                             TEXT("\x0430\x0431\x0432\x0433\x0434\x0435 ") TEXT("\x5000\x5001\x5002\x5003\x5004") #endif;
    HDC hdc;
    PAINTSTRUCT ps;
    RECT rect;

    switch (message) {
        case WM_CREATE:
            // Initialization
            dp.iDevice = IDM_DEVICE_SCREEN;
            hdlg = CreateDialogParam(((LPCREATESTRUCT)lParam)->hInstance, szAppName, hwnd, DlgProc, (LPARAM)&dp);
            return 0;

        case WM_SETFOCUS:
            // Set focus to the dialog box
            SetFocus(hdlg);
            return 0;

        case WM_COMMAND:
            switch (LOWORD(wParam)) {
                case IDM_DEVICE_SCREEN:
                case IDM_DEVICE_PRINTER:
                    // Device selection
                    CheckMenuItem(GetMenu(hwnd), dp.iDevice, MF_UNCHECKED);
                    dp.iDevice = LOWORD(wParam);
                    CheckMenuItem(GetMenu(hwnd), dp.iDevice, MF_CHECKED);
                    SendMessage(hwnd, WM_COMMAND, IDOK, 0);
                    return 0;
            }
            break;

        case WM_PAINT:
            // Painting text using the selected font
            hdc = BeginPaint(hwnd, &ps);
            // ... (Font selection and text painting code)
            EndPaint(hwnd, &ps);
            return 0;

        case WM_DESTROY:
            // Window destruction
            PostQuitMessage(0);
            return 0;
    }

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

In the PICKFONT program, the message handling section of the WndProc function is responsible for handling various messages related to the main window. It includes initialization, setting focus, command handling, and window destruction.

Initialization (WM_CREATE): The WM_CREATE message initializes the DLGPARAMS structure and creates the dialog box using CreateDialogParam. It prepares the default logical font and gathers information about the device and font characteristics.

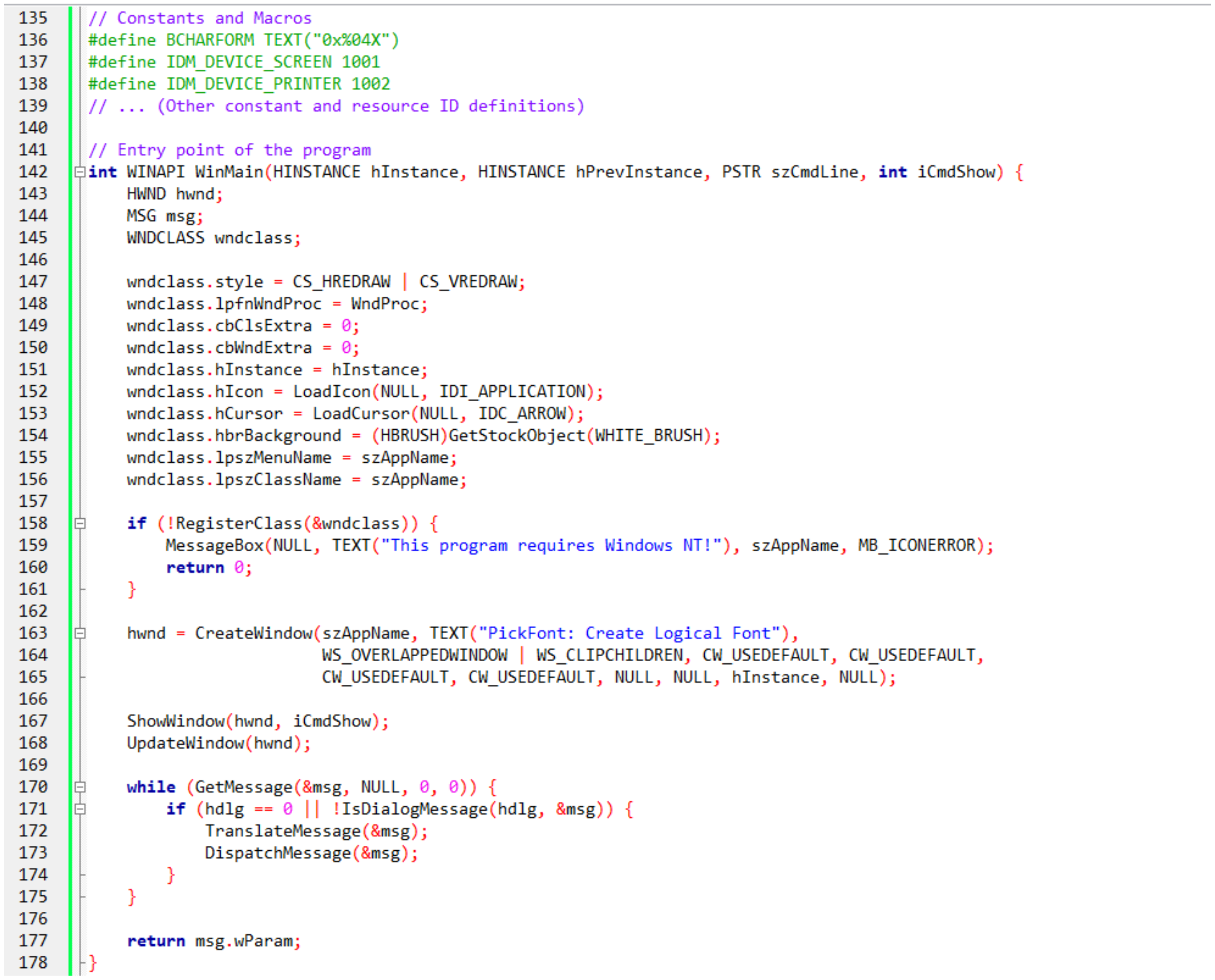
Setting Focus (WM_SETFOCUS): The WM_SETFOCUS message ensures that the dialog box receives focus when the main window gains focus. This helps maintain proper interaction with the dialog box controls.

