ð¤aí▒ß>■_
╓╥■___├─┼ãÃ╚____Ñ┴Y ┐╬Ðbjbj«W«W !
Bæ=æ=>═Å_______]©©©©8_L<╝©_÷¼ö@p▒▒▒▒▒▒▒▒│││││$G

_Í6Î▒▒▒▒▒▒Îd▒▒¼ddd▒t¢▒▒▒$Jn

J▒▒d4dÿ r┼
ì▒▒°┤N┼■¢©©$┤@QúChapter 17. TEXT AND FONTS

Displaying text was one of the first jobs we tackled in this book. Now itÆs time to explore the use of different fonts and font sizes available in Windows and to learn how to justify text.

The introduction of TrueType in Windows 3.1 greatly enhanced the ability of programmers and users to work with text in a flexible manner. TrueType is an outline font technology that was developed by Apple Computer, Inc., and Microsoft Corporation and which is supported by many font manufacturers. Because TrueType fonts are continuously scalable and can be used on both video displays and printers, true WYSIWYG (what-you-see-is-what-you-get) is now possible under Windows. TrueType also lends itself well to doing ôfancyö font manipulation, such as rotating characters, filling the interiors with patterns, or using them for clipping regions, all of which will be demonstrated in this chapter.

Simple Text Output

Let's begin by looking at the different functions Windows provides for text output, the device context attributes that affect text, and the use of stock fonts.

The Text Drawing Functions

The most common text output function is the one IÆve used in very many sample programs so far:

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

The xStart and yStart arguments are the starting position of the string in logical coordinates. Normally, this is the point at which Windows begins drawing the upper left corner of the first character. TextOut requires a pointer to the character string and the length of the string. The function does not recognize NULL-terminated character strings.

The meaning of the xStart and yStart arguments to TextOut can be altered by the SetTextAlign function. The TA_LEFT, TA_RIGHT, and TA_CENTER flags affect how xStart is used to position the string horizontally. The default is TA_LEFT. If you specify TA_RIGHT in the SetTextAlign function, subsequent TextOut calls position the right side of the last character in the string at xStart. For TA_CENTER, the center of the string is positioned at xStart.

Similarly, the TA_TOP, TA_BOTTOM, and TA_BASELINE flags affect the vertical positioning. TA_TOP is the default, which means that the string is positioned so that yStart specifies the top of the characters in the string. Using TA_BOTTOM means that the string is positioned above yStart. You can use TA_BASELINE to position a string so that the baseline is at yStart. The baseline is the line below which descenders (such as those on the lowercase p, q, and y) hang.

If you call SetTextAlign with the TA_UPDATECP flag, Windows ignores the xStart and yStart arguments to TextOut and instead uses the current position previously set by MoveToEx or LineTo, or another other function that changes the current position. The TA_UPDATECP flag also causes the TextOut function to update the current position to the end of the string

(for TA_LEFT) or the beginning of the string (for TA_RIGHT). This is useful for displaying a line of text with multiple TextOut calls. When the horizontal positioning is TA_CENTER, the current position remains the same after a TextOut call.

You'll recall that displaying columnar text in the series of SYSMETS programs in Chapter 4 required that one TextOut call be used for each column. An alternative is the TabbedTextOut function:

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

If the text string contains embedded tab characters ('\t' or 0x09), TabbedTextOut will expand the tabs into spaces based on an array of integers you pass to the function.

The first five arguments to TabbedTextOut are the same as those to TextOut. The sixth argument is the number of tab stops, and the seventh argument is an array of tab stops in units of pixels. For example, if the average character width is 8 pixels and you want a tab stop every 5 characters, then this array would contain the numbers 40, 80, 120, and so forth, in ascending order.

If the sixth and seventh arguments are 0 or NULL, tab stops are set at every eight average character widths. If the sixth argument is 1, the seventh argument points to a single integer, which is repeated incrementally for multiple tab stops. (For example, if the sixth argument is 1 and the seventh argument points to a variable containing the number 30, tab stops are set at 30, 60, 90, ... pixels.) The last argument gives the logical x-coordinate of the starting position from which tab stops are measured. This may or may not be the same as the starting position of the string.

Another advanced text output function is ExtTextOut (the Ext prefix stands for extended):

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

The fifth argument is a pointer to a rectangle structure. This is either a clipping rectangle (if iOptions is set to ETO_CLIPPED) or a background rectangle to be filled with the current background color (if iOptions is set to ETO_OPAQUE). You can specify both options or neither.

The last argument is an array of integers that specify the spacing between consecutive characters in the string. This allows a program to tighten or loosen intercharacter spacing, which is sometimes required for justifying a single word of text in a narrow column. The argument can be set to NULL for default character spacing.

A higher-level function for writing text is DrawText, which we first encountered in the HELLOWIN program in Chapter 3. Rather than specifying a coordinate starting position, you provide a structure of type RECT that defines a rectangle in which you want the text to appear:

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

As with the other text output functions, DrawText requires a pointer to the character string and the length of the string. However, if you use DrawText

with NULL-terminated strings, you can set iCount to -1, and Windows will calculate the length of the string for you.

When iFormat is set to 0, Windows interprets the text as a series of lines that are separated by carriage-return characters ('\r' or 0x0D) or linefeed characters ('\n' or 0x0A). The text begins at the upper left corner of the rectangle. A carriage return or linefeed is interpreted as a "newline" character, so Windows breaks the current line and starts a new one. The new line begins at the left side of the rectangle, spaced one character height (without external leading) below the previous line. Any text, including parts of letters, that would be displayed to the right or below the bottom of the rectangle is clipped.

You can change the default operation of DrawText by including an iFormat argument, which consists of one or more flags. The DT_LEFT flag (the default) specifies a left- justified line, DT_RIGHT specifies a right-justified line, and DT_CENTER specifies a line centered between the left and right sides of the rectangle. Because the value of DT_LEFT is 0, you needn't include the identifier if you want text to be left-justified only.

If you don't want carriage returns or linefeeds to be interpreted as newline characters, you can include the identifier DT_SINGLELINE. Windows then interprets carriage returns and linefeeds as displayable characters rather than control characters. When using DT_SINGLELINE, you can also specify whether the line is to be placed at the top of the rectangle (DT_TOP, the default), at the bottom of the rectangle (DT_BOTTOM), or halfway between the top and bottom (DT_VCENTER, the V standing for Vertical).

When displaying multiple lines of text, Windows normally breaks the lines only at carriage returns or linefeeds. If the lines are too long to fit in the rectangle, however, you can use the DT_WORDBREAK flag, which causes Windows to create breaks at the end of words within lines. For both single-line and multiple-line displays, Windows truncates any part of the text that falls outside the rectangle. You can override this by including the flag DT_NOCLIP, which also speeds up the operation of the function. When Windows spaces multiple lines of text, it normally uses the character height without external leading. If you prefer that external leading be included in the line spacing, use the flag DT_EXTERNALLEADING.

If your text contains tab characters ('\t' or 0x09), you need to include the flag DT_EXPANDTABS. By default, the tab stops are set at every eighth character position. You can specify a different tab setting by using the flag DT_TABSTOP, in which case the upper byte of iFormat contains the character-position number of each new tab stop. I recommend that you avoid using DT_TABSTOP, however, because the upper byte of iFormat is also used for some other flags.

The problem with the DT_TABSTOP flag is solved by a newer DrawTextEx function that has an extra argument:

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

The last argument is a pointer to a DRAWTEXTPARAMS function, which is defined like so:

```
typedef struct tagDRAWTEXTPARAMS
{
    UINT cbSize ;        // size of structure
    int  iTabLength ;    // size of each tab stop
    int  iLeftMargin ;   // left margin
    int  iRightMargin ;  // right margin
    UINT uiLengthDrawn ; // receives number of characters processed
} DRAWTEXTPARAMS, * LPDRAWTEXTPARAMS ;
```

The middle three fields are in units that are increments of the average character width.

Device Context Attributes for Text

Besides SetTextAlign discussed above, several other device context attributes affect text. In the default device context, the text color is black, but you can change that with:

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

As with pen colors and hatch brush colors, Windows converts the value of rgbColor to a pure color. You can obtain the current text color by calling GetTextColor.

Windows displays text in a rectangular background area that it may or may not color based on the setting of the background mode. You can change the background mode using:

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

where iMode is either OPAQUE or TRANSPARENT. The default background mode is OPAQUE, which means that Windows uses the background color to fill in the rectangular background. You can change the background color by using:

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

The value of rgbColor is converted to that of a pure color. The default background color is white.

If two lines of text are too close to each other, the background rectangle of one may obscure the text of another. For this reason, I have often wished that the default background mode were TRANSPARENT. In that case, Windows ignores the background color and doesn't color the rectangular background area. Windows also uses the background mode and background color to color the spaces between dotted and dashed lines and the area between the hatches of hatched brushes, as I discussed in Chapter 5.

Many Windows programs specify WHITE_BRUSH as the brush that Windows uses to erase the background of a window. The brush is specified in the window class structure. However, you may want to make the background of your program's window consistent with the system colors that a user can set in the Control Panel program. In that case, you would specify the background color this way in the WNDCLASS structure:

```
wndclass.hbrBackground = COLOR_WINDOW + 1 ;
```

When you want to write text to the client area, you can then set the text color and background color using the current system colors:

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

If you do this, you'll want your program to be alerted if the system colors

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

Another device context attribute that affects text is the intercharacter spacing. By default it's set to 0, which means that Windows doesn't add any space between characters. You can insert space by using the function:

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

The iExtra argument is in logical units. Windows converts it to the nearest pixel, which can be 0. If you use a negative value for iExtra (perhaps in an attempt to squeeze characters closer together), Windows takes the absolute value of the number: You can't make the value less than 0. You can obtain the current intercharacter spacing by calling GetTextCharacterExtra. Windows converts the pixel spacing to logical units before returning the value.

Using Stock Fonts

When you call TextOut, TabbedTextOut, ExtTextOut, DrawText, or DrawTextEx to write text, Windows uses the font currently selected in the device context. The font defines a particular typeface and a size. The easiest way to display text with various fonts is to use the stock fonts that Windows provides. However, the range of these is quite limited.

You can obtain a handle to a stock font by calling:

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

where iFont is one of several identifiers. You can then select that font into the device context:

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

Or you can accomplish this in one step:

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

The font selected in the default device context is called the system font and is identified by the GetStockObject argument SYSTEM_FONT. This is a proportional ANSI character set font. Specifying SYSTEM_FIXED_FONT in GetStockObject (which I did in a few programs earlier in this book) gives you a handle to a fixed-pitch font compatible with the system font used in versions of Windows prior to version 3. This is very convenient when you need all the font characters to have the same width.

The stock OEM_FIXED_FONT (also called the Terminal font) is the font that Windows uses in DOS Command Prompt windows. It incorporates a character set compatible with the original extended character set of the IBM PC. Windows uses DEFAULT_GUI_FONT for the text in window title bars, menus, and dialog boxes.

When you select a new font into a device context, you must calculate the font's character height and average character width using GetTextMetrics. If you've selected a proportional font, be aware that the average character width is really an average and that some characters have a lesser or greater width. Later in this chapter you'll learn how to determine the full width of a string made up of variable-width characters.

Although GetStockObject certainly offers the easiest access to different

fonts, you don't have much control over which font Windows gives you. You'll see shortly how you can be very specific about the typeface and size that you want.

Background on Fonts

Much of the remainder of this chapter addresses working with different fonts. Before you get involved with specific code, however, youÆll benefit from having a firm grasp of the basics of fonts as they are implemented in Windows.

The Types of Fonts

Windows supports two broad categories of fonts, called "GDI fonts" and "device fonts." The GDI fonts are stored in files on your hard disk. Device fonts are native to an output device. For example, it is very common for printers to have a collection of built-in device fonts.

GDI fonts come in three flavors: raster fonts, stroke fonts, and TrueType fonts.

A raster font is sometimes also called a bitmap font, because each character is stored as a bitmap pixel pattern. Each raster font is designed for a specific aspect ratio and character size. Windows can create larger character sizes from GDI raster fonts by simply duplicating rows or columns of pixels. However, this can be done only in integral multiples and within certain limits. For this reason, GDI raster fonts are termed "nonscalable" fonts. They cannot be expanded or compressed to an arbitrary size. The primary advantages of raster fonts are performance (because they are very fast to display) and readability (because they have been hand-designed to be as legible as possible).

Fonts are identified by typeface names. The raster fonts have typeface names of:

System (used for SYSTEM_FONT)
FixedSys (used for SYSTEM_FIXED_FONT)
Terminal (used for OEM_FIXED_FONT)
Courier
MS Serif
MS Sans Serif (used for DEFAULT_GUI_FONT)
Small Fonts

Each raster font comes in just a few (no more than six) different sizes. The Courier font is a fixed-pitch font similar in appearance to a typewriter. The word ôserifö refers to small turns that often finish the strokes of letters in a font such as the one used for this book. A ôsans serifö font (such as that used for the lettering on the cover of this book) [IS THAT TRUE ?????] doesnÆt have serifs. In early versions of Windows, the MS (Microsoft) Serif and MS Sans Serif fonts were called Tms Rmn (meaning that it was a font similar to Times Roman) and Helv (similar to Helvetica). The Small Fonts are especially designed for displaying text in small sizes.

Prior to Windows 3.1, the only other GDI fonts supplied with Windows were the stroke fonts. The stroke fonts are defined as a series of line segments in a "connect-the-dots" format. Stroke fonts are continuously scalable,

which means that the same font can be used for graphics output devices of any resolution, and the fonts can be increased or decreased to any size. However, performance is poor, legibility suffers greatly at small sizes, and at large sizes the characters look decidedly weak because their strokes are single lines. Stroke fonts are now sometimes called plotter fonts because they are particularly suitable for plotters but not for anything else. The stroke fonts have typeface names of Modern, Roman, and Script.

For both GDI raster fonts and GDI stroke fonts, Windows can "synthesize" boldface, italics, underlining, and strikethroughs without storing separate fonts for each attribute. For italics, for instance, Windows simply shifts the upper part of the character to the right.

Then there is TrueType, to which much the remainder of this chapter will be devoted.

TrueType Fonts

The individual characters of TrueType fonts are defined by filled outlines of straight lines and curves. Windows can scale these fonts by altering the coordinates that define the outlines.

When your program begins to use a TrueType font of a particular size, Windows "rasterizes" the font. This means that Windows scales the coordinates connecting the lines and curves of each character using "hints" that are included in the TrueType font file. These hints compensate for rounding errors that would otherwise cause a resultant character to be unsightly. (For example, in some fonts the two legs of a capital H should be the same width. A blind scaling of the font could result in one leg being a pixel wider than the other. The hints prevent this from happening.) The resultant outline of each character is then used to create a bitmap of the character. These bitmaps are cached in memory for future use.

Originally, Windows was equipped with 13 TrueType fonts, which have the following typeface names:

- Courier New
- Courier New Bold
- Courier New Italic
- Courier New Bold Italic
- Times New Roman
- Times New Roman Bold
- Times New Roman Italic
- Times New Roman Bold Italic
- Arial
- Arial Bold
- Arial Italic
- Arial Bold Italic
- Symbol

In more recent versions of Windows, this list has been expanded. In particular, IÆll be making use of the Lucida Sans Unicode font that includes some additional alphabets used around the world.

The three main font families are similar to the main raster fonts. Courier New is a fixed-pitch font designed to look like the output from that

antique piece of hardware known as a typewriter. Times New Roman is a clone of the Times font originally designed for the Times of London and used in many printed material. It is considered to be highly readable. Arial is clone of Helvetica, a sans-serif font. The Symbol font contains a collection of handy symbols.

Attributes or Styles?

YouÆll notice in the list of TrueType fonts shown above that bold and italic styles of Courier, Times New Roman, and Arial seem to be separate fonts with their own typeface names. This naming is very much in accordance with traditional typography. However, computer users have come to think of bold and italic as particular ôattributesö that are applied to existing fonts. Windows itself took the attribute approach early on when defining how the raster fonts were named, enumerated, and selected. With TrueType fonts, however, more traditional naming is preferred.

This conflict is not quite ever resolved in Windows. In short, as youÆll see, you can select fonts by either naming them fully or by specifying attributes. The process of font enumeration (in which an application requests a list of fonts from the system) isùas you might expectùcomplicated somewhat by this dual approach.

The Point Size

In traditional typography, you specify a font by its typeface name and its size. The type size is expressed in units called points. A point is very close to 1/72 inchùso close in face that in computer typography it is often defined as exactly 1/72 inch. The text of this book is printed in 10-point type. The point size is usually described as the height of the characters from the top of the ascenders (without diacritics) to the bottom of the descenders, for example, encompassing the full height of the letters ôbq.ö That's a convenient way to think of the type size, but it's usually not metrically accurate.

The point size of a font is actually a typographical design concept rather than a metrical concept. The size of the characters in a particular font may be greater than or less than what the point size may imply. In traditional typography you use a point size to specify the size of a font; in computer typography there are other methods to determine the actual size of the characters.

Leading and Spacing

As youÆll recall from as long ago as Chapter 4, you can obtain information about the font currently selected in the device context by calling GetTextMetrics, as weÆve also done frequently since then. Chapter 4 included the diagram shown in Figure 17-1 illustrating the vertical sizes of a font from the FONTMETRIC structure.

[Repeat Figure 4-3 here]

Figure 17-1.
Four values defining vertical character sizes in a font.

Another field of the TEXTMETRIC structure is named tmExternalLeading. The word leading (pronounced ôleddingö) is derived from the lead that typesetters insert between blocks of metal type to add white space between

lines of text. The tmInternalLeading value corresponds to the space usually reserved for diacriticis; tmExternalLeading suggests an additional space to leave between successive lines of characters. Programmers can use or ignore the external leading value.

When we refer to a font as being 8-point or 12-point, weÆre actually talking about the height of the font less internal leading. The diacritics on certain capital letters are considered to occupy the space that normally separates lines of type. The tmHeight value of the TEXTMETRIC structure thus actually refers to line spacing rather than the font point size. The point size can be derived from tmHeight minus tmInternalLeading.

The Logical Inch Problem

As I discussed in Chapter 5 (in the section entitled ôThe Size of the Deviceö on pages 133 through 138) Windows 98 defines the system font as being a 10-point font with a 12-point line spacing. Depending on whether you choose Small Fonts or Large Fonts from the Display Properties dialog, this font could have a tmHeight value of 16 pixels or 20 pixels, and a tmHeight minus tmInternalLeading value of 13 pixels or 16 pixels. Thus, the choice of the font implies a resolution of the device in dots per inch, namely 96 dpi when Small Fonts are selected and 120 dpi for Large Fonts. You can obtain this implied resolution of the device by calling GetDeviceCaps with the LOGPIXELSX or LOGPIXELSY arguments. Thus, the metrical distance occupied by 96 or 120 pixels on the screen can be said to be a ôlogical inch.ö If you start measuring your screen with a ruler and counting pixels, youÆll probably find that a logical inch is larger than an actual inch. Why is this?

