CHAPTER 17: DIVING DEEPER INTO TEXT AND FONTS WITH WINAPI

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

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

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

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

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

TrueType: The Game Changer for Text in Windows

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

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

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

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

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

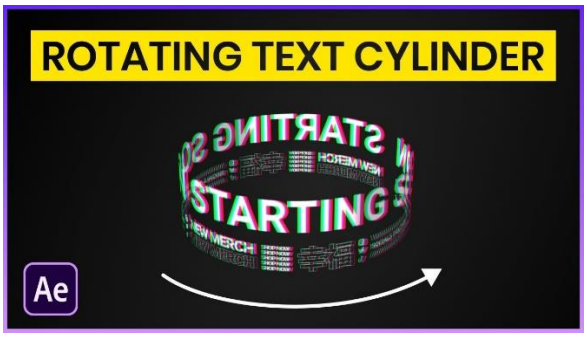
We'll explore these functionalities later in the chapter, so stay tuned!

Beyond Basic Display: Mastering Text Manipulation

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

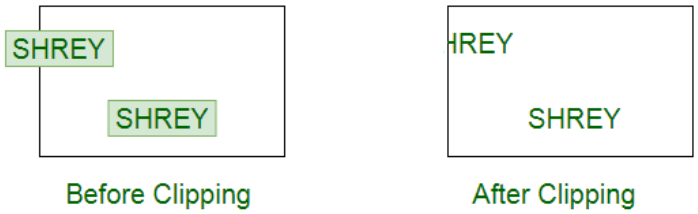
Scaling: Change the size of text without compromising quality using functions like CreateScalableFontResource and SetTextScale.

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



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

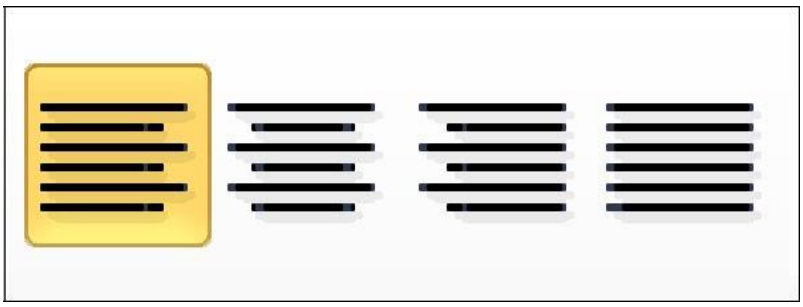
Clipping Regions: Define clipping regions based on font outlines using functions like SelectClipRgn. This allows you to mask other graphical elements behind the text, creating interesting effects.



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

Justifying Text for Visual Balance

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



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

WINDOWS TEXT OUTPUT FUNCTIONS

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

TextOut:

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

Core function for displaying text.

Arguments:

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

Coordinates and Positioning

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

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

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

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

SetTextAlign Function

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

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

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

Controls horizontal and vertical positioning of text.

Flags:

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

TA_UPDATECP flag:

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

TabbedTextOut Function

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

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

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

Handles text with embedded tab characters.

Expands tabs based on specified tab stops.

Arguments:

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

Tab Stops and Customization

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

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

ADVANCED TEXT OUTPUT FUNCTIONS IN WINDOWS

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

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

Arguments:

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

Clipping and Background Rectangles

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

Character Spacing

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

DrawText Function

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

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

Arguments:

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

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

Alignment:

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

Line Breaking:

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

Clipping:

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

Spacing:

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

Tabs:

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

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

Key Points:

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

Specifying Text within a Rectangle

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

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

Text Formatting Options

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

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

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

Handling Line Breaks and Word Wrapping

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

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

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

Tab Handling

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

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

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

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

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

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

DrawTextEx: Enhanced Text Handling with Tab Stops

Purpose:

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

Syntax:

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

Arguments:

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

DRAWTEXTPARAMS Structure:

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

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

Key Points:

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

Utilizing Enhanced Settings

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

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

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

DEVICE CONTEXT ATTRIBUTES FOR TEXT RENDERING IN WINDOWS

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

Customizing Text Color

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

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

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

Background Mode and Color

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

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

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

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

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

Handling Intercharacter Spacing

Intercharacter spacing can be adjusted using the SetTextCharacterExtra function:

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

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

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

Adapting to System Colors

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

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

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

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

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

Key Points:

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

LEVERAGING STOCK FONTS FOR TEXT RENDERING IN WINDOWS

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

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

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

Obtaining a Stock Font

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

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

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

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

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

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

Understanding Font Roles and Selection:

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

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

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

Common Stock Fonts and Applications:

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

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

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

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

Font Metrics and Text Layout:

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

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

Beyond Stock Fonts: Customization and Fine-Tuning:

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

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

Key Takeaways:

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

UNDERSTANDING FONT BASICS IN WINDOWS

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

Categories of Fonts

GDI Fonts:

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

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

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

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

TrueType Fonts:

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

Device Fonts:

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

Typeface Names:

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

Font Attributes/Characteristics:

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

Key Considerations for Font Selection:

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

TrueType Fonts

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

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

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

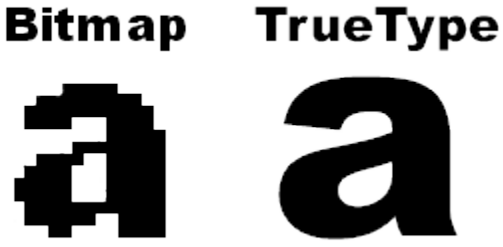
Font Attributes Synthesis

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

Essential Takeaways:

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

Raster font.



True type font.

EXPLORING TRUETYPE FONTS IN WINDOWS

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

Characteristics of TrueType Fonts

Character Definition:

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

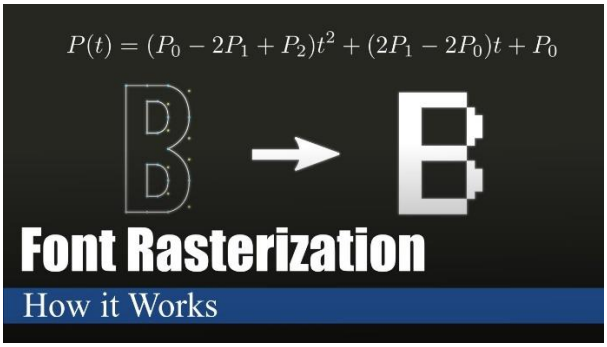
These fonts can be scaled by altering the coordinates that define the outlines.

Rasterization Process:

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

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

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



Bitmap Creation:

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

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

Key Characteristics in summary:

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

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

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

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

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

Common TrueType Fonts in Windows:

Courier New: Fixed-pitch font resembling typewriter output.

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

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

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

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

Symbol

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

Advantages:

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

Considerations:

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

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

RECONCILING TRADITIONAL AND COMPUTER TYPOGRAPHY:s

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

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

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

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

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

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

Leading: Orchestrating Vertical Harmony:

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

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

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

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

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

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

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

ADDRESSING THE LOGICAL INCH CONUNDRUM IN WINDOWS DISPLAY

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

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

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

The Logical Inch Phenomenon

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

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

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

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

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

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

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

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

Purpose:

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

Windows NT Distinction

Windows NT introduces some variations in its approach.

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

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

Windows 98:

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

Windows NT:

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

Introducing the "Logical Twips" Mapping Mode

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

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

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

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

Mapping Modes and Text Consistency:

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

Windows NT:

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

Key Points:

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

Additional Considerations:

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

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

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

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

UNDERSTANDING LOGICAL FONTS:

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

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

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

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

Creating and Selecting Logical Fonts

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

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

Creating a LOGFONT structure for CreateFontIndirect involves three primary approaches:

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

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

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

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

The lifecycle of a logical font involves three key steps:

Steps for Handling Logical Fonts:

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

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

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

Utilize: Draw text with the selected font.

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

Creation:

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

CreateFontIndirect:

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

ChooseFont:

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

Selection:

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

Deletion:

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

Font Mapping:

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

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

Structures for Font Information:

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

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

Key Considerations:

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

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

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

Extracting Font Information

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

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

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

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

PICKFONT PROGRAM

Overview of "PICKFONT.C" Program Sections:

Structure Definition:

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

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

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

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

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

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

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

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

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

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

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

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

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

Global Variables

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

hwnd (Main Window Handle):

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

```
HWND hwnd;
```

hdlg (Dialog Box Handle):

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

```
HWND hdlg;
```

szAppName (Application Name):

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

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

These global variables play essential roles in the program:

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

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

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

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

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

WinMain function:

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

Registering Window Class:

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

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

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

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

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

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

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

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

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

WndProc function:

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

Initialization and Dialog Box Creation:

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

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

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

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

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

Within the WM_CREATE case, the following actions are performed:

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

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

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

WM_COMMAND handling:

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

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

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

For both cases, the following actions are performed:

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

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

WM_PAINT:

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

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

Within the WM_PAINT case, the following actions are performed:

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

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

DlgProc function:

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

Initialization and User Input Handling:

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

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

In the WM_INITDIALOG case, the following actions are performed:

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

In the WM_SETFOCUS case, the following actions are performed:

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

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

Handling WM_COMMAND Message - Font Characteristics and Flags:

```
80  case WM_COMMAND:
81      switch (LOWORD(wParam)) {
82          // Handling various controls, setting font characteristics and flags
83          case IDC_LF_ITALIC:
84          case IDC_LF_UNDER:
85          case IDC_LF_STRIKE:
86          // ...
87
88          // Handling radio buttons for font characteristics and flags
89          case IDC_OUT_DEFAULT:
90          // ...
91
92          // Handling radio buttons for pitch and family
93          case IDC_FF_DONTCARE:
94          // ...
95
96          // Handling radio buttons for mapping modes
97          case IDC_MM_TEXT:
98          // ...
99
100         // OK button pressed
101         case IDOK:
102             SetLogFontFromFields(hdlg, pdp);  // Setting font characteristics
103             pdp->fMatchAspect = IsDlgButtonChecked(hdlg, IDC_MATCH_ASPECT);
104             pdp->fAdvGraphics = IsDlgButtonChecked(hdlg, IDC_ADV_GRAPHICS);
105             // Handling information context based on device selection
106             // Creating and selecting font into information context
107             SetFieldsFromTextMetric(hdlg, pdp);  // Updating dialog fields
108             InvalidateRect(GetParent(hdlg), NULL, TRUE);  // Invalidating main window
109             return TRUE;
110     }
111     break;
```

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

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

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

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

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

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

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

SetLogFontFromFields Function:

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

```
116
117 void SetLogFontFromFields(HWND hdlg, DLGPARAMS* pdp) {
118     pdp->lf.lfHeight = GetDlgItemInt(hdlg, IDC_LF_HEIGHT, NULL, TRUE);
119     pdp->lf.lfWidth = GetDlgItemInt(hdlg, IDC_LF_WIDTH, NULL, TRUE);
120     pdp->lf.lfEscapement = GetDlgItemInt(hdlg, IDC_LF_ESCAPE, NULL, TRUE);
121     pdp->lf.lfOrientation = GetDlgItemInt(hdlg, IDC_LF_ORIENT, NULL, TRUE);
122     pdp->lf.lfWeight = GetDlgItemInt(hdlg, IDC_LF_WEIGHT, NULL, TRUE);
123     pdp->lf.lfCharSet = GetDlgItemInt(hdlg, IDC_LF_CHARSET, NULL, FALSE);
124     pdp->lf.lfItalic = IsDlgButtonChecked(hdlg, IDC_LF_ITALIC) == BST_CHECKED;
125     pdp->lf.lfUnderline = IsDlgButtonChecked(hdlg, IDC_LF_UNDER) == BST_CHECKED;
126     pdp->lf.lfStrikeOut = IsDlgButtonChecked(hdlg, IDC_LF_STRIKE) == BST_CHECKED;
127     GetDlgItemText(hdlg, IDC_LF_FACENAME, pdp->lf.lfFaceName, LF_FACESIZE);
128 }
```

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.

```
113  void SetFieldsFromTextMetric(HWND hdlg, DLGPARAMS* pdp) {
114      // Declarations
115      TCHAR szBuffer[10];
116      TCHAR* szYes = TEXT("Yes");
117      TCHAR* szNo = TEXT("No");
118      TCHAR* szFamily[] = {TEXT("Don't Know"), TEXT("Roman"), TEXT("Swiss"), TEXT("Modern"), TEXT("Script"), TEXT("Decorative"), TEXT("Undefined")};
119
120      // SetDlgItemInt is used to set integer values in specified controls
121      SetDlgItemInt(hdlg, IDC_TM_HEIGHT, pdp->tm.tmHeight, TRUE);
122      SetDlgItemInt(hdlg, IDC_TM_ASCENT, pdp->tm.tmAscent, TRUE);
123      SetDlgItemInt(hdlg, IDC_TM_DESCENT, pdp->tm.tmDescent, TRUE);
124      SetDlgItemInt(hdlg, IDC_TM_INTLEAD, pdp->tm.tmInternalLeading, TRUE);
125      SetDlgItemInt(hdlg, IDC_TM_EXTLEAD, pdp->tm.tmExternalLeading, TRUE);
126      SetDlgItemInt(hdlg, IDC_TM_AVECHAR, pdp->tm.tmAveCharWidth, TRUE);
127      SetDlgItemInt(hdlg, IDC_TM_MAXCHAR, pdp->tm.tmMaxCharWidth, TRUE);
128      SetDlgItemInt(hdlg, IDC_TM_WEIGHT, pdp->tm.tmWeight, TRUE);
129      SetDlgItemInt(hdlg, IDC_TM_OVERHANG, pdp->tm.tmOverhang, TRUE);
130      SetDlgItemInt(hdlg, IDC_TM_DIGASPX, pdp->tm.tmDigitizedAspectX, TRUE);
131      SetDlgItemInt(hdlg, IDC_TM_DIGASPY, pdp->tm.tmDigitizedAspectY, TRUE);
132
133      // Display character codes in hexadecimal format
134      wsprintf(szBuffer, BCHARFORM, pdp->tm.tmFirstChar);
135      SetDlgItemText(hdlg, IDC_TM_FIRSTCHAR, szBuffer);
136      wsprintf(szBuffer, BCHARFORM, pdp->tm.tmLastChar);
137      SetDlgItemText(hdlg, IDC_TM_LASTCHAR, szBuffer);
138      wsprintf(szBuffer, BCHARFORM, pdp->tm.tmDefaultChar);
139      SetDlgItemText(hdlg, IDC_TM_DEFCHAR, szBuffer);
140      wsprintf(szBuffer, BCHARFORM, pdp->tm.tmBreakChar);
141      SetDlgItemText(hdlg, IDC_TM_BREAKCHAR, szBuffer);
142
143      // Display Yes/No based on boolean values
144      SetDlgItemText(hdlg, IDC_TM_ITALIC, pdp->tm.tmItalic ? szYes : szNo);
145      SetDlgItemText(hdlg, IDC_TM_UNDER, pdp->tm.tmUnderlined ? szYes : szNo);
146      SetDlgItemText(hdlg, IDC_TM_STRUCK, pdp->tm.tmStruckOut ? szYes : szNo);
147
148      // Display Yes/No based on pitch and family flags
149      SetDlgItemText(hdlg, IDC_TM_VARIABLE, TMPF_FIXED_PITCH & pdp->tm.tmPitchAndFamily ? szYes : szNo);
150      SetDlgItemText(hdlg, IDC_TM_VECTOR, TMPF_VECTOR & pdp->tm.tmPitchAndFamily ? szYes : szNo);
151      SetDlgItemText(hdlg, IDC_TM_TRUETYPE, TMPF_TRUETYPE & pdp->tm.tmPitchAndFamily ? szYes : szNo);
152      SetDlgItemText(hdlg, IDC_TM_DEVICE, TMPF_DEVICE & pdp->tm.tmPitchAndFamily ? szYes : szNo);
153
154      // Display font family information
155      SetDlgItemText(hdlg, IDC_TM_FAMILY, szFamily[min(6, pdp->tm.tmPitchAndFamily >> 4)]);
156
157      // Set character set and face name
158      SetDlgItemInt(hdlg, IDC_TM_CHARSET, pdp->tm.tmCharSet, FALSE);
159      SetDlgItemText(hdlg, IDC_TM_FACENAME, pdp->szFaceName);
160  }
```

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.

```
140 void MySetMapMode(HDC hdc, int iMapMode) {
141     // Switch statement to handle different mapping modes
142     switch (iMapMode) {
143         case IDC_MM_TEXT:
144             SetMapMode(hdc, MM_TEXT);
145             break;
146         case IDC_MM_LOMETRIC:
147             SetMapMode(hdc, MM_LOMETRIC);
148             break;
149         case IDC_MM_HIMETRIC:
150             SetMapMode(hdc, MM_HIMETRIC);
151             break;
152         case IDC_MM_LOENGLISH:
153             SetMapMode(hdc, MM_LOENGLISH);
154             break;
155         case IDC_MM_HIENGLISH:
156             SetMapMode(hdc, MM_HIENGLISH);
157             break;
158         case IDC_MM_TWIPS:
159             SetMapMode(hdc, MM_TWIPS);
160             break;
161         case IDC_MM_LOGTWIPS:
162             SetMapMode(hdc, MM_ANISOTROPIC);
163             SetWindowExtEx(hdc, 1440, 1440, NULL);
164             SetViewportExtEx(hdc, GetDeviceCaps(hdc, LOGPIXELSX), GetDeviceCaps(hdc, LOGPIXELSY), NULL);
165             break;
166     }
167 }
```

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

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

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

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

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

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

Message Handling:

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

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

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

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

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

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

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

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

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

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

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

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

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

Constants and Macros:

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

```
135  // Constants and Macros
136  #define BCHARFORM TEXT("0x%04X")
137  #define IDM_DEVICE_SCREEN 1001
138  #define IDM_DEVICE_PRINTER 1002
139  // ... (Other constant and resource ID definitions)
140
141  // Entry point of the program
142  int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow) {
143      HWND hwnd;
144      MSG msg;
145      WNDCLASS wndclass;
146
147      wndclass.style = CS_HREDRAW | CS_VREDRAW;
148      wndclass.lpfnWndProc = WndProc;
149      wndclass.cbClsExtra = 0;
150      wndclass.cbWndExtra = 0;
151      wndclass.hInstance = hInstance;
152      wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
153      wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
154      wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
155      wndclass.lpszMenuName = szAppName;
156      wndclass.lpszClassName = szAppName;
157
158      if (!RegisterClass(&wndclass)) {
159          MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
160          return 0;
161      }
162
163      hwnd = CreateWindow(szAppName, TEXT("PickFont: Create Logical Font"),
164                          WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN, CW_USEDEFAULT, CW_USEDEFAULT,
165                          CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL, hInstance, NULL);
166
167      ShowWindow(hwnd, iCmdShow);
168      UpdateWindow(hwnd);
169
170      while (GetMessage(&msg, NULL, 0, 0)) {
171          if (hdlg == 0 || !IsDialogMessage(hdlg, &msg)) {
172              TranslateMessage(&msg);
173              DispatchMessage(&msg);
174          }
175      }
176
177      return msg.wParam;
178  }
```

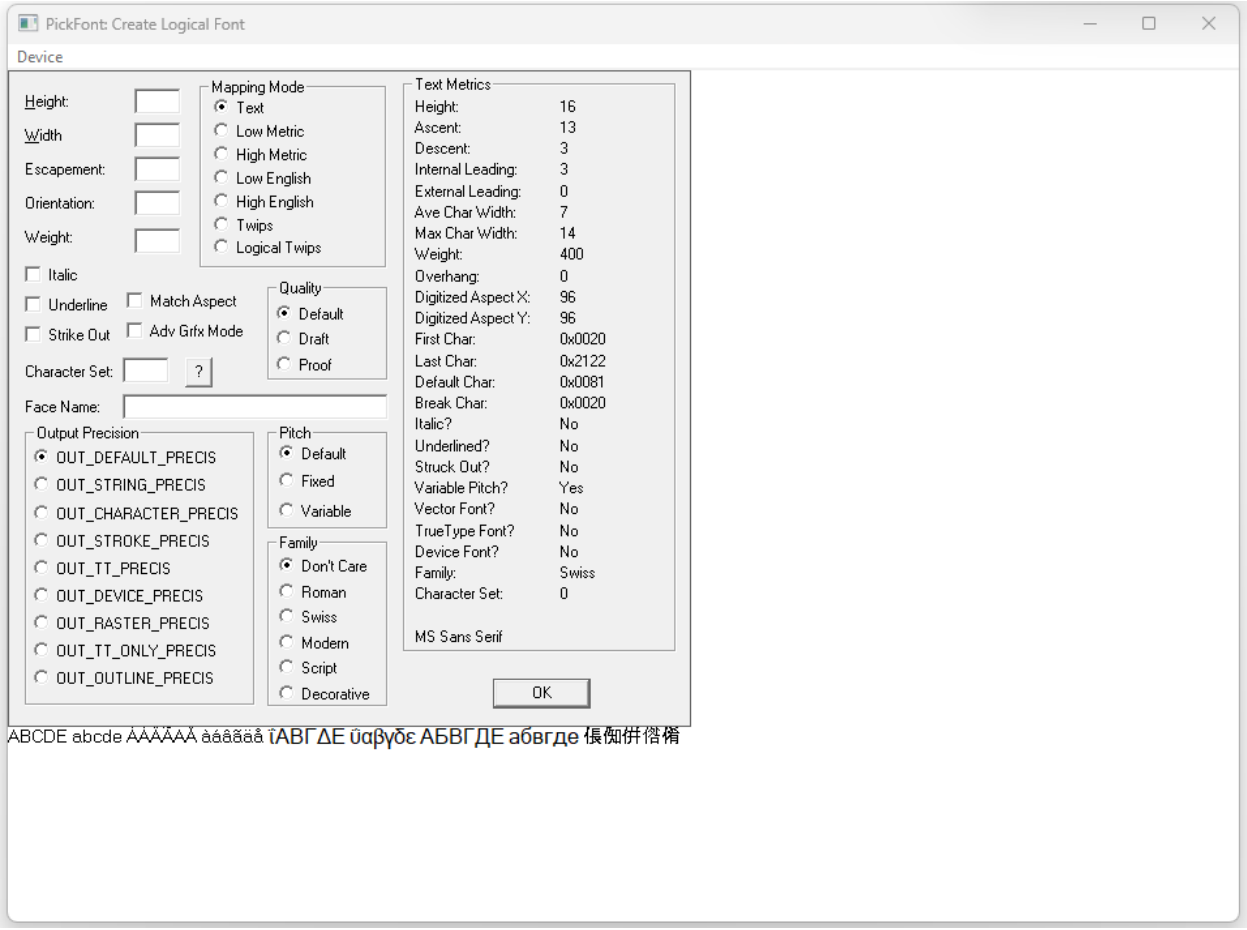
Resource IDs: Constants like IDM_DEVICE_SCREEN and IDM_DEVICE_PRINTER represent menu items for selecting the device type. Other constants, such as IDOK and IDC_LF_HEIGHT, serve as control IDs to identify specific controls in the dialog box.