Command Handling (WM_COMMAND): The WM_COMMAND message handles commands from menu items or controls. It identifies the source of the command based on the command identifier. For device selection, it updates the DLGPARAMS structure and triggers font creation and display. The WM_PAINT message is also handled to paint text on the main window using the selected font.

Window Destruction (WM_DESTROY): When the main window is being destroyed, the program posts a quit message to exit the message loop and terminate the program gracefully.

Constants and Macros:

Defines constants, resource IDs, and formatting macros used in the program. Concerning constants and macros, they provide meaningful identifiers and formatting options in the program:



```
// Constants and Macros
#define BCHARFORM TEXT("0x%04X")
#define IDM_DEVICE_SCREEN 1001
#define IDM_DEVICE_PRINTER 1002
// ... (Other constant and resource ID definitions)

// Entry point of the program
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) {
    HWND hwnd;
    MSG msg;
    WNDCLASS wndclass;

    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = WndProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = hInstance;
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName = szAppName;
    wndclass.lpszClassName = szAppName;

    if (!RegisterClass(&wndclass)) {
        MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
        return 0;
    }

    hwnd = CreateWindow(szAppName, TEXT("PickFont: Create Logical Font"),
                        WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN, CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL, hInstance, NULL);

    ShowWindow(hwnd, iCmdShow);
    UpdateWindow(hwnd);

    while (GetMessage(&msg, NULL, 0, 0)) {
        if (hdlg == 0 || !IsDialogMessage(hdlg, &msg)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }

    return msg.wParam;
}
```

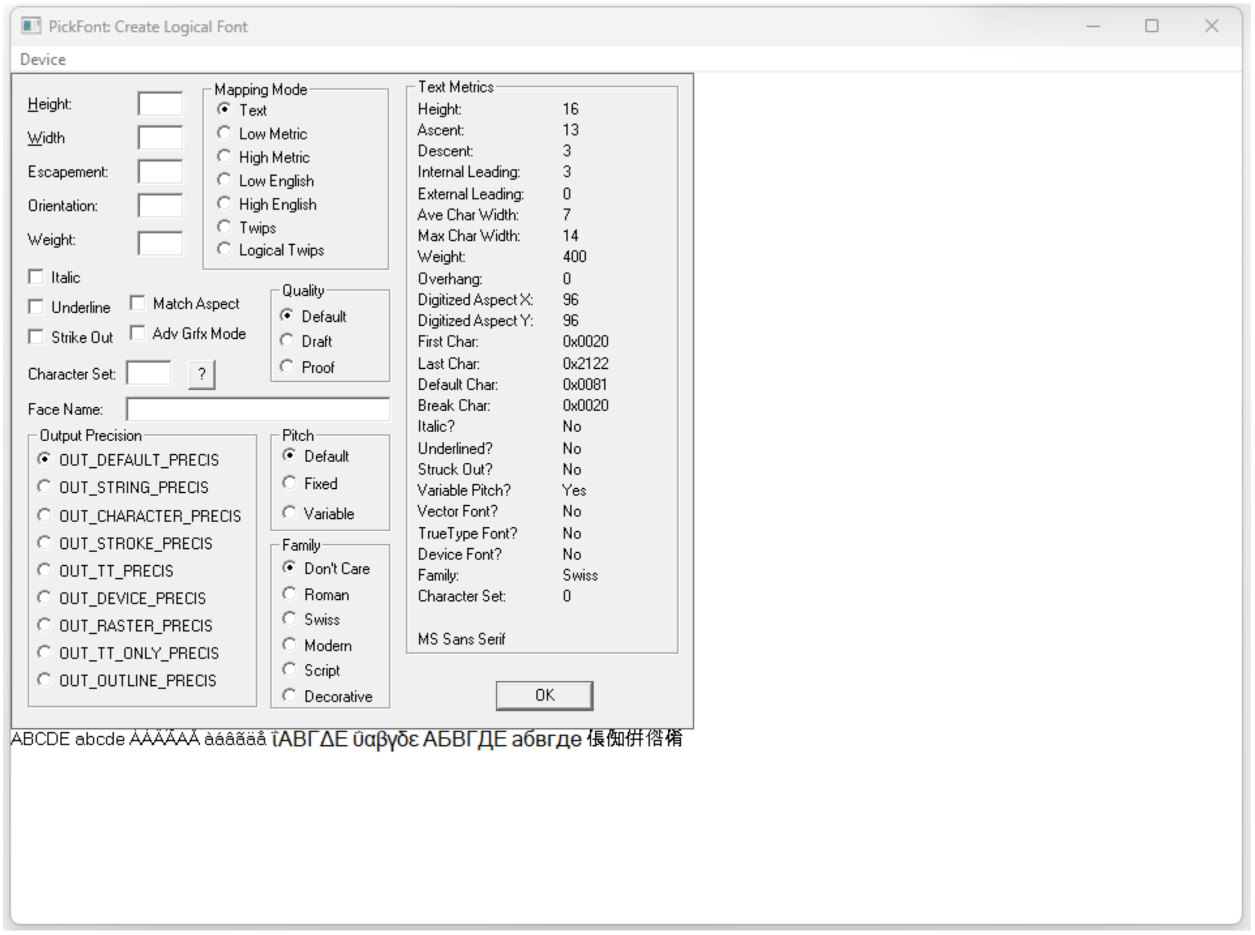
Resource IDs: Constants like IDM_DEVICE_SCREEN and IDM_DEVICE_PRINTER represent menu items for selecting the device type. Other constants, such as IDOK and IDC_LF_HEIGHT, serve as control IDs to identify specific controls in the dialog box.

