

Chapter 16

Drawing Text

This chapter covers controlling the appearance of text, using classes in the System.Drawing namespace and also by using P/Invoke to drill through to the underlying Win32 libraries for useful text drawing features that are otherwise not available in the Compact Framework.

*Author's Note: In this review chapter **C#-Specific Text is highlighted in YELLOW.***

Drawing Text.....	2
Compact Framework Text Drawing Support	2
The DrawString Function.....	3
Sample: SimpleDrawString.....	4
Font Selection.....	5
The Font Property of controls.....	6
Generic Fonts	7
Sample: GenericFonts.....	9
Creating Named Fonts	11
Enumerating Fonts	14
Sample: FontPicker	14
Sample: FontList – Win32 Font Enumeration DLL [Comment 17.60]	16
Native Code Fonts.....	20
Creating fonts in unmanaged code	21
Drawing with fonts in unmanaged code	23
Cleanup	23
Sample: RotateText.....	24
Placing Text.....	27
Text Size and the MeasureString method	28
Sample: MeasureString.....	28
Text Alignment.....	30
Sample: TextAlign	31
Word Wrap.....	36
Sample: WordWrap	37
Text Color	39
Foreground and Background Text Colors	40
Sample: TextColor.....	40
Conclusion	46

The focus of this chapter is on drawing text on the screen of a smart device using the .NET Compact Framework. As we describe in chapter 16, *Compact Framework Graphics*, the Compact Framework supports a subset of the graphical output features of the Desktop Framework. (For details on the differences, see the boxed text titled "Comparing CF and DF Text Drawing.") In general, though, the text drawing support is quite rich. You can do all of the more commonly required text drawing operations such as selecting different fonts by name, in different sizes and styles, and also control the color of drawn text. Some subtle effects are not available, but this is a small price to pay for the small memory footprint occupied by the Compact Framework.

[\[Comment 17.1\]](#)

Drawing Text

A graphical environment provides the programmer with a rich set of tools for creating a wide range of graphical images. Such an environment makes it possible to mingle pictures with text, to display text in a range of font sizes and styles, and to use an array of effects to convey subtle messages to users. In such an environment, text itself is treated as a graphical object. Drawing text in a graphical environment is paradoxically more complex than drawing text in a non-graphical environment. In a character-oriented world such as in a console program, fixed pitch fonts are used which place text in orderly rows and columns. In a graphical world, by contrast, both fixed and variable pitch fonts float in a sea of pixels. The increase in complexity allows for the free mixing of text and graphics. This freedom comes at a cost, namely the extra effort required to tame the complexity of graphical text and to use it to enhance a program's graphical output. [\[Comment 17.2\]](#)

Compact Framework Text Drawing Support

A Compact Framework program can draw text using any available font. Using TrueType fonts, that text output can be scaled to any desired size from 8 point on up to 72 point and beyond. That text can be drawn in any available color, although most programs are likely to use the default system text color. [\[Comment 17.3\]](#)

By drilling through to the underlying Win32 libraries, a Compact Framework program can do a few more things such as drawing rotated text, and accessing Clear Type fonts. Table 16-1 summarizes Compact Framework text drawing features, how to access these features, and this chapter's sample programs which illustrate each feature. [\[Comment 17.4\]](#)

Table 16-1 Text Drawing Features and Sample Programs [\[Comment 17.5\]](#)

Feature	Comment	Sample Program
Simple text drawing	Call the <code>DrawString</code> ¹ method to draw text in a control or form.	<code>SimpleDrawString</code> – Shows simplest text drawing, which involves creation of a brush (for text color) and use of the form's default font.
Simplest font creation	Create a font using the <code>FontFamily</code> enumeration to select between a fixed-pitch, serif, or sans-serif font without regard to font face name.	<code>GenericFonts</code> – Creates and draws with each of the three generic font families, in a range of styles (regular, bold, italic, strikeout, and underline)
Font Enumeration	Font enumeration involves getting a list of available font face names installed in the system. Compact Framework does not support font enumeration, so we rely on a Win32 DLL that does the font enumeration work for us	<code>FontPicker</code> – CF program that creates fonts of specific face name. <code>FontList</code> – Win32 DLL that enumerates available system fonts.
Rotate text	Compact Framework does not natively support rotated fonts. For that, you must call the Win32 font creation functions.	<code>RotateText</code> – CF program that uses <code>P/Invoke</code> to call Win32 font creation program.
ClearType fonts	ClearType is a font technology that aids in reading small text on LCD displays. Compact Framework does not support this, so you must instead call Win32 font creation functions as	