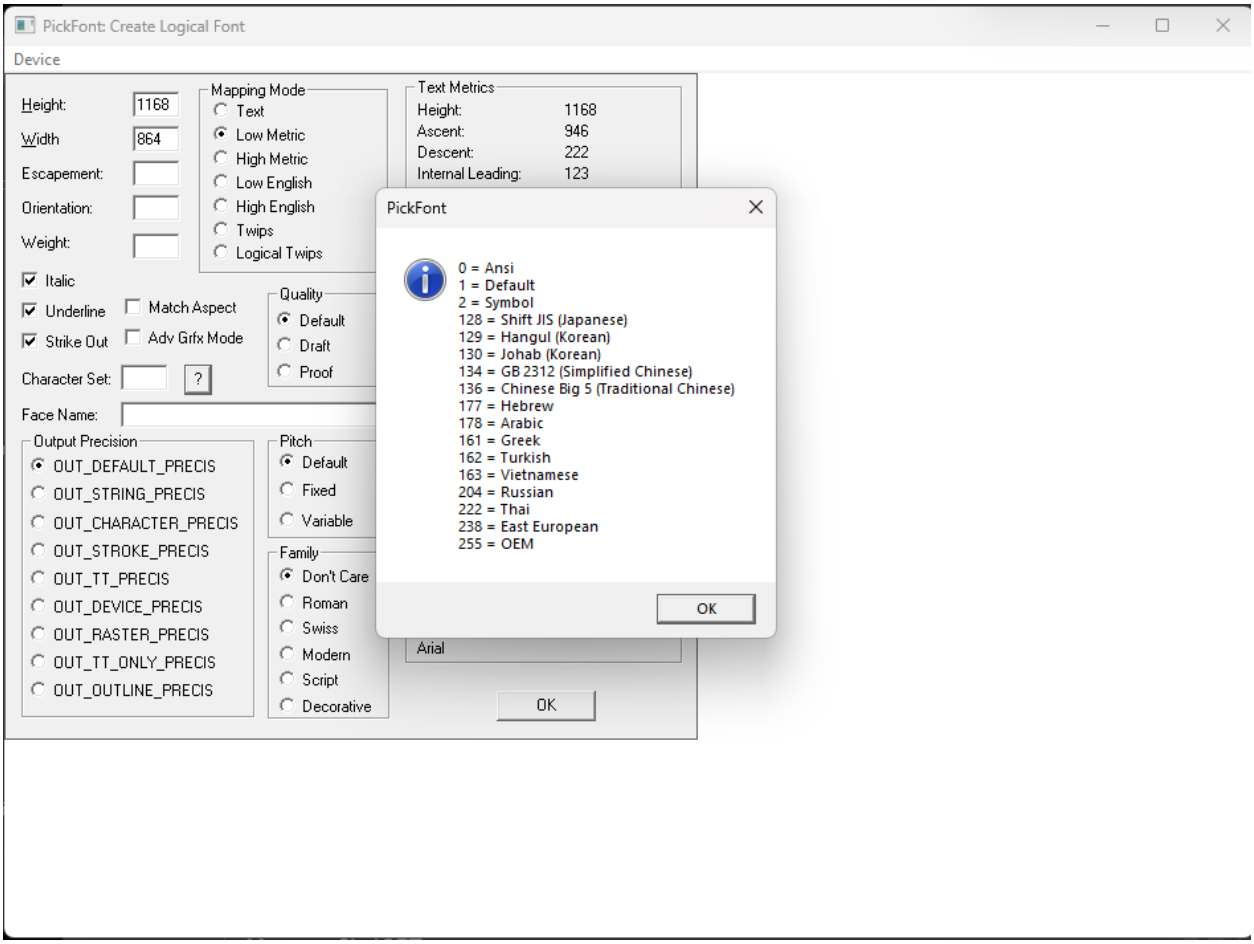
Formatting Macros: The BCHARFORM macro is used to display the BYTE character in a hexadecimal format. It ensures consistent formatting, and its definition depends on whether UNICODE is enabled.

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:

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

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

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

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

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

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

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

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

Creating a Font Using CreateFontIndirect:

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

HFONT hFont = CreateFontIndirect(&lf);

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

Accessing Font Metrics Using GetTextMetrics:

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

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

Experimenting with PICKFONT (example code):

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

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

lfHeight:

Specifies the desired height of characters in logical units.

Values:

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

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

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

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

lfWidth:

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

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

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

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

Intelligent Adjustment: To intelligently adjust the character width:

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

lfOrientation:

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

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

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

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

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

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

lfEscapement:

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

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

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

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

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

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

Examples:

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

Platform-Specific Behavior:

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

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

Key Points:

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

Now, let's address your additional questions:

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

lfWeight:

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

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

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

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

Visual representation:

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

Examples:

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

Key Points:

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

lfItalic:

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

Behavior:

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

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

lfUnderline:

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

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

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

lfStrikeOut:

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

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

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

lfCharSet:

Purpose: Specifies the character set of the font.

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

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

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

lfOutPrecision:

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

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

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

lfClipPrecision:

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

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

lfQuality:

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

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

Flags:

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

lfPitchAndFamily:

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

Pitch:

Lowest two bits:

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

Font Family:

Upper half of the byte:

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

lfFaceName:

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

Usage for TrueType Italic or Boldface Fonts:

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

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

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

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.

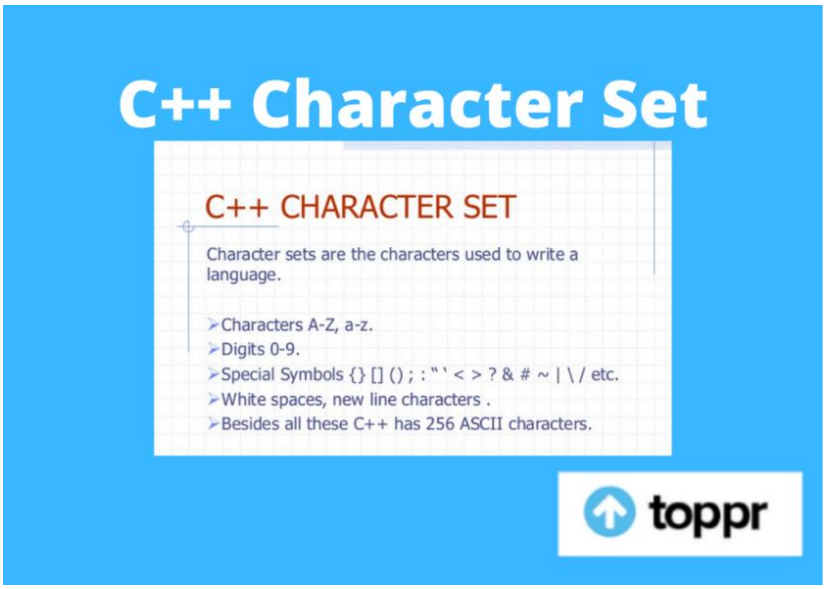
Upon setting up the logical font structure, the CreateFontIndirect function is utilized to obtain a handle to the logical font. Subsequently, when the SelectObject function is employed to select this logical font into a device context, Windows employs the font-mapping algorithm to find the most suitable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

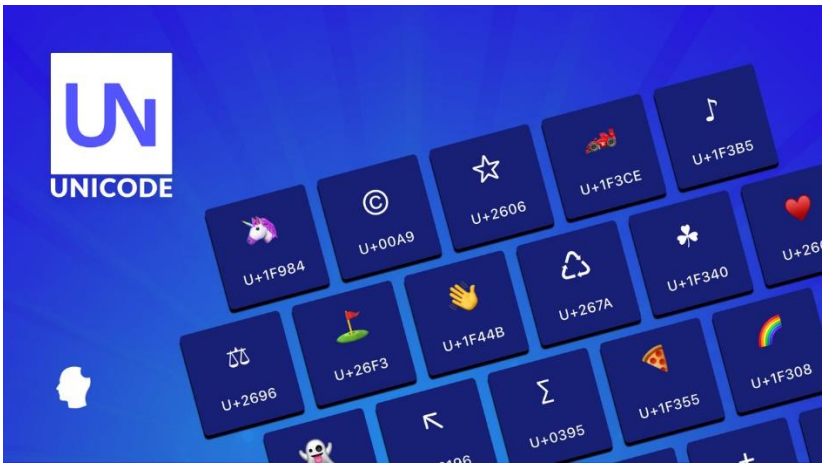
Let's break a bit…

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.



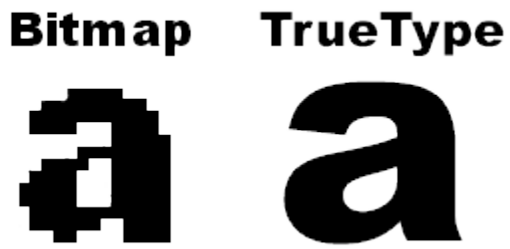
ASCII and Unicode, on the other hand, are character encoding standards. ASCII represents a set of characters that are commonly used in the English language and is a subset of Unicode. Unicode, on the other hand, is a much larger character set that encompasses a wide range of characters from various languages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

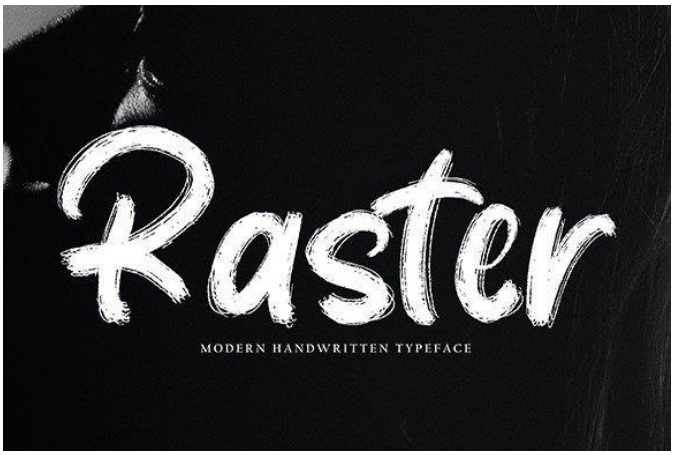
However, the introduction of TrueType "Big Fonts" allows a single TrueType font to be mapped to different character sets, including the OEM character set. To access the Symbol font or Wingdings font, SYMBOL_CHARSET (2) is now utilized.



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.

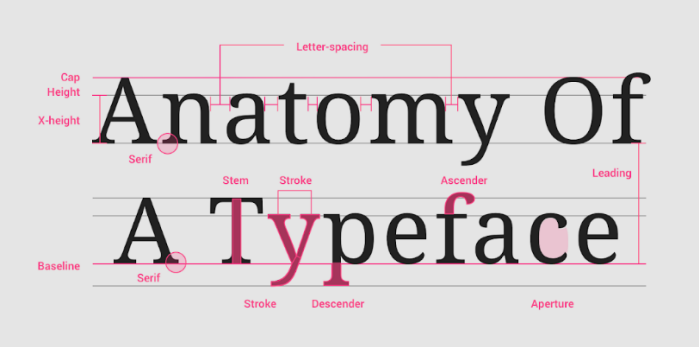
Pro tips for font-savvy developers:

- ✓ 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.

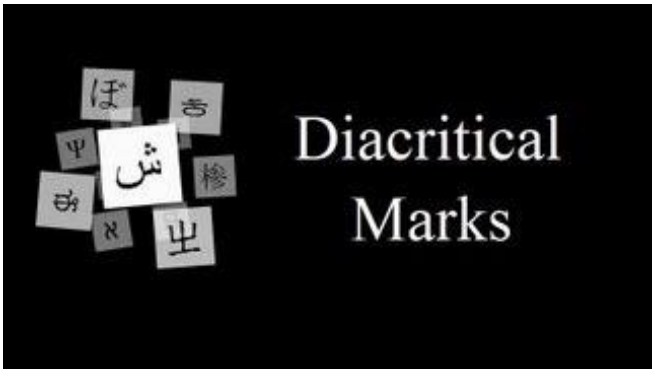
The information retrieved about a font in PICKFONT, particularly from the GetTextMetrics function, provides a detailed breakdown of various characteristics stored in the TEXTMETRIC structure. Each field in this structure holds essential details about the font's properties and dimensions, all presented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

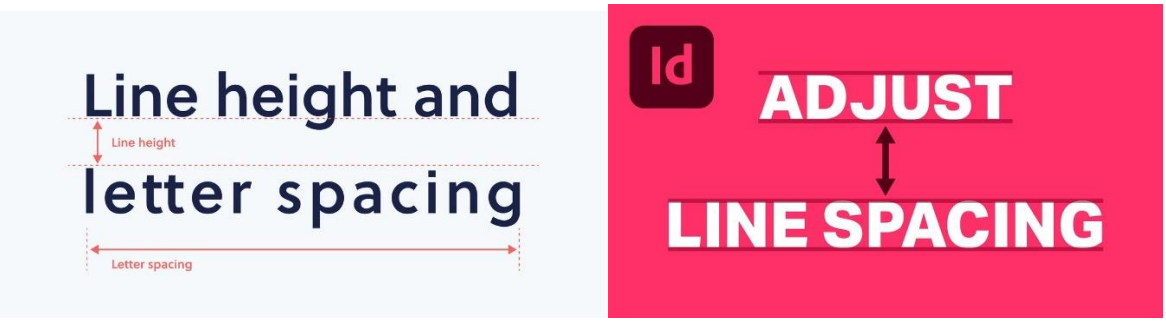
tmAscent and tmDescent: These fields denote the vertical size of the character above and below the baseline in logical units, respectively.



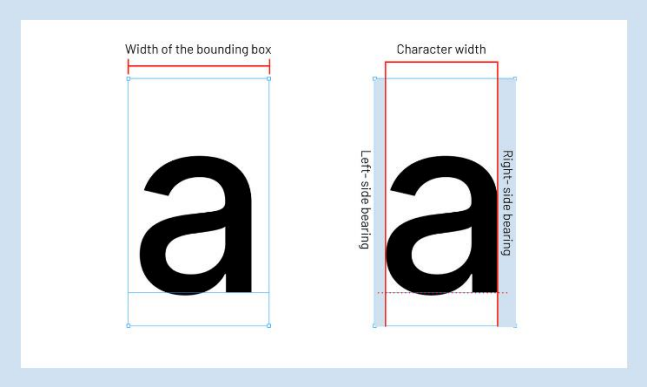
tmInternalLeading: This is a vertical size included in tmHeight, typically occupied by diacritics on some capital letters. Calculating the point size of the font involves subtracting tmInternalLeading from tmHeight.



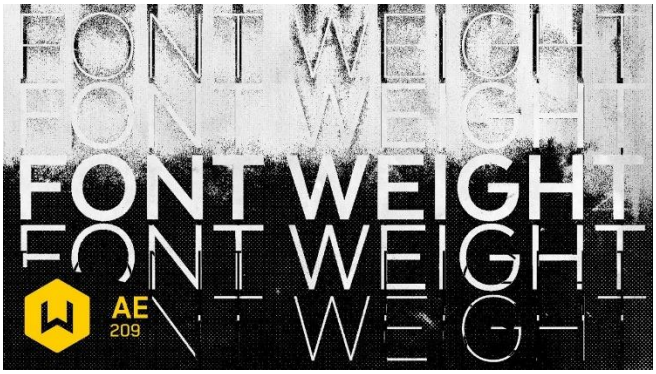
tmExternalLeading: It represents an additional amount of line spacing beyond tmHeight, recommended by the font designer for spacing successive lines of text.



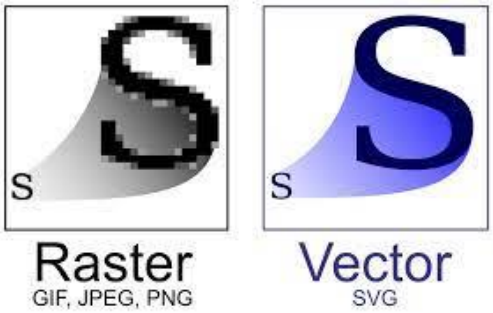
tmAveCharWidth and tmMaxCharWidth: These fields provide the average width of lowercase letters and the width of the widest character in logical units. For a fixed-pitch font, tmMaxCharWidth is the same as tmAveCharWidth.



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.



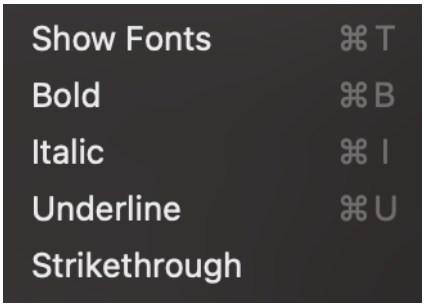
tmOverhang: Specifies the extra width added to a raster font character when synthesizing italic or boldface. For a boldface font, tmAveCharWidth minus tmOverhang equals tmAveCharWidth for the same font without boldfacing.



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.

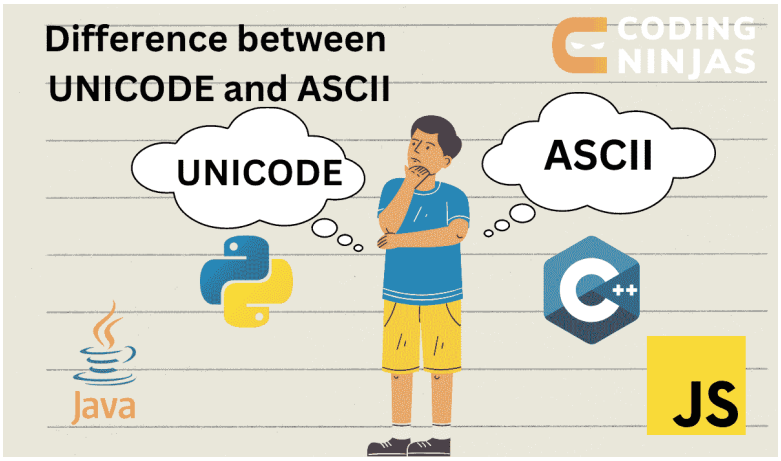


tmPitchAndFamily: This field combines information about the font's characteristics. The low-order bits indicate specific attributes, such as fixed pitch, vector (TrueType), and device font. The top four bits represent the font family, using values from the LOGFONT lfPitchAndFamily field. To summarize, 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.

