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:



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:



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:



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:



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:



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.

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:



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:



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:



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:



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:



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



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:



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:



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



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.

 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:



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.).



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.).



Extracting Font Information

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



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



PICKFONT PROGRAM

Overview of "PICKFONT.C" Program Sections:

Structure Definition:



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.



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.



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.



*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.



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:



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.



* 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): 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; 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).



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.



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:



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:



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:*



*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:



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.



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.



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.



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.



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:



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….*





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:



* 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.



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:



Accessing Font Metrics Using GetTextMetrics:



Experimenting with PICKFONT (example code):



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.



*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.



*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:*



*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.

FONT-MAPPING ALGORITHM

The Font-Mapping Algorithm plays a crucial role in the Windows operating system when rendering text, determining the appropriate real font that best matches the specified logical font. This process involves several considerations, with certain fields in the logical font structure carrying more weight than others.



Upon setting up the logical font structure, the CreateFontIndirect function is utilized to obtain a handle to the logical font. Subsequently, when the SelectObject function is employed to select this logical font into a device context, Windows employs the font-mapping algorithm to find the most suitable real font based on the user's request.



One pivotal field in this process is lfCharSet, which denotes the character set. Historically, specifying OEM\_CHARSET (255) would result in stroke fonts or the Terminal font being selected.



*Let’s break a bit…*

No, the lfCharSet field in the Font-Mapping Algorithm is not directly related to ASCII or Unicode. The lfCharSet field is a part of the logical font structure in the Windows operating system and is used to specify the character set for the font.



Character sets define the collection of characters that a font can display. Historically, when specifying OEM\_CHARSET (255) in the lfCharSet field, it was associated with the Original Equipment Manufacturer character set. This choice typically resulted in selecting stroke fonts or the Terminal font, which were fonts designed to work with the OEM character sets.



ASCII and Unicode, on the other hand, are character encoding standards. ASCII represents a set of characters that are commonly used in the English language and is a subset of Unicode. Unicode, on the other hand, is a much larger character set that encompasses a wide range of characters from various languages and symbols worldwide.



In the context of font mapping, specifying OEM\_CHARSET was more about selecting a set of characters designed for specific purposes, like working with OEM character sets, rather than specifically choosing between ASCII or Unicode. It's more about how the font is designed to handle character representation in a broader sense.

*That’s done, let’s continue from where we broke the notes…*

However, the introduction of TrueType "Big Fonts" allows a single TrueType font to be mapped to different character sets, including the OEM character set. To access the Symbol font or Wingdings font, SYMBOL\_CHARSET (2) is now utilized.



The lfPitchAndFamily field also carries significance, particularly when it contains the value FIXED\_PITCH. This value signals to Windows that a variable-width font is undesired, streamlining the font selection process.



Furthermore, lfFaceName, indicating the typeface of the font, is a crucial factor. If left as NULL, the family value in lfPitchAndFamily becomes significant when specifying a font family.

For raster fonts, Windows attempts to match the lfHeight value, even if it requires scaling a smaller font. Conversely, stroke or TrueType fonts are scaled directly to the desired height. Setting lfQuality to PROOF\_QUALITY prevents Windows from scaling a raster font, emphasizing the appearance over the requested height.



In cases where lfHeight and lfWeight values deviate significantly from the display's aspect ratio, Windows might map to a raster font designed for a device with a different aspect ratio. This was historically employed as a trick to achieve thinner or thicker fonts, although it is less necessary with TrueType fonts.

*Let’s see the algorithm re-explained in a better manner…*

Think of it as a font matchmaker. You describe your ideal font (the "logical font"), and it scours your system's font collection to find the closest real-life match.

Here's the play-by-play:

You create a font wish list: You tell Windows the font attributes you're craving, like typeface, size, and style. Think of it as filling out a dating profile for fonts.

Windows whips out its little black book: It checks its catalog of available fonts, called "real fonts," to find potential matches.

Swiping left or right: The algorithm prioritizes certain characteristics to narrow down the options. It's like the font version of Tinder!

Here's what catches the algorithm's eye:

Character set: Determines the language and symbols a font supports.

Pitch: Monospaced or variable-width? The algorithm wants to know your preference.

Typeface: The font's unique personality. Think Times New Roman vs. Comic Sans.