Formatting Macros: The BCHARFORM macro is used to display the BYTE character in a hexadecimal format. It ensures consistent formatting, and its definition depends on whether UNICODE is enabled.

The message handling section in the PICKFONT program facilitates font customization through the dialog box, while constants and macros enhance code readability and maintainability by providing clear identifiers and formatting options.

The program's main objective is to create a logical font based on user input and display it on the main window. The dialog box enables users to customize font attributes and select the device type. Overall, the program emphasizes the interaction between the main window and the dialog box for font creation and display.

The program described at once….



Height:
Width
Escapement:
Orientation:
Weight:

☐ Italic
☐ Underline
☐ Strike Out
☐ Match Aspect
☐ Adv Grfx Mode

Character Set:
Face Name:

Mapping Mode: [✓ Text] Low Metric, High Metric, Low English, High English, Twips, Logical Twips

Quality: [✓ Default] Draft, Proof

Output Precision: [✓ OUT_DEFAULT_PRECIS] OUT_STRING_PRECIS, OUT_CHARACTER_PRECIS, OUT_STROKE_PRECIS, OUT_TT_PRECIS, OUT_DEVICE_PRECIS, OUT_RASTER_PRECIS, OUT_TT_ONLY_PRECIS, OUT_OUTLINE_PRECIS

Pitch: [✓ Default] Fixed, Variable

Family: [✓ Don't Care] Roman, Swiss, Modern, Script, Decorative

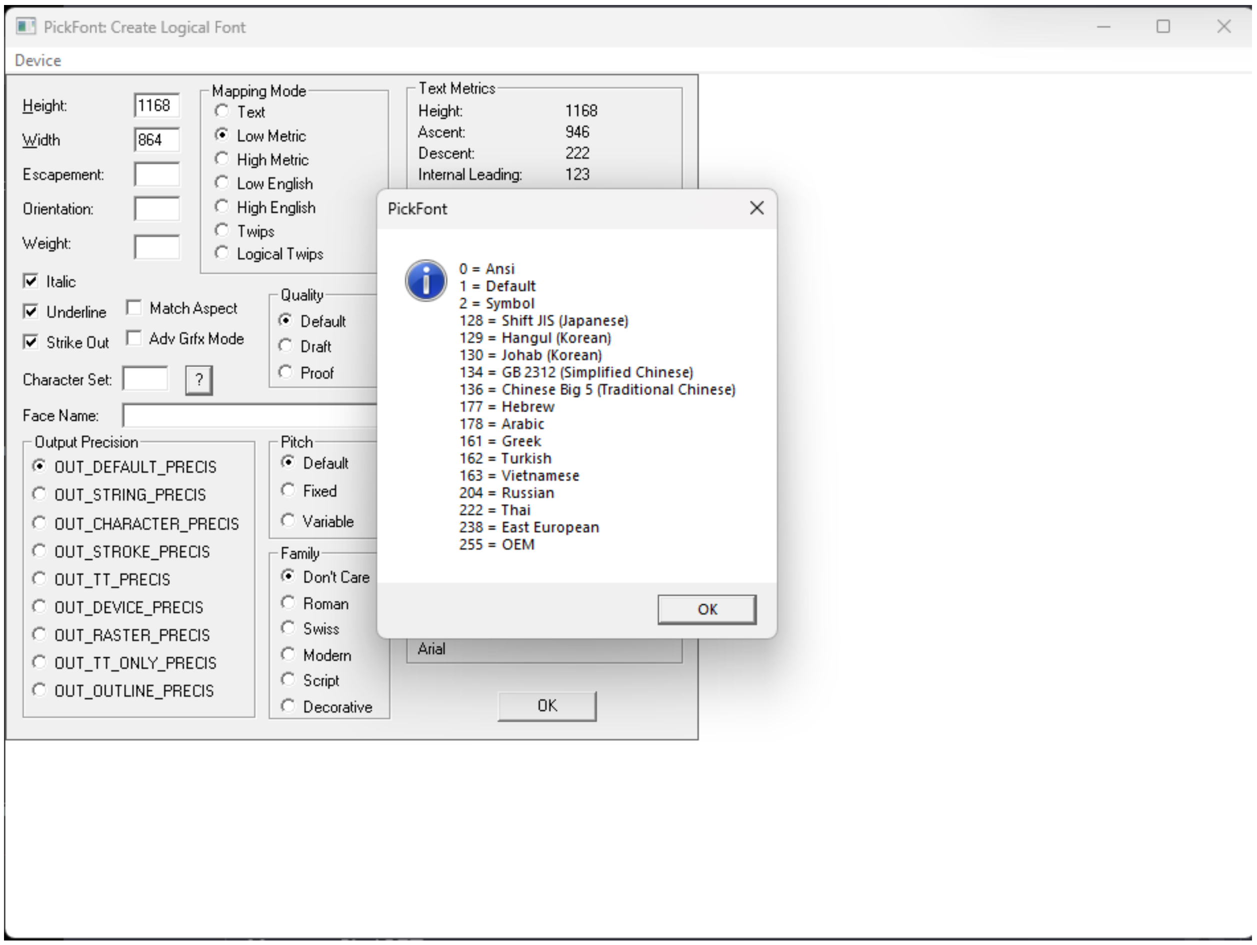
Text Metrics

Height:	16
Ascent:	13
Descent:	3
Internal Leading:	3
External Leading:	0
Ave Char Width:	7
Max Char Width:	14
Weight:	400
Overhang:	0
Digitized Aspect X:	96
Digitized Aspect Y:	96
First Char:	0x0020
Last Char:	0x2122
Default Char:	0x0081
Break Char:	0x0020
Italic?	No
Underlined?	No
Struck Out?	No
Variable Pitch?	Yes
Vector Font?	No
TrueType Font?	No
Device Font?	No
Family:	Swiss
Character Set:	0