On paper, 8-point type with about 14 characters horizontally per inch is perfectly readable. If you were programming a word processing or page-composition application, you would want to be able to show legible 8-point type on the display. But if you used the actual dimensions of the video display, there would probably not be enough pixels to show the character legibly. Even if the display had sufficient resolution, you might still have problems reading actual 8-point type on a screen. When people read print on paper, the distance between the eyes and the text is generally about a foot, but a video display is commonly viewed from a distance of two feet.

The logical inch in effect provides a magnification of the screen, allowing the display of legible fonts in a size as small as 8 points. Also, having 96 dots per logical inch makes the 640-pixel minimum display size equal to about 6.5 inches. This is precisely the width of text that prints on 8.5-inch-wide paper when you use the standard margins of an inch on each side. So the logical inch also takes advantage of the width of the screen to allow text to be displayed as large as possible.

As you may also recall from Chapter 5, Windows NT does it a little differently. In Windows NT, the LOGPIXELSX (pixels per inch) value you obtain from GetDeviceCaps is not equal to the HORZRES value (in pixels) divided by the HORZSIZE value (in millimeters), multiplied by 25.4. Similarly, LOGPIXELSY, VERTRES, and VERTSIZE are not consistent. Windows

uses the HORZRES, HORZSIZE, VERTRES, and VERTSIZE values when calculating window and offset extents for the various mapping modes; however, a program that displays text would be better off to use an assumed display resolution based on LOGPIXELSX and LOGPIXELSY. This is more consistent with Windows 98.

So, under Windows NT a program should probably not use the mapping modes provided by Windows when also displaying text in specific point sizes. The program should instead define its own mapping mode based on the logical-pixels-per-inch dimensions consistent with Windows 98. One such useful mapping mode for text I call the ôLogical Twipsö mapping mode. HereÆs how you set it:

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

With this mapping mode set, you can specify font dimensions in 20 times the point size, for example 240 for 12 points. Notice that unlike the MM_TWIPS mapping mode, the values of y increase going down the screen. This is easier when displaying successive lines of text.

Keep in mind that the discrepancy between logical inches and real inches occurs only for the display. On printer devices, there is total consistency with GDI and rulers.

The Logical Font

Now that weÆve nailed down the concept of logical inches and logical twips, itÆs time to talk about logical fonts.

A logical font is a GDI object whose handle is stored in a variable of type HFONT. A logical font is a description of a font. Like the logical pen and logical brush, it is an abstract object that becomes real only when it is a selected into a device context when an application calls SelectObject. For logical pens (for instance), you can specify any color you want for the pen, but Windows converts that to a pure color available on the device when you select the pen into the device context. Only then does Windows know about the color capabilities of the device.

Logical Font Creation and Selection

You create a logical font by calling CreateFont or CreateFontIndirect. The CreateFontIndirect function takes a pointer to a LOGFONT structure, which has 14 fields. The CreateFont function takes 14 arguments, which are identical to the 14 fields of the LOGFONT structure. These are the only two functions that create a logical font. (I mention that because there are multiple functions in Windows for some other font jobs.) Because the 14 fields are difficult to remember, CreateFont is rarely used, so IÆll focus on CreateFontIndirect.

There are three basic ways to define the fields of a LOGFONT structure in preparation for calling CreateFontIndirect:

You can simply set the fields of the LOGFONT structure to the characteristics of the font that you want. In this case, when you call SelectObject Windows uses a ôfont mappingö algorithm to attempt to give you

the font available on the device that best matches these characteristics. Depending on the fonts available on the video display or printer, the result may differ considerably from what you request.

You can enumerate all the fonts on the device and choose from those or even present them to the user with a dialog box. IÆll discuss the font enumeration functions later in this chapter. These are not used much these days because the third method does the enumeration for you.

You can take the simple approach and call the ChooseFont function, which I discussed a little in Chapter 11. You get back a LOGFONT structure that you can use directly for creating the font.

In this chapter, IÆll use the first and third approaches.

Here is the process for creating, selecting, and deleting logical fonts:

Create a logical font by calling CreateFont or CreateFontIndirect. These functions return a handle to a logical font of type HFONT.

Select the logical font into the device context using SelectObject. Windows chooses a real font that most closely matches the logical font.

Determine the size and characteristics of the real font with GetTextMetrics (and possibly some other functions). You can use this information to properly space the text that you write when this font is selected into the device context.

After youÆve finished using the font, delete the logical font by calling DeleteObject. DonÆt delete the font while it is selected in a valid device context, and donÆt delete stock fonts.

The GetTextFace function lets a program determine the face name of the font currently selected in the device context:

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

The detailed font information is available from GetTextMetrics:

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

where textmetric is a variable of type TEXTMETRIC, a structure with 20 fields.

IÆll discuss the fields of the LOGFONT and TEXTMETRIC structures in detail shortly. The structures have some similar fields, so they can be confusing. For now, just keep in mind that LOGFONT is for defining a logical font, and TEXTMETRIC is for obtaining information about the font currently selected in the device context.

The PICKFONT Program

With the PICKFONT program shown in Figure 17-1 you can define many of the fields of a LOGFONT structure. The program creates a logical font and displays the characteristics of the real font after the logical font has been selected in the screen device context. This is a very handy program for understanding how logical fonts are mapped to real fonts.