Height: Windows tries its best to match your desired font size.

Weight: From light and airy to bold and confident, the algorithm considers your desired font thickness.

Pro tips for font-savvy developers:

* TrueType fonts rock: They're flexible and scale beautifully, making them ideal matches in most cases.
* Aspect ratio matters: Keep it in mind for optimal font appearance. No one wants a squished or stretched font!
* Experiment with PICKFONT: It's like a font playground for testing different combinations and seeing the algorithm in action.

The information retrieved about a font in PICKFONT, particularly from the GetTextMetrics function, provides a detailed breakdown of various characteristics stored in the TEXTMETRIC structure. Each field in this structure holds essential details about the font's properties and dimensions, all presented in logical units.

tmHeight: This represents the height of the character in logical units. It closely corresponds to the lfHeight field specified in the LOGFONT structure. If the lfHeight is positive, tmHeight represents the line spacing of the font; if negative, tmHeight minus tmInternalLeading approximates the absolute value of lfHeight.



tmAscent and tmDescent: These fields denote the vertical size of the character above and below the baseline in logical units, respectively.



tmInternalLeading: This is a vertical size included in tmHeight, typically occupied by diacritics on some capital letters. Calculating the point size of the font involves subtracting tmInternalLeading from tmHeight.

 

tmExternalLeading: It represents an additional amount of line spacing beyond tmHeight, recommended by the font designer for spacing successive lines of text.



tmAveCharWidth and tmMaxCharWidth: These fields provide the average width of lowercase letters and the width of the widest character in logical units. For a fixed-pitch font, tmMaxCharWidth is the same as tmAveCharWidth.



tmWeight: Indicates the weight of the font on a scale from 0 through 999. In practice, it is 400 for a normal font and 700 for a boldface font.



tmOverhang: Specifies the extra width added to a raster font character when synthesizing italic or boldface. For a boldface font, tmAveCharWidth minus tmOverhang equals tmAveCharWidth for the same font without boldfacing.

 

tmDigitizedAspectX and tmDigitizedAspectY: These represent the aspect ratio for which the font is appropriate, equivalent to values obtained from GetDeviceCaps with LOGPIXELSX and LOGPIXELSY identifiers.



tmFirstChar, tmLastChar, tmDefaultChar, tmBreakChar: These fields deal with character codes. They specify the first and last characters in the font, the character used for characters not in the font (usually a rectangle), and the character used to determine word breaks when justifying text (typically the space character). These fields deal with character codes and their roles in font rendering. The tmFirstChar and tmLastChar specify the range of characters supported by the font. For Unicode fonts obtained with GetTextMetricsW, tmLastChar might exceed 255. The tmDefaultChar represents the character displayed when a requested character is not available in the font, typically a rectangle. tmBreakChar designates the character used to determine word breaks when justifying text, often set to 32, representing the space character.



tmItalic, tmUnderlined, tmStruckOut: These are flags indicating whether the font is italic, underlined, or struck through. These fields are flags that provide additional style information about the font. The tmItalic flag is nonzero for an italic font, tmUnderlined is nonzero for an underlined font, and tmStruckOut is nonzero for a strikethrough font. These flags contribute to the visual appearance of the text and help convey the stylistic attributes of the selected font.



tmPitchAndFamily: This field combines information about the font's characteristics. The low-order bits indicate specific attributes, such as fixed pitch, vector (TrueType), and device font. The top four bits represent the font family, using values from the LOGFONT lfPitchAndFamily field. To summarize, the tmPitchAndFamily field encapsulates critical information about the font's style, characteristics, and family, making it a central component for understanding and manipulating font properties in Windows applications.



To summarize, the tmPitchAndFamily field encapsulates critical information about the font's style, characteristics, and family, making it a central component for understanding and manipulating font properties in Windows applications.

tmCharSet: Identifies the character set, specifying how characters are encoded. This field is distinct from ASCII or Unicode but plays a role in character representation. The tmCharSet field in the TEXTMETRIC structure represents the character set identifier. This identifier specifies the encoding standard used for character representation in the font. It is crucial in determining how characters are mapped to specific codes, facilitating proper rendering and interpretation. Common values include ANSI\_CHARSET and SYMBOL\_CHARSET, indicating different character set standards.