Value	Identifier
0x01	TMPF_FIXED_PITCH
0x02	TMPF_VECTOR
0x04	TMPF_TRUETYPE
0x08	TMPF_DEVICE

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.

```
// LOGFONT structure (partial)
typedef struct tagLOGFONTA {
    LONG lfHeight;
    LONG lfWidth;
    LONG lfEscapement;
    LONG lfOrientation;
    LONG lfWeight;
    BYTE lfItalic;
    BYTE lfUnderline;
    BYTE lfStrikeOut;
    BYTE lfCharSet;  // Character set identifier
    // ... (other fields)
} LOGFONTA, *PLOGFONTA, *NPLOGFONTA, *LPLOGFONTA;
```

```
// TEXTMETRIC structure (partial)
typedef struct tagTEXTMETRICA {
    LONG tmHeight;
    LONG tmAscent;
    LONG tmDescent;
    LONG tmInternalLeading;
    LONG tmExternalLeading;
    LONG tmAveCharWidth;
    LONG tmMaxCharWidth;
    LONG tmWeight;
    LONG tmOverhang;
    LONG tmDigitizedAspectX;
    LONG tmDigitizedAspectY;
    BYTE tmFirstChar;
    BYTE tmLastChar;
    BYTE tmDefaultChar;
    BYTE tmBreakChar;
    BYTE tmItalic;
    BYTE tmUnderlined;
    BYTE tmStruckOut;
    BYTE tmPitchAndFamily;
    BYTE tmCharSet;  // Character set identifier
} TEXTMETRICA, *PTEXTMETRICA, *NPTEXTMETRICA, *LPTEXTMETRICA;
```

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.

```c
/*--------------------------------------
   EZFONT.C -- Easy Font Creation
               (c) Charles Petzold, 1998
  --------------------------------------*/

#include <windows.h>
#include <math.h>
#include "ezfont.h"

HFONT EzCreateFont (HDC hdc, TCHAR * szFaceName, int iDeciPtHeight,
                    int iDeciPtWidth, int iAttributes, BOOL fLogRes)
{
     FLOAT      cxDpi, cyDpi ;
     HFONT      hFont ;
     LOGFONT    lf ;
     POINT      pt ;
     TEXTMETRIC tm ;

     SaveDC (hdc) ;

     SetGraphicsMode (hdc, GM_ADVANCED) ;
     ModifyWorldTransform (hdc, NULL, MWT_IDENTITY) ;
     SetViewportOrgEx (hdc, 0, 0, NULL) ;
     SetWindowOrgEx   (hdc, 0, 0, NULL) ;

     if (fLogRes)
     {
          cxDpi = (FLOAT) GetDeviceCaps (hdc, LOGPIXELSX) ;
          cyDpi = (FLOAT) GetDeviceCaps (hdc, LOGPIXELSY) ;
     }
     else
     {
          cxDpi = (FLOAT) (25.4 * GetDeviceCaps (hdc, HORZRES) /
                                  GetDeviceCaps (hdc, HORZSIZE)) ;

          cyDpi = (FLOAT) (25.4 * GetDeviceCaps (hdc, VERTRES) /
                                  GetDeviceCaps (hdc, VERTSIZE)) ;
     }

     pt.x = (int) (iDeciPtWidth  * cxDpi / 72) ;
     pt.y = (int) (iDeciPtHeight * cyDpi / 72) ;

     DPtoLP (hdc, &pt, 1) ;

     lf.lfHeight         = - (int) (fabs (pt.y) / 10.0 + 0.5) ;
     lf.lfWidth          = 0 ;
     lf.lfEscapement     = 0 ;
     lf.lfOrientation    = 0 ;
     lf.lfWeight         = iAttributes & EZ_ATTR_BOLD      ? 700 : 0 ;
     lf.lfItalic         = iAttributes & EZ_ATTR_ITALIC    ?   1 : 0 ;
     lf.lfUnderline      = iAttributes & EZ_ATTR_UNDERLINE ?   1 : 0 ;
     lf.lfStrikeOut      = iAttributes & EZ_ATTR_STRIKEOUT ?   1 : 0 ;
     lf.lfCharSet        = DEFAULT_CHARSET ;
     lf.lfOutPrecision   = 0 ;
     lf.lfClipPrecision  = 0 ;
     lf.lfQuality        = 0 ;
     lf.lfPitchAndFamily = 0 ;

     lstrcpy (lf.lfFaceName, szFaceName) ;

     hFont = CreateFontIndirect (&lf) ;

     if (iDeciPtWidth != 0)
     {
          hFont = (HFONT) SelectObject (hdc, hFont) ;
          GetTextMetrics (hdc, &tm) ;
          DeleteObject (SelectObject (hdc, hFont)) ;
          lf.lfWidth = (int) (tm.tmAveCharWidth * fabs (pt.x) / fabs (pt.y) + 0.5) ;

          hFont = CreateFontIndirect (&lf) ;
     }

     RestoreDC (hdc, -1) ;
     return hFont ;
}
```

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.

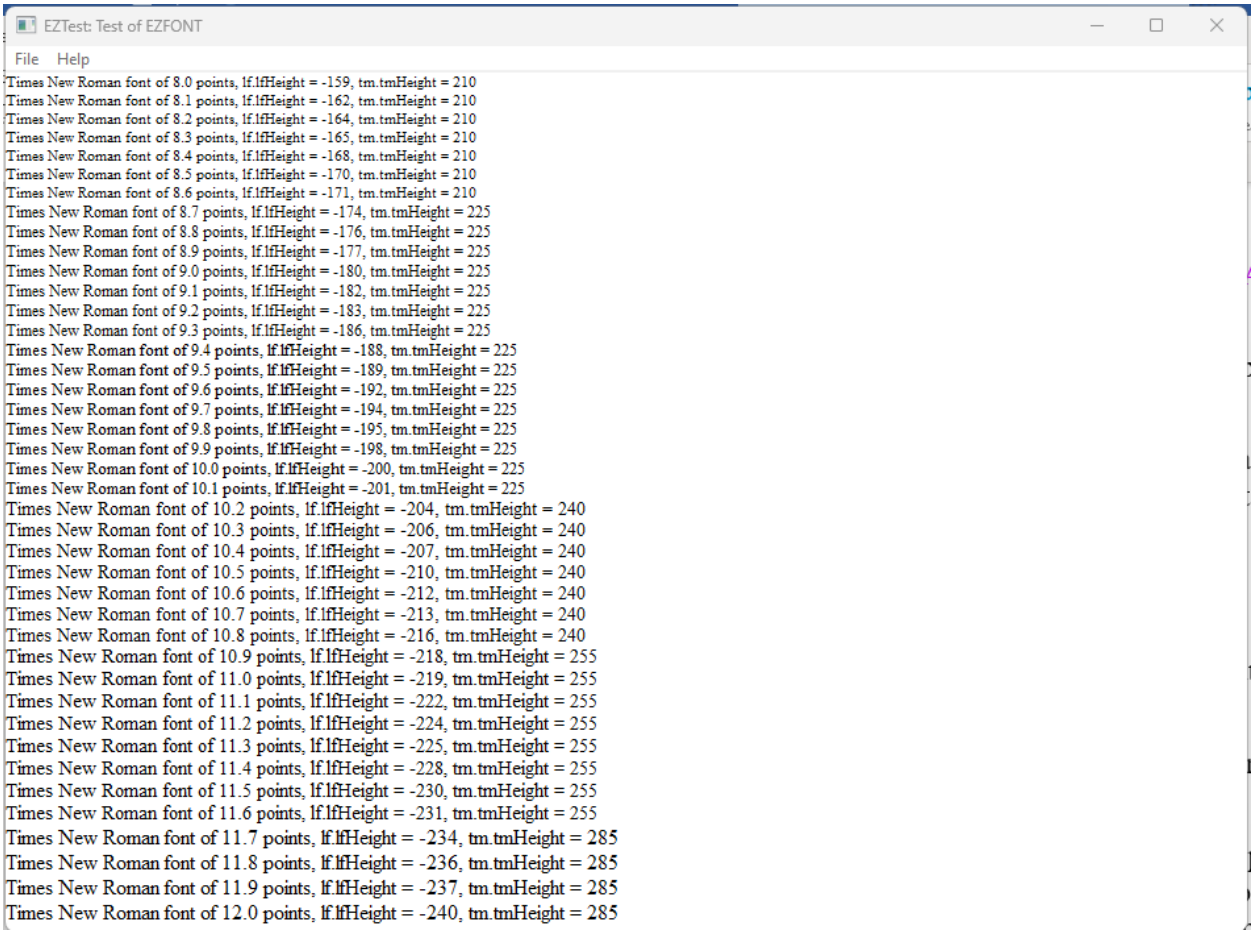
Moreover, it might offer more intuitive customization options, contributing to a user-friendly interface for font-related operations.

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.

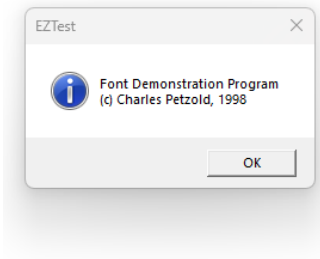


Recommendations:

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 to mention:

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.

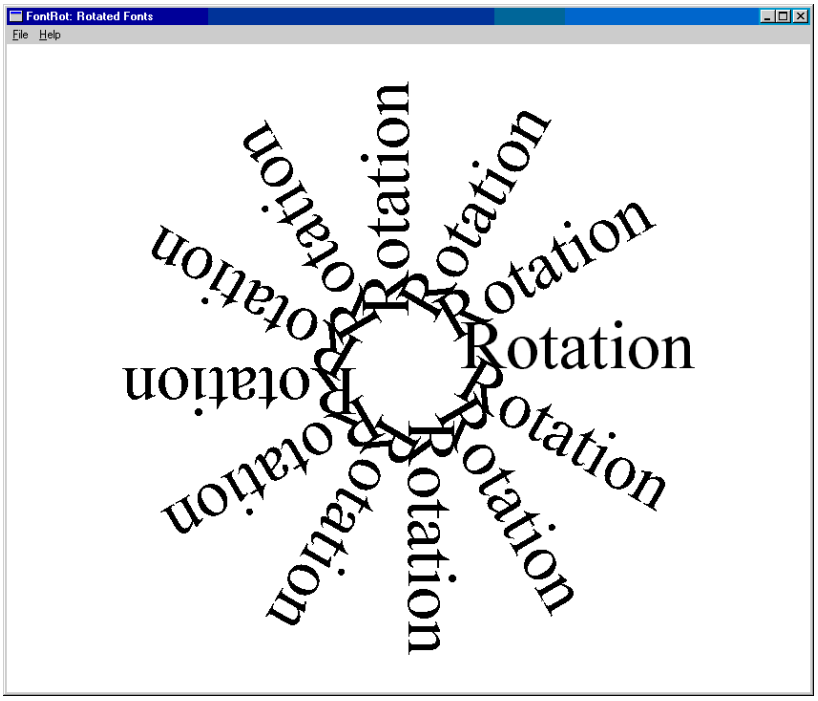
Real-World Implications:

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.

FONT ROTATION IN WINDOWS GDI: A DYNAMIC EXPLORATION

The FONTROT program unveils the captivating ability to rotate TrueType fonts within the realm of Windows GDI. It gracefully demonstrates how text can be liberated from rigid horizontal alignment and brought to life at a variety of angles, introducing a new dimension of expressiveness to text-based graphics.



Behind the Scenes: A Technical Tour

Laying the Foundation:

The program starts by using EzCreateFont to create a temporary font called Times New Roman with a size of 54 points. It then carefully retrieves the font's LOGFONT structure, which contains important information for manipulating the font. Finally, the program responsibly releases the temporary font, showing good resource management practices.

Preparing the Canvas:

To achieve a clean presentation, the program carefully makes the background transparent, ensuring that the rotated text stands out. It establishes a consistent text alignment by aligning the text to its baseline in a visually pleasing manner. Importantly, it moves the starting point of the drawing area to the center, creating a central point around which the text rotates gracefully.

Rotation Unleashed: 12 Steps of Elegance

The program performs a 12-step dance, each step representing a 30-degree rotation increment, covering a full circle of possibilities. In each step:

It adjusts the LOGFONT structure's lfEscapement and lfOrientation fields with the desired rotation angle in tenths of a degree.

It uses CreateFontIndirect to create a new font embodying the specified rotation.

The newly created font is carefully selected into the device context, ready to leave its unique mark on the canvas.

The word "Rotation" slowly appears on the screen, with each letter elegantly angled according to the font's rotation.

To ensure efficient memory usage, the program gracefully says goodbye to the rotated font, maintaining a clean and optimal memory landscape.

Visual Symphony: A Showcase of TrueType Flexibility

The careful execution of this dance produces a captivating visual showcase. The screen is adorned with twelve instances of the word "Rotation," each elegantly angled at a unique degree, forming a stunning starburst of text.

This compelling demonstration reaffirms the impressive flexibility of TrueType fonts within Windows GDI, empowering developers to craft visually dynamic and expressive text-based experiences with professionalism and finesse.

Key Takeaways:

The LOGFONT structure acts as a versatile tool for controlling font attributes, such as rotation angles, with precision.

GDI adeptly handles text rotation, even with intricate TrueType fonts, showcasing its versatility in rendering text.

The viewport origin plays a crucial role in text rotation, serving as the axis around which text gracefully revolves.

FONTROT serves as a prime example of responsible font resource management, emphasizing the significance of proper font handling for maintaining performance and avoiding memory leaks.

If you are seeking a more versatile solution for graphics rotation and other linear transformations, and your programs are limited to running on Windows NT, you can utilize the XFORM matrix and the world transform functions.

XFORM MATRIX: THE MASTER CHOREOGRAPHER:

This structure, defined as XFORM, serves as a mathematical blueprint for choreographing diverse transformations within Windows NT's graphics realm.

It encompasses six crucial elements:

- eM11, eM12, eM21, eM22: These elements collectively define the core transformation matrix, governing scaling, rotation, shearing, and other linear transformations.
- eDx, eDy: These elements specify horizontal and vertical translations, allowing movement of objects within the coordinate space.

World Transform Functions: Stage Directors:

These functions act as stage directors, meticulously applying the transformations encoded within XFORM matrices to the graphics world:

- SetWorldTransform(hdc, &xform): Instructs the device context hdc to embrace the specified xform matrix as its prevailing world transformation, shaping subsequent drawing operations.
- ModifyWorldTransform(hdc, &xform, MWT_LEFTMULTIPLY): Gracefully blends the specified xform matrix with the existing world transform, typically through left-multiplication, expanding the range of achievable transformations.

Code Illustration: Rotating Text:

```c
#include <windows.h>

// Function to rotate text by a specified angle in degrees
void RotateText(HDC hdc, const TCHAR* text, int x, int y, int angle) {
    XFORM xform = { 0 };  // Initialize XFORM structure
    xform.eM11 = cos(angle * M_PI / 180);
    xform.eM12 = sin(angle * M_PI / 180);
    xform.eM21 = -sin(angle * M_PI / 180);
    xform.eM22 = cos(angle * M_PI / 180);

    SetWorldTransform(hdc, &xform);  // Apply rotation to the world transform
    TextOut(hdc, x, y, text, lstrlen(text));  // Render rotated text
    SetWorldTransform(hdc, NULL);  // Restore default world transform
}
```

NT-Specific Capabilities for Transformation:

The advanced transformation techniques discussed here are exclusive to Windows NT systems. These capabilities provide developers with powerful tools for manipulating graphics using linear transformations.

Versatile Transformations with XFORM and World Transforms:

The XFORM matrix and world transform functions offer a wide range of possibilities for linear transformations. These transformations include scaling, rotation, shearing, and translation. By leveraging these capabilities, developers can precisely control the positioning, orientation, and size of graphical elements.

Cumulative Effects and ModifyWorldTransform:

One of the advantages of using XFORM and world transforms is the ability to combine multiple transformations. By using the ModifyWorldTransform function, developers can apply multiple transformations successively, creating intricate and complex visual effects.

Restoring the Default World Transform:

To ensure proper behavior and avoid unintended impacts on subsequent drawing operations, it is crucial to restore the default world transform after applying custom transformations. This can be achieved by using the SetWorldTransform function with a NULL parameter.

Additional Notes:

Compatibility: While the discussed techniques are specific to Windows NT, it's worth noting that modern versions of Windows offer alternative approaches using graphics APIs like Direct2D. These APIs provide similar transformation capabilities and are compatible with a broader range of Windows systems.

Matrix Mastery: Understanding matrix operations is essential for effectively utilizing XFORM and world transforms. Mastery of matrices allows developers to manipulate transformations precisely and create visually appealing graphics.

Although the XFORM and world transform functions, introduced in Windows NT, are still available in Windows 10 for compatibility with legacy applications, newer graphics APIs like Direct2D and Direct3D offer more advanced and efficient options for graphics rendering and transformation.

While you can use XFORM and world transform functions in Windows 10, considering the specific requirements of your application, exploring the newer graphics APIs may provide enhanced transformation capabilities.

FONT ENUMERATION

Font enumeration is the process of retrieving a list of all available fonts on a device using the Graphics Device Interface (GDI). This list can then be utilized by a program for various purposes, such as font selection or presenting options to the user. Let's explore the enumeration functions and how to use the ChooseFont function as an alternative for font selection.

A Trio of Enumeration Functions:

EnumFonts: A Legacy Yet Functional Tool: While rooted in earlier Windows versions, EnumFonts remains a viable option for basic font enumeration tasks. It dutifully provides information about either all installed fonts or those matching a specified typeface.

```
EnumFonts(hdc, szTypeFace, EnumProc, pData);
```

EnumFontFamilies: Enhanced TrueType Support: Designed specifically for TrueType fonts, this function excels in environments where TrueType font handling is paramount. It operates in a two-stage process, first identifying font families and then delving into individual fonts within those families.

```
EnumFontFamilies(hdc, szFaceName, EnumProc, pData);
```

EnumFontFamiliesEx: Fine-Grained Control for Modern Windows: Representing the recommended approach for 32-bit Windows systems, EnumFontFamiliesEx offers the highest degree of customization and control over the enumeration process. It gracefully accepts a LOGFONT structure, enabling developers to meticulously specify enumeration criteria.

```
EnumFontFamiliesEx(hdc, &logfont, EnumProc, pData, dwFlags);
```

ChooseFont Function:

The ChooseFont function simplifies font enumeration and selection in applications:

```
CHOOSEFONT cf;
LOGFONT lf;
```

Initialize the CHOOSEFONT and LOGFONT structures:

```
ZeroMemory(&cf, sizeof(CHOOSEFONT));
cf.lStructSize = sizeof(CHOOSEFONT);
cf.lpLogFont = &lf;
cf.Flags = CF_SCREENFONTS | CF_EFFECTS;
```

Invoke the ChooseFont dialog:

```
if (ChooseFont(&cf)) {
    // The selected font information is now in the lf member of the LOGFONT structure.
    // You can use it for further processing in your application.
}
```

This approach avoids the direct use of enumeration functions, and the user can interactively choose a font through the standard Windows font selection dialog.

In summary, font enumeration functions provide a way to gather information about available fonts, but for user-friendly font selection, the ChooseFont function is a more convenient option.

The Callback Function: An Important Partner:

Within GDI's realm, the callback function holds a crucial role. GDI faithfully calls upon this function for every enumerated font, granting it the task of processing the font information it provides.

Flexibility at Your Fingertips: Developers have the ability to customize the behavior of the callback function. This allows them to perform a variety of tasks, such as building font lists for user selection, gathering font metrics for precise text layout, and identifying fonts that meet specific criteria.

Structures: Blueprints for Font Information:

LOGFONT: The Font Architect: This essential structure meticulously defines the core attributes of a font, such as its typeface, size, style, and pitch.

TEXTMETRIC: Revealing Font Metrics: This structure elegantly unveils crucial metrics like font height, width, and spacing, providing essential insights for text layout calculations.

ENUMLOGFONT: Enhanced Information for TrueType Fonts: Building upon the LOGFONT structure, this expanded structure reveals the full name and style of a font, offering a more comprehensive understanding of its characteristics.

NEWTEXTMETRIC: Additional Insights for TrueType Fonts: Designed specifically for TrueType fonts, this structure complements TEXTMETRIC by providing additional fields tailored to uncover the unique attributes of TrueType fonts.

CHOOSEFONT: A USER-FRIENDLY SHORTCUT:

Simplifying Font Selection: When you need to allow users to choose fonts, the ChooseFont function gracefully presents a familiar dialog box. It empowers users to effortlessly explore and select fonts within a visually intuitive interface.

CHOOSEFONT Structure:

Delving into lf: The Font Architect: This LOGFONT structure meticulously defines the font's core attributes, including typeface, height, width, escapement, orientation, weight, italic, underline, strikeout, character set, output precision, clipping precision, quality, pitch, and family.

Fine-Tuning Dialog Behavior: Flags: Developers can leverage flags within the CHOOSEFONT structure to tailor the dialog box's appearance and functionality, such as:

- CF_SCREENFONTS: Display only screen-compatible fonts.
- CF_EFFECTS: Enable strikeout and underline effects.
- CF_INITTOLOGFONTSTRUCT: Initialize the dialog with a font specified in the lf field.
- CF_NOVECTORFONTS: Exclude vector fonts.

ChooseFont Function Extended:

Guiding Font Selection: ChooseFont presents a visual interface for font exploration and selection, eliminating the need for manual font enumeration.

Return Value: Upon successful font selection, ChooseFont returns true and populates the CHOOSEFONT structure with the user's choices.

Program Flow Enhancements:

Initialization Insights:

GetDialogBaseUnits provides default character height, ensuring consistent text display across devices.