MS Sans Serif

OK

ABCDE abcde ÀÁÂÃÄÅ àáâãäå ΐΑΒΓΔΕ ΰαβγδε АБВГДЕ абвгде 倀偀併偺偁



Height: 1168
Width: 864
Escapement:
Orientation:
Weight:
[✓ Italic]
[✓ Underline]
[✓ Strike Out]
Match Aspect
Adv Grfx Mode
Character Set: ?
Face Name:

Mapping Mode: Text [✓ Low Metric] High Metric Low English High English Twips Logical Twips

Quality: [✓ Default] Draft Proof

Output Precision: [✓ OUT_DEFAULT_PRECIS] OUT_STRING_PRECIS OUT_CHARACTER_PRECIS OUT_STROKE_PRECIS OUT_TT_PRECIS OUT_DEVICE_PRECIS OUT_RASTER_PRECIS OUT_TT_ONLY_PRECIS OUT_OUTLINE_PRECIS

Pitch: [✓ Default] Fixed Variable

Family: [✓ Don't Care] Roman Swiss Modern Script Decorative

Text Metrics

Height:	1168
Ascent:	946
Descent:	222
Internal Leading:	123

Arial

OK

PickFont

```
0 = Ansi
1 = Default
2 = Symbol
128 = Shift JIS (Japanese)
129 = Hangul (Korean)
130 = Johab (Korean)
134 = GB 2312 (Simplified Chinese)
136 = Chinese Big 5 (Traditional Chinese)
177 = Hebrew
178 = Arabic
161 = Greek
162 = Turkish
163 = Vietnamese
204 = Russian
222 = Thai
238 = East European
255 = OEM
```

OK

PICKFONT Program:

- Designed to demonstrate font selection and related concepts in Windows.
- Presents a modeless dialog box (Figure 17-2) for font interaction.
- Offers additional features beyond standard logical font settings.

Dialog Box Features:

- Logical Font Attributes: Allows specifying typeface, size, weight, etc.
- Mapping Modes: Controls how fonts are scaled and positioned, including a custom "Logical Twips" mode.
- Match Aspect Option: Fine-tunes font matching for visual consistency.
- Advanced Graphics Mode (Windows NT): Enables advanced font rendering features.
- Device Selection: Toggles between video display and default printer for font output.

Device Context Considerations:

Different Fonts for Different Devices: PICKFONT selects logical fonts into device contexts independently for the display and printer, potentially resulting in different fonts being used for each.

TEXTMETRIC Information: The dialog box displays font metrics from the selected device context, reflecting either screen or printer font characteristics.

Program Focus:

Font Creation and Selection: The explanation primarily explores logical font concepts rather than dialog box implementation details.

Additional Insights:

User Feedback Importance: User ratings highlight the value of clarity, conciseness, and well-structured explanations, especially for technical topics.

Context and Target Audience: Understanding the surrounding context and intended audience can guide the level of detail and specific information included in explanations.

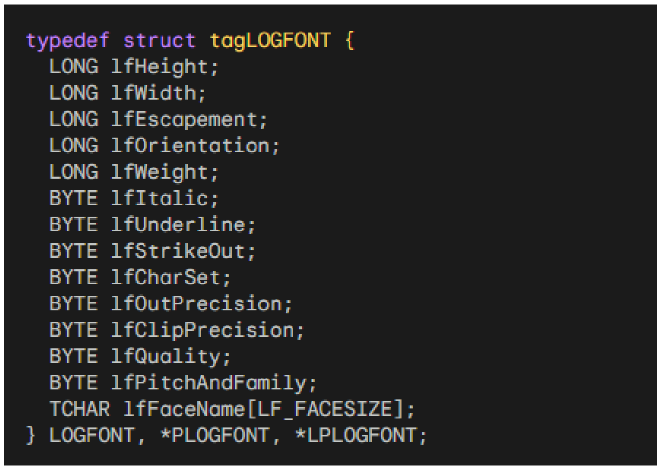
Further Exploration:

Font Enumeration Functions: While not discussed in the text, these functions allow listing available fonts on a device.

Modern Font Technologies: Exploring newer font handling and rendering techniques in contemporary Windows environments can offer insights into advanced font capabilities.

LOGFONT STRUCTURE

The LOGFONT structure plays a crucial role in font creation within the Windows operating system. This structure is integral to the process of defining a logical font, and its fields provide detailed information about the characteristics of the desired font. Here's an in-depth discussion of each field in the LOGFONT structure:

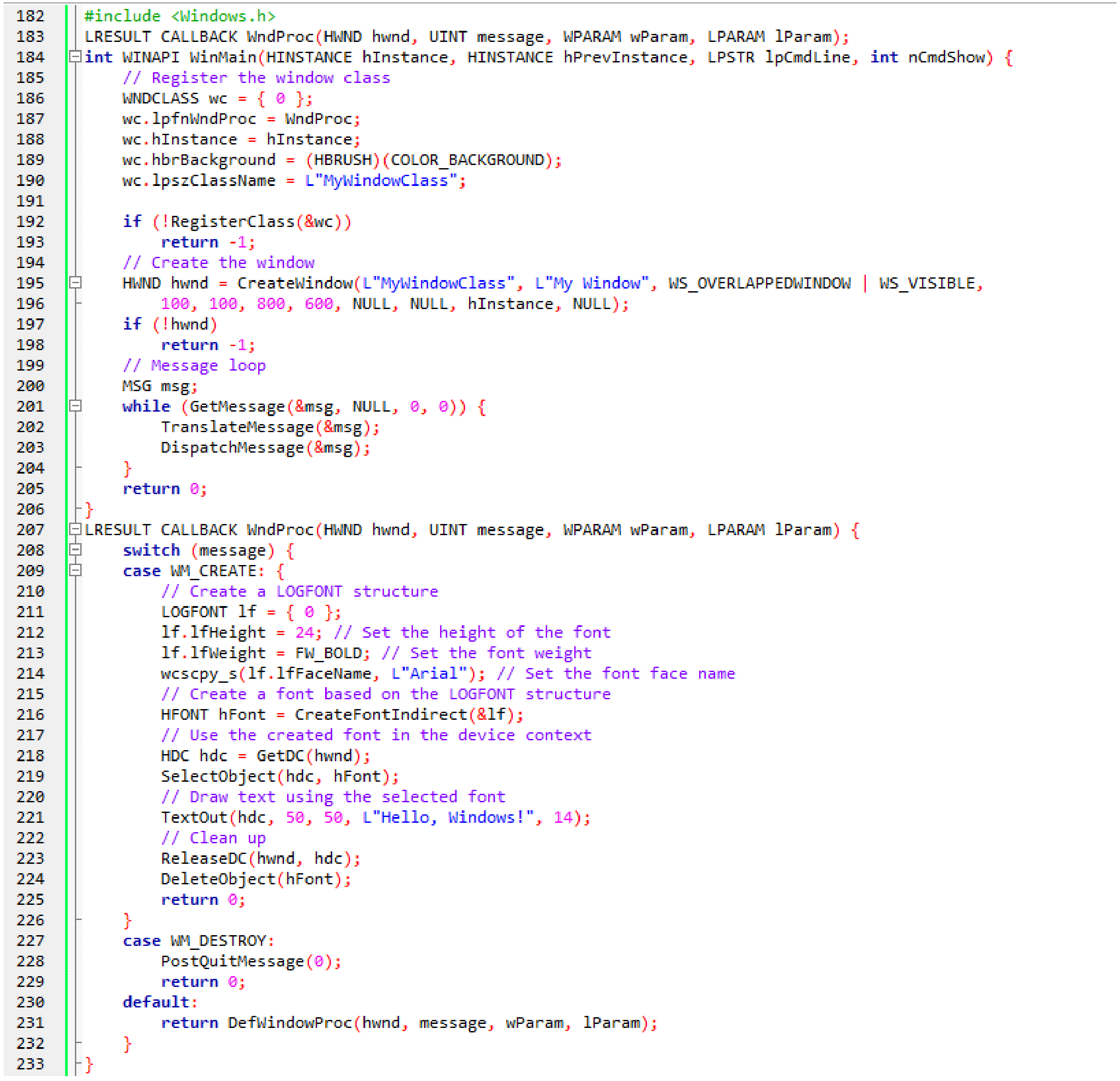


```
typedef struct tagLOGFONT {
  LONG lfHeight;
  LONG lfWidth;
  LONG lfEscapement;
  LONG lfOrientation;
  LONG lfWeight;
  BYTE lfItalic;
  BYTE lfUnderline;
  BYTE lfStrikeOut;
  BYTE lfCharSet;
  BYTE lfOutPrecision;
  BYTE lfClipPrecision;
  BYTE lfQuality;
  BYTE lfPitchAndFamily;
  TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT, *PLOGFONT, *LPLOGFONT;
```

- lfHeight: Specifies the height of the font. Positive values represent the height, while negative values represent the character cell height.
- lfWidth: Specifies the average width of characters in the font. If zero, the system selects the default width based on the font's aspect ratio.
- lfEscapement: Specifies the angle of rotation, in tenths of degrees, between the escapement vector and the x-axis of the device. Positive values indicate a counterclockwise rotation.
- lfOrientation: Specifies the angle of rotation, in tenths of degrees, between each character's baseline and the x-axis. This field is used for italicizing text.
- lfWeight: Specifies the weight of the font, such as normal, bold, or a custom weight in the range of 100 to 1000.
- lfItalic: Indicates whether the font is italicized or not.
- lfUnderline: Indicates whether the font is underlined or not.
- lfStrikeOut: Indicates whether characters in the font are struck out or not.
- lfCharSet: Specifies the character set used by the font, such as ANSI, default, or symbol character set.
- lfOutPrecision: Specifies the desired output precision for the font, such as default, TrueType, or stroke precision.
- lfClipPrecision: Specifies the clipping precision used for the font, such as default or character precision.
- lfQuality: Specifies the font quality, affecting anti-aliasing and rendering, such as default, draft, or proof quality.

- ❖ lfPitchAndFamily: Specifies the pitch (spacing) and font family of the font. The lower 4 bits represent the pitch (default, fixed, or variable), and the upper 4 bits represent the font family (e.g., Roman, Swiss).
- ❖ lfFaceName: Specifies the typeface name of the font, including the font family and style.