 

CHARACTER SETS: THE FOUNDATION OF TEXT REPRESENTATION

Numerical Codes for Characters: Text, at its core, comprises characters—letters, symbols, and punctuation marks. To seamlessly store and process text digitally, systems assign unique numerical codes to each character. These codes, collectively forming a character set, enable consistent representation and interpretation of text across different applications and platforms.



Windows Character Set Identifiers: Windows utilizes a specific system of character set identifiers, ranging from 0 to 255, to specify the character set supported by a given font. These identifiers are distinct from code pages, another encoding approach, and play a crucial role in font selection and text display within the Windows environment.



Unicode: A Universal Language

Global Character Encapsulation: To address the vast diversity of languages and writing systems worldwide, Unicode emerged as a comprehensive character set. It encompasses tens of thousands of characters, spanning virtually all scripts and symbols used in human communication.



Consistency and Compatibility: By adopting Unicode, applications and platforms can consistently represent and exchange text across diverse languages and regions, fostering seamless communication and ensuring interoperability.



Font Support for Character Sets:

LOGFONT and TEXTMETRIC Structures: These essential Windows structures play a pivotal role in font management and text rendering. They incorporate the character set identifier field, signaling a font's support for a specific character set.





TrueType Fonts: Gateway to Wider Character Coverage, TrueType fonts, often referred to as "Big Fonts," frequently encompass characters from multiple character sets, enabling broader language support and enhancing text display flexibility.

Windows Font Selection: A Dynamic Process

Font Choices Based on Character Needs: Windows actively strives to select the most suitable font to accurately render the requested characters. It considers the character set identifier specified in the LOGFONT structure and may opt for TrueType fonts or specialized fonts like Lucida Sans Unicode to accommodate a wider range of characters.

Exploring Character Sets with the PICKFONT Program:

Unicode Version: Unlocking Diverse Character Display: The Unicode version of PICKFONT exhibits a broader range of characters, including Greek, Cyrillic, and ideographs. This expanded display stems from Unicode's comprehensive character repertoire.

Non-Unicode Version: Character Rendering Constraints: In the non-Unicode version, the displayed characters hinge upon the chosen character set. Certain characters, such as accented letters or those belonging to specific alphabets, might not be universally available in all

Character Codes and Unicode Definitions:

Fixed Codes for Specific Characters: Unicode defines consistent codes for certain characters, ensuring their correct rendering across different character sets. For example, codes 0x0390 through 0x0395 always correspond to Greek alphabet letters.

Character Set Indication in LOGFONT:

While the actual character set might be Unicode, the character set ID in the LOGFONT structure signals Windows about the desired character set, guiding font selection and text display.

Adapting to Diverse Character Needs:

Windows strategically selects fonts based on character set requirements, opting for fonts like Bitstream CyberBit when default fonts lack support for character sets like SHIFTJIS\_CHARSET (Japanese), HANGUL\_CHARSET (Korean), and others.

In Conclusion:

Understanding character sets and their interplay with Windows font handling is crucial for accurate text rendering and cross-language compatibility in applications.

Unicode's comprehensive character set and Windows' dynamic font selection mechanisms work in tandem to ensure accurate and diverse text display, fostering seamless communication across languages and cultures.



Here are key points not explicitly discussed in the provided text:

Font Size and Rendering: The text doesn't delve into how font size and rendering mechanisms might affect character display across different character sets and operating systems.

Code Page Differences: While code pages are mentioned, their specific distinctions from character sets and potential implications for character handling aren't elaborated on.

Unicode Font Coverage: The range of characters supported by various Unicode fonts, beyond the mentioned examples, isn't explored.

Font Selection Criteria: Windows' precise font selection logic for different character sets and operating systems, beyond the general guidelines provided, could be further elucidated.

Non-Unicode Font Limitations: The potential drawbacks or challenges associated with using non-Unicode fonts for multilingual text display aren't explicitly discussed.

Unicode Font Compatibility: The text doesn't address potential compatibility issues that might arise when using Unicode fonts across different applications or platforms.

Font Fallback Mechanisms: The strategies employed by Windows or applications when encountering characters not supported by the selected font aren't detailed.

Additional Considerations for Discussion:

Font Substitution: How Windows handles font substitution when a requested character isn't available in the selected font.

Character Mapping: The potential need for character mapping or re-encoding when working with text across different character sets.