¹ The fully-qualified name is `System.Drawing.Graphics`.

Feature	Comment	Sample Program
	illustrated in the RotateText sample.	
Calculate size of graphical text	Optimal positioning of text requires calculating the size of the bounding box of drawn text. This is accomplished using the MeasureString method.	MeasureString – Shows how to use results from MeasureString method in drawing.
Setting text alignment	By default, text is aligned to the upper-left corner of the text box. Compact Framework does support alternative text alignment settings, so different alignments require modifying the (x,y) location where you draw text.	TextAlign – Shows twelve ways to align text by mixing and matching four vertical alignments and three horizontal alignments
Word wrap	Compact Framework supports two versions of the DrawString method, but only one of them supports word wrap. The version which uses a rectangle supports word wrap. In addition to wrapping the text to the indicated rectangle, the text is clipped to that rectangle as well – that is, text is only drawn within the specified rectangle.	WordWrap – shows use of the version of the DrawString method which supports wrapping and clipping a text string to a specified output rectangle.
Text color	Set text foreground color by creating a brush. Set text background color by drawing a colored rectangle before drawing the text.	TextColors – Shows three ways to pick colors: (1) using named colors, (2) using system colors, (3) with the color picker dialog box (requires P/Invoke)

The DrawString Function

All Compact Framework text drawing is done with the `DrawString` method, a member of the `Graphics` class², available in two overloaded implementations. We start our discussion with the simpler of the two overloaded functions, which accepts a pair of (x,y) values for the location where the text is to be drawn. This method does not provide automatic word-wrapping support. So if a string is too long for the available drawing space, the "extra" characters disappear. A second version of `DrawString`, which we discuss later in this chapter, does word wrapping for you. [\[Comment 17.6\]](#)

The simpler version of `DrawString`, which takes five parameters, is defined as follows: [\[Comment 17.7\]](#)

```
public void DrawString(
    string str,
    Font font,
    Brush brText,
    float x,
    float y);
```

[\[Comment 17.8\]](#)

This first parameter, `str`, identifies the string to draw. While automatic word-wrap is not supported, a carriage-return or linefeed character within the string causes the string to display in

² The fully qualified name is `System.Drawing.Graphics`.

multiple lines. (In C#, insert a new line using the `\n` or `\r` character; in VB, a new line is caused by including any of the following constants in a string: `vbLf`, `vbCr`, or `vbCrLf`.) [\[Comment 17.9\]](#)

The second parameter, `font`, is the font used for drawing the characters. This could be the default font (the `Font` property) of a control, or a font which you dynamically create. [\[Comment 17.10\]](#)

The third parameter, `brText`, identifies the brush which specifies the color of the text foreground pixels; in the world of .NET programming, the background pixels are always left untouched³ when you draw text. [\[Comment 17.11\]](#)

The fourth and fifth parameters, `x` and `y`, indicate the text drawing location. This location is the upper-left corner of the rectangle which bounds the text. These coordinate values are single-precision floating point values, which is different from the integer coordinates used to draw raster and vector graphics. [\[Comment 17.12\]](#)