PICKFONT.C

```
/*-----------------------------------------
   PICKFONT.C -- Create Logical Font
                 (c) Charles Petzold, 1998
  ----------------------------------------*/
```

```
#include <windows.h>
#include "resource.h"

     // Structure shared between main window and dialog box

typedef struct
{
     int        iDevice, iMapMode ;
     BOOL       fMatchAspect ;
     BOOL       fAdvGraphics ;
     LOGFONT    lf ;
     TEXTMETRIC tm ;
     TCHAR      szFaceName [LF_FULLFACESIZE] ;
}
DLGPARAMS ;

     // Formatting for BCHAR fields of TEXTMETRIC structure

#ifdef UNICODE
#define BCHARFORM TEXT ("0x%04X")
#else
#define BCHARFORM TEXT ("0x%02X")
#endif

     // Global variables

HWND  hdlg ;
TCHAR szAppName[] = TEXT ("PickFont") ;

     // Forward declarations of functions

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
BOOL    CALLBACK DlgProc (HWND, UINT, WPARAM, LPARAM) ;
void SetLogFontFromFields    (HWND hdlg, DLGPARAMS * pdp) ;
void SetFieldsFromTextMetric (HWND hdlg, DLGPARAMS * pdp) ;
void MySetMapMode (HDC hdc, int iMapMode) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     HWND     hwnd ;
     MSG      msg ;
     WNDCLASS wndclass ;

     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
```

```
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;

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

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

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

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

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
     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 ")
                                    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:
          dp.iDevice = IDM_DEVICE_SCREEN ;

          hdlg = CreateDialogParam (((LPCREATESTRUCT) lParam)->hInstance,
                                    szAppName, hwnd, DlgProc, (LPARAM) &dp)
;
          return 0 ;

     case WM_SETFOCUS:
          SetFocus (hdlg) ;
          return 0 ;

     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 ;

     case WM_PAINT:
          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

          GetClientRect (hdlg, &rect) ;
          rect.bottom += 1 ;
          DPtoLP (hdc, (PPOINT) &rect, 2) ;

               // Create and select the font; display the text

          SelectObject (hdc, CreateFontIndirect (&dp.lf)) ;
          TextOut (hdc, rect.left, rect.bottom, szText, lstrlen (szText)) ;

          DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY:
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}

BOOL CALLBACK DlgProc (HWND hdlg, UINT message, WPARAM wParam, LPARAM
lParam)
{
     static DLGPARAMS * pdp ;
     static PRINTDLG    pd = { sizeof (PRINTDLG) } ;
     HDC                hdcDevice ;
     HFONT              hFont ;

     switch (message)
     {
     case WM_INITDIALOG:
               // Save pointer to dialog-parameters structure in WndProc

          pdp = (DLGPARAMS *) lParam ;

          SendDlgItemMessage (hdlg, IDC_LF_FACENAME, EM_LIMITTEXT,
```

```
                              LF_FACESIZE - 1, 0) ;

          CheckRadioButton (hdlg, IDC_OUT_DEFAULT, IDC_OUT_OUTLINE,
                                  IDC_OUT_DEFAULT) ;

          CheckRadioButton (hdlg, IDC_DEFAULT_QUALITY, IDC_PROOF_QUALITY,
                                  IDC_DEFAULT_QUALITY) ;

          CheckRadioButton (hdlg, IDC_DEFAULT_PITCH, IDC_VARIABLE_PITCH,
                                  IDC_DEFAULT_PITCH) ;

          CheckRadioButton (hdlg, IDC_FF_DONTCARE, IDC_FF_DECORATIVE,
                                  IDC_FF_DONTCARE) ;

          CheckRadioButton (hdlg, IDC_MM_TEXT, IDC_MM_LOGTWIPS,
                                  IDC_MM_TEXT) ;

          SendMessage (hdlg, WM_COMMAND, IDOK, 0) ;

                              // fall through
     case WM_SETFOCUS:
          SetFocus (GetDlgItem (hdlg, IDC_LF_HEIGHT)) ;
          return FALSE ;

     case WM_COMMAND:
          switch (LOWORD (wParam))
          {
          case IDC_CHARSET_HELP:
               MessageBox (hdlg,
                           TEXT ("0 = Ansi\n")
                           TEXT ("1 = Default\n")
                           TEXT ("2 = Symbol\n")
                           TEXT ("128 = Shift JIS (Japanese)\n")
                           TEXT ("129 = Hangul (Korean)\n")
                           TEXT ("130 = Johab (Korean)\n")
                           TEXT ("134 = GB 2312 (Simplified Chinese)\n")
                           TEXT ("136 = Chinese Big 5 (Tradtional Chinese)
\n")
                           TEXT ("177 = Hebrew\n")
                           TEXT ("178 = Arabic\n")
                           TEXT ("161 = Greek\n")
                           TEXT ("162 = Turkish\n")
                           TEXT ("163 = Vietnamese\n")
                           TEXT ("204 = Russian\n")
                           TEXT ("222 = Thai\n")
                           TEXT ("238 = East European\n")
                           TEXT ("255 = OEM"),
```

```
                    szAppName, MB_OK | MB_ICONINFORMATION) ;
          return TRUE ;

          // These radio buttons set the lfOutPrecision field

     case IDC_OUT_DEFAULT:
          pdp->lf.lfOutPrecision = OUT_DEFAULT_PRECIS ;
          return TRUE ;

     case IDC_OUT_STRING:
          pdp->lf.lfOutPrecision = OUT_STRING_PRECIS ;
          return TRUE ;

     case IDC_OUT_CHARACTER:
          pdp->lf.lfOutPrecision = OUT_CHARACTER_PRECIS ;
          return TRUE ;

     case IDC_OUT_STROKE:
          pdp->lf.lfOutPrecision = OUT_STROKE_PRECIS ;
          return TRUE ;

     case IDC_OUT_TT:
          pdp->lf.lfOutPrecision = OUT_TT_PRECIS ;
          return TRUE ;

     case IDC_OUT_DEVICE:
          pdp->lf.lfOutPrecision = OUT_DEVICE_PRECIS ;
          return TRUE ;

     case IDC_OUT_RASTER:
          pdp->lf.lfOutPrecision = OUT_RASTER_PRECIS ;
          return TRUE ;

     case IDC_OUT_TT_ONLY:
          pdp->lf.lfOutPrecision = OUT_TT_ONLY_PRECIS ;
          return TRUE ;

     case IDC_OUT_OUTLINE:
          pdp->lf.lfOutPrecision = OUT_OUTLINE_PRECIS ;
          return TRUE ;

          // These three radio buttons set the lfQuality field

     case IDC_DEFAULT_QUALITY:
          pdp->lf.lfQuality = DEFAULT_QUALITY ;
          return TRUE ;
```

```
case IDC_DRAFT_QUALITY:
     pdp->lf.lfQuality = DRAFT_QUALITY ;
     return TRUE ;

case IDC_PROOF_QUALITY:
     pdp->lf.lfQuality = PROOF_QUALITY ;
     return TRUE ;

     // These three radio buttons set the lower nibble
     //   of the lpPitchAndFamily field

case IDC_DEFAULT_PITCH:
     pdp->lf.lfPitchAndFamily =
          (0xF0 & pdp->lf.lfPitchAndFamily) | DEFAULT_PITCH ;
     return TRUE ;

case IDC_FIXED_PITCH:
     pdp->lf.lfPitchAndFamily =
          (0xF0 & pdp->lf.lfPitchAndFamily) | FIXED_PITCH ;
     return TRUE ;

case IDC_VARIABLE_PITCH:
     pdp->lf.lfPitchAndFamily =
          (0xF0 & pdp->lf.lfPitchAndFamily) | VARIABLE_PITCH ;
     return TRUE ;

     // These six radio buttons set the upper nibble
     //   of the lpPitchAndFamily field

case IDC_FF_DONTCARE:
     pdp->lf.lfPitchAndFamily =
          (0x0F & pdp->lf.lfPitchAndFamily) | FF_DONTCARE ;
     return TRUE ;

case IDC_FF_ROMAN:
     pdp->lf.lfPitchAndFamily =
          (0x0F & pdp->lf.lfPitchAndFamily) | FF_ROMAN;
     return TRUE ;

case IDC_FF_SWISS:
     pdp->lf.lfPitchAndFamily =
          (0x0F & pdp->lf.lfPitchAndFamily) | FF_SWISS ;
     return TRUE ;

case IDC_FF_MODERN:
     pdp->lf.lfPitchAndFamily =
          (0x0F & pdp->lf.lfPitchAndFamily) | FF_MODERN ;
```

```
               return TRUE ;

          case IDC_FF_SCRIPT:
               pdp->lf.lfPitchAndFamily =
                    (0x0F & pdp->lf.lfPitchAndFamily) | FF_SCRIPT ;
               return TRUE ;

          case IDC_FF_DECORATIVE:
               pdp->lf.lfPitchAndFamily =
                    (0x0F & pdp->lf.lfPitchAndFamily) | FF_DECORATIVE ;
               return TRUE ;

               // Mapping mode:

          case IDC_MM_TEXT:
          case IDC_MM_LOMETRIC:
          case IDC_MM_HIMETRIC:
          case IDC_MM_LOENGLISH:
          case IDC_MM_HIENGLISH:
          case IDC_MM_TWIPS:
          case IDC_MM_LOGTWIPS:
               pdp->iMapMode = LOWORD (wParam) ;
               return TRUE ;

               // OK button pressed
               // -----------------

          case IDOK:
                    // Get LOGFONT structure

               SetLogFontFromFields (hdlg, pdp) ;

                    // Set Match-Aspect and Advanced Graphics flags

               pdp->fMatchAspect = IsDlgButtonChecked (hdlg,
IDC_MATCH_ASPECT) ;
               pdp->fAdvGraphics = IsDlgButtonChecked (hdlg,
IDC_ADV_GRAPHICS) ;

                    // Get Information Context

               if (pdp->iDevice == IDM_DEVICE_SCREEN)
               {
                    hdcDevice = CreateIC (TEXT ("DISPLAY"), NULL, NULL,
NULL) ;
               }
               else
```

```
               {
                    pd.hwndOwner = hdlg ;
                    pd.Flags = PD_RETURNDEFAULT | PD_RETURNIC ;
                    pd.hDevNames = NULL ;
                    pd.hDevMode = NULL ;

                    PrintDlg (&pd) ;

                    hdcDevice = pd.hDC ;
               }
                    // Set the mapping mode and the mapper flag

               MySetMapMode (hdcDevice, pdp->iMapMode) ;
               SetMapperFlags (hdcDevice, pdp->fMatchAspect) ;

                    // Create font and select it into IC

               hFont = CreateFontIndirect (&pdp->lf) ;
               SelectObject (hdcDevice, hFont) ;

                    // Get the text metrics and face name

               GetTextMetrics (hdcDevice, &pdp->tm) ;
               GetTextFace (hdcDevice, LF_FULLFACESIZE, pdp->szFaceName) ;
               DeleteDC (hdcDevice) ;
               DeleteObject (hFont) ;

                    // Update dialog fields and invalidate main window

               SetFieldsFromTextMetric (hdlg, pdp) ;
               InvalidateRect (GetParent (hdlg), NULL, TRUE) ;
               return TRUE ;
          }
          break ;
     }
     return FALSE ;
}
void SetLogFontFromFields (HWND hdlg, DLGPARAMS * pdp)
{
     pdp->lf.lfHeight      = GetDlgItemInt (hdlg, IDC_LF_HEIGHT,  NULL,
TRUE) ;
     pdp->lf.lfWidth       = GetDlgItemInt (hdlg, IDC_LF_WIDTH,   NULL,
TRUE) ;
     pdp->lf.lfEscapement  = GetDlgItemInt (hdlg, IDC_LF_ESCAPE,  NULL,
TRUE) ;
     pdp->lf.lfOrientation = GetDlgItemInt (hdlg, IDC_LF_ORIENT,  NULL,
TRUE) ;
```

```
     pdp->lf.lfWeight      = GetDlgItemInt (hdlg, IDC_LF_WEIGHT,  NULL,
TRUE) ;
     pdp->lf.lfCharSet     = GetDlgItemInt (hdlg, IDC_LF_CHARSET, NULL,
FALSE) ;

     pdp->lf.lfItalic =
                    IsDlgButtonChecked (hdlg, IDC_LF_ITALIC) == BST_CHECKED
;
     pdp->lf.lfUnderline =
                    IsDlgButtonChecked (hdlg, IDC_LF_UNDER)  == BST_CHECKED
;
     pdp->lf.lfStrikeOut =
                    IsDlgButtonChecked (hdlg, IDC_LF_STRIKE) == BST_CHECKED
;

     GetDlgItemText (hdlg, IDC_LF_FACENAME, pdp->lf.lfFaceName,
LF_FACESIZE) ;
}

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

     SetDlgItemInt (hdlg, IDC_TM_HEIGHT,   pdp->tm.tmHeight,
TRUE) ;
     SetDlgItemInt (hdlg, IDC_TM_ASCENT,   pdp->tm.tmAscent,
TRUE) ;
     SetDlgItemInt (hdlg, IDC_TM_DESCENT,  pdp->tm.tmDescent,
TRUE) ;
     SetDlgItemInt (hdlg, IDC_TM_INTLEAD,  pdp->tm.tmInternalLeading,
TRUE) ;
     SetDlgItemInt (hdlg, IDC_TM_EXTLEAD,  pdp->tm.tmExternalLeading,
TRUE) ;
     SetDlgItemInt (hdlg, IDC_TM_AVECHAR,  pdp->tm.tmAveCharWidth,
TRUE) ;
     SetDlgItemInt (hdlg, IDC_TM_MAXCHAR,  pdp->tm.tmMaxCharWidth,
TRUE) ;
     SetDlgItemInt (hdlg, IDC_TM_WEIGHT,   pdp->tm.tmWeight,
TRUE) ;
     SetDlgItemInt (hdlg, IDC_TM_OVERHANG, pdp->tm.tmOverhang,
TRUE) ;
```

```
     SetDlgItemInt (hdlg, IDC_TM_DIGASPX,  pdp->tm.tmDigitizedAspectX,
TRUE) ;
     SetDlgItemInt (hdlg, IDC_TM_DIGASPY,  pdp->tm.tmDigitizedAspectY,
TRUE) ;

     wsprintf (szBuffer, BCHARFORM, pdp->tm.tmFirstChar) ;
     SetDlgItemText (hdlg, IDC_TM_FIRSTCHAR, szBuffer) ;

     wsprintf (szBuffer, BCHARFORM, pdp->tm.tmLastChar) ;
     SetDlgItemText (hdlg, IDC_TM_LASTCHAR, szBuffer) ;

     wsprintf (szBuffer, BCHARFORM, pdp->tm.tmDefaultChar) ;
     SetDlgItemText (hdlg, IDC_TM_DEFCHAR, szBuffer) ;

     wsprintf (szBuffer, BCHARFORM, pdp->tm.tmBreakChar) ;
     SetDlgItemText (hdlg, IDC_TM_BREAKCHAR, szBuffer) ;

     SetDlgItemText (hdlg, IDC_TM_ITALIC, pdp->tm.tmItalic     ? szYes :
szNo) ;
     SetDlgItemText (hdlg, IDC_TM_UNDER,  pdp->tm.tmUnderlined ? szYes :
szNo) ;
     SetDlgItemText (hdlg, IDC_TM_STRUCK, pdp->tm.tmStruckOut  ? szYes :
szNo) ;

     SetDlgItemText (hdlg, IDC_TM_VARIABLE,
               TMPF_FIXED_PITCH & pdp->tm.tmPitchAndFamily ? szYes :
szNo) ;

     SetDlgItemText (hdlg, IDC_TM_VECTOR,
               TMPF_VECTOR & pdp->tm.tmPitchAndFamily ? szYes : szNo) ;

     SetDlgItemText (hdlg, IDC_TM_TRUETYPE,
               TMPF_TRUETYPE & pdp->tm.tmPitchAndFamily ? szYes : szNo) ;

     SetDlgItemText (hdlg, IDC_TM_DEVICE,
               TMPF_DEVICE & pdp->tm.tmPitchAndFamily ? szYes : szNo) ;

     SetDlgItemText (hdlg, IDC_TM_FAMILY,
               szFamily [min (6, pdp->tm.tmPitchAndFamily >> 4)]) ;

     SetDlgItemInt  (hdlg, IDC_TM_CHARSET,   pdp->tm.tmCharSet, FALSE) ;
     SetDlgItemText (hdlg, IDC_TM_FACENAME, pdp->szFaceName) ;
}

void MySetMapMode (HDC hdc, int iMapMode)
{
     switch (iMapMode)
```

```
     {
     case IDC_MM_TEXT:       SetMapMode (hdc, MM_TEXT) ;       break ;
     case IDC_MM_LOMETRIC:   SetMapMode (hdc, MM_LOMETRIC) ;   break ;
     case IDC_MM_HIMETRIC:   SetMapMode (hdc, MM_HIMETRIC) ;   break ;
     case IDC_MM_LOENGLISH:  SetMapMode (hdc, MM_LOENGLISH) ;  break ;
     case IDC_MM_HIENGLISH:  SetMapMode (hdc, MM_HIENGLISH) ;  break ;
     case IDC_MM_TWIPS:      SetMapMode (hdc, MM_TWIPS) ;      break ;
     case IDC_MM_LOGTWIPS:
          SetMapMode (hdc, MM_ANISOTROPIC) ;
          SetWindowExtEx (hdc, 1440, 1440, NULL) ;
          SetViewportExtEx (hdc, GetDeviceCaps (hdc, LOGPIXELSX),
                                 GetDeviceCaps (hdc, LOGPIXELSY), NULL) ;
          break ;
     }
}
PICKFONT.RC
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////
//
// Dialog

PICKFONT DIALOG DISCARDABLE  0, 0, 348, 308
STYLE WS_CHILD | WS_VISIBLE | WS_BORDER
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT           "&Height:",IDC_STATIC,8,10,44,8
    EDITTEXT        IDC_LF_HEIGHT,64,8,24,12,ES_AUTOHSCROLL
    LTEXT           "&Width",IDC_STATIC,8,26,44,8
    EDITTEXT        IDC_LF_WIDTH,64,24,24,12,ES_AUTOHSCROLL
    LTEXT           "Escapement:",IDC_STATIC,8,42,44,8
    EDITTEXT        IDC_LF_ESCAPE,64,40,24,12,ES_AUTOHSCROLL
    LTEXT           "Orientation:",IDC_STATIC,8,58,44,8
    EDITTEXT        IDC_LF_ORIENT,64,56,24,12,ES_AUTOHSCROLL
    LTEXT           "Weight:",IDC_STATIC,8,74,44,8
    EDITTEXT        IDC_LF_WEIGHT,64,74,24,12,ES_AUTOHSCROLL
    GROUPBOX        "Mapping Mode",IDC_STATIC,97,3,96,90,WS_GROUP
    CONTROL
"Text",IDC_MM_TEXT,"Button",BS_AUTORADIOBUTTON,104,13,56,
                    8
    CONTROL         "Low
Metric",IDC_MM_LOMETRIC,"Button",BS_AUTORADIOBUTTON,
                    104,24,56,8
    CONTROL         "High Metric",IDC_MM_HIMETRIC,"Button",
```

```
                    BS_AUTORADIOBUTTON,104,35,56,8
    CONTROL         "Low English",IDC_MM_LOENGLISH,"Button",
                    BS_AUTORADIOBUTTON,104,46,56,8
    CONTROL         "High English",IDC_MM_HIENGLISH,"Button",
                    BS_AUTORADIOBUTTON,104,57,56,8
    CONTROL
"Twips",IDC_MM_TWIPS,"Button",BS_AUTORADIOBUTTON,104,68,
                    56,8
    CONTROL         "Logical Twips",IDC_MM_LOGTWIPS,"Button",
                    BS_AUTORADIOBUTTON,104,79,64,8
    CONTROL         "Italic",IDC_LF_ITALIC,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,8,90,48,12
    CONTROL         "Underline",IDC_LF_UNDER,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,8,104,48,12
    CONTROL         "Strike Out",IDC_LF_STRIKE,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,8,118,48,12
    CONTROL         "Match
Aspect",IDC_MATCH_ASPECT,"Button",BS_AUTOCHECKBOX |
                    WS_TABSTOP,60,104,62,8
    CONTROL         "Adv Grfx Mode",IDC_ADV_GRAPHICS,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,60,118,62,8
    LTEXT           "Character Set:",IDC_STATIC,8,137,46,8
    EDITTEXT        IDC_LF_CHARSET,58,135,24,12,ES_AUTOHSCROLL
    PUSHBUTTON      "?",IDC_CHARSET_HELP,90,135,14,14
    GROUPBOX        "Quality",IDC_STATIC,132,98,62,48,WS_GROUP
    CONTROL         "Default",IDC_DEFAULT_QUALITY,"Button",
                    BS_AUTORADIOBUTTON,136,110,40,8
    CONTROL         "Draft",IDC_DRAFT_QUALITY,"Button",BS_AUTORADIOBUTTON,
                    136,122,40,8
    CONTROL         "Proof",IDC_PROOF_QUALITY,"Button",BS_AUTORADIOBUTTON,
                    136,134,40,8
    LTEXT           "Face Name:",IDC_STATIC,8,154,44,8
    EDITTEXT        IDC_LF_FACENAME,58,152,136,12,ES_AUTOHSCROLL
    GROUPBOX        "Output Precision",IDC_STATIC,8,166,118,133,WS_GROUP
    CONTROL         "OUT_DEFAULT_PRECIS",IDC_OUT_DEFAULT,"Button",
                    BS_AUTORADIOBUTTON,12,178,112,8
    CONTROL         "OUT_STRING_PRECIS",IDC_OUT_STRING,"Button",
                    BS_AUTORADIOBUTTON,12,191,112,8
    CONTROL         "OUT_CHARACTER_PRECIS",IDC_OUT_CHARACTER,"Button",
                    BS_AUTORADIOBUTTON,12,204,112,8
    CONTROL         "OUT_STROKE_PRECIS",IDC_OUT_STROKE,"Button",
                    BS_AUTORADIOBUTTON,12,217,112,8
    CONTROL         "OUT_TT_PRECIS",IDC_OUT_TT,"Button",BS_AUTORADIOBUTTON,
                    12,230,112,8
    CONTROL         "OUT_DEVICE_PRECIS",IDC_OUT_DEVICE,"Button",
                    BS_AUTORADIOBUTTON,12,243,112,8
    CONTROL         "OUT_RASTER_PRECIS",IDC_OUT_RASTER,"Button",
```

```
                    BS_AUTORADIOBUTTON,12,256,112,8
    CONTROL         "OUT_TT_ONLY_PRECIS",IDC_OUT_TT_ONLY,"Button",
                    BS_AUTORADIOBUTTON,12,269,112,8
    CONTROL         "OUT_OUTLINE_PRECIS",IDC_OUT_OUTLINE,"Button",
                    BS_AUTORADIOBUTTON,12,282,112,8
    GROUPBOX        "Pitch",IDC_STATIC,132,166,62,50,WS_GROUP
    CONTROL
"Default",IDC_DEFAULT_PITCH,"Button",BS_AUTORADIOBUTTON,
                    137,176,52,8
    CONTROL
"Fixed",IDC_FIXED_PITCH,"Button",BS_AUTORADIOBUTTON,137,
                    189,52,8
    CONTROL         "Variable",IDC_VARIABLE_PITCH,"Button",
                    BS_AUTORADIOBUTTON,137,203,52,8
    GROUPBOX        "Family",IDC_STATIC,132,218,62,82,WS_GROUP
    CONTROL         "Don't
Care",IDC_FF_DONTCARE,"Button",BS_AUTORADIOBUTTON,
                    137,229,52,8
    CONTROL
"Roman",IDC_FF_ROMAN,"Button",BS_AUTORADIOBUTTON,137,241,
                    52,8
    CONTROL
"Swiss",IDC_FF_SWISS,"Button",BS_AUTORADIOBUTTON,137,253,
                    52,8
    CONTROL         "Modern",IDC_FF_MODERN,"Button",BS_AUTORADIOBUTTON,137,
                    265,52,8
    CONTROL         "Script",IDC_FF_SCRIPT,"Button",BS_AUTORADIOBUTTON,137,
                    277,52,8
    CONTROL         "Decorative",IDC_FF_DECORATIVE,"Button",
                    BS_AUTORADIOBUTTON,137,289,52,8
    DEFPUSHBUTTON   "OK",IDOK,247,286,50,14
    GROUPBOX        "Text Metrics",IDC_STATIC,201,2,140,272,WS_GROUP
    LTEXT           "Height:",IDC_STATIC,207,12,64,8
    LTEXT           "0",IDC_TM_HEIGHT,281,12,44,8
    LTEXT           "Ascent:",IDC_STATIC,207,22,64,8
    LTEXT           "0",IDC_TM_ASCENT,281,22,44,8
    LTEXT           "Descent:",IDC_STATIC,207,32,64,8
    LTEXT           "0",IDC_TM_DESCENT,281,32,44,8
    LTEXT           "Internal Leading:",IDC_STATIC,207,42,64,8
    LTEXT           "0",IDC_TM_INTLEAD,281,42,44,8
    LTEXT           "External Leading:",IDC_STATIC,207,52,64,8
    LTEXT           "0",IDC_TM_EXTLEAD,281,52,44,8
    LTEXT           "Ave Char Width:",IDC_STATIC,207,62,64,8
    LTEXT           "0",IDC_TM_AVECHAR,281,62,44,8
    LTEXT           "Max Char Width:",IDC_STATIC,207,72,64,8
    LTEXT           "0",IDC_TM_MAXCHAR,281,72,44,8
    LTEXT           "Weight:",IDC_STATIC,207,82,64,8
```

```
    LTEXT           "0",IDC_TM_WEIGHT,281,82,44,8
    LTEXT           "Overhang:",IDC_STATIC,207,92,64,8
    LTEXT           "0",IDC_TM_OVERHANG,281,92,44,8
    LTEXT           "Digitized Aspect X:",IDC_STATIC,207,102,64,8
    LTEXT           "0",IDC_TM_DIGASPX,281,102,44,8
    LTEXT           "Digitized Aspect Y:",IDC_STATIC,207,112,64,8
    LTEXT           "0",IDC_TM_DIGASPY,281,112,44,8
    LTEXT           "First Char:",IDC_STATIC,207,122,64,8
    LTEXT           "0",IDC_TM_FIRSTCHAR,281,122,44,8
    LTEXT           "Last Char:",IDC_STATIC,207,132,64,8
    LTEXT           "0",IDC_TM_LASTCHAR,281,132,44,8
    LTEXT           "Default Char:",IDC_STATIC,207,142,64,8
    LTEXT           "0",IDC_TM_DEFCHAR,281,142,44,8
    LTEXT           "Break Char:",IDC_STATIC,207,152,64,8
    LTEXT           "0",IDC_TM_BREAKCHAR,281,152,44,8
    LTEXT           "Italic?",IDC_STATIC,207,162,64,8
    LTEXT           "0",IDC_TM_ITALIC,281,162,44,8
    LTEXT           "Underlined?",IDC_STATIC,207,172,64,8
    LTEXT           "0",IDC_TM_UNDER,281,172,44,8
    LTEXT           "Struck Out?",IDC_STATIC,207,182,64,8
    LTEXT           "0",IDC_TM_STRUCK,281,182,44,8
    LTEXT           "Variable Pitch?",IDC_STATIC,207,192,64,8
    LTEXT           "0",IDC_TM_VARIABLE,281,192,44,8
    LTEXT           "Vector Font?",IDC_STATIC,207,202,64,8
    LTEXT           "0",IDC_TM_VECTOR,281,202,44,8
    LTEXT           "TrueType Font?",IDC_STATIC,207,212,64,8
    LTEXT           "0",IDC_TM_TRUETYPE,281,212,44,8
    LTEXT           "Device Font?",IDC_STATIC,207,222,64,8
    LTEXT           "0",IDC_TM_DEVICE,281,222,44,8
    LTEXT           "Family:",IDC_STATIC,207,232,64,8
    LTEXT           "0",IDC_TM_FAMILY,281,232,44,8
    LTEXT           "Charecter Set:",IDC_STATIC,207,242,64,8
    LTEXT           "0",IDC_TM_CHARSET,281,242,44,8
    LTEXT           "0",IDC_TM_FACENAME,207,262,128,8
END

////////////////////////////////////////////////////////////////////////
//
// Menu

PICKFONT MENU DISCARDABLE
BEGIN
    POPUP "&Device"
    BEGIN
        MENUITEM "&Screen",                     IDM_DEVICE_SCREEN, CHECKED
        MENUITEM "&Printer",                    IDM_DEVICE_PRINTER
    END
```

END
RESOURCE.H

```
// Microsoft Developer Studio generated include file.
// Used by PickFont.rc

#define IDC_LF_HEIGHT                   1000
#define IDC_LF_WIDTH                    1001
#define IDC_LF_ESCAPE                   1002
#define IDC_LF_ORIENT                   1003
#define IDC_LF_WEIGHT                   1004
#define IDC_MM_TEXT                     1005
#define IDC_MM_LOMETRIC                 1006
#define IDC_MM_HIMETRIC                 1007
#define IDC_MM_LOENGLISH                1008
#define IDC_MM_HIENGLISH                1009
#define IDC_MM_TWIPS                    1010
#define IDC_MM_LOGTWIPS                 1011
#define IDC_LF_ITALIC                   1012
#define IDC_LF_UNDER                    1013
#define IDC_LF_STRIKE                   1014
#define IDC_MATCH_ASPECT                1015
#define IDC_ADV_GRAPHICS                1016
#define IDC_LF_CHARSET                  1017
#define IDC_CHARSET_HELP                1018
#define IDC_DEFAULT_QUALITY             1019
#define IDC_DRAFT_QUALITY               1020
#define IDC_PROOF_QUALITY               1021
#define IDC_LF_FACENAME                 1022
#define IDC_OUT_DEFAULT                 1023
#define IDC_OUT_STRING                  1024
#define IDC_OUT_CHARACTER               1025
#define IDC_OUT_STROKE                  1026
#define IDC_OUT_TT                      1027
#define IDC_OUT_DEVICE                  1028
#define IDC_OUT_RASTER                  1029
#define IDC_OUT_TT_ONLY                 1030
#define IDC_OUT_OUTLINE                 1031
#define IDC_DEFAULT_PITCH               1032
#define IDC_FIXED_PITCH                 1033
#define IDC_VARIABLE_PITCH              1034
#define IDC_FF_DONTCARE                 1035
#define IDC_FF_ROMAN                    1036
#define IDC_FF_SWISS                    1037
#define IDC_FF_MODERN                   1038
#define IDC_FF_SCRIPT                   1039
#define IDC_FF_DECORATIVE               1040
#define IDC_TM_HEIGHT                   1041
```

```
#define IDC_TM_ASCENT                   1042
#define IDC_TM_DESCENT                  1043
#define IDC_TM_INTLEAD                  1044
#define IDC_TM_EXTLEAD                  1045
#define IDC_TM_AVECHAR                  1046
#define IDC_TM_MAXCHAR                  1047
#define IDC_TM_WEIGHT                   1048
#define IDC_TM_OVERHANG                 1049
#define IDC_TM_DIGASPX                  1050
#define IDC_TM_DIGASPY                  1051
#define IDC_TM_FIRSTCHAR                1052
#define IDC_TM_LASTCHAR                 1053
#define IDC_TM_DEFCHAR                  1054
#define IDC_TM_BREAKCHAR                1055
#define IDC_TM_ITALIC                   1056
#define IDC_TM_UNDER                    1057
#define IDC_TM_STRUCK                   1058
#define IDC_TM_VARIABLE                 1059
#define IDC_TM_VECTOR                   1060
#define IDC_TM_TRUETYPE                 1061
#define IDC_TM_DEVICE                   1062
#define IDC_TM_FAMILY                   1063
#define IDC_TM_CHARSET                  1064
#define IDC_TM_FACENAME                 1065
#define IDM_DEVICE_SCREEN               40001
#define IDM_DEVICE_PRINTER              40002
```

Figure 17-1.
The PICKFONT program.

Figure 14-2 shows a typical PICKFONT screen. The left side of the PICKFONT display is a modeless dialog box that allows you to select most of the fields of the logical font structure. The right side of the dialog box shows the results of GetTextMetrics after the font is selected in the device context. Below the dialog box, the program displays a string of characters using this font. Because the modeless dialog box is so big, youÆre best off running this program on a display size of 1024 by 768 or larger.