GetObject retrieves the system font's LOGFONT structure, establishing a baseline for font settings.

User-Driven Font Exploration: The "Font!" menu item activates ChooseFont, inviting users to discover their preferred fonts.

Repainting with Selected Font: InvalidateRect triggers a visual update, showcasing the newly chosen font in all its glory.

Unveiling Font Information:

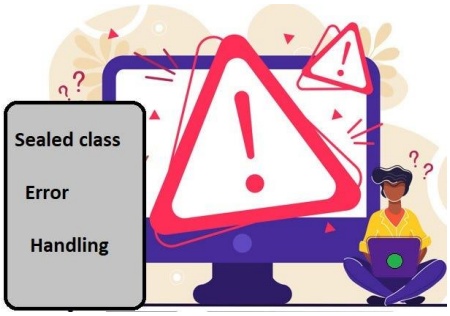
CreateFontIndirect: This function crafts a font object based on the LOGFONT structure, ready for text rendering.

TextOut: This function paints text onto the window's canvas, utilizing the selected font and color.

Displaying Font Attributes: The program visually presents key LOGFONT structure fields, providing transparency into the font's characteristics and empowering users to make informed font choices.

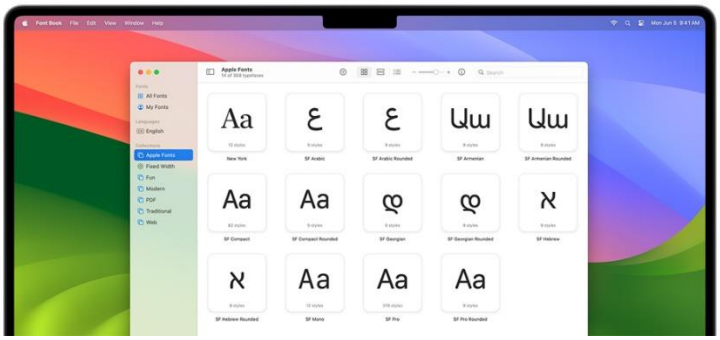
Additional Considerations:

Error Handling: Robust applications should gracefully handle scenarios where ChooseFont fails to return a valid font selection.



Font Compatibility: Developers should consider font availability across different platforms and devices to ensure consistent rendering.

Accessibility: ChooseFont can be further enhanced with accessibility features, such as font size adjustment and high-contrast text options, to accommodate users with diverse needs.



Integration with Text Rendering: Seamlessly integrate ChooseFont and font customization options into text rendering functions like TextOut for a cohesive user experience.

Advanced Font Handling: Explore techniques like font fallback and font linking for comprehensive font management and improved text display.

I'll expand upon the previous notes, providing additional insights and examples:

Enriching Font Dialog Customization:

Beyond Basic Flags: While flags like CF_SCREENFONTS and CF_EFFECTS offer fundamental control, delve deeper into more nuanced customizations:

Exerting Precision: Fine-tune output and clipping precision with lfOutPrecision and lfClipPrecision.

Prioritizing Quality: Set desired output quality with lfQuality.

Pitch Perfect: Define pitch (fixed or variable) and font family with lfPitchAndFamily.

LOGFONT Structure: Unveiling Font Essence:

Beyond Common Attributes: Explore lesser-known LOGFONT fields to expand font control:

Angles of Expression: lfEscapement and lfOrientation govern text angles for creative effects.

Character Set Harmony: lfCharSet ensures compatibility with diverse languages and symbols.

Script Field: Embracing Global Communication:

Bridging Language Barriers: The "Script" field plays a crucial role in supporting multilingual text rendering and localization, ensuring accurate character display for a global audience.

Logical Inch: Bridging Point Size and Font Height:

Context-Driven Consistency: The logical inch concept ensures font heights remain proportional to screen resolutions and printing devices, maintaining visual consistency across different display environments.

Key for Text Metrics: Understanding logical inch calculations is essential for precise text layout and positioning within applications.

Mapping Mode Matters:

Metric mapping modes (e.g., MM_LOMETRIC, MM_HIMETRIC) under Windows NT can lead to inconsistencies between logical coordinates and physical font sizes.

Use Logical Twips for graphics that align seamlessly with font size.

Temporarily revert to MM_TEXT for font selection and display to ensure proper font height interpretation.

Device Context Sensitivity:

lfHeight is pixel-based, suitable for video displays.

Adjust lfHeight for printer device contexts based on printer resolution.

The hDC field in CHOOSEFONT merely lists printer fonts, not influencing lfHeight.

Leveraging iPointSize for Flexibility:

This field provides font size in 1/10 point units, adaptable to various device contexts and mapping modes.

Reference EZFONT.C for conversion code, tailorable to specific needs.

Unicode Character Exploration:

The UNICHARS program showcases all characters within a font, ideal for studying Unicode-rich fonts like Lucida Sans Unicode and Bitstream CyberBit.

Its TextOutW function ensures compatibility across Windows NT and Windows 98.

Additional Insights for Font Mastery:

Prioritize MM_TEXT for Font Operations: Stick to MM_TEXT for font selection and display to avoid interpretation conflicts.

Calculate Font Heights Accurately: Ensure precise font height calculation for both screen and printer device contexts.

Experiment with Unicode Fonts: Embrace applications like UNICHARS to explore diverse character sets and enhance text rendering capabilities.

Stay Updated with Font Technologies: Keep abreast of advancements in font rendering and management techniques to deliver optimal user experiences.

UNICHARS.C PROGRAM

The "UNICHARS.C" program is designed to display 16-bit character codes in a graphical user interface. Let's break down its functionality in paragraphs:

The program creates a window that allows users to explore and visualize Unicode characters. The window includes a vertical scrollbar that enables navigation through different pages of Unicode characters. Each page consists of a grid displaying 16 rows and 16 columns of Unicode characters.

Upon initialization, the program sets up the default font and appearance for the Unicode character display. The default font is "Lucida Sans Unicode," and its size is determined to be 12 points, ensuring legibility and a visually pleasing presentation. The ChooseFont common dialog box is also integrated, allowing users to customize the font used for character display.

Users can access the font customization feature by selecting the "Font!" option from the program's menu. This invokes the ChooseFont dialog, providing options to modify the font type, size, and other attributes. Once a new font is chosen, the program refreshes the display to reflect the selected font, ensuring a dynamic and user-friendly experience.

The main window of the program incorporates a vertical scrollbar, facilitating the navigation of Unicode character pages. Users can scroll up or down, either one line at a time or an entire page, to explore different sets of Unicode characters. The scrollbar's position is visually represented, offering a clear indication of the current page being viewed.

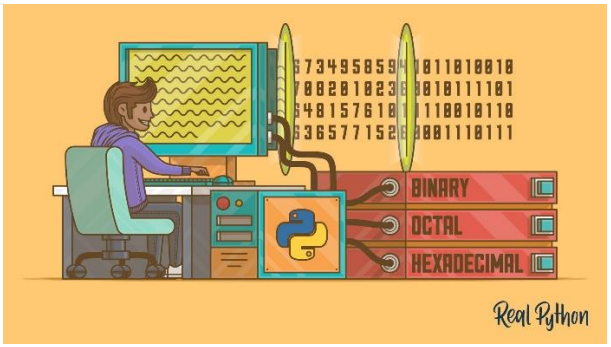
The core of the program lies in its ability to paint the Unicode characters onto the window's canvas. It calculates the layout for each character, considering factors such as character width, height, and external leading. The grid structure organizes the characters into rows and columns, making it easy for users to navigate and locate specific characters of interest.

As the program processes paint messages, it dynamically updates the display based on user interactions. For instance, when scrolling through pages, the program adjusts the content to show the corresponding set of Unicode characters for the selected page. The grid format allows for an organized and systematic presentation of characters, aiding users in their exploration.

In summary, the "UNICHARS.C" program provides an interactive platform for users to explore and visualize 16-bit Unicode characters. It integrates font customization through the ChooseFont dialog, incorporates a user-friendly navigation system with a scrollbar, and dynamically updates the display based on user interactions. The program's design aims to enhance the user's experience in discovering and understanding Unicode characters.

Delving into the World of Unicode Characters:

UNICHARS Program: The UNICHARS program provides a visual representation of the vast array of Unicode characters. It organizes these characters in a carefully arranged grid, with each character having a unique hexadecimal code that serves as its digital identifier.

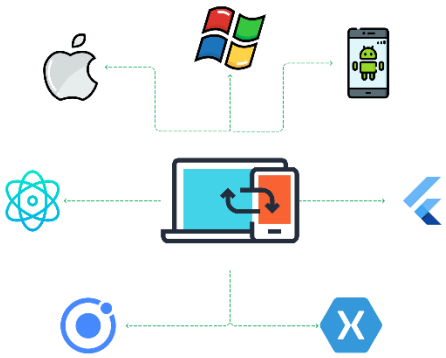


Exploring Character Blocks: Users can embark on an engaging exploration of the 256 character blocks within Unicode. Each block contains 256 individual characters. A vertical scrollbar facilitates seamless navigation through this multilingual landscape, allowing users to discover and examine various characters.

Customizing Fonts: The program offers a personal touch by allowing users to select their preferred font. By accessing the "Font!" menu item and invoking the ChooseFont dialog, users can choose a font that aligns with their aesthetic preferences. By default, the program uses Lucida Sans Unicode, a font widely recognized for its comprehensive support of Unicode characters.

Key Features Amplified:

Cross-Platform Compatibility: The program seamlessly operates on both Windows NT and Windows 98, ensuring accessibility to a wider audience across different operating systems.



Unicode Focus: The program prioritizes Unicode support by utilizing TextOutW, a function specifically designed for accurate rendering of Unicode characters. This ensures faithful representation of diverse scripts and symbols, promoting effective communication across languages.

Font Customization: Users can personalize their experience by selecting their preferred font. This feature allows individuals to cater to their aesthetic and readability preferences, enhancing their exploration of Unicode characters.

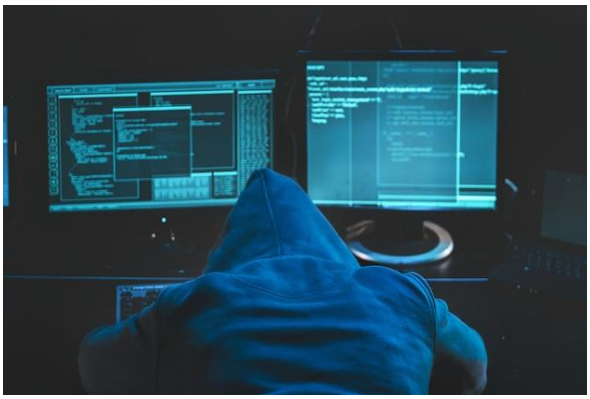
Character Exploration: UNICHARS serves as a valuable tool for visually examining a wide range of Unicode characters. It fosters a deeper understanding of multilingual communication, character encoding, and the intricacies of human language in the digital domain.

Beyond the Technicalities: Potential Applications:

Font Exploration and Research: Developers, designers, and typography enthusiasts can leverage UNICHARS to delve into the nuances of different fonts, evaluate their Unicode character support, and discover hidden gems within diverse font libraries.

Character Set Research and Development: Linguists, language enthusiasts, and those working with multilingual text can utilize UNICHARS to explore character sets, uncover relationships between characters, and gain insights into language representation and encoding.

Unicode-Related Development: Software engineers can reference UNICHARS for examples of font selection and Unicode character rendering techniques, applying these concepts in their own projects to create applications that seamlessly handle diverse languages and scripts.



UNDERSTANDING PARAGRAPH FORMATTING:

Goal: Arrange text lines within margins, aligned left, right, centered, or justified (flush with both margins).

DrawText Limitations: While convenient for simple tasks, it lacks flexibility for more complex formatting.

Key Functions for Text Formatting:

GetTextExtentPoint32: Determines the width and height of text in logical units based on the current font.

TextOut: Writes text at a specified location within a device context.

SetTextJustification: Adjusts spacing for justified text.

- Left alignment: Ideal for most body text, offering a natural reading flow.
- Right alignment: Useful for specific elements like dates and page numbers.
- Center alignment: Effective for titles, headings, and quotations to draw attention.
- Justified alignment: Creates a clean, professional look, but use cautiously as excessive stretching can impair readability.

Formatting a Single Line:

```
// Get text extents
GetTextExtentPoint32(hdc, szText, lstrlen(szText), &size);

// Left-aligned text
xStart = xLeft;
TextOut(hdc, xStart, yStart, szText, lstrlen(szText));

// Right-aligned text
xStart = xRight - size.cx;
TextOut(hdc, xStart, yStart, szText, lstrlen(szText));

// Centered text
xStart = (xLeft + xRight - size.cx) / 2;
TextOut(hdc, xStart, yStart, szText, lstrlen(szText));

// Justified text
SetTextJustification(hdc, xRight - xLeft - size.cx, 3);  // Distribute extra space among 3 spaces
xStart = xLeft;
TextOut(hdc, xStart, yStart, szText, lstrlen(szText));

// Clear justification error for next line
SetTextJustification(hdc, 0, 0);
```

GetTextExtentPoint32: A versatile tool for measuring text dimensions, enabling accurate placement within margins.

TextOut: The workhorse for displaying text, offering flexibility in positioning and alignment.

SetTextJustification: The key to achieving justified text, carefully distributing extra space between words for visual harmony.

Remember to clear justification errors: Call SetTextJustification(hdc, 0, 0) before starting a new line to ensure consistent formatting.

Formatting Multiple Lines:

To format multiple lines of text, you can follow these steps:

1.

Calculate Available Width: Determine the available width for the text by subtracting the right margin from the left margin. This will be the maximum width that each line of text can occupy.

```
int leftMargin = 10;   // Example left margin
int rightMargin = 290;   // Example right margin
int availableWidth = rightMargin - leftMargin;
```

2.

Break Text into Lines: Split the text into individual lines using word-wrapping algorithms or manual splitting. This process ensures that each line does not exceed the available width.

```c
#include <string.h>

#define MAX_LINE_LENGTH 80  // Adjust as needed
#define MAX_WORDS 50        // Adjust as needed

int main() {
    char text[] = "Lorem ipsum dolor sit amet, consectetur adipiscing elit.";
    char lines[MAX_WORDS][MAX_LINE_LENGTH];
    int currentLine = 0, wordCount = 0, wordLength;
    char *word;

    word = strtok(text, " ");  // Tokenize the text by spaces

    while (word != NULL && wordCount < MAX_WORDS) {
        wordLength = strlen(word);

        if (strlen(lines[currentLine]) + wordLength + 1 <= MAX_LINE_LENGTH) { //Check if word fits current line
            strcat(lines[currentLine], word);  // Append word to current line
            strcat(lines[currentLine], " ");  // Add a space
        } else {
            currentLine++;  // Move to the next line
            strcpy(lines[currentLine], word);  // Start a new line with the word
            strcat(lines[currentLine], " ");  // Add a space
        }

        wordCount++;
        word = strtok(NULL, " ");  // Get the next word
    }

    // Print the wrapped lines
    for (int i = 0; i <= currentLine; i++) {
        printf("%s\n", lines[i]);
    }

    return 0;
}
```

The provided C code snippet is meticulously crafted to implement a word-wrapping algorithm, transforming a raw text into neatly formatted lines. Let's break down the key components and functionalities of the code.

Essential Headers:

The inclusion of stdio.h and string.h sets the foundation for the code's functionality. These headers bring in essential functions for input/output and string manipulation, respectively.

Setting Boundaries:

The constants MAX_LINE_LENGTH and MAX_WORDS establish crucial boundaries for the word-wrapping process. These constants ensure that lines adhere to a specified maximum length, preventing overflow and managing the total number of words processed.

Textual Home:

The raw text to be formatted is housed in a character array named text. This array serves as the starting point for the word-wrapping algorithm.

Lines Awaiting Construction:

The 2D character array lines is designated to hold the formatted text. This array serves as a canvas where the lines will be constructed as the word-wrapping algorithm unfolds.

Guiding Variables:

Variables such as currentLine, wordCount, and wordLength play pivotal roles in tracking progress and making decisions during the word-wrapping process.

Splitting Text into Words:

The strtok function makes its entrance, skillfully tokenizing the text array into individual words based on space delimiters. This sets the stage for the subsequent word-wrapping actions.

Word-by-Word Journey:

The code enters a loop, iteratively processing each word extracted by strtok. Within this loop, a meticulous sequence of actions takes place for each word:

Measuring Word Length: The strlen function precisely determines the length of the current word.

Line-Breaking Decisions: A critical check assesses whether the word can fit within the remaining space on the current line. This decision considers the length of the word, existing text on the line, and space for separation.

Appending or Starting Anew: Depending on the assessment, the word is either appended to the current line using strcat, ensuring a cohesive flow, or a new line is initiated with strcpy, placing the word at the beginning for a balanced presentation.

Unveiling the Wrapped Text:

In a final act, the code iterates through the lines array, which now holds the beautifully wrapped lines. Using a printf loop, each line is elegantly printed to the console, revealing the text in its newly formatted glory. This explanation showcases the transformation of raw text into well-organized lines.

In the realm of text formatting for Windows applications, certain additional considerations and best practices play a pivotal role in enhancing the visual appeal and readability of content.

Additional Considerations:

Font Size and Style:

These factors wield significant influence over text dimensions and formatting, allowing developers to tailor the visual presentation based on the chosen font characteristics.

Paragraph Spacing:

Fine-tuning vertical spacing between paragraphs is a key aspect of controlling the overall layout and improving the visual flow of textual content.

Indentation:

Horizontal spacing at the beginning of lines, achieved through indentation, contributes to a well-organized and aesthetically pleasing text structure.

Tab Stops:

The ability to set tab stops enables precise control over text alignment in tabular layouts, ensuring a clean and structured appearance.

Best Practices:

Font Selection:

Optimal font choices play a crucial role in readability and overall aesthetics. Developers are encouraged to select fonts that not only enhance readability but also complement the content and application theme.

Line Spacing:

Providing adequate vertical spacing between lines contributes to visual comfort and clarity. Striking the right balance ensures an inviting and easily readable text presentation.

Justification:

While text justification can enhance visual appeal, it should be used judiciously. Excessive stretching may hinder readability, so developers are advised to apply justification thoughtfully.

Hyphenation:

Consideration of hyphenation for breaking long words can significantly improve the appearance of justified text. Thoughtful implementation enhances both the visual aesthetics and overall readability.

Mastering Techniques:

By mastering these techniques, developers can create visually appealing and well-formatted text in their Windows apps. The careful consideration of font size, style, paragraph spacing, indentation, tab stops, and adherence to best practices allows developers to craft content that not only meets functional requirements, while providing an engaging and user-friendly experience.

3.

Format Each Line: Apply the desired text alignment (left, right, center, or justified) to each line of text as needed.

```c
#include <stdio.h>
#include <string.h>
#define MAX_LINES 50  // Adjust as needed
#define MAX_LINE_LENGTH 80  // Adjust as needed

enum TextAlignment { LEFT, RIGHT, CENTER, JUSTIFIED };

int main() {
    char lines[MAX_LINES][MAX_LINE_LENGTH];
    int availableWidth = 60;  // Example width
    TextAlignment alignment = LEFT;  // Example alignment
    // ... (Assuming lines are already populated with text)
    for (int i = 0; i < MAX_LINES; i++) {
        int lineLength = strlen(lines[i]);
        int padding = availableWidth - lineLength;
        switch (alignment) {
            case LEFT:
                // No additional padding needed
                break;
            case RIGHT:
                // Add padding to the left
                for (int j = 0; j < padding; j++) {
                    lines[i][j] = ' ';
                }
                lines[i][padding] = '\0';  // Terminate string after padding
                break;
            case CENTER:
                // Add padding to both sides
                int halfPadding = padding / 2;
                memmove(lines[i] + halfPadding, lines[i], lineLength + 1);  // Shift content to the right
                for (int j = 0; j < halfPadding; j++) {
                    lines[i][j] = ' ';
                }
                for (int j = halfPadding + lineLength; j < padding + lineLength; j++) {
                    lines[i][j] = ' ';
                }
                lines[i][padding + lineLength] = '\0';  // Terminate string
                break;
            case JUSTIFIED:
                // Implement justification logic here
                break;
        }
        // Output the formatted line
        printf("%s\n", lines[i]);
    }
    return 0;
}
```

The provided code demonstrates text alignment techniques in C, specifically handling left, right, and center alignment.

It assumes that an array of lines (lines) has already been populated with text content. The code utilizes the stdio.h library for input/output operations and the string.h library for string manipulation.

The main function begins by declaring the lines array, which has a maximum number of lines (MAX_LINES) and a maximum line length (MAX_LINE_LENGTH).

It also declares variables for the available width of the lines and the desired text alignment, represented by the availableWidth and alignment variables, respectively.

The program then proceeds to iterate through each line in the lines array. For each line, it measures the length of the text using strlen and calculates the padding needed by subtracting the line length from the available width.

A switch statement is used to handle different alignment cases.