Sample: SimpleDrawString

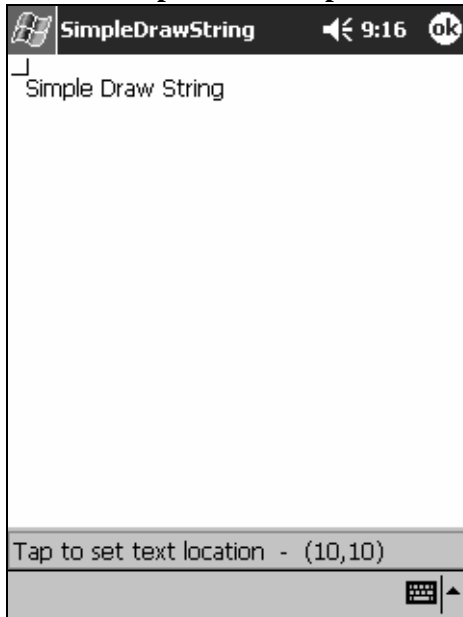
Our first sample program shows the simplest way to draw in a form. Figure 16-1 shows the program's output. This program uses the form's default font to draw a character string using the system default window text color. [\[Comment 17.13\]](#)

We highlight the text placement location with a pair of lines, to make it obvious that the (x,y) location identifies the upper-left corner of an imaginary box that bounds the text string. The Compact Framework does not have any built-in facility to request alternative text alignments, which you might use to center a string over a column of data or center the text inside a graphical image. The `TextAlign` sample, shown later in this chapter, shows some simple techniques for manually creating eleven alternative text alignments to the default upper-left alignment. [\[Comment 17.14\]](#)

Text Drawing Performance

The `SimpleDrawString` sample shows one way to draw text, but this is not necessarily the fastest way to draw text. We notice a 10% improvement in drawing speed when we create a font ahead of time, instead of using the `Font` property of the control. For most purposes, the `Font` property works well for drawing text. To squeeze the fastest text drawing from the Compact Framework, however, we suggest you cache the font and passing that font to the `DrawString` function. [\[Comment 17.15\]](#)

³ Win32 programmers may recall that the `Background Color` attribute in a DC allows a text drawing operation to also affect the background pixels.

Figure 16-1 Output from SimpleDrawString sample [\[Comment 17.16\]](#)**Listing 16-1** Fragment from SimpleDrawString.cs showing Paint event handler

```
private float xDraw = 10;
private float yDraw = 10;

private void
FormMain_Paint(object sender, PaintEventArgs e)
{
    Brush brText = new SolidBrush(SystemColors.WindowText);
    e.Graphics.DrawString("Simple Draw String", Font, brText,
        xDraw, yDraw);

    // Highlight origin.
    int x = (int)xDraw;
    int y = (int)yDraw;
    Pen penBlack = new Pen(Color.Black);
    e.Graphics.DrawLine(penBlack, x, y, x-8, y);
    e.Graphics.DrawLine(penBlack, x, y, x, y-8);

    // Cleanup
    brText.Dispose();
    penBlack.Dispose();
}
```

Font Selection

We now turn our attention to the subject of fonts. Font selection provides the primary way to control the appearance of text. Windows CE supports two basic font technologies: bitmap fonts

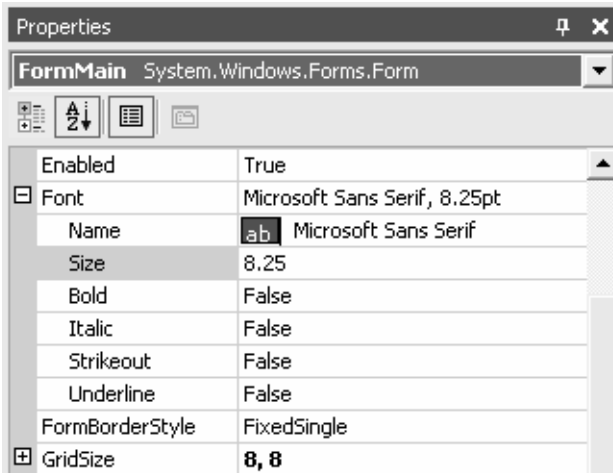
and TrueType fonts. A given platform can support only one of these two technologies. The decision about which to support is made by the platform creator, and cannot be changed by an application program. A given platform would support bitmap fonts to minimize the size of the operating system image. The benefits of supporting TrueType fonts include the ability to draw scalable fonts, and the ability to draw rotated text. [\[Comment 17.19\]](#)