INCLUDEPICTURE "pickfont.gif" * MERGEFORMAT \d

Figure 17-2.
A typical PICKFONT display (Unicode version under Windows NT).

The modeless dialog box also contains some options that are not part of the logical font structure. These are the mapping mode (including my Logical Twips mode) the Match Aspect option, which changes the way Windows matches a logical font to a real font, and the ôAdv Grfx Mode,ö which sets the advanced graphics mode in Windows NT. IÆll discuss these in more detail shortly.

From the Device menu you can select the default printer rather than the video display. In this case, PICKFONT selects the logical font into the

printer device context and displays the TEXTMETRIC structure from the printer. The program then selects the logical font into the window device context for displaying the sample string. Thus, the text displayed by the program may use a different font (a screen font) than the font described by the list of the TEXTMETRIC fields (which is a printer font).

Much of the PICKFONT program contains the logical necessary to maintain the dialog box, so I wonÆt go into detail on the workings of the program. Instead, IÆll explain what youÆre doing when you create and select a logical font.

The Logical Font Structure

To create a logical font, you can call CreateFont, a function that has 14 arguments. Generally, itÆs easier to define a structure of type LOGFONT:

```
LOGFONT lf ;
```

and then define the fields of this structure. When finish, you call CreateFontIndirect with a pointer to the structure:

```
hFont = CreatFontIndirect (&lf) ;
```

You donÆt need to set each and every field of the LOGFONT structure. If your logical font structure is defined as a static variable, all the fields will be initialized to 0. The 0 values are generally defaults. You can then use that structure directly witout any changes, and CreateFontIndirect will return a handle to a font. When you select that font into the device context, youÆll get a reasonable default font. You can be as specific or as vague as you want in the LOGFONT structure, and the Windows will attempt to match your requests with a real font.

As I discuss each field of the LOGFONT structure, you may want to test them out using the PICKFONT program. Be sure to press Enter or the OK button when you want the program to use any fields youÆve entered.

[NOTE: The stuff that follows is adapted from the Windows 3.1 edition of the book, page 692 and following. However, I havenÆt tried to mimic the indented appearance of the text, specifically the paragraphs that are indented like the bulleted paragraphs but without bullets.]

The first two fields of the LOGFONT structure are in logical units, so they depend on the current setting of the mapping mode:

lfHeightùThis is the desired height of the characters (including internal leading but not external leading) in logical units. You can set lfHeight to 0 for a default size, or you can set it to a positive or negative value depending on what you want the field to represent. If you set lfHeight to a positive value, youÆre implying that you want this value to be a height that includes internal leading. In effect, youÆre really requesting a font that is appropriate for a line spacing of lfHeight. If you set lfHeight to a negative value, then Windows treats the absolute value of that number as a desired font height compatible with the point size. This is an important distinction: If you want a font of a particular point size, convert that point size to logical units, and set the lfHeight field to the negative of that value. If lfHeight is positive, then the tmHeight field of the resultant TEXTMETRIC structure will be roughly that value. (ItÆs sometimes a little off, probably due to rounding.) If lfHeight is negative, then it

will roughly match the tmHeight field of the TEXTMETRIC structure less the tmInternalLeading field.

lfWidthùThis is the desired width of the characters in logical units. In most cases, youÆll want to set this value to 0 and let Windows choose a font based solely on the height. Using a non-zero value does not work well with raster fonts, but with TrueType fonts, you can easily use this to get a font that has wider or slimmer characters than normal. This field corresponds to the tmAveCharWidth field of the TEXTMETRIC structure. To use the lfWidth field intelligently, first set up the LOGFONT structure with a lfWidth field set to zero, create the logical font, select it into a device context, and then call GetTextMetrics. Get the tmAveCharWidth field, adjust it up or down, probably by a percentage, and then create a second font using that adjusted tmAveCharWidth value for lfWidth.

The next two fields specify the ôescapementö and ôorientationö of the text. In theory, lfEscapement allows character strings to be written at an angle (but with the baseline of each character still parallel to the horizontal axis) and lfOrientation allows individual characters to be tilted. These fields have never quite worked as advertised, and even today they donÆt work as they should except in one case: YouÆre using a TrueType font, youÆre running Windows NT, and you call SetGraphicsMode with the CM_ADVANCED flag set. You can accomplish the final requirement in PICKFONT by checking the ôAdv Grfx Modeö checkbox.

To experiment with these fields in PICKFONT, be aware that the units are in tenths of a degree and indicate a counter-clockwise rotation. ItÆs easy to enter values that cause the sample text string to disappear! For this reason, use values between 0 and û600 (or so), or values between 3000 and 3600.

lfEscapementùThis is an angle in tenths of a degree, measured from the horizontal in a counterclockwise direction. It specifies how the successive characters of a string are placed when you write text. Here are some examples:

ValuePlacement of Characters0Run from left to right (default)900Go up1800Run from right to left2700Go downIn Windows 98, this value sets both the escapement and orientation of TrueType text. In Windows NT, this value also normally sets both the escapement and orientation of TrueType text, except when you call SetGraphicsMode with the GM_ADVANCED argument, in which case it works as documented.

lfOrientationùThis is an angle in tenths of a degree, measured from the horizontal in a counterclockwise direction. It affects the appearance of each individual character. Here are some examples:

ValueCharacter Appearance0Normal (default)900Tipped 90 degrees to the right1800Upside down2700Tipped 90 degrees to the leftThis field has no effect except with a TrueType font under Windows NT with the graphics mode set to GM_ADVANCED, in which case it works as documented.

The remaining 10 fields follow:

lfWeightùThis field allows you to specify boldface. The WINGDI.H header files defines a bunch of values to use with this field:

ValueIdentifier0FW_DONTCARE100FW_THIN200FW_EXTRALIGHT or
FW_ULTRALIGHT300FW_LIGHT400FW_NORMAL or
FW_REGULAR500FW_MEDIUM600FW_SEMIBOLD or
FW_DEMIBOLD700FW_BOLD800FW_EXTRABOLD or FW_ULTRABOLD900FW_HEAVY or
FW_BLACKIn reality, this table is much more ambitious than anything that
was ever implemented. You can use 0 or 400 for normal and 700 for bold.
lfItalicùWhen nonzero, this specifies italics. Windows can synthesize
italics on GDI raster fonts. That is, Windows simply shifts some rows of
the character bitmap to mimic italic. With TrueType fonts, Windows uses the
actual italic or oblique versions of the fonts.
lfUnderlineùWhen nonzero, this specifies underlining, which is always
synthesized on GDI fonts. That is, the Windows GDI simply draws a line
under each character, including spaces.
lfStrikeOutùWhen nonzero, this specifies that the font should have a line
drawn through the charactrers. This is also synthesized on GDI fonts.
lfCharSetùThis is a byte value that specifies the character set of the
font. IÆll have more to say about this field in the upcoming section
ôCharacter Sets and Unicode.ö In PICKFONT, you can press the button with
the question mark to obtain a list of the character set codes you can use.
Notice that the lfCharSet field is the only field where a zero does not
indicate a default value. A zero value is equivalent to ANSI_CHARSET, the
ANSI character set used in the United States and Western Europe. The
DEFAULT_CHARSET code (which equals 1) indicates the default character set
for the machine on which the program is running.
lfOutPrecisionùThis specifies how Windows should attempt to match the
desired font sizes and characteristics with actual fonts. ItÆs a rather
complex field that you probably wonÆt be using much. Check the
documentation of the LOGFONT structure for more detail. Note that you can
use the OUT_TT_ONLY_PRECIS flag to ensure that you always get a TrueType
font.
lfClipPrecisionùThis field specifies how characters are to be clipped when
they partially lie outside the clipping region. This field is not used much
and is not implemented in the PICKFONT program.
lfQualityùThis is an instruction to Windows regarding the matching of a
desired font with a real font. It really only has meaning with raster
fonts, and should not affect TrueType fonts. The DRAFT_QUALITY flag
indicates that you want GDI to scale raster fonts to achieve the size you
want; the PROOF_QUALITY flag indicates no scaling should be done. The
PROOF_QUALITY fonts are the most attractive, but they may be smaller than
what you request. YouÆll probably use DEFAULT_QUALITY (or 0) in this field.
 lfPitchAndFamilyùThis byte is composed of two parts. You can use the C
bitwise OR operator to combine two identifiers for this field. The lowest
two bits specify whether the font has a fixed pitch (all characters are the
same width) or a variable pitch:
ValueIdentifier0DEFAULT_PITCH1FIXED_PITCH2VARIABLE_PITCHThe upper half of
this byte specifies the font family:
ValueIdentifier0x00FW_DONTCARE0x10FF_ROMAN (variable widths,

serifs)0x20FF_SWISS (variable widths, no serifs)0x30FF_MODERN (fixed pitch)0x40FF_SCRIPT (mimics handwriting)0x50FF_DECORATIVElfFaceNameùThis is the actual text name of a typeface (such as Courier, Arial, or Times New Roman). This field is a byte array that is LF_FACESIZE (or 32 characters) wide. If you want a TrueType italic or boldface font, you can get it in one of two ways. You can use the complete typeface name (such as Times New Roman Italic) in the lfFaceName field, or you can use the base name (Times New Roman) and set the lfItalic field.

The Font-Mapping Algorithm

After you set up the logical font structure, you call CreateFontIndirect to get a handle to the logical font. When you call SelectObject to select that logical font into a device context, Windows finds the real font that most closely matches the request. In doing so, it uses a ôfont-mapping algorithm.ö Certain fields of the structure are more important than other fields.

The best way to get a feel for font mapping is to spend some time experimenting with PICKFONT. Here are some general guidelines:

The lfCharSet (character set) field is very important. It used to be that if you specified OEM_CHARSET (255), youÆd get either one of the stroke fonts or the Terminal font, because these are the only fonts that used the OEM character sets. However, with the advent of TrueType ôBig Fontsö (discussed earlier in this book on page 256), a single TrueType font can be mapped to different character sets, including the OEM character set. YouÆll need to use SYMBOL_CHARSET (2) to get the Symbol font or the Wingdings font.

A pitch value of FIXED_PITCH in the lfPitchAndFamily field is important, because you are in effect telling Windows that you donÆt want to deal with a variable-width font.

The lfFaceName field is important, because youÆre being specific about the typeface of the font that you want. If you leave lfFaceName set to NULL and set the family value in the lfPitchAndFamily field to a value other than FF_DONTCARE, that field becomes important because youÆre being specific about the font family.

For raster fonts, Windows will attempt to match the lfHeight value even if it needs to increase the size of a smaller font. The height of the actual font will always be less than or equal to that of the requested font unless there is no font small enough to satisfy your request. For stroke or TrueType fonts, Windows will simply scale the font to the desired height. You can prevent Windows from scaling a raster font by setting lfQuality to PROOF_QUALITY. By doing so, youÆre telling Windows that the requested height of the font is less important than the appearance of the font.

If you specify lfHeight and lfWeight values that are out of line for the particular aspect ratio of the display, Windows can map to a raster font that is designed for a display or other device of a different aspect ratio. This used to be a trick to get to get a thin or thick font. (This is not really necessary with TrueType, of course.) In general, youÆll probably want to avoid matching with a font for another device, which you can do in

PICKFONT by clicking the check box marked Match Aspect. If this box is checked, PICKFONT makes a call to SetMapperFlags with a TRUE argument.

Finding Out About the Font

At the right side of the modeless dialog box in PICKFONT is the information obtained from the GetTextMetrics function after the font has been selected in a device context. (Notice that you can use PICKFONTÆs device menu to indicate whether you want this device context to be the screen or the default printer. The results may be different because different fonts may be available on the printer.) At the very bottom of the list in PICKFONT is the typeface name available from GetTextFace.

All the size values that Windows copies into the TEXTMETRIC structure are in logical units except for the digitized aspect ratios. The fields of the TEXTMETRIC structure are as follows:

tmHeightùThe height of the character in logical units. This is the value that should approximate the lfHeight field specified in the LOGFONT structure, if that value was positive, in which case it respresents the line spacing of the font rather than the point size. If the lfHeight field of the LOGFONT structure was negative, the tmHeight field minus the tmInternalLeading field should approximate the absolute value of the lfHeight field.

tmAscentùThe vertical size of the character above the baseline in logical units.

tmDescentùThe vertical size of the character below the baseline in logical units.

tmInternalLeadingùA vertical size included in the tmHeight value that is usually occupied by diacritics on some capital letters. Once again, you can calculate the point size of the font by subtracting the tmInternalLeading value from the tmHeight value.

tmExternalLeadingùAn additional amount of line spacing beyond tmHeight recommended by the designer of the font for spacing successive lines of text.

tmAveCharWidthùThe average width of lowercase letters in the font.

tmMaxCharWidthùThe width of the widest charactrer in logical units. For a fixed-pitch font, this value is the same as tmAveCharWidth.

tmWeightùThe weight of the font ranging from 0 through 999. In reality, the field will be 400 or a normal font and 700 for a boldface font.

tmOverhangùThe amount of extra width (in logical units) that Windows adds to a raster font character when synthesizing italic or boldface. When a raster font is italicized, the tmAveCharWidth value remains unchanged, because a string of italicized characters has the same overall width as the same string of normal characters. For boldfacing, Windows must slightly expand the width of each character. For a boldface font, the tmAveCharWidth value less the tmOverhang value equals the tmAveCharWidth value for the same font without boldfacing.

tmDigitizedAspectX and tmDigitizedAspectYùThe aspect ratio for which the font is appropriate. These are equivalent to values obtained from GetDeviceCaps with the LOGPIXELSX and LOGPIXELSY identifiers.

tmFirstCharùThe character code of the first character in the font.
tmLastCharùThe character code of the last character in the font. If the TEXTMETRIC structure is obtained by a call to GetTextMetricsW (the wide character version of the function) then this value may be greater than 255.
tmDefaultCharùThe character code that Windows uses to display characters that are not in the font.
tmBreakCharùThe character that Windows (and your programs) should use to determine word breaks when justifying text. Unless youÆre using something bizarre (like an EBCDIC font), this will be 32ùthe space character.
tmItalicùNonzero for an italic font.
tmUnderlinedùNonzero for an underlined font.
tmStruckOutùNonzero for a strikethrough font.
tmPitchAndFamilyùThe four low-order bits are flags that indicate some characteristics about the font, indicated by the following identifiers defined in WINGDI.H:
ValueIdentifier0x01TMPF_FIXED_PITCH0x02TMPF_VECTOR0x04TMPF_TRUETYPE0x08TMPF_DEVICEDespite the name of the TMPF_FIXED_PITCH flag, the lowest bit is 1 if the font characters have a variable pitch. The second lowest bit (TMPF_VECTOR) will be 1 for TrueType fonts and for fonts that use other scalable outline technologies, such as PostScript. The TMPF_DEVICE flag indicates a device font (that is, a font built into a printer) rather than a GDI-based font.
The top four bits of this field indicates the font family, which are the same values used in the LOGFONT lfPitchAndFamily field.
tmCharSetùThe character set identifier.

Character Sets and Unicode

I discussed the concept of the Windows character set in Chapter 6, where we had to deal with international issues involving the keyboard. In the LOGFONT and TEXTMETRIC structures, the character set of the desired font (or the actual font) is indicated by a one-byte number between 0 and 255. The character set identifiers are defined in WINGDI.H like so:

```
#define ANSI_CHARSET            0
#define DEFAULT_CHARSET         1
#define SYMBOL_CHARSET          2
#define MAC_CHARSET             77
#define SHIFTJIS_CHARSET        128
#define HANGEUL_CHARSET         129
#define HANGUL_CHARSET          129
#define JOHAB_CHARSET           130
#define GB2312_CHARSET          134
#define CHINESEBIG5_CHARSET     136
#define GREEK_CHARSET           161
#define TURKISH_CHARSET         162
#define VIETNAMESE_CHARSET      163
#define HEBREW_CHARSET          177
#define ARABIC_CHARSET          178
#define BALTIC_CHARSET          186
```

```
#define RUSSIAN_CHARSET         204
#define THAI_CHARSET            222
#define EASTEUROPE_CHARSET      238
#define OEM_CHARSET             255
```

The character set is similar in concept to the codepage, but the character set is specific to Windows and is always less than or equal to 255.

As with all of the programs in this book, you can compile PICKFONT both with and without the UNICODE identifier defined. As usual, on the companion CD-ROM, the two versions of the program are located in the DEBUG and RELEASE directories, respectively.

Notice that the character string that PICKFONT displays towards the bottom of its window is longer in the Unicode version of the program. In both versions, the character string begins with the character codes 0x40 through 0x45 and 0x60 through 0x65. Regardless of the character set you choose (except for SYMBOL_CHARSET), these character codes will display as the first five upper and lower case letters of the Latin alphabet (A through E and a through e).

When running the non-Unicode version of the PICKFONT program, the next 12 charactersùthe character codes 0xC0 through 0xC5 and 0xE0 through 0xE5ùwill be dependent upon the character set you choose. For ANSI_CHARSET, these character codes correspond to accented versions of the uppercase and lowercase letter A. For GREEK_CHARSET, these codes will correspond to letters of the Greek alphabet. For RUSSIAN_CHARSET, they will be letters of the Cyrillic alphabet. Notice that the font may change when you select one of these character sets. This is because a raster font may not have these characters, but a TrueType font probably will. YouÆll recall that most TrueType fonts are ôbig fontsö and include characters for several different character sets. If youÆre running a Far Eastern version of Windows, these characters will be interpreted as double-byte characters and will display as ideographs rather than letters.

When running the Unicode version of PICKFONT under Windows NT, the codes 0xC0 through 0xC5 and 0xE0 through 0xE5 will always (except for SYMBOL_CHARSET) be accented versions of the uppercase and lowercase letter A because thatÆs how these codes are defined in Unicode. The program also displays character codes 0x0390 through 0x0395 and 0x03B0 through 0x03B5. These will always correspond to letters of the Greek alphabet because thatÆs how these codes are defined in Unicode. Similarly the program displays character codes 0x0410 through 0x0415 and 0x0430 through 0x0435, which always correspond to letters in the Cyrillic alphabet. However, notice that these characters may not be present in a default font. You may have to select the GREEK_CHARSET or RUSSIAN_CHARSET to get them. In this case, the character set ID in the LOGFONT structure doesnÆt change the actual character set; the character set is always Unicode. The character set ID instead indicates that characters from this character set are desired.

Now select HEBREW_CHARSET (code 177). The Hebrew alphabet is not included in WindowsÆ usual big fonts, so the operating system picks Lucida Sans

Unicode, as you can verify at the bottom right corner of the modeless dialog box.