These members provide detailed information about the font's appearance, style, and characteristics.



```
#include <Windows.h>
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
    // Register the window class
    WNDCLASS wc = { 0 };
    wc.lpfnWndProc = WndProc;
    wc.hInstance = hInstance;
    wc.hbrBackground = (HBRUSH)(COLOR_BACKGROUND);
    wc.lpszClassName = L"MyWindowClass";

    if (!RegisterClass(&wc))
        return -1;
    // Create the window
    HWND hwnd = CreateWindow(L"MyWindowClass", L"My Window", WS_OVERLAPPEDWINDOW | WS_VISIBLE,
        100, 100, 800, 600, NULL, NULL, hInstance, NULL);
    if (!hwnd)
        return -1;
    // Message loop
    MSG msg;
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return 0;
}
LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    switch (message) {
    case WM_CREATE: {
        // Create a LOGFONT structure
        LOGFONT lf = { 0 };
        lf.lfHeight = 24; // Set the height of the font
        lf.lfWeight = FW_BOLD; // Set the font weight
        wcscpy_s(lf.lfFaceName, L"Arial"); // Set the font face name
        // Create a font based on the LOGFONT structure
        HFONT hFont = CreateFontIndirect(&lf);
        // Use the created font in the device context
        HDC hdc = GetDC(hwnd);
        SelectObject(hdc, hFont);
        // Draw text using the selected font
        TextOut(hdc, 50, 50, L"Hello, Windows!", 14);
        // Clean up
        ReleaseDC(hwnd, hdc);
        DeleteObject(hFont);
        return 0;
    }
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    default:
        return DefWindowProc(hwnd, message, wParam, lParam);
    }
}
```

When utilizing the LOGFONT structure, you can be as specific or as general as needed. If certain fields are left uninitialized (set to 0), default values are assumed. The flexibility of this structure allows for fine-tuning font specifications, and using CreateFontIndirect with a pointer to the LOGFONT structure enables the creation of fonts tailored to specific requirements.

The PICKFONT program serves as a useful tool to experiment with these fields and observe their impact on font appearance. Pressing Enter or the OK button in the program applies the entered fields, providing a real-time preview of the font based on the specified characteristics.

Creating a Font Using CreateFontIndirect:

```
LOGFONT lf = {0}; // Initialize all fields to defaults
lf.lfHeight = -12; // Font height in points
strcpy(lf.lfFaceName, "Arial"); // Font typeface

HFONT hFont = CreateFontIndirect(&lf);

// Select the font into a device context (hdc)
SelectObject(hdc, hFont);
```

Accessing Font Metrics Using GetTextMetrics:

```
TEXTMETRIC tm;
GetTextMetrics(hdc, &tm);

// Access font metrics like tmHeight, tmAveCharWidth, etc.
```

Experimenting with PICKFONT (example code):

```
// Assuming PICKFONT is a function that displays a dialog for font selection
LOGFONT lf = PICKFONT();

// Create and use the selected font
// ...
```

lfHeight:

Specifies the desired height of characters in logical units.

Values:

- 0: Default height is used.
- Positive values: Specify the desired height of characters, including the internal leading (line spacing).
- Negative values: Specify the font height compatible with the desired point size. The absolute value of the negative height is used.

Mapping Mode: It is important to understand the current mapping mode to interpret the lfHeight value correctly.

Point Size Conversion: For negative values of lfHeight, it is necessary to convert the desired point size to logical units using the current mapping mode to determine the actual font height.

TEXTMETRIC: The tmHeight field in the TEXTMETRIC structure roughly matches the lfHeight value for positive lfHeight values. For negative lfHeight values, subtracting tmInternalLeading from tmHeight in the TEXTMETRIC structure gives a value that roughly matches lfHeight.

lfWidth:

Purpose: Specifies the desired width of characters in logical units.

Common Value: 0. When lfWidth is set to 0, the operating system chooses an appropriate width based on the specified lfHeight value, maintaining the font's aspect ratio.

TrueType Font Adjustment: You can use lfWidth to achieve wider or slimmer characters by specifying a positive or negative value. This adjustment affects the character width while preserving the font's height.

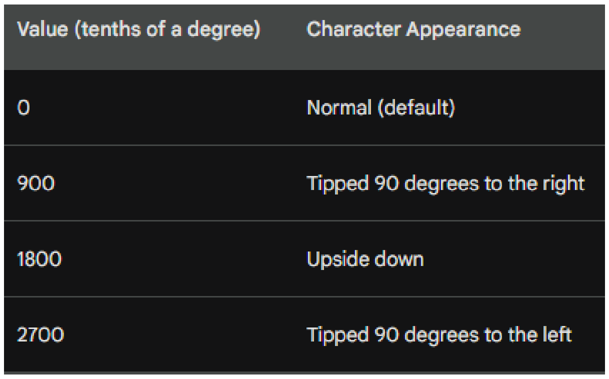
TEXTMETRIC: The lfWidth member corresponds to the tmAveCharWidth field in the TEXTMETRIC structure. The tmAveCharWidth represents the average width of characters in the font.

Intelligent Adjustment: To intelligently adjust the character width:

- Create a font with lfWidth set to 0.
- Use the GetTextMetrics function to retrieve the TEXTMETRIC structure, which contains information about the font's metrics, including the average character width (tmAveCharWidth).
- Adjust the tmAveCharWidth value, such as by applying a percentage adjustment.
- Create a new font with the adjusted tmAveCharWidth value assigned to lfWidth.

lfOrientation:

Purpose: Specifies the angle of individual character rotation, measured counterclockwise from the horizontal baseline. Controls the angle of individual character rotation, measured counterclockwise from the horizontal.