What font support is found on Windows Mobile devices? The Pocket PC supports TrueType fonts. As we show later, there are three fonts on a typical Pocket PC: Tahoma, Courier New, and Bookdings. The SmartPhone supports bitmap fonts. A typical SmartPhone has two fonts: an 11-point bold Nina, and a 16-point Nina. [\[Comment 17.20\]](#)

The Font Property of controls

The built-in controls use fonts. As shown in figure 16-2, you select a control's font at design time in the properties window of the Windows Forms Designer provided as part of Visual Studio .NET. A word of caution is in order because the forms designer displays a font picker dialog for whatever fonts are installed on your desktop system. You can, for example, select fonts such as *Arial* or *Wingdings*. But if you run your program on a device without these fonts, you may get results different from what you expect. [\[Comment 17.21\]](#)

Figure 16-2 Control font selection in the Windows Form Designer [\[Comment 17.22\]](#)



The reason is that font support in Windows CE – just like in desktop Windows – uses a process that is sometimes called "font mapping". This process takes a request for a font and tries to map that request to a font that is actually present in the system. It works best when the font name or font family is present, and when the size you request is available. It fails when either of these conditions is not met, and under such a situation this feature is sometimes called "font mangling." The solution is to make sure that you request font names and sizes which are available. [\[Comment 17.23\]](#)

As shown in the `SimpleDrawString` sample, you can draw in a form or in a control by accessing the `Font` property of the form or control. Here is the line of code that shows this being done: [\[Comment 17.24\]](#)

```
e.Graphics.DrawString("Simple Draw String", Font, brText,
xDraw, yDraw);
```

The `Font` property is a read/write property, so you can modify a control's font at run-time using any of the font creation techniques in this chapter. [\[Comment 17.25\]](#)

Compact Framework controls do not support ambient properties

Programmers familiar with the Desktop .NET Framework might notice that Compact Framework controls do not support the concept of ambient properties. On the desktop, controls use the ambient properties from the parent control if that property is not specified for the control itself. The following properties of controls in the Desktop Framework are ambient properties: `BackColor`, `Cursor`, `Font`, and `ForeColor`. [\[Comment 17.26\]](#)

The built-in Compact Framework controls do not support ambient properties. To change the set the font for a Compact Framework control, you cannot simply set the font in the control's parent. You must instead also change the font for each and every control. [\[Comment 17.27\]](#)

Generic Fonts

There are several ways to get a font, but the easiest is to ask for a generic font. The reason is that the only other way to ask for a font is by face name – using names like “Arial”, “Times New Roman”, or “Courier New.” And while this doesn’t seem that complicated, the fact is that different systems ship with different available fonts. The three fonts just named, for example, are three fonts you find in all the desktop versions of Windows starting with Windows 95. So if you are writing solely for that environment, you can probably get away with asking for those three fonts by name. [\[Comment 17.28\]](#)

On a Windows CE-powered device, the set of available fonts is a bit different. What you find depends on how a specific device was configured. On the Pocket PC 2002, for example, the following fonts are available: [\[Comment 17.29\]](#)

- `Bookdings` – A special symbol font used by the PocketPC itself to draw elements of the user-interface, such as checkmarks, scrollbar arrows, and other symbols. [\[Comment 17.30\]](#)
- `Courier New` – fixed pitch serif font. [\[Comment 17.31\]](#)
- `Tahoma` – Sans-serif, variable-pitch font [\[Comment 17.32\]](#)

To ask for fonts by name, you must first figure out what specific fonts are installed on a given system. The process for doing that is known as "font enumeration." The Compact Framework does not support font enumeration, so you must use `P/Invoke` to enumerate fonts in a Compact Framework program (see the `FontPicker` and `FontList` samples to see how that is done). [\[Comment 17.33\]](#)