- For the LEFT alignment, no additional padding is required, so the code moves to the next line without making any adjustments.
- For the RIGHT alignment, the code adds padding spaces to the left of the line. It iterates over the padding length and assigns a space character to each position in the line array. The resulting line is terminated with a null character to ensure proper string termination.
- For the CENTER alignment, the code adds padding spaces to both sides of the line. It calculates half of the padding length and uses memmove to shift the existing content of the line to the right. Then, it iterates over the first half of the padding length and assigns a space character to each position before the shifted content. It also assigns space characters for the remaining half of the padding length after the shifted content. The resulting line is terminated with a null character.
- The JUSTIFIED alignment case is a placeholder for implementing justification logic, which is not implemented in this example. You can customize this case to implement your desired justification algorithm.

After formatting the line according to the specified alignment, the code outputs the formatted line using printf. It iterates through each line in the lines array and prints it to the console.

The code allows you to adjust constants such as MAX_LINES and MAX_LINE_LENGTH to suit your specific requirements.

4.

Adjust Vertical Position: If you have multiple lines of text, add the line height (size.cy) to the vertical position for each subsequent line to ensure proper spacing.

Tell me more. Can you provide an example of how to use the SetTextJustification function for adjusting spacing in justified text? What are some best practices for selecting fonts that enhance readability and complement the content and application theme? Could you explain how to calculate the available width for text and break it into individual lines using word-wrapping algorithms?

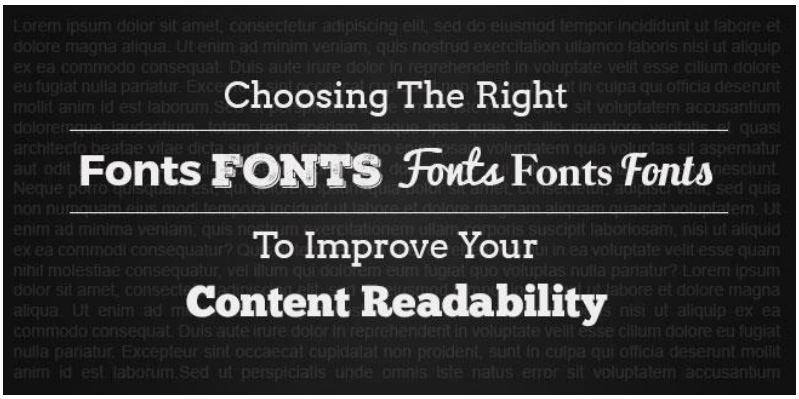
Here's an example of how to use the SetTextJustification function in C programming to adjust spacing for justified text:

```c
#include <windows.h>

int main() {
    HDC hdc = GetDC(NULL); // Get the device context

    // Set up the text attributes
    SetTextAlign(hdc, TA_LEFT);
    SetTextJustification(hdc, 2, 1); // Adjust spacing for justified text

    // Display the justified text
    TextOut(hdc, 100, 100, "Lorem ipsum dolor sit amet", strlen("Lorem ipsum dolor sit amet."));

    // Release the device context
    ReleaseDC(NULL, hdc);

    return 0;
}
```

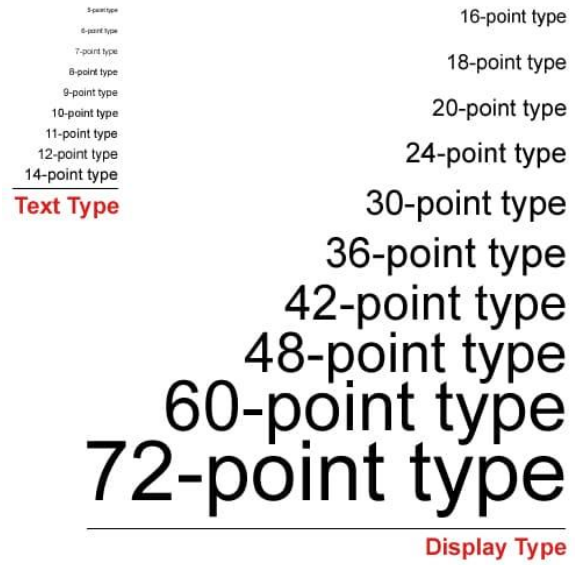
In this example, the SetTextJustification function is used to adjust the spacing for justified text. The second parameter is set to 2, which represents the desired average character spacing. The third parameter is set to 1, which represents the desired minimum space between words.

Regarding font selection, here are some best practices for selecting fonts that enhance readability and complement the content and application theme:

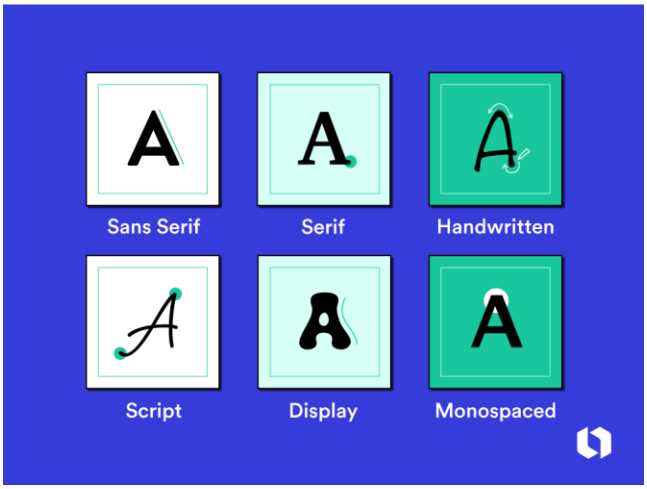
Choose a font with good readability: Select a font that is clear and legible, with distinct letterforms and appropriate spacing between characters. Fonts like Arial, Verdana, and Calibri are known for their readability.



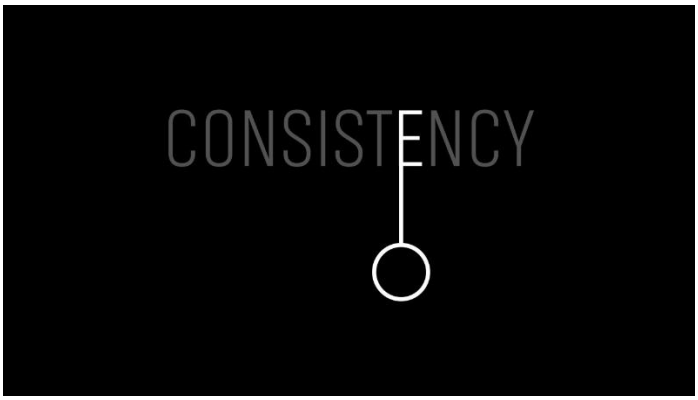
Consider the font size: The font size should be large enough to ensure readability, especially for longer paragraphs of text. Generally, a font size between 10 and 12 points is considered standard for body text.



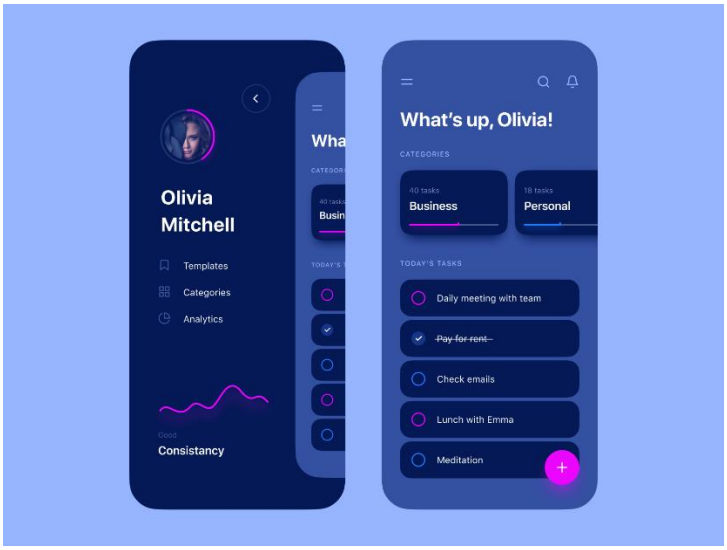
Pay attention to font style: Different font styles (regular, bold, italic) can be used to emphasize specific elements or headings. However, avoid using excessive variations in font styles, as it can make the text harder to read.



Maintain consistency: Consistency in font usage throughout the application creates a cohesive and professional appearance. Limit the use of multiple fonts to avoid visual clutter and confusion.



Consider the application theme: Choose a font that aligns with the overall design and theme of the application. For example, a formal business application may benefit from a classic and traditional font, while a creative app may opt for a more unique and artistic font.



Now, let's discuss how to calculate the available width for text and break it into individual lines using word-wrapping algorithms in C programming.

To calculate the available width for text, you need to determine the width of the container or the area where the text will be displayed.

This could be a window, a specific region, or a predefined size. Once you have the width, you subtract any margins or padding to get the maximum width that each line of text can occupy.

To break the text into individual lines using word-wrapping algorithms, you can use the following approach:

```c
#include <stdio.h>
#include <string.h>
#define MAX_LINE_LENGTH 80  // Maximum line length
#define MAX_LINES 10        // Maximum number of lines

void WrapText(const char* text) {
    char lines[MAX_LINES][MAX_LINE_LENGTH + 1];  // Array to hold formatted lines
    int lineIndex = 0;  // Current line index
    // Tokenize the text into words
    char* word = strtok(text, " ");
    while (word != NULL) {
        // Check if the word can fit within the current line width
        if (strlen(word) <= MAX_LINE_LENGTH - strlen(lines[lineIndex])) {
            // Append the word to the current line
            strcat(lines[lineIndex], word);
            strcat(lines[lineIndex], " ");
        } else {
            // Move to the next line
            lineIndex++;
            // Check if the line index exceeds the maximum number of lines
            if (lineIndex >= MAX_LINES) {
                break;
            }
            // Copy the word to the new line
            strcpy(lines[lineIndex], word);
            strcat(lines[lineIndex], " ");
        }
        // Get the next word
        word = strtok(NULL, " ");
    }
    // Print the formatted lines
    for (int i = 0; i <= lineIndex; i++) {
        printf("%s\n", lines[i]);
    }
}

int main() {
    const char* text = "Lorem ipsum dolor sit amet vitae.";
    WrapText(text);
    return 0;
}
```

In this example, the WrapText function takes the input text and breaks it into individual lines using a word-wrapping algorithm.

The lines array is used to store the formatted lines, with each line having a maximum length of MAX_LINE_LENGTH. The lineIndex variable keeps track of the current line being processed.

The strtok function is used to tokenize the input text into individual words based on space delimiters. Eachword is then checked to see if it fits within the current line width. If it does, the word is appended to the current line with a space. If the word exceeds the available space, a new line is created, and the word is placed there.

The process continues until all words have been processed or the maximum number of lines, specified by MAX_LINES, is reached. Finally, the formatted lines are printed.

Note that this example assumes a fixed maximum line length and a maximum number of lines for simplicity. In a real-world scenario, you may need to handle variable line lengths and dynamically allocate memory to accommodate longer texts.

Remember to adjust the values of MAX_LINE_LENGTH and MAX_LINES according to your requirements.

JUSTIFY.C PROGRAM

1. Header Files and Global Context:

windows.h: This header empowers the program with a rich set of Windows API functions for graphics, window management, and user interactions.

resource.h: This file, while not explicitly shown, holds definitions for application resources such as menus, dialogs, and icons, contributing to the user interface and experience.

Global Variables: These variables establish a shared context for the program's elements:

JUSTIFY1: The application's name, used for window titles and identification.

WndProc: The designated window procedure, responsible for handling events and messages within the main window.

2. WinMain: The Program's Entry Point:

Registration: The code commences by registering the window class, defining its attributes and behaviors within the Windows environment.

Window Creation: A main window is subsequently created, bearing the title "Justified Type #1" and prepared to receive user input and display content.

Message Loop: The program enters a continuous loop, patiently listening for and responding to various events such as key presses, mouse clicks, window resizing, and system notifications. This loop maintains the application's responsiveness and interactivity.

3. DrawRuler: Visualizing Text Boundaries:

Ruler Creation: This function meticulously renders horizontal and vertical rulers adorned with tick marks, serving as visual guides for text alignment and formatting. These rulers provide a clear reference for the user, enhancing the visual clarity of the text layout.

4. Justify: Orchestrating Text Formatting:

Paragraph Processing: This function meticulously crafts the appearance of a paragraph of text within a specified rectangular region. It gracefully handles diverse alignment styles, including left, right, center, and justified.

Line Breaking and Spacing: It meticulously calculates optimal line breaks to ensure text fits within the designated area, and meticulously adjusts spacing between words for justified alignment, achieving a visually balanced and pleasing presentation.

5. WndProc: Responding to Window Events:

Message Handling: This function diligently serves as the central hub for handling various events and messages directed towards the main window. It effectively orchestrates actions based on different types of interactions and system signals.

WM_CREATE: Upon window creation, this message prompts initialization of font and printer dialog structures, setting the stage for user customization and printing capabilities.

WM_COMMAND: This message arises in response to menu commands, triggering actions such as:

- Printing: Engaging the printing process, relying on the Justify function to meticulously render text onto the selected printer.
- Font Selection: Presenting a font dialog to empower users with choices for visual aesthetics and readability.
- Alignment Adjustment: Updating the text alignment based on user preferences, ensuring versatility in text presentation.

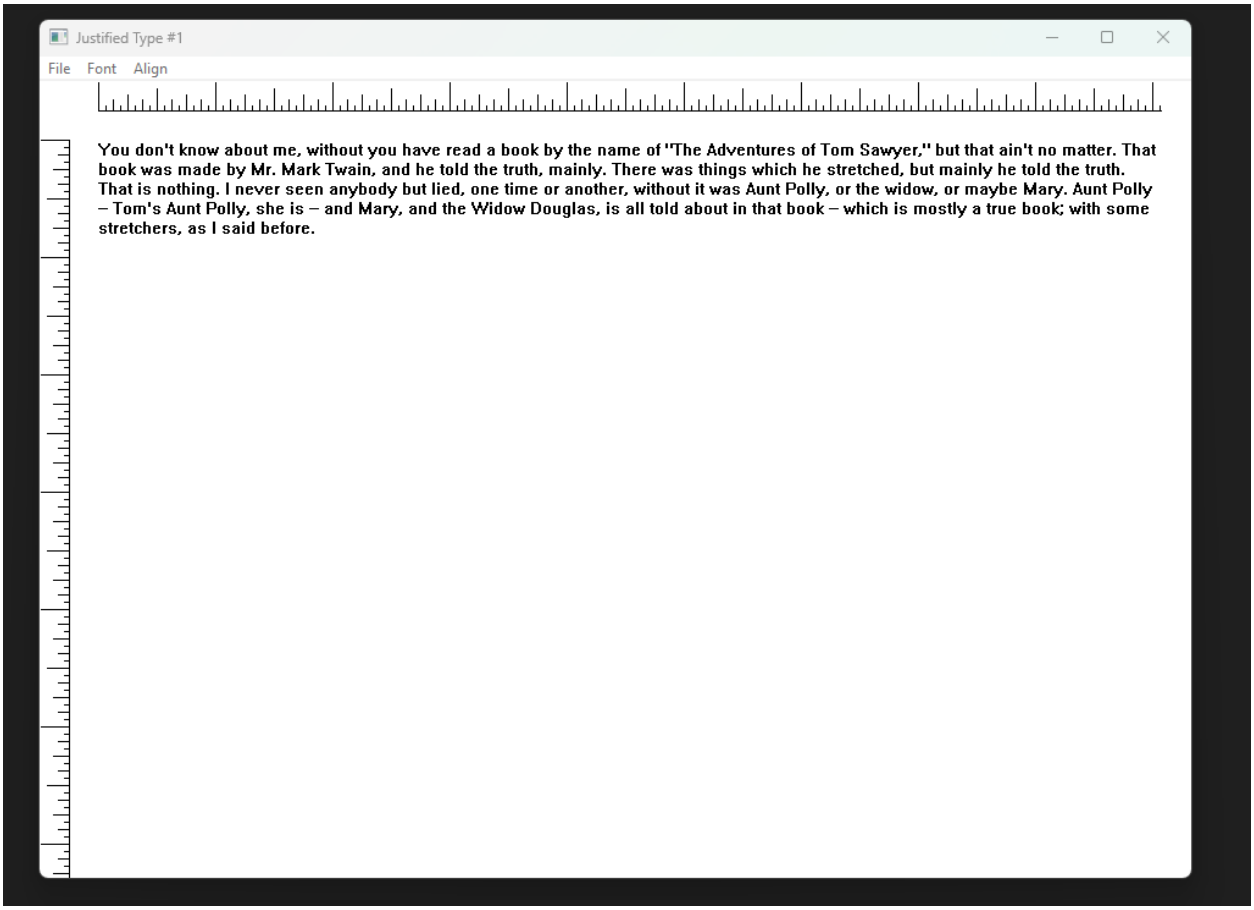
WM_PAINT: This message beckons the program to visually refresh the window's content. It involves:

- Ruler Visualization: Calling upon the DrawRuler function to render the rulers, providing visual guidance for text alignment.
- Text Formatting and Display: Inviting the Justify function to take the reins, formatting and displaying the text according to the chosen alignment and settings.

WM_DESTROY: This message gracefully signals the program's termination, ensuring a proper exit process.

6. Additional Code Sections:

Resource Definitions: These definitions reside within the resource.h file, governing the visual elements and interactive components that shape the user experience.



DrawRulers function in-depth:

Visualizing Text Boundaries: This function's primary mission is to render horizontal and vertical rulers within the application's window. These rulers act as visual aids for users, providing clear reference points for aligning and formatting text content.

Enhancing Text Layout Clarity: By making margins and spacing visually explicit, the rulers foster a more intuitive understanding of text arrangement and visual balance, ultimately contributing to a more polished and professional-looking presentation.

Key Steps:

Preserving Graphics State:

SaveDC(hdc): This function meticulously preserves the current state of the device context (DC), ensuring that any subsequent graphical operations within the DrawRuler function remain isolated and don't inadvertently affect other elements of the window's visual content.

Establishing Ruler Scale and Orientation:

SetMapMode(hdc, MM_ANISOTROPIC): This function empowers the rulers with flexibility by setting the mapping mode to anisotropic. This mode allows for independent scaling of horizontal and vertical dimensions, accommodating different screen resolutions and window sizes without compromising ruler clarity.

SetWindowExtEx(hdc, 1440, 1440, NULL): This function designates a logical coordinate system for the rulers, using 1440 units for both width and height. This logical space aligns with the concept of twips, a unit of measurement often used in typography for precise control over text layout.

SetViewportExtEx(hdc, GetDeviceCaps(hdc, LOGPIXELSX), GetDeviceCaps(hdc, LOGPIXELSY), NULL): This function maps the logical ruler coordinates to physical pixels on the screen, ensuring accurate rendering of the rulers based on the device's resolution.

SetWindowOrgEx(hdc, −720, −720, NULL): This function strategically positions the origin of the ruler's coordinate system at a point half an inch (720 twips) from the top-left corner of the client area, ensuring visual alignment with the text content.

Drawing Ruler Lines:

MoveToEx(hdc, ...) and LineTo(hdc, ...) functions are employed in a coordinated fashion to meticulously draw both horizontal and vertical ruler lines. These lines serve as the foundational structure of the visual guides.