PICKFONT also displays character codes 0x5000 through 0x5004, which correspond to a few of the many Chinese, Japanese, and Korean ideographs. YouÆll see these if youÆre running a Far Eastern version of Windows, or you can download a free Unicode font that is more extensive than Lucida Sans Unicode. This is the Bitstream CyberBit font, available at http://www.bitstream.com/products/world/cyberbits. (Just to give you an idea of the difference, Lucida Sans Unicode is about 300K. Bitstream CyberBit is about 13 megabytes.) If you have this font installed, Windows will select it if you want a character set not supported by Lucida Sans Unicode, such as SHIFTJIS_CHARSET (Japanese), HANGUL_CHARSET (Korean), JOHAB_CHARSET (Korean), GB2312_CHARSET (Simplified Chinese), or CHINESEBIG5_CHARSET (Traditional Chinese).

IÆll present a program that lets you view all the characters of a Unicode font later in this chapter.

The EZFONT System

The introduction of TrueTypeùand its basis in traditional typographyùhas provided Windows with a solid foundation for displaying text in its many varieties. However, some of the Windows font-selection functions are based on older technology, in which raster fonts on the screen had to approximate printer device fonts. In next next section IÆll discribe font enumeration, which lets a program obtain a list of all the fonts available on the video display or pinter. However, the ChooseFont dialog box (to be discussed shortly) largely eliminates the necessity for font enumeration by a program.

Because the standard TrueType fonts are available on every system, and because these fonts can be used for both the screen and the printer, itÆs not necessary for a program to enumerate fonts in order to select one, or to blindly request a certain font type that may need to be approximated. A program could simply and precisely select TrueType fonts that it knows to exist on the system (unless, of course, the user has deliberately deleted them). It really should be almost as simple as specifying the name of the font (probably one of the 13 names listed above) and its point size. I call this approach EZFONT ("easy font") and the two files you need are shown in Figure 17-3.

EZFONT.H

```
/*----------------------
   EZFONT.H header file
  ----------------------*/

HFONT EzCreateFont (HDC hdc, TCHAR * szFaceName, int iDeciPtHeight,
                    int iDeciPtWidth, int iAttributes, BOOL fLogRes) ;

#define EZ_ATTR_BOLD          1
#define EZ_ATTR_ITALIC        2
#define EZ_ATTR_UNDERLINE     4
```

```
#define EZ_ATTR_STRIKEOUT     8
EZFONT.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 ;
}
```

Figure 17-3.
The EZFONT files.

EZFONT.C has only one function, called EzCreateFont, which you can use like so:

```
hFont = EzCreateFont (hdc, szFaceName, iDeciPtHeight, iDeciPtWidth,
                      iAttributes, fLogRes) ;
```

The function returns a handle to a font. The font can be selected in the device context by calling SelectObject. You should then call GetTextMetrics or GetOutlineTextMetrics to determine the actual size of the font dimensions in logical coordinates. Before your program terminates, you should delete any created fonts by calling DeleteObject.

The szFaceName argument is any TrueType typeface name. The closer you stick to the standard fonts, the less chance there is that the font wonÆt exist

on the system.
The third argument indicates the desired point size, but it's specified in "decipoints," which are 1/10th of a point. Thus, if you want a point size of 12-1/2, use a value of 125.
Normally, the fourth argument should be set to zero or identical to the third argument. However, you can create a TrueType font with a wider or narrower size by setting this argument to something different. This is sometimes called the "em-width" of the font and describes the width of the font in points. Don't confuse this with the average width of the font characters or anything like that. Back in the early days of typography, a capital 'M' was as wide as it was high. So, the concept of an "em-square" came into being, and that's the origin of the em-width measurement. When the em-width equals the em-height (the point size of the font), the character widths are as the font designer intended. A smaller or wider em-width lets you create slimmer or wider characters.
The iAttributes argument can be set to one or more of the following values defined in EZFONT.H:

```
EZ_ATTR_BOLD
EZ_ATTR_ITALIC
EZ_ATTR_UNDERLINE
EZ_ATTR_STRIKEOUT
```

You could use EZ_ATTR_BOLD or EZ_ATTR_ITALIC or include the style as part of the complete TrueType typeface name.
Finally, you set the last argument to TRUE to base the visible font size on the "logical resolution" returned by the GetDeviceCaps function using the LOGPIXELSX and LOGPIXELSY arguments. Otherwise, the font size is based on the resolution as calculated from the HORZRES, HORZSIZE, VERTRES, and VERTSIZE values. This only makes a different for the video display under Windows NT.
The EzCreateFont function begins by making some adjustments that are only recognized by Windows NT. These are the calls to the SetGraphicsMode and ModifyWorldTransform functions, which have no effect in Windows 98. The Windows NT world transform should have the effect of modifying the visible size of the font, so the world transform is set to the defaultùno transformùbefore the font size is calculated.
EzCreateFont basically sets the fields of a LOGFONT structure and calls CreateFontIndirect, which returns a handle to the font. The big chore of the EzCreateFont function is to convert a point size to logical units for the lfHeight field of the LOGFONT structure. It turns out that the point size must be converted to device units (pixels) first, and then to logical units. To perform the first step, the function uses GetDeviceCaps. Getting from pixels to logical units would seem to involve a fairly simple call to the DPtoLP ("device point to logical point") function. But in order for the DPtoLP conversion to work correctly, the same mapping mode must be in effect when you later display text using the created font. This means that you should set your mapping mode before calling the EzCreateFont function. In most cases, you use only one mapping mode for drawing on a particular

area of the window, so this requirement should not be a problem.
The EZTEST program in Figure 17-4 tests out the EZFONT files, but not too rigourously. This program uses the EZTEST files shown above and also includes FONTDEMO files that are used in some later programs in this book.

EZTEST.C

```
/*----------------------------------------
   EZTEST.C -- Test of EZFONT
               (c) Charles Petzold, 1998
  ----------------------------------------*/

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

TCHAR szAppName [] = TEXT ("EZTest") ;
TCHAR szTitle   [] = TEXT ("EZTest: Test of EZFONT") ;

void PaintRoutine (HWND hwnd, HDC hdc, int cxArea, int cyArea)
{
     HFONT      hFont ;
     int        y, iPointSize ;
     LOGFONT    lf ;
     TCHAR      szBuffer [100] ;
     TEXTMETRIC tm ;

          // Set Logical Twips mapping mode

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

          // Try some fonts

     y = 0 ;

     for (iPointSize = 80 ; iPointSize <= 120 ; iPointSize++)
     {
          hFont = EzCreateFont (hdc, TEXT ("Times New Roman"),
                                iPointSize, 0, 0, TRUE) ;

          GetObject (hFont, sizeof (LOGFONT), &lf) ;

          SelectObject (hdc, hFont) ;
          GetTextMetrics (hdc, &tm) ;

          TextOut (hdc, 0, y, szBuffer,
               wsprintf (szBuffer,
```

```
                         TEXT ("Times New Roman font of %i.%i points, ")
                         TEXT ("lf.lfHeight = %i, tm.tmHeight = %i"),
                         iPointSize / 10, iPointSize % 10,
                         lf.lfHeight, tm.tmHeight)) ;

          DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
          y += tm.tmHeight ;
     }
}
```

FONTDEMO.C

```
/*--------------------------------------------
   FONTDEMO.C -- Font Demonstration Shell Program
                 (c) Charles Petzold, 1998
  --------------------------------------------*/

#include <windows.h>
#include "..\\EZTest\\EzFont.h"
#include "..\\EZTest\\resource.h"

extern  void     PaintRoutine (HWND, HDC, int, int) ;
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

HINSTANCE hInst ;

extern TCHAR szAppName [] ;
extern TCHAR szTitle [] ;

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

     hInst = hInstance ;

     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  = szResource ;
     wndclass.lpszClassName = szAppName ;
```

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

     hwnd = CreateWindow (szAppName, szTitle,
                          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 message, WPARAM wParam, LPARAM
lParam)
{
     static DOCINFO  di = { sizeof (DOCINFO), TEXT ("Font Demo:
Printing") } ;
     static int      cxClient, cyClient ;
     static PRINTDLG pd = { sizeof (PRINTDLG) } ;
     BOOL            fSuccess ;
     HDC             hdc, hdcPrn ;
     int             cxPage, cyPage ;
     PAINTSTRUCT     ps ;

     switch (message)
     {
     case WM_COMMAND:
          switch (wParam)
          {
          case IDM_PRINT:

                    // Get printer DC

               pd.hwndOwner = hwnd ;
```

```
               pd.Flags     = PD_RETURNDC | PD_NOPAGENUMS |
PD_NOSELECTION ;

               if (!PrintDlg (&pd))
                    return 0 ;

               if (NULL == (hdcPrn = pd.hDC))
               {
                    MessageBox (hwnd, TEXT ("Cannot obtain Printer DC"),
                                szAppName, MB_ICONEXCLAMATION | MB_OK) ;
                    return 0 ;
               }
                    // Get size of printable area of page

               cxPage = GetDeviceCaps (hdcPrn, HORZRES) ;
               cyPage = GetDeviceCaps (hdcPrn, VERTRES) ;

               fSuccess = FALSE ;

                    // Do the printer page

               SetCursor (LoadCursor (NULL, IDC_WAIT)) ;
               ShowCursor (TRUE) ;

               if ((StartDoc (hdcPrn, &di) > 0) && (StartPage (hdcPrn) >
0))
               {
                    PaintRoutine (hwnd, hdcPrn, cxPage, cyPage) ;

                    if (EndPage (hdcPrn) > 0)
                    {
                         fSuccess = TRUE ;
                         EndDoc (hdcPrn) ;
                    }
               }
               DeleteDC (hdcPrn) ;

               ShowCursor (FALSE) ;
               SetCursor (LoadCursor (NULL, IDC_ARROW)) ;

               if (!fSuccess)
                    MessageBox (hwnd,
                                TEXT ("Error encountered during printing"),
                                szAppName, MB_ICONEXCLAMATION | MB_OK) ;
               return 0 ;

          case IDM_ABOUT:
```

```
               MessageBox (hwnd, TEXT ("Font Demonstration Program\n")
                                 TEXT ("(c) Charles Petzold, 1998"),
                           szAppName, MB_ICONINFORMATION | MB_OK) ;
               return 0 ;
          }
          break ;

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

     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;

          PaintRoutine (hwnd, hdc, cxClient, cyClient) ;

          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

FONTDEMO.RC

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
//
// Menu

FONTDEMO MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&Print...",                   IDM_PRINT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About...",                   IDM_ABOUT
    END
END
```

RESOURCE.H

```
// Microsoft Developer Studio generated include file.
// Used by FontDemo.rc

#define IDM_PRINT                       40001
#define IDM_ABOUT                       40002
```

Figure 17-4.

The EZTEST program.

The PaintRoutine is EZTEST.C sets its mapping mode to Logical Twips, and then calls creates Times New Roman fonts with sizes ranging from 8 points to 12 points in 0.1 point intervals. The program output may be a little disturbing when you first run it. Many of the lines of text use a font that is obviously the same size, and indeed the tmHeight font on the TEXTMETRIC function reports these fonts as having the same height. WhatÆs happening here is a result of the rasterization process. The discrete pixels of the display canÆt allow for every possible size. However, the FONTDEMO shell program allows printing the output as well. Here youÆll find that the font sizes are more accurately differentiated.

Font Rotation

As you may have discovered by experimenting with PICKFONT, the lfOrientation and lfEscapement fields of the LOGFONT structure allow you to rotate TrueType text. If you think about it, this shouldnÆt be much of stretch for GDI. Formulas to rotate coordinate points around an origin are well known.

Although EzCreateFont does not allow you to specify a rotation angle for the font, itÆs fairly easy to make an adjustment after calling the function, as the FONTROT (ôFont Rotateö) program demonstrates. Figure 7-5 Shows the FONTROT.C file; the program also requires the EZFONT files and the FONTDEMO files shown earlier.

FONTROT.C

```
/*----------------------------------------
   FONTROT.C -- Rotated Fonts
                (c) Charles Petzold, 1998
  ----------------------------------------*/

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

TCHAR szAppName [] = TEXT ("FontRot") ;
TCHAR szTitle   [] = TEXT ("FontRot: Rotated Fonts") ;

void PaintRoutine (HWND hwnd, HDC hdc, int cxArea, int cyArea)
{
     static TCHAR szString [] = TEXT ("   Rotation") ;
     HFONT        hFont ;
     int          i ;
     LOGFONT      lf ;
```

```
     hFont = EzCreateFont (hdc, TEXT ("Times New Roman"), 540, 0, 0,
TRUE) ;
     GetObject (hFont, sizeof (LOGFONT), &lf) ;
     DeleteObject (hFont) ;

     SetBkMode (hdc, TRANSPARENT) ;
     SetTextAlign (hdc, TA_BASELINE) ;
     SetViewportOrgEx (hdc, cxArea / 2, cyArea / 2, NULL) ;

     for (i = 0 ; i < 12 ; i ++)
     {
          lf.lfEscapement = lf.lfOrientation = i * 300 ;
          SelectObject (hdc, CreateFontIndirect (&lf)) ;

          TextOut (hdc, 0, 0, szString, lstrlen (szString)) ;

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

Figure 7-5.
The FONTROT program.

FONTROT calls EzCreateFont just to obtain the LOGFONT structure associated with a 54-point Times New Roman font. The program then deletes that font. In the for loop, for each angle in 30 degree increments, a new font is created and the text is displayed. The results are shown in Figure 7-6.

INCLUDEPICTURE "FontRot.gif" * MERGEFORMAT \d

Figure 17-6.
The FONTROT display.

If youÆre interested in a more-generalized approach to graphics rotation and other linear transformation, and you know that your programs will be restricted to running under Windows NT, you can use the XFORM matric and the world transform functions.

Font Enumeration

Font enumeration is the process of obtaining from GDI a list of all fonts available on a device. A program can then select one of these fonts or display them in a dialog box for selection by the user. IÆll first briefly describe the enumeration functions, and then show how to use the ChooseFont function, which fortunately makes font enumeration much less necessary for an application.

The Enumeration Functions

Back in the old days of Windows, font enumeration required use of the EnumFonts function:

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

A program could enumerate all fonts (by setting the second argument to NULL) or just those of a particular typeface. The third argument is an enumeration callback function; the fourth argument is optional data passed

to that function. GDI calls the callback function once for each font in the system, passing to it both LOGFONT and TEXTMETRIC structures that defined the font, plus some flags indicating the type of font.

The EnumFontFamilies function was designed to better enumerate TrueType fonts under Windows 3.1:

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

Generally, EnumFontFamilies is called first with a NULL second argument. The EnumProc callback function is called once for each font family (such as Times New Roman). Then the application calls EnumFontFamilies again with that typeface name and a different callback function. GDI calls the second callback function for each font in the family (such as Times New Roman Italic). The callback function is passed an ENUMLOGFONT structure (which is a LOGFONT structure plus a ôfull nameö field and a ôstyleö field containing, for example, the text name ôItalicö or ôBoldö) and a TEXTMETRIC structure for non-TrueType fonts and a NEWTEXTMETRIC structure for TrueType fonts. The NEWTEXTMETRIC structure adds four fields to the information in the TEXTMETRIC structure.

The EnumFontFamiliesEx function is recommended for applications running under the 32-bit versions of Windows:

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

The second argument is a pointer to a LOGFONT structure for which the lfCharSet and lfFaceName fields indicate what fonts are to be enumerated. The callback function gets information about each font in the form of ENUMLOGFONTEX and NEWTEXTMETRICEX structures.

The ChooseFont Dialog

We had a little introduction to the ChooseFont common dialog box back in Chapter 11. Now that weÆve encountered font enumeration, the inner workings of the ChooseFont function should be obvious. The ChooseFont function takes a pointer to a CHOOSEFONT structure as its only argument and displays a dialog box listing all the fonts. On return from ChooseFont, a LOGFONT structure (which is part of the CHOOSEFONT structure) lets you create a logical font.

The CHOSFONT program, shown in Figure 17-7, demonstrates using the ChooseFont function and displays the fields of the LOGFONT structure that the function defines. The program also displays the same string of text as PICKFONT.

CHOSFONT.C

```
/*----------------------------------------
   CHOSFONT.C -- ChooseFont Demo
                 (c) Charles Petzold, 1998
  ----------------------------------------*/

#include <windows.h>
#include "resource.h"

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

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

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

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

     hwnd = CreateWindow (szAppName, TEXT ("ChooseFont"),
                          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 message, WPARAM wParam, LPARAM
lParam)
{
     static CHOOSEFONT cf ;
```

```
     static int        cyChar ;
     static LOGFONT    lf ;
     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 ;
     int               y ;
     PAINTSTRUCT       ps ;
     TCHAR             szBuffer [64] ;
     TEXTMETRIC        tm ;

     switch (message)
     {
     case WM_CREATE:

               // Get text height

          cyChar = HIWORD (GetDialogBaseUnits ()) ;

               // Initialize the LOGFONT structure

          GetObject (GetStockObject (SYSTEM_FONT), sizeof (lf), &lf) ;

               // Inialize the CHOOSEFONT structure

          cf.lStructSize    = sizeof (CHOOSEFONT) ;
          cf.hwndOwner      = hwnd ;
          cf.hDC            = NULL ;
          cf.lpLogFont      = &lf ;
          cf.iPointSize     = 0 ;
          cf.Flags          = CF_INITTOLOGFONTSTRUCT |
```

```
                         CF_SCREENFONTS | CF_EFFECTS ;
          cf.rgbColors      = 0 ;
          cf.lCustData      = 0 ;
          cf.lpfnHook       = NULL ;
          cf.lpTemplateName = NULL ;
          cf.hInstance      = NULL ;
          cf.lpszStyle      = NULL ;
          cf.nFontType      = 0 ;
          cf.nSizeMin       = 0 ;
          cf.nSizeMax       = 0 ;
          return 0 ;

     case WM_COMMAND:
          switch (LOWORD (wParam))
          {
          case IDM_FONT:
               if (ChooseFont (&cf))
                    InvalidateRect (hwnd, NULL, TRUE) ;
               return 0 ;
          }
          return 0 ;

     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;

               // Display sample text using selected font

          SelectObject (hdc, CreateFontIndirect (&lf)) ;
          GetTextMetrics (hdc, &tm) ;
          SetTextColor (hdc, cf.rgbColors) ;
          TextOut (hdc, 0, y = tm.tmExternalLeading, szText, lstrlen
(szText)) ;

               // Display LOGFONT structure fields using system font

          DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT))) ;
          SetTextColor (hdc, 0) ;

          TextOut (hdc, 0, y += tm.tmHeight, szBuffer,
               wsprintf (szBuffer, TEXT ("lfHeight = %i"), lf.lfHeight)) ;

          TextOut (hdc, 0, y += cyChar, szBuffer,
               wsprintf (szBuffer, TEXT ("lfWidth = %i"), lf.lfWidth)) ;

          TextOut (hdc, 0, y += cyChar, szBuffer,
               wsprintf (szBuffer, TEXT ("lfEscapement = %i"),
                         lf.lfEscapement)) ;
```

```
          TextOut (hdc, 0, y += cyChar, szBuffer,
               wsprintf (szBuffer, TEXT ("lfOrientation = %i"),
                         lf.lfOrientation)) ;

          TextOut (hdc, 0, y += cyChar, szBuffer,
               wsprintf (szBuffer, TEXT ("lfWeight = %i"), lf.lfWeight)) ;

          TextOut (hdc, 0, y += cyChar, szBuffer,
               wsprintf (szBuffer, TEXT ("lfItalic = %i"), lf.lfItalic)) ;

          TextOut (hdc, 0, y += cyChar, szBuffer,
               wsprintf (szBuffer, TEXT ("lfUnderline = %i"),
lf.lfUnderline)) ;

          TextOut (hdc, 0, y += cyChar, szBuffer,
               wsprintf (szBuffer, TEXT ("lfStrikeOut = %i"),
lf.lfStrikeOut)) ;

          TextOut (hdc, 0, y += cyChar, szBuffer,
               wsprintf (szBuffer, TEXT ("lfCharSet = %i"),
lf.lfCharSet)) ;

          TextOut (hdc, 0, y += cyChar, szBuffer,
               wsprintf (szBuffer, TEXT ("lfOutPrecision = %i"),
                         lf.lfOutPrecision)) ;

          TextOut (hdc, 0, y += cyChar, szBuffer,
               wsprintf (szBuffer, TEXT ("lfClipPrecision = %i"),
                         lf.lfClipPrecision)) ;

          TextOut (hdc, 0, y += cyChar, szBuffer,
               wsprintf (szBuffer, TEXT ("lfQuality = %i"),
lf.lfQuality)) ;

          TextOut (hdc, 0, y += cyChar, szBuffer,
               wsprintf (szBuffer, TEXT ("lfPitchAndFamily = 0x%02X"),
                         lf.lfPitchAndFamily)) ;

          TextOut (hdc, 0, y += cyChar, szBuffer,
               wsprintf (szBuffer, TEXT ("lfFaceName = %s"),
lf.lfFaceName)) ;

          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY:
```

```
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

CHOSFONT.RC

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
//
// Menu

CHOSFONT MENU DISCARDABLE
BEGIN
    MENUITEM "&Font!",                      IDM_FONT
END
```

RESOURCE.H

```
// Microsoft Developer Studio generated include file.
// Used by ChosFont.rc

#define IDM_FONT                        40001
```

Figure 17-7.
The CHOSFONT program.