Value (tenths of a degree)	Character Appearance
0	Normal (default)
900	Tipped 90 degrees to the right
1800	Upside down
2700	Tipped 90 degrees to the left

Limited Functionality: Similar to lfEscapement, the lfOrientation value is often limited to working as expected with TrueType fonts under Windows NT when advanced graphics mode is enabled. The specific behavior may depend on the operating system, font rendering engine, and graphics capabilities.

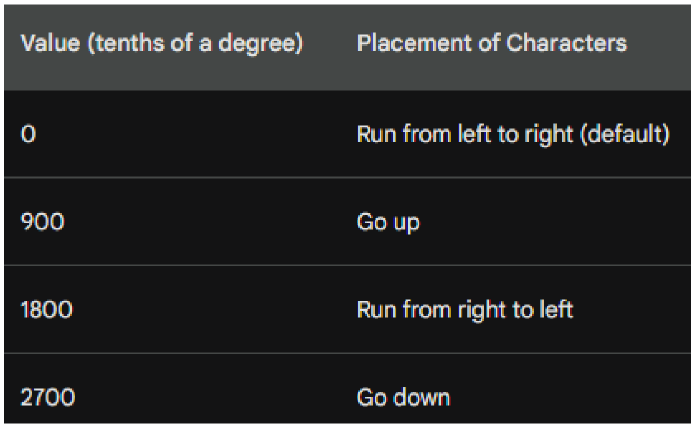
Units: The lfOrientation value is specified in tenths of a degree, where each unit represents 1/10th of a degree of rotation.

Character Appearance: Positive lfOrientation values tilt characters to the right, causing the characters to appear slanted or italicized in that direction. Negative lfOrientation values tilt characters to the left, creating a slant in the opposite direction.

KeyPoints: Counterclockwise rotation. Positive values tilt characters to the right, negative values tilt to the left.

lfEscapement:

Description: Specifies the angle, in tenths of a degree, measured counterclockwise from the horizontal baseline. It controls how successive characters of a text string are placed when writing text.



Value (tenths of a degree)	Placement of Characters
0	Run from left to right (default)
900	Go up
1800	Run from right to left
2700	Go down

Limited Functionality: The lfEscapement value often works as expected only with TrueType fonts under Windows NT when advanced graphics mode is enabled. The precise behavior may vary depending on the operating system, font rendering engine, and graphics capabilities.

Experimentation: When using the PICKFONT program, you can experiment with values between 0 and -600 or 3000 and 3600 for the lfEscapement member. This range allows you to rotate the text at various angles and observe the results.

Units: The lfEscapement value is specified in tenths of a degree. Each unit corresponds to 1/10th of a degree of rotation.

Counterclockwise Rotation: Positive lfEscapement values rotate the text counterclockwise, causing the text to appear rotated upwards. Negative lfEscapement values rotate the text clockwise, causing the text to appear rotated downwards.

Examples:

- ✓ 0: Text runs from left to right (default orientation).
- ✓ 900: Text goes up, with each character rotated counterclockwise by 90 degrees.
- ✓ 1800: Text runs from right to left, with each character rotated by 180 degrees.
- ✓ 2700: Text goes down, with each character rotated clockwise by 90 degrees.

Platform-Specific Behavior:

- ✓ Windows 98: Sets the escapement and orientation of TrueType text.
- ✓ Windows NT: Normally sets the escapement and orientation of TrueType text, except when the GM_ADVANCED flag is used.
- ✓ GM_ADVANCED: When the GM_ADVANCED flag is used, the lfEscapement value works as documented in Windows NT, allowing advanced control over text placement and rotation.

It's important to note that the behavior and support for lfEscapement may vary depending on the platform, operating system version, and font rendering capabilities. Experimentation with different values, such as those between 0 and -600 or 3000 and 3600 in the PICKFONT program, can help understand and visualize the effects of text rotation and placement.

Key Points:

- ✓ Counterclockwise Rotation: lfEscapement values result in counterclockwise rotation of the text.
- ✓ Positive Values: Positive lfEscapement values rotate the text upwards.
- ✓ Negative Values: Negative lfEscapement values rotate the text downwards.

Now, let's address your additional questions:

- ✓ Orientation (lfOrientation): The lfOrientation member in the LOGFONT structure controls the angle of individual character rotation, measured counterclockwise from the horizontal baseline. It allows you to tilt characters individually, creating a slanted or italicized appearance. Positive values tilt characters to the right, while negative values tilt characters to the left.
- ✓ GM_ADVANCED flag: In Windows NT, the behavior of lfEscapement is typically set according to the TrueType font's design. However, when the GM_ADVANCED flag is used, it overrides the default behavior and allows advanced control over text placement and rotation. The lfEscapement value works as documented in Windows NT, providing precise control over the angle of text placement.
- ✓ Limitations and Considerations: The functionality of lfEscapement may vary depending on the platform, font rendering engine, and operating system. It is often most reliable and predictable when used with TrueType fonts under Windows NT with advanced graphics mode enabled. The support and behavior of lfEscapement with non-TrueType fonts or under different operating systems may be limited or less consistent. Therefore, it is essential to experiment and test different values to ensure the desired text orientation and placement are achieved.

lfWeight:

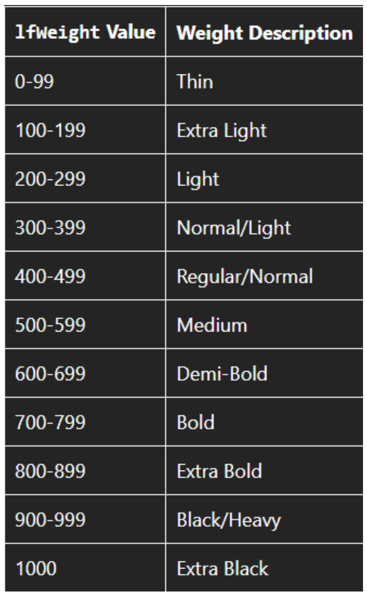
Purpose: Specifies the font weight, which determines the thickness or boldness of the characters in a font.