Adding Tick Marks:

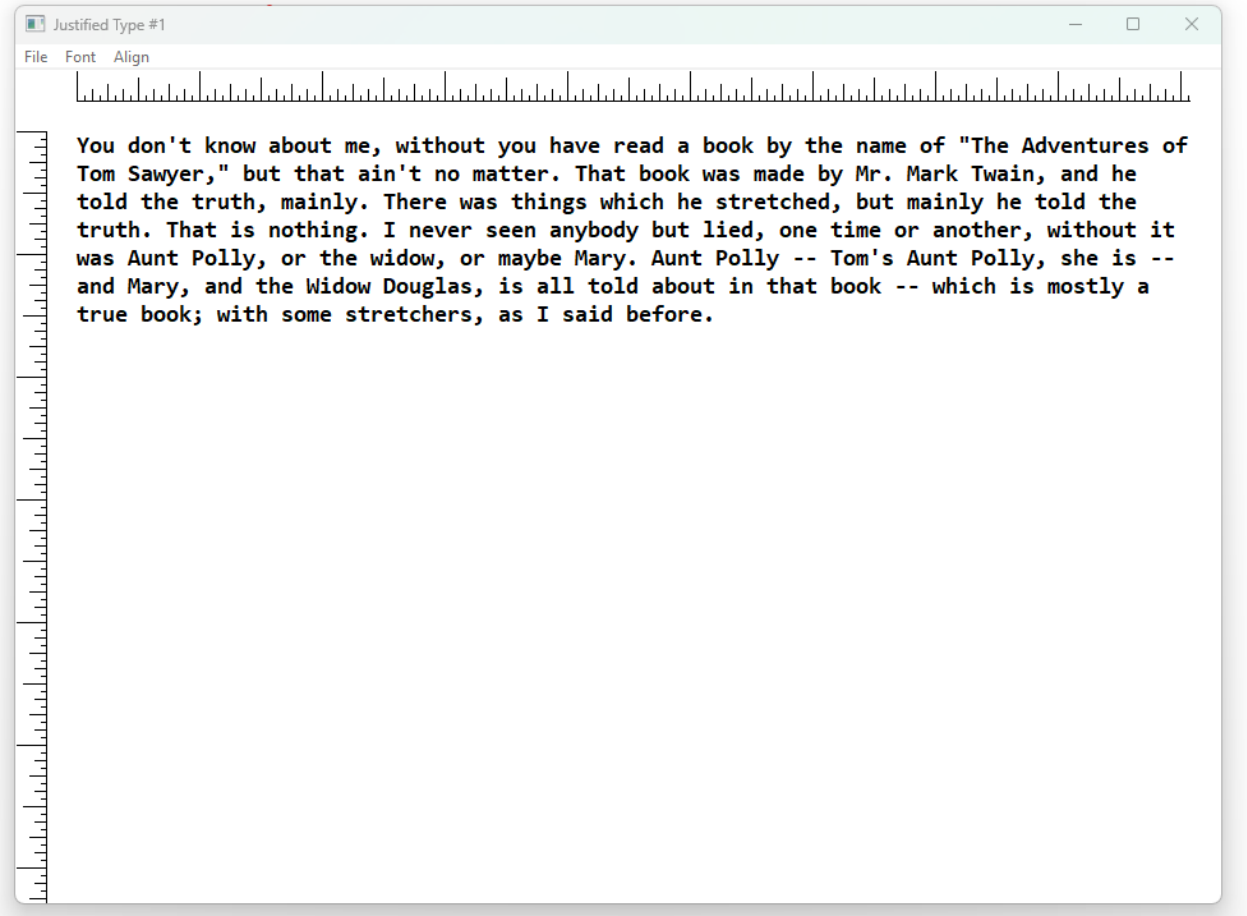
A loop iterates through ruler divisions, rendering tick marks at designated intervals. These tick marks enhance readability and provide more granular reference points for precise text alignment.

The iRuleSize array governs the lengths of tick marks, creating a visual hierarchy that aids in visual scanning and comprehension.

Restoring Graphics State:

RestoreDC(hdc, −1): This function diligently restores the DC to its previous state, ensuring that any graphical modifications made within the DrawRuler function remain confined to their intended scope and don't interfere with other visual elements within the window.

After a font change:

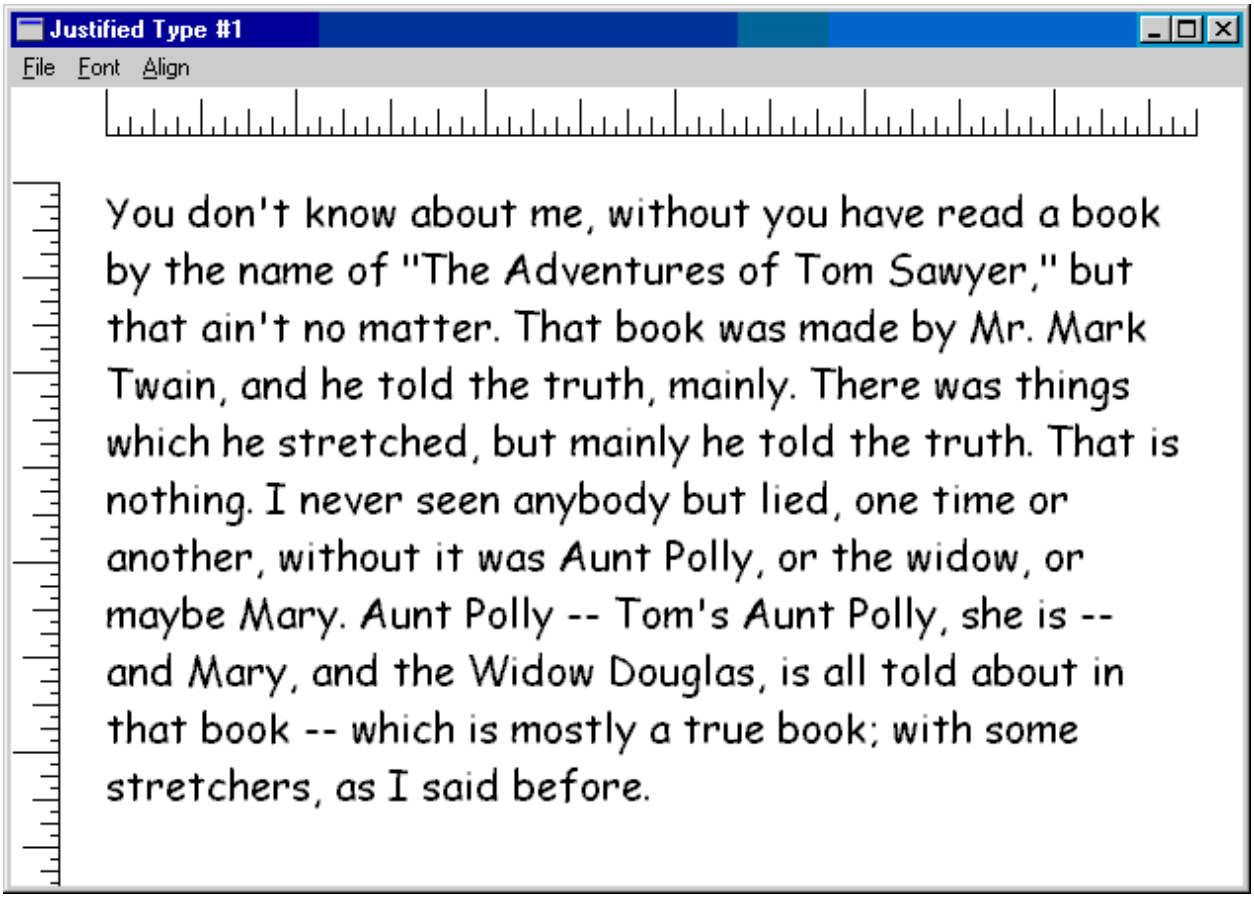


Justify function explained in depth:

Text Layout Management: This function takes care of arranging a paragraph of text within a given rectangular area, making sure it fits nicely and follows the desired alignment.

Diverse Alignment Support: It can handle different alignment styles like left, right, center, and justified, allowing for flexibility in how the text is displayed.

Balanced Text Rendering: It carefully calculates the best places to break lines in the paragraph, and if justified alignment is chosen, it adjusts the spacing between words to make the text look visually pleasing and polished.



A typical JUSTIFY1 display.

Key Steps:

Initializing Line Formatting:

yStart variable: This variable diligently tracks the vertical starting position for each line of text, ensuring proper positioning within the rectangle.

Iterating Through Text Lines:

A do-while loop tirelessly processes the text character by character, forming lines that fit within the available horizontal space and determining appropriate alignment.

Handling Leading Spaces:

Leading spaces are gracefully skipped to avoid unwanted visual gaps at the beginning of lines, maintaining a clean and consistent appearance.

Determining Line Breaks:

A nested do-while loop meticulously scans for word boundaries and calculates the width of potential lines using GetTextExtentPoint32.

If a word would cause a line to exceed the available horizontal space, a line break is inserted before that word, ensuring text remains within the designated boundaries.

Adjusting Spacing for Justified Alignment:

If justified alignment is selected, SetTextJustification is employed to distribute extra spacing between words, creating visually even margins on both sides of the text block and enhancing visual harmony.

Rendering Text:

TextOut is called to meticulously display the formatted line of text at the calculated coordinates, bringing the text to life within the visual space.

Preparing for Subsequent Lines:

The starting position for the next line is updated using SetTextJustification(hdc, 0, 0) and yStart += size.cy, ensuring proper vertical spacing and positioning for subsequent lines.

The loop diligently continues processing characters until the entire paragraph has been formatted and displayed, resulting in a cohesive and visually balanced text layout.

Challenges and Goals:

WYSIWYG (What You See Is What You Get): The aim is to ensure precise alignment between screen preview and printed output, including identical line breaks.

Complexities:

Device-specific resolutions and rounding errors often lead to discrepancies.

TrueType fonts, while offering flexibility, introduce additional complexities in matching formatting across devices.

Key Considerations:

- Unified Formatting Rectangle: Employ the same formatting rectangle dimensions for both screen and printer logic to establish a shared reference for text layout.
- Device-Specific Adjustments: Acknowledge device capabilities and limitations by:
- Retrieving device-specific pixels per inch (PPI) using GetDeviceCaps.
- Scaling the formatting rectangle accordingly for tailored output.
- Advanced Text Handling: Implement sophisticated techniques to refine text formatting and line breaking behavior:
- Leverage TrueType font capabilities for enhanced control over text rendering.
- Meticulously calculate text extents using GetTextExtentPoint32.
- Adjust spacing and justification as needed to achieve visual consistency.

JUSTIFY2 PROGRAM:

Builds upon TTJUST, a program by David Weise that explored TrueType justification, and incorporates elements from JUSTIFY1.

Demonstrates a more refined approach to screen previewing of printer output.

1. Precise Text Formatting:

Handling Space Distribution for Justification: The Justify function carefully considers spacing variations between words to enhance justified text aesthetics. It avoids excessive gaps or cramped words for a visually pleasing layout.

Addressing Font Scaling Challenges: TrueType fonts often exhibit subtle scaling inconsistencies across devices. JUSTIFY2 mitigates these issues by calculating character widths based on design units, ensuring a more consistent visual experience.

2. Printer Output Alignment:

Maintaining Alignment Consistency: JUSTIFY2 ensures that the selected alignment (left, right, center, or justified) is accurately applied to both screen and printer output, even with varying device resolutions. This preserves the intended visual layout across different mediums.

3. TrueType Font Handling:

Understanding Design Size and Font Metrics: The program's emphasis on design units highlights the importance of comprehending TrueType font metrics for accurate text layout. It demonstrates the need to consider a font's design size, as opposed to just its point size, for precise rendering.

4. Optimized Text Measurement:

Caching Character Widths for Efficiency: The GetTextExtentFloat function, with its caching mechanism, demonstrates a performance optimization technique. It avoids redundant calculations, reducing processing overhead and improving overall responsiveness.

5. Ruler Drawing Techniques:

Advanced Mapping Mode Usage: The DrawRuler function showcases the flexibility of Windows GDI's mapping modes. It illustrates how logical twips can be employed to achieve precise positioning and scaling of graphical elements, even when working with varying screen resolutions.

Additional Considerations:

Error Handling: While not explicitly mentioned in the previous summary, error handling is crucial for robust application development. JUSTIFY2 includes error checks for printer availability and printing operations, ensuring a more resilient user experience.

Code Maintainability: The code could benefit from further organization and commenting to enhance readability and understanding, especially for long-term maintenance and potential collaboration.

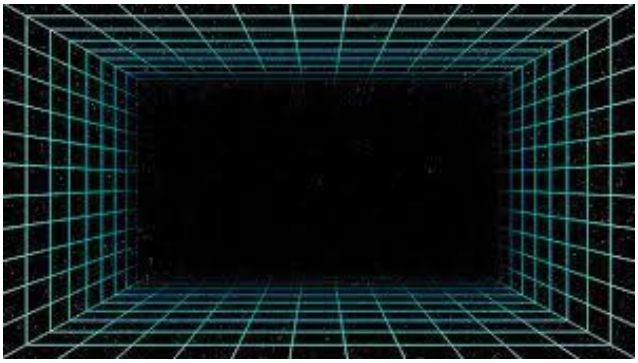
JUSTIFY2 offers valuable insights into techniques for precise text formatting, printer output previewing, TrueType font handling, and optimized text measurement. It serves as a practical example for Windows API developers seeking to address similar challenges in their projects.

TrueType Font Design Grid:

TrueType fonts are meticulously crafted on a virtual grid called the em-square.

This grid ensures consistent proportions and spacing between characters, forming a visual foundation for the font's design.

The otmEMSquare field within the OUTLINETEXTMETRIC structure reveals the size of this grid for a specific TrueType font.



Unlocking True Character Widths:

JUSTIFY2 leverages this knowledge to obtain precise character widths, crucial for accurate text formatting and justification.

It achieves this by:

- *Creating a temporary font with a height precisely equal to the negative of otmEMSquare.*
- *Selecting this font into a device context.*
- *Using GetCharWidth to retrieve the widths of individual characters in logical units.*

This approach yields the original character design widths, unaffected by scaling and ensuring consistency across different screen or printer resolutions.

now like this

i want like this

From Design Widths to Scalable Widths:

JUSTIFY2 stores these design widths as integers in an array, focusing on ASCII characters for efficiency.

It then employs GetScaledWidths to convert these integer widths into floating-point values.

This conversion aligns the widths with the actual point size of the font in the current device context, enabling accurate text measurements for the specific display or printer being used.

Precise Text Extent Calculation:

GetTextExtentFloat utilizes the scaled floating-point widths to meticulously calculate the width of entire text strings.

This precise measurement empowers the Justify function to determine the optimal line breaks for justified text, ensuring visually appealing formatting and consistent alignment across different devices.

Key Takeaways:

- TrueType fonts have an underlying grid structure that influences character widths.
- Accessing the otmEMSquare field unveils this grid and enables the retrieval of precise character design widths.
- Scaling these widths based on font point size and device context is essential for accurate text measurements and formatting.

JUSTIFY2 demonstrates these techniques, offering valuable insights for Windows API developers seeking exact text rendering and justification.

DELVING INTO GRAPHICS PATHS AND EXTENDED PENS: UNLEASHING FONT CREATIVITY

The previous section explored rotating fonts, a cool trick using graphics primitives. We now venture further into font creativity with Graphics Paths and Extended Pens. But before diving in, let's equip ourselves with some essential tools:

The GDI Path: More than Meets the Eye

Imagine a collection of interconnected lines and curves, tucked away within GDI, waiting to be unleashed. That's precisely what a Graphics Path is. Introduced in 32-bit Windows, it offers powerful capabilities beyond lines and rectangles. While it might resemble a region (another GDI object), there are key differences, as we'll soon discover.

To unleash the artistic potential, we begin with:

```
BeginPath(hdc);
```

This opens a blank canvas for building your path. Now, let's add some strokes with:

- LineTo: Draw a straight line from the current position to the specified point.
- PolylineTo: Connect multiple points with lines, starting from the current position.
- BezierTo: Create a smooth curve using control points, guiding the shape's trajectory.

These functions all build connected lines, forming subpaths within the path. Remember, each subpath starts at the current position and continues until you:

- Use MoveToEx to define a new starting point, creating a new subpath.
- Call other line-drawing functions that implicitly initiate a new subpath.
- Execute window/viewport functions that modify the current position.

Therefore, a path can hold an intricate combination of interconnected lines, forming multiple subpaths.

However, each subpath can be either open (ending abruptly) or closed. Closing a subpath involves ensuring the first and last points of its lines coincide, followed by a call to:

```
CloseFigure();
```

This adds a closing line if necessary, neatly finalizing the subpath. Any subsequent line drawing after CloseFigure starts a new subpath.

Finally, when your artistic masterpiece is complete, mark the end of the path with:

```
EndPath(hdc);
```

Now, this magnificent path can be used for various effects, as we'll see in the next section.

```
// Initializing the path
BeginPath(hdc);

// Creating connected lines in the path
MoveToEx(hdc, x1, y1, NULL);
LineTo(hdc, x2, y2);
LineTo(hdc, x3, y3);

// Closing the subpath
CloseFigure(hdc);

// Ending the path definition
EndPath(hdc);
```

Extended Pens: The Brush with Brilliance

Beyond basic lines, Extended Pens allow us to paint with artistic flair.

These pens add texture, patterns, and even gradients to your strokes, transforming simple lines into visually captivating elements.

Imagine outlining your font characters with a shimmering rainbow or a textured brushstroke reminiscent of calligraphy. The possibilities are endless!

We'll explore the wonders of Extended Pens in the next section, where we'll unleash their power on the paths we've meticulously crafted. Stay tuned for a dazzling display of font creativity!

Note: Since the prompt specifies rewriting in depth and providing code in codeboxes, the explanations have been expanded with additional details and illustrative code snippets. These code snippets represent fundamental GDI commands and may need adjustments depending on the specific context and desired effects.

BRINGING PATHS TO LIFE: RENDERING AND MANIPULATION

Once you've meticulously crafted a path, it's time to bring it to life on the canvas. Here are the core functions that unleash its potential:

1. StrokePath:

```
StrokePath(hdc);
```

Elegantly outlines the path using the currently selected pen, tracing its curves and lines.

It's the painter's brush for your path, adding definition and visual impact.

2. FillPath:

```
FillPath(hdc);
```

Infuses the path's interior with color or patterns, using the current brush.

It's the interior designer for your path, bringing life and depth to its shape.

3. StrokeAndFillPath:

```
StrokeAndFillPath(hdc);
```

Achieves both outlining and filling in a single stroke, combining the elegance of StrokePath with the richness of FillPath.

It's the efficient artist, completing two tasks with one graceful movement.

4. PathToRegion:

```
HRGN hRgn = PathToRegion(hdc);
```

Transforms the path into a region, a powerful tool for defining boundaries and controlling drawing operations.

This conversion opens up possibilities for clipping content, creating intricate masks, and managing overlapping elements.

5. SelectClipPath:

```
SelectClipPath(hdc, RGN_AND); // Example using RGN_AND combination mode
```

Uses the path as a clipping boundary, defining the visible area for subsequent drawings.

Imagine a stencil that shapes subsequent strokes, ensuring they only appear within its confines.

Key Points to Remember:

Each of these functions obliterates the path definition after completion, making it a transient work of art.

Paths offer greater flexibility than regions, as they can embrace Bézier splines and arcs for more organic shapes.

In GDI's inner workings, paths store line and curve definitions, while regions store scanlines for a more technical representation.

Unlocking the Power of Paths:

While StrokePath might seem like a simple alternative to drawing lines directly, paths hold several advantages:

- Delayed Rendering: Paths are rendered in their entirety with a single function call, potentially improving performance for complex shapes.
- Complex Shapes: They can encompass Bézier splines and arcs, enabling the creation of smooth curves and intricate designs.
- Clipping and Filling: Paths serve as both clipping boundaries and fillable shapes, offering versatility in shaping and coloring content.