As usual with the common dialog boxes, a Flags field in the CHOOSEFONT structure lets you pick lots of options. The CF_INITLOGFONTSTRUCT flag that CHOSFONT specifies causes Windows to initialize the dialog box selection based on the LOGFONT structure passed to the ChooseFont structure. You can use flags to specify TrueType fonts only (CF_TTONLY) or fixed-pitch fonts only (CF_FIXEDPITCHONLY) or no symbol fonts (CF_SCRIPTSONLY). You can display screen font (CF_SCREENFONTS), printer fonts (CF_PRINTERFONTS) or both (CF_BOTH). In the latter two cases, the hDC field of the CHOOSEFONT structure must reference a printer device context. The CHOSFONT program uses the CF_SCREENFONTS flag.

The CF_EFFECTS flag (the third flag that the CHOSFONT program uses) forces the dialog box to include check boxes for underlining and strikeout, and also allows the selection of a text color. ItÆs not hard to implement text color in your code, so try it.

Notice the Script field in the Font dialog displayed by ChooseFont. This lets the user select a character set; the appropriate character set ID is returned in the LOGFONT structure.

The ChooseFont function uses the logical inch to calculate the lfHeight field from the point size. For example, suppose you have Small Fonts installed from the Display Properties dialog. That means that GetDeviceCaps with a video display device context and the argument LOGPIXELSY returns 96.

If you use ChooseFont to choose a 72-point Times Roman Font, you really want a 1-inch tall font. When ChooseFont returns, the lfHeight field of the LOGFONT structure will equal û96 (note the minus sign), meaning that the point size of the font is equivalent to should 96 pixels, or one logical inch.

Good. ThatÆs probably what we want. But keep the following in mind:

If you set one of the metric mapping modes under Windows NT, logical coordinates will be inconsistent with the physical size of the font. For example, if you draw a ruler next to the text based on a metric mapping modes, it will be not match the font. You should use the Logical Twips mapping mode described above to draw graphics that are consistent with the font size.

If youÆre going to be using any non-MM_TEXT mapping mode, make sure the mapping mode is not set when you select the font into the device context and display the text. Otherwise, GDI will interpret the lfHeight field of the LOGFONT structure as being expressed in logical coordinates.

The lfHeight field of the LOGFONT structure set by ChooseFont is always in pixels and it is only appropriate for the video display. When you create a font for a printer device context, you must adjust the lfHeight value. The ChooseFont function uses the hDC field of the CHOOSEFONT structure only for obtaining printer fonts to be listed in the dialog box. This device context handle does not affect the value of lfHeight.

Fortunately, the CHOOSEFONT structure includes an iPointSize field that provides the size of the selected font in units of 1/10 of a point. Regardless of the device context and mapping mode, you can always convert this field to a logical size and use that for the lfHeight field. The appropriate code may be found in the EZFONT.C file. You can probably simplify it based on your needs.

Another program that uses ChooseFont is UNICHARS, shown in Figure 17-8. This program lets you study all the characters of a font, and is particularly useful for studying the Lucida Sans Unicode (which it uses by default for display) or the Bitstream CyberBit font. UNICHARS always uses the TextOutW function for displaying the font characters, so you can run it under Windows NT or Windows 98.

UNICHARS.C

```
/*-----------------------------------------------
   UNICHARS.C -- Displays 16-bit character codes
                 (c) Charles Petzold, 1998
  ----------------------------------------------*/

#include <windows.h>
#include "resource.h"

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

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
```

```
{
     static TCHAR szAppName[] = TEXT ("UniChars") ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS     wndclass ;

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

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

     hwnd = CreateWindow (szAppName, TEXT ("Unicode Characters"),
                          WS_OVERLAPPEDWINDOW | WS_VSCROLL,
                          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 message, WPARAM wParam, LPARAM
lParam)
{
     static CHOOSEFONT cf ;
     static int        iPage ;
     static LOGFONT    lf ;
```

```
     HDC               hdc ;
     int               cxChar, cyChar, x, y, i, cxLabels ;
     PAINTSTRUCT       ps ;
     SIZE              size ;
     TCHAR             szBuffer [8] ;
     TEXTMETRIC        tm ;
     WCHAR             ch ;

     switch (message)
     {
     case WM_CREATE:
          hdc = GetDC (hwnd) ;
          lf.lfHeight = - GetDeviceCaps (hdc, LOGPIXELSY) / 6 ;  // 12
points
          lstrcpy (lf.lfFaceName, TEXT ("Lucida Sans Unicode")) ;
          ReleaseDC (hwnd, hdc) ;

          cf.lStructSize = sizeof (CHOOSEFONT) ;
          cf.hwndOwner   = hwnd ;
          cf.lpLogFont   = &lf ;
          cf.Flags       = CF_INITTOLOGFONTSTRUCT | CF_SCREENFONTS ;

          SetScrollRange (hwnd, SB_VERT, 0, 255, FALSE) ;
          SetScrollPos   (hwnd, SB_VERT, iPage,  TRUE ) ;
          return 0 ;

     case WM_COMMAND:
          switch (LOWORD (wParam))
          {
          case IDM_FONT:
               if (ChooseFont (&cf))
                    InvalidateRect (hwnd, NULL, TRUE) ;
               return 0 ;
          }
          return 0 ;

     case WM_VSCROLL:
          switch (LOWORD (wParam))
               {
               case SB_LINEUP:         iPage -=  1 ;  break ;
               case SB_LINEDOWN:       iPage +=  1 ;  break ;
               case SB_PAGEUP:         iPage -= 16 ;  break ;
               case SB_PAGEDOWN:       iPage += 16 ;  break ;
               case SB_THUMBPOSITION:  iPage = HIWORD (wParam) ;  break ;

               default:
                    return 0 ;
```

```
          }

     iPage = max (0, min (iPage, 255)) ;

     SetScrollPos (hwnd, SB_VERT, iPage, TRUE) ;
     InvalidateRect (hwnd, NULL, TRUE) ;
     return 0 ;

case WM_PAINT:
     hdc = BeginPaint (hwnd, &ps) ;

     SelectObject (hdc, CreateFontIndirect (&lf)) ;

     GetTextMetrics (hdc, &tm) ;
     cxChar = tm.tmMaxCharWidth ;
     cyChar = tm.tmHeight + tm.tmExternalLeading ;

     cxLabels = 0 ;

     for (i = 0 ; i < 16 ; i++)
     {
          wsprintf (szBuffer, TEXT (" 000%1X: "), i) ;
          GetTextExtentPoint (hdc, szBuffer, 7, &size) ;

          cxLabels = max (cxLabels, size.cx) ;
     }

     for (y = 0 ; y < 16 ; y++)
     {
          wsprintf (szBuffer, TEXT (" %03X_: "), 16 * iPage + y) ;
          TextOut (hdc, 0, y * cyChar, szBuffer, 7) ;

          for (x = 0 ; x < 16 ; x++)
          {
               ch = (WCHAR) (256 * iPage + 16 * y + x) ;
               TextOutW (hdc, x * cxChar + cxLabels,
                              y * cyChar, &ch, 1);
          }
     }

     DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT)));
     EndPaint (hwnd, &ps) ;
     return 0 ;

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

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

UNICHARS.RC

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

//////////////////////////////////////////////////////////////////////
//
// Menu

UNICHARS MENU DISCARDABLE
BEGIN
    MENUITEM "&Font!",                      IDM_FONT
END
```

RESOURCE.H

```
// Microsoft Developer Studio generated include file.
// Used by Unichars.rc

#define IDM_FONT                        40001
```

Figure 17-8.
The UNICHARS program.

Paragraph Formatting

Equipped with the ability to select and create logical fonts, it's time to try out hand at text formatting. The process involves placing each line of text within margins in one of four ways: aligned on the left margin, aligned on the right margin, centered between the margins, or justifiedùthat is, running from one margin to the other, with equal spaces between the words. For the first three jobs, you can use the DrawText function with the DT_WORDBREAK argument, but this approach has limitations. For instance, you can't determine what part of the text DrawText was able to fit within the rectangle. DrawText is convenient for some simple jobs, but for more complex formatting tasks, you'll probably want to employ TextOut.

Simple Text Formatting

One of the most useful functions for working with text is GetTextExtentPoint32. (This is a function whose name reveals some changes since the early versions of Windows.) The function tells you the width and height of a character string based on the current font selected in the device context:

```
GetTextExtentPoint32 (hdc, pString, iCount, &size) ;
```

The width and height of the text in logical units are returned in the cx and cy fields of the SIZE structure. I'll begin with an example using one line of text. Let's say that you have selected a font into your device context and now want to write the text:

```
TCHAR * szText [] = TEXT ("Hello, how are you?") ;
```
You want the text to start at the vertical coordinate yStart, within margins set by the coordinates xLeft and xRight. Your job is to calculate the xStart value for the horizontal coordinate where the text begins. This job would be considerably easier if the text were displayed using a fixed-pitch font, but that's not the general case. First, you get the text extents of the string:
```
GetTextExtentPoint32 (hdc, szText, lstrlen (szText), &size) ;
```
If size.cx is larger than (xRight - xLeft), then the line is too long to fit within the margins. Let's assume it can fit.
To align the text on the left margin, you simply set xStart equal to xLeft and then write the text:
```
TextOut (hdc, xStart, yStart, szText, lstrlen (szText)) ;
```
This is easy. You can now add the size.cy to yStart, and you're ready to write the next line of text.
To align the text on the right margin, you use this formula for xStart:
```
xStart = xRight - size.cx ;
```
To center the text between the left and right margins, use this formula:
```
xStart = (xLeft + xRight - size.cx) / 2 ;
```
Now here's the tough jobùto justify the text within the left and right margins. The distance between the margins is (xRight - xLeft). Without justification, the text is size.cx wide. The difference between these two values, which is:
```
xRight - xLeft - size.cx
```
must be equally distributed among the three space characters in the character string. It sounds like a terrible job, but it's not too bad. To do it, you call:
```
SetTextJustification (hdc, xRight - xLeft - size.cx, 3)
```
The second argument is the amount of space that must be distributed among the space characters in the character string. The third argument is the number of space characters, in this case 3. Now set xStart equal to xLeft and write the text with TextOut:
```
TextOut (hdc, xStart, yStart, szText, lstrlen (szText)) ;
```
The text will be justified between the xLeft and xRight margins.
Whenever you call SetTextJustification, it accumulates an error term if the amount of space doesn't distribute evenly among the space characters. This error term will affect subsequent GetTextExtentPoint32 calls. Each time you start a new line, you should clear out the error term by calling:
```
SetTextJustification (hdc, 0, 0) ;
```
Working with Paragraphs
If you're working with a whole paragraph, you have to start at the beginning and scan through the string looking for space characters. Every time you encounter a space character (or another character that can be used to break the line), you call GetTextExtentPoint32 to determine if the text still fits between the left and right margins. When the text exceeds the space allowed for it, you backtrack to the previous blank. Now you have determined the character string for the line. If you want to justify the

line, call SetTextJustification and TextOut, clear out the error term, and proceed to the next line.

The JUSTIFY1 program, shown in Figure 7-9, does this job for the first paragraph of Mark TwainÆs The Adventures of Huckleberry Finn. You can pick the font you want from a dialog box and you can also use a menu selection to change the alignment (left, right, centered, or justified). Figure 7-10 shows a typeical JUSTIFY1 display.