Unicode Best Practices: Recommendations for developers to ensure optimal text handling and cross-language compatibility in their applications.

EZFONT PROGRAM

EZFONT's Role in Windows Font Management:

Addressing Legacy Font Handling Challenges: EZFONT emerged as a response to the complexities associated with font enumeration and approximation in earlier Windows font-handling mechanisms.

Capitalizing on TrueType's Universality: TrueType fonts, being widely available on Windows systems, offered a foundation for simplifying font selection and management, which EZFONT effectively leverages.

 

EzCreateFont Function in Depth:

Device Context Preparation:

Preserving Original State: Saving the device context's state ensures that any subsequent modifications within the function remain isolated, preventing unintended side effects in the application.

Advanced Graphics Mode: This mode enables precise font control and rendering capabilities.

Resetting Transformations: Resetting transformation matrices to identity ensures accurate font measurements and calculations, unaffected by prior transformations.

Resolution Handling:

Physical vs. Logical Resolution: EZFONT accommodates both physical and logical resolution scenarios, providing flexibility for different device configurations.

DPI Calculation: Determining the horizontal and vertical dots per inch (DPI) is crucial for scaling fonts accurately across devices with varying resolutions.

Font Size Calculation:

Tenths of a Point to Device Units: The desired font height and width, specified in tenths of a point, are converted to device units using the calculated DPI.

Device Units to Logical Units: This subsequent conversion ensures consistency in font size rendering across different devices, independent of their physical resolutions.

LOGFONT Configuration:

Encapsulating Font Properties: The LOGFONT structure serves as a comprehensive container for font attributes, including height, width, typeface, character set, and styling properties (bold, italic, underline, strikeout).

Font Creation and Adjustment:

Initial Font Creation: The CreateFontIndirect function is responsible for generating a font based on the specified LOGFONT settings.

Width Adjustment (Optional): If a specific font width is requested, EZFONT retrieves font metrics using GetTextMetrics and adjusts the LOGFONT's width accordingly, ensuring the desired visual appearance.

Device Context Restoration:

Preserving Application State: Restoring the device context to its original state maintains consistency for the application's subsequent drawing operations.

Font Handle Return:

Handle for Text Rendering: The function returns a font handle (HFONT), which serves as a reference to the newly created font, enabling its use in various text rendering tasks within the application.

Additional Considerations:

* Potential for Further Simplification: EZFONT could potentially streamline font enumeration tasks for developers who still require them, further simplifying font management.
* Integration with ChooseFont Dialog Box: EZFONT might offer a means to simplify font choices presented within the ChooseFont dialog box, enhancing user experience.
* Point Size Specification: Use decipoints (1/10ths of a point) for height.
* Em-Width Adjustment: Control overall character widths independently of height.
* Font Attributes: Set bold, italic, underline, and strikeout styles.
* Logical vs. Physical Resolution: Choose based on device type and font scaling needs.
* Mapping Mode Consistency: Ensure the same mapping mode is used during font creation and text rendering.
* Memory Management: Delete created fonts using DeleteObject before program termination.
* Windows NT-Specific Adjustments: Necessary for accurate font sizing on Windows NT systems.
* Font Enumeration Simplification: EZFONT could potentially simplify font enumeration tasks as well.
* ChooseFont Dialog Integration: Might offer a way to simplify font choices within the dialog.

Windows NT-Specific Adjustments:

In the context of Windows 10 development, the specific adjustments made in EZFONT for Windows NT may no longer be necessary.

Windows 10 has undergone significant evolution since the Windows NT era, and the calls to SetGraphicsMode and ModifyWorldTransform, designed for Windows NT, can be safely omitted.

This suggests that Windows 10's advancements may have obviated the need for these particular adjustments in achieving accurate font sizing.

Font Enumeration Simplification:

Despite the advancements in Windows 10's font handling capabilities, EZFONT retains relevance for developers seeking to simplify font enumeration.

Its potential to streamline font selection processes remains valuable, offering a straightforward approach to filter font choices and simplify the selection of TrueType fonts.

Developers can leverage EZFONT to enhance the efficiency of font-related tasks, even within the context of Windows 10.

ChooseFont Dialog Integration:

Integrating EZFONT with the ChooseFont dialog in Windows 10 applications could yield several benefits for the user experience.