Values: The lfWeight value can range from 0 to 1000, with predefined constants available for common weights:

- FW_DONTCARE (0): The weight is not specified or doesn't matter.
- FW_THIN (100): A very thin or light weight.
- FW_NORMAL (400): The default weight. Often equivalent to regular or normal weight.
- FW_BOLD (700): A bold weight.
- FW_BLACK (900): The heaviest or blackest weight available.

Intermediate Values: Values between the predefined constants provide a gradual range of weights, allowing for finer adjustments.

Visual representation:



`lfWeight` Value	Weight Description
0-99	Thin
100-199	Extra Light
200-299	Light
300-399	Normal/Light
400-499	Regular/Normal
500-599	Medium
600-699	Demi-Bold
700-799	Bold
800-899	Extra Bold
900-999	Black/Heavy
1000	Extra Black

Examples:

- lfWeight set to FW_NORMAL (400) will result in the default, regular weight.
- lfWeight set to FW_BOLD (700) will render the text in a bold weight.
- lfWeight set to FW_THIN (100) will produce very thin or light-weight characters.
- Custom values between the predefined constants can be used to specify intermediate weights.

Key Points:

- Not all values are fully implemented.
- Commonly used values: 0 or 400 for normal, 700 for bold.

lfItalic:

Purpose: When set to a nonzero value, it specifies that the font should be rendered in italic style.

Behavior:

- For GDI raster fonts: Windows synthesizes italics by shifting some rows of the character bitmap to mimic an italic appearance.
- For TrueType fonts: Windows uses the actual italic or oblique version of the font if available.

Example: Setting lfItalic to a nonzero value will render the text in an italicized style.

lfUnderline:

Purpose: When set to a nonzero value, it specifies that the font should be rendered with an underline.

Behavior: The Windows GDI draws a line underneath each character, including spaces, to create the underline effect.

Example: Setting lfUnderline to a nonzero value will render the text with an underline.

lfStrikeOut:

Purpose: When set to a nonzero value, it specifies that the font should have a line drawn through the characters, creating a strikeout effect.

Behavior: The strikeout line is synthesized by the Windows GDI for both GDI raster fonts and TrueType fonts.

Example: Setting lfStrikeOut to a nonzero value will render the text with a strikeout effect.

lfCharSet:

Purpose: Specifies the character set of the font.

Value: It is a byte value that represents the character set.

Default: A zero value is equivalent to ANSI_CHARSET, which corresponds to the ANSI character set used in the United States and Western Europe.

Additional Information: The DEFAULT_CHARSET code, equal to 1, indicates the default character set for the machine on which the program is running.

lfOutPrecision:

Purpose: Specifies how Windows should attempt to match the desired font sizes and characteristics with actual fonts.

Usage: This field is complex and may not be used frequently. Referring to the documentation of the LOGFONT structure provides more detailed information.

Special Flag: The OUT_TT_ONLY_PRECIS flag can be used to ensure that you always get a TrueType font.

lfClipPrecision:

Purpose: Specifies how characters should be clipped when they lie partially outside the clipping region.

Usage: This field is not commonly used and is not implemented in the PICKFONT program.

lfQuality:

Purpose: Provides an instruction to Windows regarding the matching of the desired font with an actual font.

Behavior: Primarily relevant for raster fonts, it does not significantly affect TrueType fonts.

Flags:

- DRAFT_QUALITY: Indicates that GDI should scale raster fonts to achieve the desired size.
- PROOF_QUALITY: Indicates that no scaling should be done. These fonts are visually attractive but may be smaller than the requested size.
- DEFAULT_QUALITY (or 0): Typically used for this field to specify the default font quality.

lfPitchAndFamily:

Purpose: This byte is composed of two parts, combined using the bitwise OR operator. It specifies the pitch (fixed or variable) and font family of the font.

Pitch:

Lowest two bits:

- DEFAULT_PITCH (0): The font has a default pitch, meaning it can have both variable and fixed pitch characters.
- FIXED_PITCH (1): The font has a fixed pitch, where all characters have the same width.
- VARIABLE_PITCH (2): The font has a variable pitch, meaning characters can have different widths.

Font Family:

Upper half of the byte:

- FW_DONTCARE (0x00): No specific font family is specified or preferred.
- FF_ROMAN (0x10): A font family with variable widths and serifs (such as Times New Roman).
- FF_SWISS (0x20): A font family with variable widths and no serifs (such as Arial).
- FF_MODERN (0x30): A font family with fixed pitch (monospaced).
- FF_SCRIPT (0x40): A font family that mimics handwriting.
- FF_DECORATIVE (0x50): A font family used for decorative or artistic purposes.

lfFaceName:

Purpose: This field represents the actual text name of the typeface or font, such as "Courier," "Arial," or "Times New Roman." It is a byte array that is LF_FACESIZE (32 characters) wide.

Usage for TrueType Italic or Boldface Fonts:

To obtain a TrueType italic or boldface font, you have two options:

- Use the complete typeface name, including style information, in the lfFaceName field. For example, "Times New Roman Italic" or "Arial Bold."
- Use the base typeface name (without style information) in the lfFaceName field and set the lfItalic or lfWeight fields to indicate the desired style (italic or bold).

By combining the values of lfPitchAndFamily and specifying the appropriate lfFaceName, you can define the characteristics of the desired font, such as pitch, font family, and typeface name.