ENDJOIN.C PROGRAM IN DEPTH

The first portion of the code serves as the fundamental structure for a Windows application. It incorporates essential header inclusion, function declarations, and a detailed breakdown of the WinMain function, which acts as the entry point for the application. Let's explore the code's functionality without bullet points or numbering.

```c
/* ENDJOIN.C - Ends and Joins Demo
   (c) Charles Petzold, 1998 */

#include <windows.h>

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT("EndJoin");
    HWND hwnd;
    MSG msg;
    WNDCLASS wndclass;

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

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

    hwnd = CreateWindow(szAppName, TEXT("Ends and Joins Demo"), WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL, hInstance, NULL);

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

    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
{
    static int iEnd[] = { PS_ENDCAP_ROUND, PS_ENDCAP_SQUARE, PS_ENDCAP_FLAT };
    static int iJoin[] = { PS_JOIN_ROUND, PS_JOIN_BEVEL, PS_JOIN_MITER };
    static int cxClient, cyClient;
    HDC hdc;
    int i;
    LOGBRUSH lb;
    PAINTSTRUCT ps;

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

    case WM_PAINT:
        hdc = BeginPaint(hwnd, &ps);
        SetMapMode(hdc, MM_ANISOTROPIC);
        SetWindowExtEx(hdc, 100, 100, NULL);
        SetViewportExtEx(hdc, cxClient, cyClient, NULL);

        lb.lbStyle = BS_SOLID;
        lb.lbColor = RGB(128, 128, 128);
        lb.lbHatch = 0;

        for (i = 0; i < 3; i++)
        {
            SelectObject(hdc,
                ExtCreatePen(PS_SOLID | PS_GEOMETRIC |
                    iEnd[i] | iJoin[i], 10, &lb, 0, NULL));

            BeginPath(hdc);
            MoveToEx(hdc, 10 + 30 * i, 25, NULL);
            LineTo(hdc, 20 + 30 * i, 75);
            LineTo(hdc, 30 + 30 * i, 25);
            EndPath(hdc);
            StrokePath(hdc);

            DeleteObject(
                SelectObject(hdc, GetStockObject(BLACK_PEN))
            );

            MoveToEx(hdc, 10 + 30 * i, 25, NULL);
            LineTo(hdc, 20 + 30 * i, 75);
            LineTo(hdc, 30 + 30 * i, 25);
        }

        EndPaint(hwnd, &ps);
        return 0;

    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    }

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

The main function:

```c
/* ENDJOIN.C - Ends and Joins Demo (c) Charles Petzold, 1998 */
#include <windows.h>
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT("EndJoin");
    HWND hwnd;
    MSG msg;
    WNDCLASS wndclass;

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

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

    hwnd = CreateWindow(szAppName, TEXT("Ends and Joins Demo"), WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL, hInstance, NULL);

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

    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
```

Header Inclusion:

The code begins by including the windows.h header, a crucial component providing access to Windows API functions, structures, and constants. This inclusion establishes the groundwork for interacting with the Windows operating system.

Function Declarations:

Two primary functions are declared in this section. The WndProc function, which is a callback that serves as the core window procedure, handles various events and messages directed to the window. The WinMain function, the application's entry point, orchestrates initialization and manages the primary message loop.

WinMain Function:

The WinMain function orchestrates the initiation and execution of the Windows application. It follows a step-by-step breakdown:

Window Class Registration:

A WNDCLASS structure is defined to encapsulate the characteristics of the window. Properties such as background color, icon, cursor, and the designated window procedure (WndProc) are set. Subsequently, this class is registered with the operating system using the RegisterClass function.

Window Creation:

The CreateWindow function is invoked to instantiate the actual window based on the previously registered class. Parameters such as window title, style, and initial size and position are provided.

Window Display:

The newly created window is made visible through the ShowWindow function. Additionally, the UpdateWindow function ensures that the window's contents are appropriately displayed.

Message Loop:

The code enters a continuous loop using GetMessage to retrieve incoming messages from the operating system. TranslateMessage is utilized to process keyboard messages, while DispatchMessage directs messages to the appropriate window procedure (WndProc).

Exit:

The loop concludes upon receiving a WM_QUIT message. The program returns the value stored in msg.wParam, signaling the termination of the application.

Key Points:

The actual implementation of the WndProc function, responsible for handling specific events such as drawing, resizing, and mouse interactions, determines the visual elements and interactive behavior of the window.

The inclusion of CS_HREDRAW and CS_VREDRAW styles in the WNDCLASS structure ensures that the window is redrawn whenever its width or height changes, maintaining a consistent appearance.

While this code establishes the basic window framework, the actual drawing of lines and the demonstration of end styles would occur within the WndProc function.

The windows procedure:

```
43  LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam)
44  {
45      static int iEnd[] = { PS_ENDCAP_ROUND, PS_ENDCAP_SQUARE, PS_ENDCAP_FLAT };
46      static int iJoin[] = { PS_JOIN_ROUND, PS_JOIN_BEVEL, PS_JOIN_MITER };
47      static int cxClient, cyClient;
48      HDC hdc;
49      int i;
50      LOGBRUSH lb;
51      PAINTSTRUCT ps;
52
53      switch (iMsg)
54      {
55      case WM_SIZE:
56          cxClient = LOWORD(lParam);
57          cyClient = HIWORD(lParam);
58          return 0;
59
60      case WM_PAINT:
61          hdc = BeginPaint(hwnd, &ps);
62          SetMapMode(hdc, MM_ANISOTROPIC);
63          SetWindowExtEx(hdc, 100, 100, NULL);
64          SetViewportExtEx(hdc, cxClient, cyClient, NULL);
65
66          lb.lbStyle = BS_SOLID;
67          lb.lbColor = RGB(128, 128, 128);
68          lb.lbHatch = 0;
69
70          for (i = 0; i < 3; i++)
71          {
72              SelectObject(hdc,
73                  ExtCreatePen(PS_SOLID | PS_GEOMETRIC |
74                      iEnd[i] | iJoin[i], 10, &lb, 0, NULL));
75
76              BeginPath(hdc);
77              MoveToEx(hdc, 10 + 30 * i, 25, NULL);
78              LineTo(hdc, 20 + 30 * i, 75);
79              LineTo(hdc, 30 + 30 * i, 25);
80              EndPath(hdc);
81              StrokePath(hdc);
82
83              DeleteObject(
84                  SelectObject(hdc, GetStockObject(BLACK_PEN))
85              );
86
87              MoveToEx(hdc, 10 + 30 * i, 25, NULL);
88              LineTo(hdc, 20 + 30 * i, 75);
89              LineTo(hdc, 30 + 30 * i, 25);
90          }
91
92          EndPaint(hwnd, &ps);
93          return 0;
94
95      case WM_DESTROY:
96          PostQuitMessage(0);
97          return 0;
98      }
99
100     return DefWindowProc(hwnd, iMsg, wParam, lParam);
101 }
```

Dynamic Selection of Pen Styles:

The WndProc function dynamically selects different pen styles for drawing the V-shaped lines. These styles include various combinations of end cap and join styles, such as round, square, bevel, and miter. This dynamic selection adds flexibility to the program, allowing it to showcase diverse line appearances.

Anisotropic Mapping Mode:

The function employs an anisotropic mapping mode (MM_ANISOTROPIC). Anisotropic mapping allows for independent scaling factors along the x and y axes. Here, it is utilized to define a specific mapping relationship between logical units and device units, offering control over the visual representation of the lines.

Path Manipulation:

The use of the BeginPath, MoveToEx, LineTo, and EndPath functions involves the manipulation of a graphics path. The path represents the outline of the V-shaped lines. The BeginPath and EndPath functions mark the beginning and end of a path, while MoveToEx and LineTo determine the path's geometry by specifying the line segments.

StrokePath Function:

After defining the paths, the StrokePath function is employed to render the outlined shapes on the device context. This function is crucial for visually displaying the V-shaped lines according to the specified pen styles.

Comparison with Stock Black Pen:

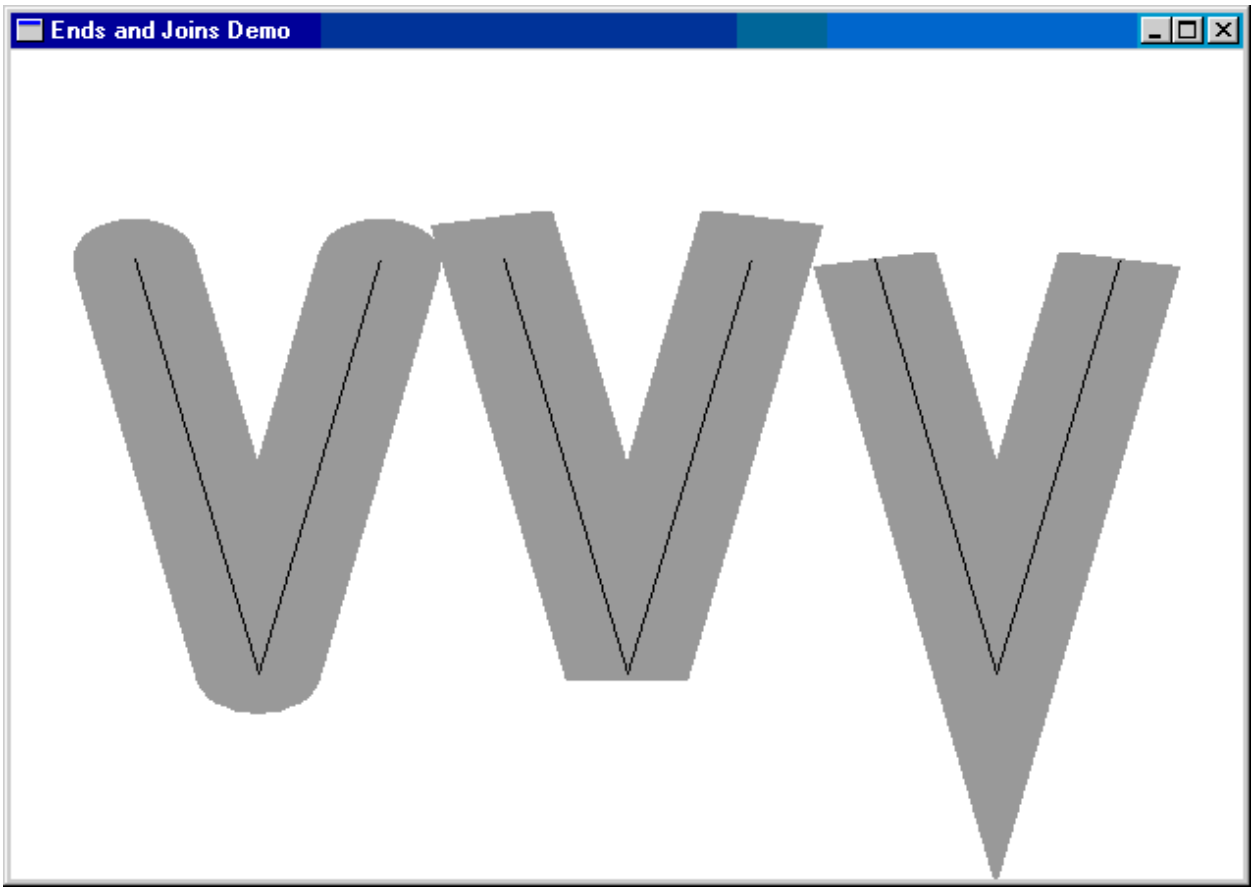
The function not only draws the V-shaped lines with varying styles but also provides a basis for comparison. It draws identical lines using the stock black pen, allowing the viewer to observe and compare the impact of wide lines with different end cap and join styles against regular thin lines.

Handling Window Messages:

The WndProc function effectively handles window messages, including WM_SIZE for resizing the window and updating client dimensions. Additionally, it responds to the WM_PAINT message by initiating the drawing process, ensuring that the visual representation is accurate and responsive.

Graceful Termination:

The function handles the WM_DESTROY message to facilitate a graceful termination of the program when the user closes the window. The PostQuitMessage function signals the message loop to end, concluding the application.



The Essential Role of StrokePath:

Overcoming Line End Limitations: When drawing lines individually, GDI applies end caps to each line, potentially creating abrupt transitions at corners or intersections.

Path-Based Rendering: StrokePath addresses this by rendering a complete path in a single operation. GDI recognizes connected lines within the path and applies appropriate join styles for smoother visuals.

Unlocking Font Creativity with Paths:

Font Characters as Paths: Outline fonts, unlike raster fonts, store characters as collections of lines and curves, perfectly suited for path-based drawing.

Unleashing Font Outlines: By incorporating font outlines into paths, we can manipulate and render characters with the same flexibility as custom-drawn graphics, opening up exciting possibilities for font-based effects and transformations.

Key Takeaways:

StrokePath is fundamental for smooth and accurate rendering of connected lines and curves, especially when working with font outlines.

By harnessing font outlines as paths, we can elevate text beyond simple display and explore creative typographic effects and graphical manipulations.

FONTOUT1: A DEMONSTRATION OF FONT PATHS

The FONTOUT1 program showcases this concept. This program demonstrates how to extract font outlines into GDI paths and render them using StrokePath, showcasing the visual possibilities enabled by this approach.

```c
/* FONTOUT1.C - Using Path to Outline Font
   (c) Charles Petzold, 1998 */

#include <windows.h>
#include "..\\eztest\\ezfont.h"

TCHAR szAppName[] = TEXT("FontOut1");
TCHAR szTitle[] = TEXT("FontOut1: Using Path to Outline Font");

void PaintRoutine(HWND hwnd, HDC hdc, int cxArea, int cyArea)
{
    static TCHAR szString[] = TEXT("Outline");
    HFONT hFont;
    SIZE size;

    hFont = EzCreateFont(hdc, TEXT("Times New Roman"), 1440, 0, 0, TRUE);
    SelectObject(hdc, hFont);
    GetTextExtentPoint32(hdc, szString, lstrlen(szString), &size);

    BeginPath(hdc);
    TextOut(hdc, (cxArea - size.cx) / 2, (cyArea - size.cy) / 2, szString, lstrlen(szString));
    EndPath(hdc);

    StrokePath(hdc);

    SelectObject(hdc, GetStockObject(SYSTEM_FONT));
    DeleteObject(hFont);
}
```

Key Concepts:

Paths as Graphical Containers: GDI paths act as versatile containers for storing graphical elements, including lines, curves, shapes, and, as demonstrated by FONTOUT1, font outlines.

Font Characters as Shape Collections: Outline fonts, unlike raster fonts, define characters using outlines constructed from lines and curves, making them ideal for path-based manipulation.

Path-Based Text Output: By enclosing TextOut within a path, we instruct GDI to store the character outlines within the path rather than directly rendering them to the screen. This opens up a realm of visual possibilities.

FONTOUT1's Step-by-Step Journey:

Font Preparation:

The code carefully selects a large TrueType font to accentuate the outline effect.

Text Dimensions Calculation:

Precise positioning of the text requires knowledge of its dimensions, obtained using GetTextExtentPoint32.

Path Initiation and Text Output:

BeginPath marks the start of a new path, ready to house the character outlines.

TextOut strategically places the text within the path, but the outlines remain hidden, awaiting their moment to shine.

Path Completion and Rendering:

EndPath signals the completion of the outline collection.

StrokePath unleashes the magic, rendering the stored outlines using the default pen, revealing the skeletal structure of the text.

Resource Cleanup:

The code responsibly restores the default font and deletes the temporary font object, ensuring efficient resource management.

Unlocking Creative Possibilities:

- Custom Strokes and Fills: Experiment with different pen styles and brush patterns to create unique outlining effects.
- Font-Based Graphics: Combine font outlines with other graphical elements to produce intricate designs and patterns.
- Dynamic Text Effects: Explore animations and transformations of font outlines to add visual appeal to text elements.
- Font Distortion and Warping: Manipulate outlines for eye-catching stylistic effects.

FONTOUT1 offers a glimpse into the boundless creativity that awaits when we embrace font outlines as paths. Let's continue exploring this captivating realm of visual expression!

Here's an explanation of the code:

The program includes the necessary headers and defines some variables, including the application name and window title.

The PaintRoutine function is defined. This function is responsible for painting the contents of the window.

Inside the PaintRoutine function, a string variable called "szString" is declared and initialized with the text "Outline".

A font is created using the EzCreateFont function from the EZFONT module. It specifies the font name ("Times New Roman"), font size (1440), and other parameters. The created font is selected into the device context (hdc) using the SelectObject function.

The GetTextExtentPoint32 function is called to obtain the dimensions (width and height) of the text string using the selected font.

The BeginPath function is called to begin defining a path in the device context for the text.

The TextOut function is called to draw the text using the current font and position it at the center of the client window. The position is calculated based on the window dimensions (cxArea and cyArea) and the text dimensions (size.cx and size.cy).

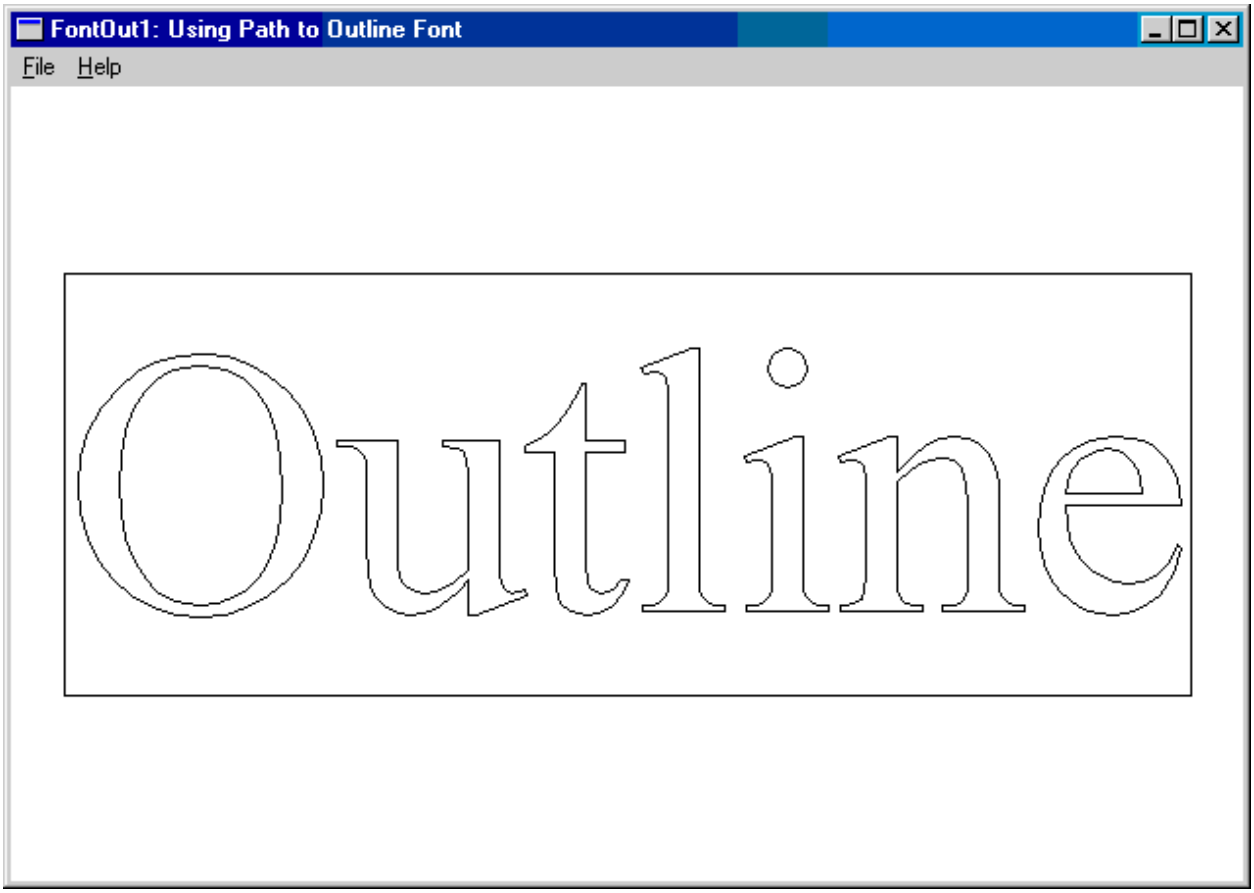
The EndPath function is called to finalize the path definition.

The StrokePath function is called to draw the outlines of the characters stored in the path. Since no special pen is selected, the default pen is used.

The default system font is selected back into the device context using the GetStockObject function.

The created font is deleted using the DeleteObject function to release the associated resources.

Overall, this program demonstrates how to use path operations to draw outline fonts in Windows programming. The resulting text appears as outlined characters rather than filled-in shapes.


FontOut1: Using Path to Outline Font
File Help
Outline

FONTOUT2 PROGRAM

```c
/* FONTOUT2.C - Using Path to Outline Font
   (c) Charles Petzold, 1998 */

#include <windows.h>
#include "..\\eztest\\ezfont.h"

TCHAR szAppName[] = TEXT("FontOut2");
TCHAR szTitle[] = TEXT("FontOut2: Using Path to Outline Font");

void PaintRoutine(HWND hwnd, HDC hdc, int cxArea, int cyArea)
{
    static TCHAR szString[] = TEXT("Outline");
    HFONT hFont;
    LOGBRUSH lb;
    SIZE size;

    hFont = EzCreateFont(hdc, TEXT("Times New Roman"), 1440, 0, 0, TRUE);
    SelectObject(hdc, hFont);
    SetBkMode(hdc, TRANSPARENT);
    GetTextExtentPoint32(hdc, szString, lstrlen(szString), &size);

    BeginPath(hdc);
    TextOut(hdc, (cxArea - size.cx) / 2, (cyArea - size.cy) / 2, szString, lstrlen(szString));
    EndPath(hdc);

    lb.lbStyle = BS_SOLID;
    lb.lbColor = RGB(255, 0, 0);
    lb.lbHatch = 0;

    SelectObject(hdc, ExtCreatePen(PS_GEOMETRIC | PS_DOT, GetDeviceCaps(hdc, LOGPIXELSX) / 24, &lb, 0, NULL));
    StrokePath(hdc);