This integration could facilitate the filtering of font choices to those reliably available on the system, simplifying the selection process.

Moreover, it might offer more intuitive customization options, contributing to a user-friendly interface for font-related operations.

Key Considerations for Windows 10:

In the Windows 10 environment, it's crucial to acknowledge the updated font handling APIs that provide a comprehensive set of options for font selection, creation, and management.

When designing font-related features for Windows 10 applications, prioritizing user experience is paramount.

This involves offering intuitive font selection tools, ensuring consistent font rendering across diverse devices and resolutions, and providing clear feedback during font-related operations.



Recommendations:

For developers working with EZFONT in a Windows 10 environment, thorough compatibility testing is recommended.

Identifying and addressing any compatibility issues will help ensure the smooth functioning of EZFONT in the latest operating system.

Additionally, exploring Windows 10's advanced font handling APIs can provide developers with expanded capabilities for font management, aligning with the modern features offered by the operating system. Ultimately, a user-centric design approach should guide the development of font-related features to deliver an optimal experience for Windows 10 users.



EZTEST.C:

Focus on Font Testing: This program's primary purpose is to rigorously evaluate the EZFONT system's ability to accurately create and render TrueType fonts within a Windows environment. It serves as a comprehensive test suite for ensuring font functionality.

Key Functionalities:

Font Size Exploration: The program dynamically creates TrueType fonts spanning a range of point sizes, meticulously examining font generation and rendering across a spectrum of sizes.

Font Metrics Retrieval: It probes font metrics using GetObject and GetTextMetrics, gathering essential information about font dimensions and characteristics, validating their consistency and accuracy.

Font Information Display: Descriptive text is rendered alongside each generated font, showcasing its point size and corresponding metric values, providing a visual and informative assessment of font behavior.

Logical Twips for Scalability: The program employs the logical twips mapping mode to establish a reliable foundation for consistent font scaling across diverse devices and screen resolutions, ensuring accurate font appearance regardless of hardware configuration.

Window Procedure (WndProc):

Initialization:

The program starts by initializing variables, including a static DOCINFO structure and a PRINTDLG structure.

It defines variables to store the client area dimensions and a handle to the printer DC.

Command Handling (WM\_COMMAND):

It handles commands, including IDM\_PRINT (Print) and IDM\_ABOUT (About).

When IDM\_PRINT is selected, it opens a Print dialog, obtains the printer DC, and sets up the printing environment.

It calls the PaintRoutine to perform font-related painting on the printer DC.

It handles the printing process, including starting and ending the document and pages.

When IDM\_ABOUT is selected, it displays an informational message box about the font demonstration program.

Window Size Handling (WM\_SIZE):

It updates the variables storing the client area dimensions when the window size changes.

Painting (WM\_PAINT):

It calls BeginPaint and EndPaint to set up and finish the painting process.

It calls the PaintRoutine function to handle the actual painting of font-related information on the window DC.

Cleanup and Quit (WM\_DESTROY):

When the window is closed, it posts a quit message to terminate the program.

2. PaintRoutine Function:

Device Context Setup:

It sets the logical mapping mode to logical twips and adjusts the window and viewport extents for consistency.

Font Testing Loop:

It iterates through a range of point sizes from 80 to 120.

For each point size, it creates a Times New Roman font using EzCreateFont, retrieves font information, and prints details about the font.

Cleanup:

It deletes the created font object to release resources.

3. WinMain Function:

Window Class Registration:

It registers the window class with the window procedure (WndProc) and other attributes.

Window Creation:

It creates the main window with specified attributes.

Message Loop:

It enters the message loop, handling messages until the program is terminated.

Return:

It returns the wParam of the last message as the program's exit code.

Conclusion:

The EZTEST.C program is designed to create a window, handle user commands for printing and displaying information about the font demonstration program, and paint font-related details in the client area. It works in conjunction with the EzCreateFont function and is part of a font demonstration system.

FONTDEMO.C:

Visual Font Showcase: This program excels in providing an engaging and interactive demonstration of font capabilities, seamlessly integrating the EZFONT system to offer a user-friendly visual experience.

Notable Features:

Window Creation and Message Handling: It establishes a standard Windows window to serve as a canvas for font rendering, skillfully handling various system messages to ensure smooth window interactions and responsiveness.