JUSTIFY1.C

```
/*----------------------------------------
   JUSTIFY1.C -- Justified Type Program #1
                 (c) Charles Petzold, 1998
  ----------------------------------------*/

#include <windows.h>
#include "resource.h"

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

TCHAR szAppName[] = TEXT ("Justify1") ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     HWND     hwnd ;
     MSG      msg ;
     WNDCLASS wndclass ;

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

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

     hwnd = CreateWindow (szAppName, TEXT ("Justified Type #1"),
                          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 ;
}

void DrawRuler (HDC hdc, RECT * prc)
{
     static int iRuleSize [16] = { 360, 72, 144, 72, 216, 72, 144, 72,
                                   288, 72, 144, 72, 216, 72, 144, 72 } ;
     int        i, j ;
     POINT      ptClient ;

     SaveDC (hdc) ;

          // Set Logical Twips mapping mode

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

          // Move the origin to a half inch from upper left

     SetWindowOrgEx (hdc, -720, -720, NULL) ;

          // Find the right margin (quarter inch from right)

     ptClient.x = prc->right ;
     ptClient.y = prc->bottom ;
     DPtoLP (hdc, &ptClient, 1) ;
     ptClient.x -= 360 ;

          // Draw the rulers

     MoveToEx (hdc, 0,          -360, NULL) ;
     LineTo   (hdc, ptClient.x, -360) ;
     MoveToEx (hdc, -360,          0, NULL) ;
```

```
     LineTo   (hdc, -360, ptClient.y) ;

     for (i = 0, j = 0 ; i <= ptClient.x ; i += 1440 / 16, j++)
     {
          MoveToEx (hdc, i, -360, NULL) ;
          LineTo   (hdc, i, -360 - iRuleSize [j % 16]) ;
     }

     for (i = 0, j = 0 ; i <= ptClient.y ; i += 1440 / 16, j++)
     {
          MoveToEx (hdc, -360, i, NULL) ;
          LineTo   (hdc, -360 - iRuleSize [j % 16], i) ;
     }

     RestoreDC (hdc, -1) ;
}

void Justify (HDC hdc, PTSTR pText, RECT * prc, int iAlign)
{
     int   xStart, yStart, cSpaceChars ;
     PTSTR pBegin, pEnd ;
     SIZE  size ;

     yStart = prc->top ;
     do                              // for each text line
     {
          cSpaceChars = 0 ;          // initialize number of spaces in line

          while (*pText == ' ')      // skip over leading spaces
               pText++ ;

          pBegin = pText ;           // set pointer to char at beginning of
line

          do                         // until the line is known
          {
               pEnd = pText ;        // set pointer to char at end of line

                    // skip to next space

               while (*pText != '\0' && *pText++ != ' ') ;

               if (*pText == '\0')
                    break ;

                    // after each space encountered, calculate extents
```

```
            cSpaceChars++ ;
            GetTextExtentPoint32(hdc, pBegin, pText - pBegin - 1, &size)
;
        }
        while (size.cx < (prc->right - prc->left)) ;

        cSpaceChars-- ;               // discount last space at end of
line

        while (*(pEnd - 1) == ' ')    // eliminate trailing spaces
        {
            pEnd-- ;
            cSpaceChars-- ;
        }

            // if end of text and no space characters, set pEnd to end

        if (*pText == '\0' || cSpaceChars <= 0)
            pEnd = pText ;

        GetTextExtentPoint32 (hdc, pBegin, pEnd - pBegin, &size) ;

        switch (iAlign)               // use alignment for xStart
        {
        case IDM_ALIGN_LEFT:
            xStart = prc->left ;
            break ;

        case IDM_ALIGN_RIGHT:
            xStart = prc->right - size.cx ;
            break ;

        case IDM_ALIGN_CENTER:
            xStart = (prc->right + prc->left - size.cx) / 2 ;
            break ;

        case IDM_ALIGN_JUSTIFIED:
            if (*pText != '\0' && cSpaceChars > 0)
                 SetTextJustification (hdc,
                                       prc->right - prc->left - size.cx,
                                       cSpaceChars) ;
            xStart = prc->left ;
            break ;
        }
            // display the text

        TextOut (hdc, xStart, yStart, pBegin, pEnd - pBegin) ;
```

```
               // prepare for next line

          SetTextJustification (hdc, 0, 0) ;
          yStart += size.cy ;
          pText = pEnd ;
     }
     while (*pText && yStart < prc->bottom - size.cy) ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
     static CHOOSEFONT cf ;
     static DOCINFO    di = { sizeof (DOCINFO), TEXT ("Justify1: Printing")
} ;
     static int        iAlign = IDM_ALIGN_LEFT ;
     static LOGFONT    lf ;
     static PRINTDLG   pd ;
     static TCHAR      szText[] = {
                              TEXT ("You don't know about me, without you
")
                              TEXT ("have read a book by the name of \"The
")
                              TEXT ("Adventures of Tom Sawyer,\" but that
")
                              TEXT ("ain't no matter. That book was made by
")
                              TEXT ("Mr. Mark Twain, and he told the truth,
")
                              TEXT ("mainly. There was things which he ")
                              TEXT ("stretched, but mainly he told the
truth. ")
                              TEXT ("That is nothing. I never seen anybody
")
                              TEXT ("but lied, one time or another, without
")
                              TEXT ("it was Aunt Polly, or the widow, or ")
                              TEXT ("maybe Mary. Aunt Polly -- Tom's Aunt
")
                              TEXT ("Polly, she is -- and Mary, and the
Widow ")
                              TEXT ("Douglas, is all told about in that
book ")
                              TEXT ("-- which is mostly a true book; with
")
                              TEXT ("some stretchers, as I said before.") }
```

```
;
     BOOL               fSuccess ;
     HDC                hdc, hdcPrn ;
     HMENU              hMenu ;
     int                iSavePointSize ;
     PAINTSTRUCT        ps ;
     RECT               rect ;

     switch (message)
     {
     case WM_CREATE:
               // Initialize the CHOOSEFONT structure

          GetObject (GetStockObject (SYSTEM_FONT), sizeof (lf), &lf) ;

          cf.lStructSize    = sizeof (CHOOSEFONT) ;
          cf.hwndOwner      = hwnd ;
          cf.hDC            = NULL ;
          cf.lpLogFont      = &lf ;
          cf.iPointSize     = 0 ;
          cf.Flags          = CF_INITTOLOGFONTSTRUCT | CF_SCREENFONTS |
                              CF_EFFECTS ;
          cf.rgbColors      = 0 ;
          cf.lCustData      = 0 ;
          cf.lpfnHook       = NULL ;
          cf.lpTemplateName = NULL ;
          cf.hInstance      = NULL ;
          cf.lpszStyle      = NULL ;
          cf.nFontType      = 0 ;
          cf.nSizeMin       = 0 ;
          cf.nSizeMax       = 0 ;

          return 0 ;

     case WM_COMMAND:
          hMenu = GetMenu (hwnd) ;

          switch (LOWORD (wParam))
          {
          case IDM_FILE_PRINT:
                                   // Get printer DC

               pd.lStructSize = sizeof (PRINTDLG) ;
               pd.hwndOwner   = hwnd ;
               pd.Flags       = PD_RETURNDC | PD_NOPAGENUMS |
PD_NOSELECTION ;
```

```
          if (!PrintDlg (&pd))
               return 0 ;

          if (NULL == (hdcPrn = pd.hDC))
          {
               MessageBox (hwnd, TEXT ("Cannot obtain Printer DC"),
                           szAppName, MB_ICONEXCLAMATION | MB_OK) ;
               return 0 ;
          }
               // Set margins of 1 inch

          rect.left   = GetDeviceCaps (hdcPrn, LOGPIXELSX) -
                        GetDeviceCaps (hdcPrn, PHYSICALOFFSETX) ;

          rect.top    = GetDeviceCaps (hdcPrn, LOGPIXELSY) -
                        GetDeviceCaps (hdcPrn, PHYSICALOFFSETY) ;

          rect.right  = GetDeviceCaps (hdcPrn, PHYSICALWIDTH) -
                        GetDeviceCaps (hdcPrn, LOGPIXELSX) -
                        GetDeviceCaps (hdcPrn, PHYSICALOFFSETX) ;

          rect.bottom = GetDeviceCaps (hdcPrn, PHYSICALHEIGHT) -
                        GetDeviceCaps (hdcPrn, LOGPIXELSY) -
                        GetDeviceCaps (hdcPrn, PHYSICALOFFSETY) ;

               // Display text on printer

          SetCursor (LoadCursor (NULL, IDC_WAIT)) ;
          ShowCursor (TRUE) ;

          fSuccess = FALSE ;

          if ((StartDoc (hdcPrn, &di) > 0) && (StartPage (hdcPrn) >
0))
          {
                    // Select font using adjusted lfHeight

               iSavePointSize = lf.lfHeight ;
               lf.lfHeight = -(GetDeviceCaps (hdcPrn, LOGPIXELSY) *
                                    cf.iPointSize) / 720 ;

               SelectObject (hdcPrn, CreateFontIndirect (&lf)) ;
               lf.lfHeight = iSavePointSize ;

                    // Set text color

               SetTextColor (hdcPrn, cf.rgbColors) ;
```

```
                    // Display text

               Justify (hdcPrn, szText, &rect, iAlign) ;

               if (EndPage (hdcPrn) > 0)
               {
                    fSuccess = TRUE ;
                    EndDoc (hdcPrn) ;
               }
          }
          ShowCursor (FALSE) ;
          SetCursor (LoadCursor (NULL, IDC_ARROW)) ;

          DeleteDC (hdcPrn) ;

          if (!fSuccess)
               MessageBox (hwnd, TEXT ("Could not print text"),
                           szAppName, MB_ICONEXCLAMATION | MB_OK) ;
          return 0 ;

     case IDM_FONT:
          if (ChooseFont (&cf))
               InvalidateRect (hwnd, NULL, TRUE) ;
          return 0 ;

     case IDM_ALIGN_LEFT:
     case IDM_ALIGN_RIGHT:
     case IDM_ALIGN_CENTER:
     case IDM_ALIGN_JUSTIFIED:
          CheckMenuItem (hMenu, iAlign, MF_UNCHECKED) ;
          iAlign = LOWORD (wParam) ;
          CheckMenuItem (hMenu, iAlign, MF_CHECKED) ;
          InvalidateRect (hwnd, NULL, TRUE) ;
          return 0 ;
     }
     return 0 ;

case WM_PAINT:
     hdc = BeginPaint (hwnd, &ps) ;

     GetClientRect (hwnd, &rect) ;
     DrawRuler (hdc, &rect) ;

     rect.left  += GetDeviceCaps (hdc, LOGPIXELSX) / 2 ;
     rect.top   += GetDeviceCaps (hdc, LOGPIXELSY) / 2 ;
     rect.right -= GetDeviceCaps (hdc, LOGPIXELSX) / 4 ;
```

```
          SelectObject (hdc, CreateFontIndirect (&lf)) ;
          SetTextColor (hdc, cf.rgbColors) ;

          Justify (hdc, szText, &rect, iAlign) ;

          DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT)));
          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY:
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

JUSTIFY1.RC

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
//
// Menu

JUSTIFY1 MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&Print",                      IDM_FILE_PRINT
    END
    POPUP "&Font"
    BEGIN
        MENUITEM "&Font...",                    IDM_FONT
    END
    POPUP "&Align"
    BEGIN
        MENUITEM "&Left",                       IDM_ALIGN_LEFT, CHECKED
        MENUITEM "&Right",                      IDM_ALIGN_RIGHT
        MENUITEM "&Centered",                   IDM_ALIGN_CENTER
        MENUITEM "&Justified",                  IDM_ALIGN_JUSTIFIED
    END
END
```

RESOURCE.H

```
// Microsoft Developer Studio generated include file.
// Used by Justify1.rc
```

```
#define IDM_FILE_PRINT                  40001
#define IDM_FONT                        40002
#define IDM_ALIGN_LEFT                  40003
#define IDM_ALIGN_RIGHT                 40004
#define IDM_ALIGN_CENTER                40005
#define IDM_ALIGN_JUSTIFIED             40006
```

Figure 7-9.
The JUSTIFY1 program.

INCLUDEPICTURE "Justify1.gif" * MERGEFORMAT \d

Figure 7-10.
A typical JUSTIFY1 display.

JUSTIFY1 displays a ruler (in logical inches, of course) across the top and down the left side of the client area. The DrawRuler function draws the ruler. A rectangle structure defines the area in which the text must be justified.

The bulk of the work involved with formatting this text is in the Justify function. Rhe function starts searching for blanks at the beginning of the text and uses GetTextExtentPoint32 to measure each line. When the length of the line exceeds the width of the display area, JUSTIFY1 returns to the previous space and uses the line up to that point. Depending on the value of the iAlign constant, the line is left aligned, right aligned, centered, or justified.

JUSTIFY isn't perfect. It doesnÆt have any logic for hyphens, for example. Also, the justification logic falls apart when there are fewer than two words in each line. Even if we solve this problem (which isn't a particularly difficult one), the program still won't work properly when a single word is too long to fit within the left and right margins. Of course, matters can become even more complex when you start working with programs that can use multiple fonts on the same line (as Windows word processors do with apparant ease). But nobody ever claimed this stuff was easy. It's just easier than if you were doing all the work yourself.

Previewing Printer Output

Some text is not strictly for viewing on the screen. Some text is for printing. And often in that case, the screen preview of the text must match the formatting of the printer output precisely. ItÆs not enough to show the same fonts and sizes and character formatting. With TrueType, thatÆs a snap. WhatÆs also needed is for each line in a paragraph to break at the same place. This is the hard part of WYSIWYG.

JUSTIFY1 includes a Print option, but what it does is simply set 1 inch margins at the top, left, and right side of the page. Thus, the formatting is completely independent of the screen display. HereÆs an interesting exercise: Change a few lines in JUSTIFY1 so that both the screen and the printer logic are based on a six-inch formatting rectangle. To do this, change the definitions of rect.right in both the WM_PAINT and Print command logic. In the WM_PAINT logic, the statement is:

```
rect.right = rect.left + 6 * GetDeviceCaps (hdc, LOGPIXELSX) ;
```

In the Print command logic, the statement is:
rect.right = rect.left + 6 * GetDeviceCaps (hdcPrn, LOGPIXELSX) ;
If you select a TrueType font, the line breaks on the screen should be the same as on the printer output.
But they arenÆt. Even though the two devices are using the same font in the same point size and displaying text in the same formatting rectangle, the different display resolutions and rounding errors cause the line breaks to occur at different places. Obviously a more sophisticated approach is needed for the screen previewing of printer output.
A stab at such an approach is demonstrated by the JUSTIFY2 program shown in Figure 17-11. The code is JUSTIFY2 is based on a program called TTJUST (ôTrueType Justifyö) written by MicrosoftÆs David Weise, which was in turn based on a version of the JUSTIFY1 program in an earlier edition of this book. To symbolize the increased complexity of this program, the Mark Twain has been replaced with the first paragraph from Herman Melville's Moby Dick.
JUSTIFY2.C

```
/*----------------------------------------
   JUSTIFY2.C -- Justified Type Program #2
                 (c) Charles Petzold, 1998
  ----------------------------------------*/

#include <windows.h>
#include "resource.h"

#define OUTWIDTH 6       // Width of formatted output in inches
#define LASTCHAR 127     // Last character code used in text

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

TCHAR szAppName[] = TEXT ("Justify2") ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     HWND     hwnd ;
     MSG      msg ;
     WNDCLASS wndclass ;

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

```
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;

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

     hwnd = CreateWindow (szAppName, TEXT ("Justified Type #2"),
                          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 ;
}

void DrawRuler (HDC hdc, RECT * prc)
{
     static int iRuleSize [16] = { 360, 72, 144, 72, 216, 72, 144, 72,
                                   288, 72, 144, 72, 216, 72, 144, 72 } ;
     int        i, j ;
     POINT      ptClient ;

     SaveDC (hdc) ;

          // Set Logical Twips mapping mode

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

          // Move the origin to a half inch from upper left

     SetWindowOrgEx (hdc, -720, -720, NULL) ;
```

```
          // Find the right margin (quarter inch from right)

     ptClient.x = prc->right ;
     ptClient.y = prc->bottom ;
     DPtoLP (hdc, &ptClient, 1) ;
     ptClient.x -= 360 ;

          // Draw the rulers

     MoveToEx (hdc, 0,                -360, NULL) ;
     LineTo   (hdc, OUTWIDTH * 1440, -360) ;
     MoveToEx (hdc, -360,               0, NULL) ;
     LineTo   (hdc, -360,      ptClient.y) ;

     for (i = 0, j = 0 ; i <= ptClient.x && i <= OUTWIDTH * 1440 ;
                         i += 1440 / 16, j++)
     {
          MoveToEx (hdc, i, -360, NULL) ;
          LineTo   (hdc, i, -360 - iRuleSize [j % 16]) ;
     }

     for (i = 0, j = 0 ; i <= ptClient.y ; i += 1440 / 16, j++)
     {
          MoveToEx (hdc, -360, i, NULL) ;
          LineTo   (hdc, -360 - iRuleSize [j % 16], i) ;
     }

     RestoreDC (hdc, -1) ;
}

/*------------------------------------------------------------------------
   GetCharDesignWidths:  Gets character widths for font as large as the
                         original design size
  ------------------------------------------------------------------------*/

UINT GetCharDesignWidths (HDC hdc, UINT uFirst, UINT uLast, int * piWidths)
{
     HFONT             hFont, hFontDesign ;
     LOGFONT           lf ;
     OUTLINETEXTMETRIC otm ;

     hFont = GetCurrentObject (hdc, OBJ_FONT) ;
     GetObject (hFont, sizeof (LOGFONT), &lf) ;

          // Get outline text metrics (we'll only be using a field that is
          //   independent of the DC the font is selected into)
```

```
    otm.otmSize = sizeof (OUTLINETEXTMETRIC) ;
    GetOutlineTextMetrics (hdc, sizeof (OUTLINETEXTMETRIC), &otm) ;

         // Create a new font based on the design size

    lf.lfHeight = - (int) otm.otmEMSquare ;
    lf.lfWidth  = 0 ;
    hFontDesign = CreateFontIndirect (&lf) ;

         // Select the font into the DC and get the character widths

    SaveDC (hdc) ;
    SetMapMode (hdc, MM_TEXT) ;
    SelectObject (hdc, hFontDesign) ;

    GetCharWidth (hdc, uFirst, uLast, piWidths) ;
    SelectObject (hdc, hFont) ;
    RestoreDC (hdc, -1) ;

         // Clean up

    DeleteObject (hFontDesign) ;

    return otm.otmEMSquare ;
}

/*-----------------------------------------------------------------------
   GetScaledWidths:  Gets floating point character widths for selected
                     font size
  -----------------------------------------------------------------------*/

void GetScaledWidths (HDC hdc, double * pdWidths)
{
     double  dScale ;
     HFONT   hFont ;
     int     aiDesignWidths [LASTCHAR + 1] ;
     int     i ;
     LOGFONT lf ;
     UINT    uEMSquare ;

          // Call function above

     uEMSquare = GetCharDesignWidths (hdc, 0, LASTCHAR, aiDesignWidths) ;

          // Get LOGFONT for current font in device context

     hFont = GetCurrentObject (hdc, OBJ_FONT) ;
```

```
     GetObject (hFont, sizeof (LOGFONT), &lf) ;

          // Scale the widths and store as floating point values

     dScale = (double) -lf.lfHeight / (double) uEMSquare ;

     for (i = 0 ; i <= LASTCHAR ; i++)
          pdWidths[i] = dScale * aiDesignWidths[i] ;
}

/*-------------------------------------------------------------
   GetTextExtentFloat:  Calculates text width in floating point
  -------------------------------------------------------------*/

double GetTextExtentFloat (double * pdWidths, PTSTR psText, int iCount)
{
     double dWidth = 0 ;
     int    i ;

     for (i = 0 ; i < iCount ; i++)
          dWidth += pdWidths [psText[i]] ;

     return dWidth ;
}

/*-------------------------------------------------------------------
   Justify:  Based on design units for screen/printer compatibility
  -------------------------------------------------------------------*/

void Justify (HDC hdc, PTSTR pText, RECT * prc, int iAlign)
{
     double dWidth, adWidths[LASTCHAR + 1] ;
     int    xStart, yStart, cSpaceChars ;
     PTSTR  pBegin, pEnd ;
     SIZE   size ;

          // Fill the adWidths array with floating point character widths

     GetScaledWidths (hdc, adWidths) ;

          // Call this function just once to get size.cy (font height)

     GetTextExtentPoint32(hdc, pText, 1, &size) ;

     yStart = prc->top ;
     do                            // for each text line
     {
```

```
          cSpaceChars = 0 ;        // initialize number of spaces in line

          while (*pText == ' ')    // skip over leading spaces
               pText++ ;

          pBegin = pText ;         // set pointer to char at beginning of
line

          do                       // until the line is known
          {
               pEnd = pText ;      // set pointer to char at end of line

                    // skip to next space

               while (*pText != '\0' && *pText++ != ' ') ;

               if (*pText == '\0')
                    break ;

                    // after each space encountered, calculate extents

               cSpaceChars++ ;
               dWidth = GetTextExtentFloat (adWidths, pBegin,
                                                    pText - pBegin - 1) ;
          }
          while (dWidth < (double) (prc->right - prc->left)) ;

          cSpaceChars-- ;               // discount last space at end of
line

          while (*(pEnd - 1) == ' ')    // eliminate trailing spaces
          {
               pEnd-- ;
               cSpaceChars-- ;
          }

               // if end of text and no space characters, set pEnd to end

          if (*pText == '\0' || cSpaceChars <= 0)
               pEnd = pText ;

          dWidth = GetTextExtentFloat (adWidths, pBegin, pText - pBegin -
1) ;

          switch (iAlign)               // use alignment for xStart
          {
          case IDM_ALIGN_LEFT:
```

```
               xStart = prc->left ;
               break ;

          case IDM_ALIGN_RIGHT:
               xStart = prc->right - (int) (dWidth + .5) ;
               break ;

          case IDM_ALIGN_CENTER:
               xStart = (prc->right + prc->left - (int) (dWidth + .5)) /
2 ;
               break ;

          case IDM_ALIGN_JUSTIFIED:
               if (*pText != '\0' && cSpaceChars > 0)
                    SetTextJustification (hdc,
                                          prc->right - prc->left -
                                                       (int) (dWidth + .5),
                                          cSpaceChars) ;
               xStart = prc->left ;
               break ;
          }
               // display the text

          TextOut (hdc, xStart, yStart, pBegin, pEnd - pBegin) ;

               // prepare for next line

          SetTextJustification (hdc, 0, 0) ;
          yStart += size.cy ;
          pText = pEnd ;
     }
     while (*pText && yStart < prc->bottom - size.cy) ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
     static CHOOSEFONT cf ;
     static DOCINFO    di = { sizeof (DOCINFO), TEXT ("Justify2: Printing")
} ;
     static int        iAlign = IDM_ALIGN_LEFT ;
     static LOGFONT    lf ;
     static PRINTDLG   pd ;
     static TCHAR      szText[] = {
                              TEXT ("Call me Ishmael. Some years ago --
never ")
                              TEXT ("mind how long precisely -- having
```

```
little ")
                                TEXT ("or no money in my purse, and nothing
")
                                TEXT ("particular to interest me on shore, I
")
                                TEXT ("thought I would sail about a little
and ")
                                TEXT ("see the watery part of the world. It
is ")
                                TEXT ("a way I have of driving off the
spleen, ")
                                TEXT ("and regulating the circulation.
Whenever ")
                                TEXT ("I find myself growing grim about the
")
                                TEXT ("mouth; whenever it is a damp, drizzly
")
                                TEXT ("November in my soul; whenever I find
")
                                TEXT ("myself involuntarily pausing before ")
                                TEXT ("coffin warehouses, and bringing up the
")
                                TEXT ("rear of every funeral I meet; and ")
                                TEXT ("especially whenever my hypos get such
an ")
                                TEXT ("upper hand of me, that it requires a
")
                                TEXT ("strong moral principle to prevent me
")
                                TEXT ("from deliberately stepping into the ")
                                TEXT ("street, and methodically knocking ")
                                TEXT ("people's hats off -- then, I account
it ")
                                TEXT ("high time to get to sea as soon as I
")
                                TEXT ("can. This is my substitute for pistol
")
                                TEXT ("and ball. With a philosophical
flourish ")
                                TEXT ("Cato throws himself upon his sword; I
")
                                TEXT ("quietly take to the ship. There is ")
                                TEXT ("nothing surprising in this. If they
but ")
                                TEXT ("knew it, almost all men in their
degree, ")
                                TEXT ("some time or other, cherish very
```