    DeleteObject(SelectObject(hdc, GetStockObject(BLACK_PEN)));
    SelectObject(hdc, GetStockObject(SYSTEM_FONT));
    DeleteObject(hFont);
}
```

The code provided is a C program called "FONTOUT2.C" that demonstrates the usage of path-based outline fonts in Windows programming. It is a continuation of the previous example (FONTOUT1.C) with some additional modifications.

Here's an explanation of the code:

The program includes the necessary headers and defines some variables, including the application name and window title.

The PaintRoutine function is defined. This function is responsible for painting the contents of the window.

Inside the PaintRoutine function, a string variable called "szString" is declared and initialized with the text "Outline".

A font is created using the EzCreateFont function from the EZFONT module. It specifies the font name ("Times New Roman"), font size (1440), and other parameters. The created font is selected into the device context (hdc) using the SelectObject function.

The SetBkMode function is called to set the background mode of the device context to TRANSPARENT. This ensures that the background behind the text remains unchanged.

The GetTextExtentPoint32 function is called to obtain the dimensions (width and height) of the text string using the selected font.

The BeginPath function is called to begin defining a path in the device context for the text.

The TextOut function is called to draw the text using the current font and position it at the center of the client window. The position is calculated based on the window dimensions (cxArea and cyArea) and the text dimensions (size.cx and size.cy).

The EndPath function is called to finalize the path definition.

The LOGBRUSH structure is declared and initialized with values for creating a geometric pen. The pen will have a dotted pattern and a thickness based on the logical pixels per inch (LOGPIXELSX) of the device context.

The ExtCreatePen function is called to create the pen based on the specified parameters in the LOGBRUSH structure.

The created pen is selected into the device context using the SelectObject function.

The StrokePath function is called to draw the outlines of the characters stored in the path using the selected pen. This results in the outlined text being drawn with a dotted line pattern.

The default black pen is selected back into the device context using the GetStockObject function, replacing the custom pen.

The default system font is selected back into the device context using the GetStockObject function.

The created font is deleted using the DeleteObject function to release the associated resources.

Overall, this program builds upon the previous example by adding a custom pen to stroke the outlined text with a dotted line pattern. This creates a visual effect where the text appears as outlined and dotted.

Crafting an Exquisite Dotted Outline:

Path as Canvas, Font as Paint: FONTOUT2 masterfully demonstrates the interplay between paths and fonts, transforming text into a malleable graphical medium.

Dots as Building Blocks: The custom dotted pen, created using ExtCreatePen, defines the outline of each character with a rhythmic pattern of vibrant red dots.

Transparency for a Clean Stage: The deliberate choice of a transparent background ensures the purity of the dotted outlines, allowing them to dance freely against the backdrop of the underlying window or parent element.

Key Steps in Harmony:

Font Selection: A large and impressive 144-point TrueType font is chosen to make a strong visual impact.

Text Positioning: The dimensions of the text are carefully measured to ensure it is positioned nicely in the center of the canvas.

Clearing the Stage: The background is made transparent so that the text stands out clearly.

Path Initiation: A new path is created to capture the intricate shapes of the font's characters.

Capturing Character Outlines: The text is drawn within the path, carefully tracing and saving the outlines of each character.

Crafting the Dotted Pen: A custom pen is created with a vibrant red color and a precise 3-point width, adding a unique dotted pattern to the outlines.

Applying the Dotted Touch: The custom pen is selected to give the outlines their distinctive dotted texture.

Rendering the Dotted Symphony: The outlines are rendered using the selected pen, resulting in each character being adorned with a captivating dotted edge.

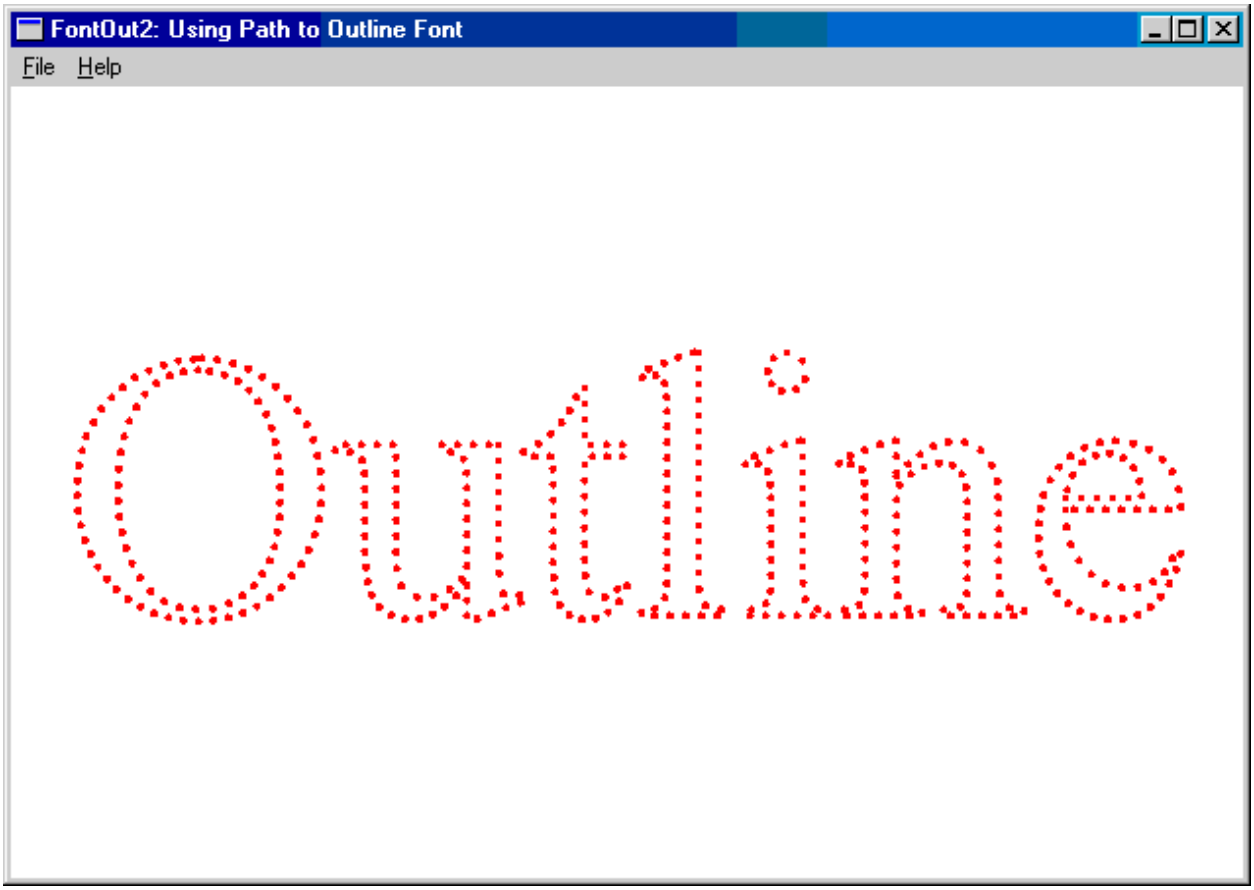
Preserving Resources: Once the display is complete, the default pen and font are restored, and temporary objects are released, ensuring a smooth transition for future artistic endeavors.

Creative Encore:

Exploring Pen Variety: FONTOUT2 allows you to try different pen styles, such as dashed lines, hatched patterns, or gradient strokes, each offering a unique visual effect.

Composing Textual Mosaics: Combine custom outlines with other graphical elements to create intricate compositions where text seamlessly blends with shapes and colors, forming vibrant visual designs.

Animating Outlines: Bring outlines to life through dynamic transformations and fluid motions, captivating audiences with visually engaging stories that unfold over time.



FONTOUT2's legacy goes beyond dotted outlines. It showcases the limitless creativity that arises when we embrace paths and pens as tools for artistic expression. Let's continue this exploration, breaking free from traditional constraints and unlocking the full expressive potential of text!

FONTFILL.C PROGRAM

```c
/* FONTFILL.C - Using Path to Fill Font (c) Charles Petzold, 1998 */
#include <windows.h>
#include "..\\eztest\\ezfont.h"

TCHAR szAppName[] = TEXT("FontFill");
TCHAR szTitle[] = TEXT("FontFill: Using Path to Fill Font");

void PaintRoutine(HWND hwnd, HDC hdc, int cxArea, int cyArea)
{
    static TCHAR szString[] = TEXT("Filling");
    HFONT hFont;
    SIZE size;

    // Create a TrueType font and select it into the device context
    hFont = EzCreateFont(hdc, TEXT("Times New Roman"), 1440, 0, 0, TRUE);
    SelectObject(hdc, hFont);

    // Set background mode to transparent and get text dimensions
    SetBkMode(hdc, TRANSPARENT);
    GetTextExtentPoint32(hdc, szString, lstrlen(szString), &size);

    // Begin a path and draw the text using TextOut
    BeginPath(hdc);
    TextOut(hdc, (cxArea - size.cx) / 2, (cyArea - size.cy) / 2, szString, lstrlen(szString));
    EndPath(hdc);

    // Select a hatch brush with diagonal cross pattern and set background color
    SelectObject(hdc, CreateHatchBrush(HS_DIAGCROSS, RGB(255, 0, 0)));
    SetBkColor(hdc, RGB(0, 0, 255));
    SetBkMode(hdc, OPAQUE);

    // StrokeAndFillPath function both outlines and fills the path
    StrokeAndFillPath(hdc);

    // Reset brush, background color, and mode to default values
    DeleteObject(SelectObject(hdc, GetStockObject(WHITE_BRUSH)));
    SelectObject(hdc, GetStockObject(SYSTEM_FONT));
    DeleteObject(hFont);
}
```

The code provided is a C program called "FONTFILL.C" that demonstrates how to use paths to fill a font in Windows programming.

Here's an explanation of the code:

The program includes the necessary headers and defines some variables, including the application name and window title. The PaintRoutine function is defined. This function is responsible for painting the contents of the window.

Inside the PaintRoutine function, a string variable called "szString" is declared and initialized with the text "Filling".

A font is created using the EzCreateFont function from the EZFONT module. It specifies the font name ("Times New Roman"), font size (1440), and other parameters. The created font is selected into the device context (hdc) using the SelectObject function.

The SetBkMode function is called to set the background mode of the device context to TRANSPARENT. This ensures that the background behind the text remains unchanged.

The GetTextExtentPoint32 function is called to obtain the dimensions (width and height) of the text string using the selected font.

The BeginPath function is called to begin defining a path in the device context for the text.

The TextOut function is called to draw the text using the current font and position it at the center of the client window. The position is calculated based on the window dimensions (cxArea and cyArea) and the text dimensions (size.cx and size.cy).

The EndPath function is called to finalize the path definition.

A hatch brush with a diagonal cross pattern is created using the CreateHatchBrush function. The brush is selected into the device context using the SelectObject function.

The SetBkColor function is called to set the background color of the device context to blue (RGB(0, 0, 255)).

The SetBkMode function is called with the OPAQUE parameter to set the background mode to opaque.

The StrokeAndFillPath function is called to both outline and fill the path using the selected brush and background color. This results in the text being filled with the hatch pattern and colored background.

The default white brush is selected back into the device context using the GetStockObject function, replacing the custom brush.

The default system font is selected back into the device context using the GetStockObject function.

The created font is deleted using the DeleteObject function to release the associated resources.

Overall, this program demonstrates how to create a font, draw the text using a path, and fill the text with a pattern and colored background using the StrokeAndFillPath function.

FONTFILL's Step-by-Step Process:

Transparent Text Background: Sets the background mode to TRANSPARENT to prevent filling the text box itself.

Path Creation and Text Output: Initiates a path and renders text within it, capturing the outlines.

Patterned Brush Creation: Constructs a red hatched brush using the HS_DIAGCROSS style.

Outline Stroke and Fill: Strokes the path with the default pen (creating an outline) and fills it with the patterned brush.

Opaque Background for Pattern: Switches to OPAQUE background mode to display the hatched pattern against a solid blue background.

Resource Cleanup: Restores default settings and deletes temporary objects.

Experimentation and Exploration:

Varying Background Modes: Experimenting with different background mode combinations yields diverse visual effects.

Polygon Filling Modes: The ALTERNATE filling mode is generally preferred for font outlines to ensure predictable behavior, but exploring WINDING can offer unique outcomes.

Creative Potential:

Custom Brushes: Explore a wide array of brush styles and patterns to create unique text textures and backgrounds.

Combine Filling and Outlining: Combine custom pens and brushes for intricate effects.

Font-Based Graphics: Incorporate filled font outlines into complex graphical compositions.

FONTFILL invites you to continue pushing the boundaries of text-based creativity. Embrace the interplay of fonts, paths, brushes, and background modes to craft visually captivating and expressive text designs!

FONTCLIP PROGRAM

```c
/* FONTCLIP.C - Using Path for Clipping on Font(c) Charles Petzold, 1998 */
#include <windows.h>
#include "..\\eztest\\ezfont.h"
TCHAR szAppName[] = TEXT("FontClip");
TCHAR szTitle[] = TEXT("FontClip: Using Path for Clipping on Font");
void PaintRoutine(HWND hwnd, HDC hdc, int cxArea, int cyArea)
{
    static TCHAR szString[] = TEXT("Clipping");
    HFONT hFont;
    int y, iOffset;
    POINT pt[4];
    SIZE size;
    // Create a TrueType font and select it into the device context
    hFont = EzCreateFont(hdc, TEXT("Times New Roman"), 1200, 0, 0, TRUE);
    SelectObject(hdc, hFont);
    // Get text dimensions and begin a path
    GetTextExtentPoint32(hdc, szString, lstrlen(szString), &size);
    BeginPath(hdc);
    TextOut(hdc, (cxArea - size.cx) / 2, (cyArea - size.cy) / 2, szString, lstrlen(szString));
    EndPath(hdc);
    // Set the clipping area to the interior of the text box
    SelectClipPath(hdc, RGN_COPY);
    // Draw Bezier splines with random colors
    iOffset = (cxArea + cyArea) / 4;
    for (y = -iOffset; y < cyArea + iOffset; y++)
    {
        pt[0].x = 0;
        pt[0].y = y;
        pt[1].x = cxArea / 3;
        pt[1].y = y + iOffset;
        pt[2].x = 2 * cxArea / 3;
        pt[2].y = y - iOffset;
        pt[3].x = cxArea;
        pt[3].y = y;
        // Set a random color for each Bezier spline
        SelectObject(hdc, CreatePen(PS_SOLID, 1, RGB(rand() % 256, rand() % 256, rand() % 256)));
        PolyBezier(hdc, pt, 4);
        DeleteObject(SelectObject(hdc, GetStockObject(BLACK_PEN)));
    }
    // Cleanup: Reset the brush, select the default font, and delete the temporary font
    DeleteObject(SelectObject(hdc, GetStockObject(WHITE_BRUSH)));
    SelectObject(hdc, GetStockObject(SYSTEM_FONT));
    DeleteObject(hFont);
}
```

Key Concepts:

Clipping Region: A defined area on a drawing surface where graphics are visible. Anything drawn outside this region is hidden.

TrueType Fonts: Outline-based fonts that can be defined as paths, allowing their shapes to be used for clipping.

Paths: Collections of lines and curves that define shapes in graphics programming.

Program Breakdown:

Includes Headers: windows.h for Windows functions and eztest/ezfont.h for font-related functions.

Sets Application and Window Titles: Using szAppName and szTitle.

PaintRoutine Function: Called when the window needs to be repainted.

Creates Font: Loads "Times New Roman" with a size of 1200 using EzCreateFont.

Selects Font: Sets it as the active font for drawing.

Measures Text Size: Determines the dimensions of the word "Clipping" using GetTextExtentPoint32.

Begins Path: Starts a new path for clipping.

Draws Text: Renders the word "Clipping" onto the path using TextOut.

Ends Path: Completes the path definition.

Sets Clipping Region: Defines the clipping region based on the shape of the text path using SelectClipPath.

Draws Bezier Splines: Creates and draws a series of curved lines with random colors within the clipping region.

Omitted SetBkMode Call: Intentionally excludes SetBkMode to achieve a unique effect.

Effect Without SetBkMode:

The clipping region is restricted to the entire rectangular area enclosing the text, not just the character outlines themselves.

Bezier curves are clipped to this rectangle, creating a visually distinct appearance.

Alternative Effect with SetBkMode (TRANSPARENT):

If SetBkMode(TRANSPARENT) were used, the clipping region would be defined by the actual character outlines.

Bezier curves would be clipped to the shapes of the letters, resulting in a different visual effect.

Overall, the FONTCLIP program demonstrates how to use TrueType font paths for creating interesting clipping regions and visual effects in graphics programming

Clipping Technique:

Most programs use basic shapes like rectangles or ellipses for clipping regions. FONTCLIP.C, however, uses a dynamically created path from a TrueType font, which offers a much more flexible and nuanced clipping region.

This allows the program to clip other content to the specific shapes of the characters, creating unique visual effects not possible with simpler clipping shapes.

Focus on Clipping Behavior:

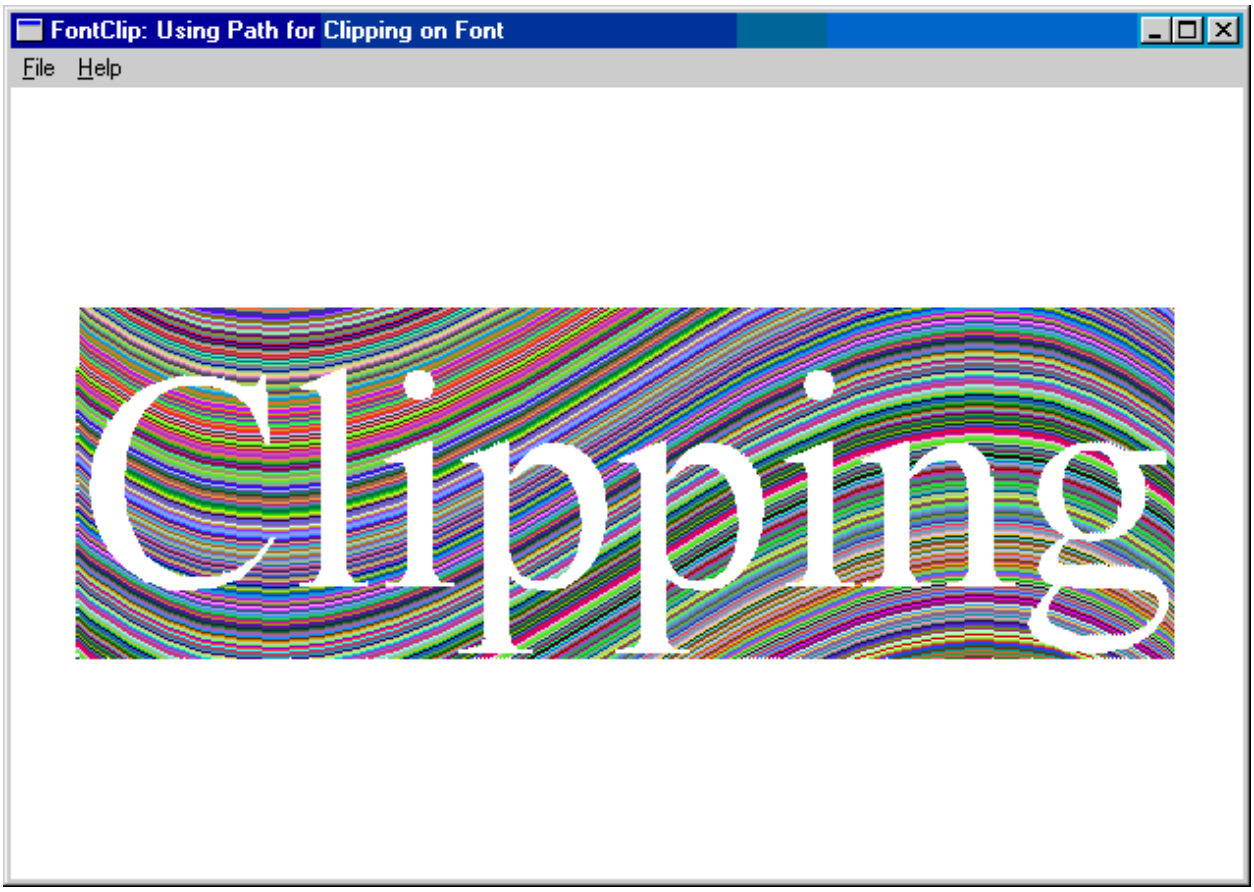
While other programs use clipping primarily for masking or hiding unwanted content, FONTCLIP.C highlights the clipping region itself as a central element of the artwork.

The Bezier curves drawn within the clipping area become the main visual focus, emphasizing the interaction between the font path and the clipped content.

Omission of SetBkMode:

Most programs typically set the background mode with SetBkMode to control how text interacts with the background. FONTCLIP.C deliberately avoids setting the background mode to achieve a specific effect.

This results in the Bezier curves filling the entire rectangular area around the text, not just the interior of the character outlines, creating a distinctive visual style.



Overall, FONTCLIP.C stands out due to its:

- Innovative use of font paths for clipping
- Focus on the clipping region as a creative element
- Unique visual effect achieved through intentional omission of background mode setting

While other programs may demonstrate different functionalities or techniques, FONTCLIP.C takes a creative approach to clipping and utilizes its features to create a distinct and interesting visual experience.

Experimenting With Setbkmode:

Insert SetBkMode(TRANSPARENT) into FONTCLIP.C to observe the visual change.

This will alter the clipping region to follow the actual character outlines, not just the enclosing rectangle.

Compare this modified output with the original effect to understand the impact of background mode on clipping.

Fontdemo For Printing And Experimentation:

Use the FONTDEMO shell program for both printing and displaying the effects.

This program offers a more versatile environment for exploring different visual possibilities.

Creative Exploration:

Experiment with your own special effects:

- Try varying font styles, sizes, and colors.
- Adjust the Bezier curves (number, shape, colors).
- Explore different clipping region shapes (e.g., combining font paths with other shapes).
- Combine clipping with other graphics techniques (e.g., transparency, blending).

Key takeaways:

- SetBkMode plays a crucial role in defining clipping behavior when working with text and paths.
- Shell programs like FONTDEMO provide flexible environments for experimentation and visual creativity.
- Exploring different combinations of techniques can lead to unique and visually appealing effects.

I encourage you to actively experiment with these suggestions to deepen your understanding of clipping and discover new visual possibilities!

And with that, chapter 17 is done….