Font Rendering Orchestration: The program invokes the PaintRoutine function (shared with EZTEST.C) to execute the actual font rendering process, leveraging EZFONT's font creation utilities to produce visually appealing text elements.

User-Friendly Menu: It offers a convenient menu system, empowering users to print the displayed fonts for physical reference or to access program information, fostering accessibility and exploration.

Printing Functionality: The program incorporates robust printing capabilities, allowing users to effortlessly capture the visual font demonstration on paper, extending its utility beyond onscreen experiences.

1. Include Statements and External Declarations:

Include Statements:

It includes necessary header files, such as windows.h, EzFont.h, and resource.h.

External Declarations:

It declares external functions, including PaintRoutine and WndProc, as well as external variables like szAppName and szTitle.

2. WndProc (Window Procedure):

Window Class Registration:

It registers the window class with the window procedure (WndProc) and other attributes.

Window Creation:

It creates the main window with specified attributes.

Message Loop:

It enters the message loop, handling messages until the program is terminated.

Return:

It returns the wParam of the last message as the program's exit code.

3. PaintRoutine Function:

Print Command Handling (WM\_COMMAND):

It handles the IDM\_PRINT command.

It opens a Print dialog, obtains the printer DC, and sets up the printing environment.

It calls the PaintRoutine to perform font-related painting on the printer DC.

It handles the printing process, including starting and ending the document and pages.

About Command Handling (WM\_COMMAND):

It handles the IDM\_ABOUT command.

It displays an informational message box about the font demonstration program.

Window Size Handling (WM\_SIZE):

It updates the variables storing the client area dimensions when the window size changes.

Painting (WM\_PAINT):

It calls BeginPaint and EndPaint to set up and finish the painting process.

It calls the PaintRoutine function to handle the actual painting of font-related information on the window DC.

Printing Cleanup (WM\_DESTROY):

It handles cleanup related to printing when the window is destroyed.

4. WinMain Function:

Window Class Registration:

It registers the window class with the window procedure (WndProc) and other attributes.

Window Creation:

It creates the main window with specified attributes.

Message Loop:

It enters the message loop, handling messages until the program is terminated.

Return:

It returns the wParam of the last message as the program's exit code.

Conclusion:

The FONTDEMO.C program is designed to create a window, handle user commands for printing and displaying information about the font demonstration program, and paint font-related details in the client area. It works in conjunction with the EzCreateFont function and is part of a font demonstration system.

Intertwined Yet Distinct:

Shared Font Rendering Logic: Both programs strategically utilize the PaintRoutine function to streamline font rendering tasks, promoting code reusability and maintainability.

EZFONT as the Font Engine: FONTDEMO.C effectively harnesses the font creation prowess of the EZFONT system to fuel its visual demonstrations, showcasing the practical applications of EZFONT's capabilities.

Diverse Purposes, Harmonious Collaboration: While EZTEST.C focuses on meticulous font testing, FONTDEMO.C excels in providing an engaging user experience, underscoring the versatility of the EZFONT system in both development and demonstration scenarios.

*Here's a breakdown of the key points:*

Rasterization Limitations:

The PaintRoutine function in EZTEST.C encounters challenges in precisely rendering font sizes due to the inherent constraints of rasterization on displays.

Fonts are ultimately composed of pixels, which are discrete units of screen space.

Rendering font sizes with very fine increments (0.1 points in this case) can lead to situations where multiple sizes appear visually identical on screen, as the pixel grid cannot accommodate every subtle size variation.

Printing vs. On-Screen Rendering:

FONTDEMO.C offers the ability to print the font output, often resulting in more accurate size differentiation compared to on-screen rendering.

Printers typically have higher resolutions and employ different rendering techniques, allowing them to represent a wider range of font sizes with greater precision.

Mapping Modes and Font Scaling:

The use of logical twips mapping mode in EZTEST.C aims to promote consistent font scaling across devices.

However, it's crucial to acknowledge that mapping modes primarily affect font scaling behavior, not the underlying rasterization process itself.

Real-World Implications:

This phenomenon highlights the importance of understanding the nuances of font rendering and rasterization when designing applications that heavily rely on precise font size control.

It's essential to anticipate potential discrepancies between on-screen and printed output, especially when working with fonts at very small sizes or with very fine size adjustments.