```
nearly ")
                              TEXT ("the same feelings towards the ocean
with ")
                              TEXT ("me.") } ;
     BOOL              fSuccess ;
     HDC               hdc, hdcPrn ;
     HMENU             hMenu ;
     int               iSavePointSize ;
     PAINTSTRUCT       ps ;
     RECT              rect ;

     switch (message)
     {
     case WM_CREATE:
               // Initialize the CHOOSEFONT structure

          hdc = GetDC (hwnd) ;
          lf.lfHeight = - GetDeviceCaps (hdc, LOGPIXELSY) / 6 ;
          lstrcpy (lf.lfFaceName, TEXT ("Times New Roman")) ;
          ReleaseDC (hwnd, hdc) ;

          cf.lStructSize    = sizeof (CHOOSEFONT) ;
          cf.hwndOwner      = hwnd ;
          cf.hDC            = NULL ;
          cf.lpLogFont      = &lf ;
          cf.iPointSize     = 0 ;
          cf.Flags          = CF_INITTOLOGFONTSTRUCT | CF_SCREENFONTS |
                              CF_TTONLY | CF_EFFECTS ;
          cf.rgbColors      = 0 ;
          cf.lCustData      = 0 ;
          cf.lpfnHook       = NULL ;
          cf.lpTemplateName = NULL ;
          cf.hInstance      = NULL ;
          cf.lpszStyle      = NULL ;
          cf.nFontType      = 0 ;
          cf.nSizeMin       = 0 ;
          cf.nSizeMax       = 0 ;

          return 0 ;

     case WM_COMMAND:
          hMenu = GetMenu (hwnd) ;

          switch (LOWORD (wParam))
          {
          case IDM_FILE_PRINT:
                                   // Get printer DC
```

```
               pd.lStructSize = sizeof (PRINTDLG) ;
               pd.hwndOwner   = hwnd ;
               pd.Flags       = PD_RETURNDC | PD_NOPAGENUMS |
PD_NOSELECTION ;

               if (!PrintDlg (&pd))
                    return 0 ;

               if (NULL == (hdcPrn = pd.hDC))
               {
                    MessageBox (hwnd, TEXT ("Cannot obtain Printer DC"),
                                szAppName, MB_ICONEXCLAMATION | MB_OK) ;
                    return 0 ;
               }
                    // Set margins for OUTWIDTH inches wide

               rect.left  = (GetDeviceCaps (hdcPrn, PHYSICALWIDTH) -
                             GetDeviceCaps (hdcPrn, LOGPIXELSX) * OUTWIDTH)
/ 2
                           - GetDeviceCaps (hdcPrn, PHYSICALOFFSETX) ;

               rect.right = rect.left +
                             GetDeviceCaps (hdcPrn, LOGPIXELSX) *
OUTWIDTH ;

                    // Set margins of 1 inch at top and bottom

               rect.top    = GetDeviceCaps (hdcPrn, LOGPIXELSY) -
                             GetDeviceCaps (hdcPrn, PHYSICALOFFSETY) ;

               rect.bottom = GetDeviceCaps (hdcPrn, PHYSICALHEIGHT) -
                             GetDeviceCaps (hdcPrn, LOGPIXELSY) -
                             GetDeviceCaps (hdcPrn, PHYSICALOFFSETY) ;

                    // Display text on printer

               SetCursor (LoadCursor (NULL, IDC_WAIT)) ;
               ShowCursor (TRUE) ;

               fSuccess = FALSE ;

               if ((StartDoc (hdcPrn, &di) > 0) && (StartPage (hdcPrn) >
0))
               {
                         // Select font using adjusted lfHeight
```

```
               iSavePointSize = lf.lfHeight ;
               lf.lfHeight = -(GetDeviceCaps (hdcPrn, LOGPIXELSY) *
                                   cf.iPointSize) / 720 ;

               SelectObject (hdcPrn, CreateFontIndirect (&lf)) ;
               lf.lfHeight = iSavePointSize ;

                    // Set text color

               SetTextColor (hdcPrn, cf.rgbColors) ;

                    // Display text

               Justify (hdcPrn, szText, &rect, iAlign) ;

               if (EndPage (hdcPrn) > 0)
               {
                    fSuccess = TRUE ;
                    EndDoc (hdcPrn) ;
               }
          }
          ShowCursor (FALSE) ;
          SetCursor (LoadCursor (NULL, IDC_ARROW)) ;

          DeleteDC (hdcPrn) ;

          if (!fSuccess)
               MessageBox (hwnd, TEXT ("Could not print text"),
                           szAppName, MB_ICONEXCLAMATION | MB_OK) ;
          return 0 ;

     case IDM_FONT:
          if (ChooseFont (&cf))
               InvalidateRect (hwnd, NULL, TRUE) ;
          return 0 ;

     case IDM_ALIGN_LEFT:
     case IDM_ALIGN_RIGHT:
     case IDM_ALIGN_CENTER:
     case IDM_ALIGN_JUSTIFIED:
          CheckMenuItem (hMenu, iAlign, MF_UNCHECKED) ;
          iAlign = LOWORD (wParam) ;
          CheckMenuItem (hMenu, iAlign, MF_CHECKED) ;
          InvalidateRect (hwnd, NULL, TRUE) ;
          return 0 ;
     }
     return 0 ;
```

```
     case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps) ;

          GetClientRect (hwnd, &rect) ;
          DrawRuler (hdc, &rect) ;

          rect.left  += GetDeviceCaps (hdc, LOGPIXELSX) / 2 ;
          rect.top   += GetDeviceCaps (hdc, LOGPIXELSY) / 2 ;
          rect.right = rect.left + OUTWIDTH * GetDeviceCaps (hdc,
LOGPIXELSX) ;

          SelectObject (hdc, CreateFontIndirect (&lf)) ;
          SetTextColor (hdc, cf.rgbColors) ;

          Justify (hdc, szText, &rect, iAlign) ;

          DeleteObject (SelectObject (hdc, GetStockObject (SYSTEM_FONT)));
          EndPaint (hwnd, &ps) ;
          return 0 ;

     case WM_DESTROY:
          PostQuitMessage (0) ;
          return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

JUSTIFY2.RC

```
//Microsoft Developer Studio generated resource script.

#include "resource.h"
#include "afxres.h"

//////////////////////////////////////////////////////////////////////
//
// Menu

JUSTIFY2 MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&Print",                      IDM_FILE_PRINT
    END
    POPUP "&Font"
    BEGIN
        MENUITEM "&Font...",                    IDM_FONT
    END
```

```
    POPUP "&Align"
    BEGIN
        MENUITEM "&Left",                       IDM_ALIGN_LEFT, CHECKED
        MENUITEM "&Right",                      IDM_ALIGN_RIGHT
        MENUITEM "&Centered",                   IDM_ALIGN_CENTER
        MENUITEM "&Justified",                  IDM_ALIGN_JUSTIFIED
    END
END
```

RESOURCE.H

```
// Microsoft Developer Studio generated include file.
// Used by Justify2.rc

#define IDM_FILE_PRINT                  40001
#define IDM_FONT                        40002
#define IDM_ALIGN_LEFT                  40003
#define IDM_ALIGN_RIGHT                 40004
#define IDM_ALIGN_CENTER                40005
#define IDM_ALIGN_JUSTIFIED             40006
```

Figure 17-11.
The JUSTIFY2 program.

JUSTIFY2 only works with TrueType fonts. In its GetCharDesignWidths function the program uses the GetOutlineTextMetrics function to get a seemingly unimportant piece of information. This is the OUTLINETEXTMETRIC field otmEMSquare.

A TrueType font is designed on a grid called an em-square. The word ôemö refers to the width of a square piece of type, that is, a width that is equal to the point size of the font. All the characters of any particular TrueType font are designed on the same grid, although they generally have different widths. The otmEMSquare field of the OUTLINETEXTMETRIC structure gives the dimension of this em-square for any particular font. For most TrueType fonts, youÆll find that the otmEMSquare field is equal to 2048, which means that the font was designed on a 2048-by-2048 grid.

HereÆs the key: You can set up a LOGFONT structure for the particular TrueType typeface name but with an lfHeight field equal to the negative of the otmEMSquare value. After creating that font and selecting it into a device context, you can call GetCharWidth. This function gives you the width of individual characters in the font in logical units. Normally, these character widths are not exact because theyÆve been scaled to a different font size. But with a font based on the otmEMSquare size, these widths are always exact integers independent of any device context.

The GetCharDesignWidths function obtains the original character design widths in this manner and stores them in an integer array. The JUSTIFY2 program knows that itÆs text only uses ASCII characters, so this array neednÆt be very large. The GetScaledWidths function converts these integer widths to floating point widths based on the actual point size of the font in the deviceÆs logical coordinates. The GetTextExtentFloat function uses those floating point widths to calculate the width of a whole string.

ThatÆs the function the new Justify function uses to calculate the widths of lines of text.

The Fun and Fancy Stuff

Expressing font characters in terms of outlines opens up lots of potential in combining fonts with other graphics techniques. Earlier we saw how fonts can be rotated. This final section shows some other tricks. But before we continue, letÆs look at two important preliminaries: graphics paths and extended pens.

The GDI Path

A path is a collection of straight lines and curves stored internally to GDI. Paths were introduced in the 32-bit versions of Windows. The path may initially seem very similar to the region, and indeed, you can convert a path to a region and use a path for clipping. However, we'll see shortly how they differ.

The begin a path definition, you simply call:

```
BeginPath (hdc) ;
```

After this call, any line you draw (straight lines, arcs, and Bezier splines) will be stored internally to GDI as a path and not rendered on the device context. Often a path consists of connected lines. To make connected lines, you use the LineTo, PolylineTo, and BezierTo functions, all of which draw lines beginning at the current position. If you change the current position using MoveToEx, or if you call any of the other line-drawing functions, or if you call one of the window/viewport functions that cause a change in the current position, you create a new subpath within the entire path. Thus, a path contains one or more subpaths, where each subpath is a series of connected lines.

Each subpath within the path can be open or closed. A closed subpath is one in which the first point of the first connected line is the same as the last point of the last connected line, and moreover, the subpath is concluded by a call to CloseFigure. CloseFigure will close the subpath with a straight line if necessary. Any subsequent line-drawing function begins a new subpath. Finally, you end the path definition by calling.

```
EndPath (hdc) ;
```

At this point you then call one of the following five functions:

```
StrokePath (hdc) ;
FillPath (hdc) ;
StrokeAndFillPath (hdc) ;
hRgn = PathToRegion (hdc) ;
SelectClipPath (hdc, iCombine) ;
```

Each of these functions destroys the path definition after completion. StrokePath draws the path using the current pen. You may wonder: What's the point? Why can't I just skip all this path stuff and draw the lines normally? I'll tell you why shortly.

The other four functions close any open paths with straight lines. FillPath fills the path using the current brush according to the current polygon filling mode. StrokeAndFillPath does both jobs in one shot. You can also convert the path to a region or use the path for a clipping area.

The iCombine parameter is one of the RGN_ constants use with the CombineRgn function, and indicates how the path is combined with the current clipping region.
Paths are more flexible than regions for filling and clipping because regions can be defined only by combinations of rectangles, ellipses, and polygons. Paths can be composed of Bezier splines and (at least in Windows NT) arcs. In GDI, paths and regions are stored quite differently. The path is a
 a o a   at   t ating ing ing ing by   t
intointointointoell spl splcan can can can cas c exc exc exc exc exc exc
exc exc exc (at ley, ey, ey, ey, ey, ey, eyJustJustJusxc exc exc exc exc
exc exc exc ex
                    on,ou   pat   xt xt xt xt xt xt
xt xt
t rile (dWidth <th <th <th <th <th <th
th ine paramaramaramaramarased ofd otCuotCuotCuotCuotC, NU, Ns cNs cNs cNs
cNs cNs cNs cNsstJusxJusnRusnRusnRusnRusn   n   n   n
t rilerilerilerilerilerilerileriler
lering spaces
          {
               pEnd-- ;
               cSpaceChars-- ;
          }

               // if end of tef tef tef tef tef tef t
f t
f t
f t
f t
f t
f t
f t
f t
f t
f t
f t
fusnRuslipslipslipslipslipslipsl, 72, 7 ca ca ca ca
caerilerprirprirprirprirprirp
irp
irp_Rrp_Rrp_Rrp_Rrp_   _   _   font in thn thn thn thn t- ;
  Cur Cur Cur Cur Cur C
r C
r C thn thn thn t- ;
  Cur Cur Cur Cur CuCheuCheuCheuCh
f t
f t
ot
ot

```
ot
ot
pslips   ñ   ñ   ñ   ñ    ca
Rrp_Rrousrousrousrousrouaft inlip.lip.lip.lip.lipULLpULLpULLpULt  ri  ri
ri  ri  rif end{
 ñ{
 ñ{
 ñ{
 ñ{
 ur Curs c.s c.s c.s c.s
cM_PcM_PcM_PcM_e4M_e4M_e4M_e4M_e4M_e4M_e4M_e4M_e4Mize ize ize ize ize
ca "st "st "st "st "st "
t "
t "4Mize izeoizeoizeoizeoizesrot  ot   "st "st "st "st
"s.les.les.les.l8ñ.l8ñ.l8ñ.l8ñ.l8  ri  .l8ñ.l8ñ.l8ñ.l8ñ.l8{
 ñ{
1 p
1 p
1 p
1 p
1 n fc.sp_R.p_R.p_R.p_R.p_R 2 R 2 R 2 R 2 a col col col col coft oft oft
oft72, 7 ca ca ca ca caerilerprirprirprirprirprirp
irp
irp
irp
irp
irp
i
p
i
p
i  n   n   n   n   n ip.lipCh ñCh ñCh ñCh ñCh ñ.l8ñ.   .   .   .   .
 .
 . Creatreatreatreatrea1 n fct72Rt72Rt72Rt72Rt72   2   2   2  li  li  li
li  li  li   <=  <=  <= 0 ;
  ;
  ;
  ;
  ;
  ;
  ;
  ;
  ;
  otCuotCuotCuotCuotCuotCuotCuotCuot
uot3Eot3Eot3Eot3Eot3Eot3
  3
  3
```

```
  3
 0EST0EST0EST0ESTñCh ñ.3Eot lit lit lit lit lit lit l
t leo leo leo leo leo leo le       spl spl spl spl spl spl sp<= p<= ws  ws
ws  ws  ws  ws  ws++)
++)
++ñ
++ñ
++ñ
++ñ
++ñEx (Ex  co siz siz siz siz siz siz si
 si a c a c a c a c a c a
c a
c aC/////////////////ST0ESTileoileoileoileoilelit li a i a i a i a i a i a
i a i a i a i a i a i a i a i a i a lines)nes)nes)nes)nes li  l ñCs  ws  ws  ws
ws    <=  <=  an  an  an  an  a+ñ
++ñt72ñt72ñt72ñt72ñt72ñt72ñt7T   o   =   =   =   =   =   = IN
 IN
 IN
 IN
 INI INI INI INI INI INI INI INI
NI  INI INI INI INI INI OPU OPU OPU OP! OP! OP! OP! OP! OP! OP! OP! OP! OP!
OP! OP! OP! OP!u w!u w!u w!u w!u   ws   3
   <=  <=  <=  <=  ++ñ
++T0E+ñ
++ñt++ñt++ñt++ñtz stz stz stz stt stt stt stt stt stt sty ino inI
//////leoINI INI INI INI INIileIileIileIileIAIeIAIeIAIeIAIeIAIeIAIeIAIeIAIeIAIe
AIeri eri eri eri eri t
! OleoOleoOleoOleoOleoOleoOled sao sa,slipslipslipslipsln  an  and by
Justify2.rc

#define IDM_FILE_PRINT                    ty ino ino ino ino ino ino ino ino
inoftnoftnoftnoftnoftnoftnoftnoftnoftnxtnxtnxtnxtn CueIAis Ais Ais Ais
Ais Ais Ais Ais A
s A,s A,s A,s A,s A,s A,s A,s A,s A,s A,s A,s A,s A,s A,s A,s A,s A,s A,s A,s
A,s A,   ,   ,   ,   ,  gur gur gur gur gur aor aor aor aor aino ins Axs
Axs Axs Axs ApULtno   x   x   x   x
rinofAIefAIefAIefAIefAIefAIefAIefAIaveDaveat eat eat eat eat eatc, Mc,
Mc,fAIavAIavAIavAIavAIavAIavAIavAIavA
avA;
  ;
  ;
  ;
  ;
  ;
  ;
  ;
  ;
```

;
;
;
;

Metrietrietrietrietr aor aor aor aor aino ins Axs Axs Axs Axs ApULtno x
x x x rinofAIefAIefAIefAIefAIefAIefAIaveDaveat eat eat eat eat eat
eatc, Mc, Mc,fAIavAIavAIavAIavAIavAIavAIavAIavA
avA;
;
;
;
;
;
;
;
;
;
;
;

Metrietrietrietrietr aor aor aor aor aor ains Axs Ac, Mc, Mc, Mc, Mc, Mcx
x x inI I I I I
reaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI

aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa

IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaI

```
AIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIa
IAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI
aIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIA
IaIAIaIAIaIAIaIAIaIAIaIAIaIAIaIAI ;

aIAIaIAIaIAIaIAIaIAIaAIaaAIaaAIaaAIaaAIaaAIaaAIaaaIAaaIAaaIAoIaIaIAaIAIIaIAIaIn
IaIaIAIAIaIAAIaIAaIAIaIAIaIAIaIAIintCF_INITTOITTIaIAIaIAIzxc
aIAIaIAIaIAIaIAIaIAIaIAaIAAaIAAaIAAaIAAaIAAaIAIAIIAIaIA,   ,   ,   ,AaIAAaIAAaI
AAaIAAaIAAaIAAaIAAaIAAaIAAaIAAaIAAaIAAaIAAaIAAaI   =   =e
aIAIAAaaIAAIaAIaaIAIIIAIh consi IAI IA IAI IAIaIAIaIAIaIAI Ns ,AaIAAAIaIt
f t
f t
f
tAIaIIaIAIaIAIaIAIIaIIAIsIAIsIAIsIAIsIAIsIAIsIAIsIAIsIAIsIAIsIAIsIAIsIAIsIA
IsIAIsIAIsIAIIAIIAI
```