Alternatively, you request a generic font, as shown here: [\[Comment 17.34\]](#)

```
Font font = new Font(FontFamily.GenericSansSerif, 10,
                    FontStyle.Regular);
```

This is one of two constructors for creating fonts (the other uses a face name and a font size). This function takes three parameters, and is defined as follows: [\[Comment 17.35\]](#)

```
public Font(
    FontFamily family,
    float emSize,
    FontStyle style);
```

The first parameter, `family`, is a value from the `FontFamily` enumeration⁴ which contains these three values: [\[Comment 17.36\]](#)

- `FontFamily.GenericMonospace` – Requests a fixed-pitch font, meaning that the width of every character in the font is the same. (like Courier New)
- `FontFamily.GenericSansSerif` – Requests a variable-pitch font without serifs (“Arial” on the desktop Windows, and “Tahoma” on the Pocket PC.)
- `FontFamily.GenericSerif` – Requests a variable-pitch font with serifs (like “Times New Roman” on desktop Windows).

The second parameter, `emSize`, is the point size of the desired font. On Windows CE systems which support the TrueType scalable font technology – such as the Pocket PC – each font is available in any size you desire. Generally speaking, 8 point is considered the smallest size which is readable on paper and 10 point is commonly the smallest size used on a display screen. And while you can create fonts which are one-half inch (36 point) or a full inch (72 point) tall, such sizes easily overwhelm the available screen space of most Windows CE-powered devices. [\[Comment 17.37\]](#)

The third parameter, `style`, is from the `FontStyle` enumeration that lets you fine-tune the appearance of the requested font. This enumeration provides five different font style modifiers, which you can combine together in any combination using the logical OR (`'|'`) operator: [\[Comment 17.38\]](#)

- `FontStyle.Bold` – Requests a bold font.
- `FontStyle.Italic` – Requests an italicized font.
- `FontStyle.Regular` – Requests a regular font (default).
- `FontStyle.Strikeout` – Requests a strikeout font.
- `FontStyle.Underline` – Requests an underlined font.

For example, the following code creates three different fonts using three different font families and three different font styles: [\[Comment 17.39\]](#)

```
// Create monospace bold 10 pt font.
Font fontMono = new Font(FontFamily.GenericMonospace, 10,
                        FontStyle.Bold);

// Create Sans Serif italic 10 pt font.
Font fontSans = new Font(FontFamily.GenericSansSerif, 10,
                        FontStyle.Italic);

// Create Sans Serif italic 10 pt font.
Font fontSerif = new Font(FontFamily.GenericSerif, 10,
                        FontStyle.Italic);
```

When a program creates drawing objects – whether fonts, pens, brushes, or a `Graphic` object – that program must dispose of the drawing objects. Failure to dispose of drawing objects creates memory leaks, which can eventually result in crashing the system. To destroy the three font objects created above, call the `Dispose` method for each of them as shown here: [\[Comment 17.40\]](#)

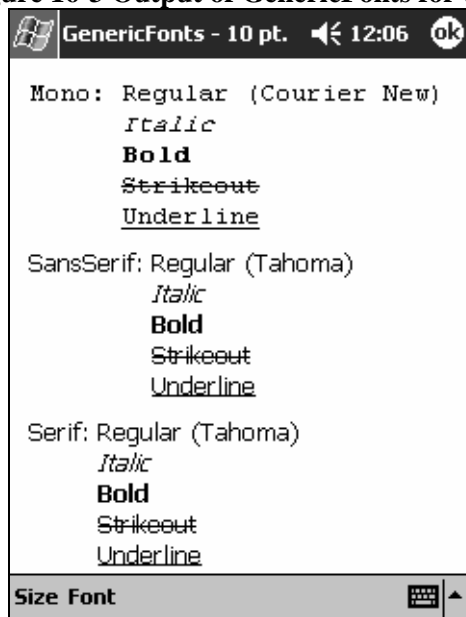
⁴ We call this an enumeration, but to be accurate this is a class instead of an enumeration because the desktop framework implementation provides several useful methods.

```
fontMono.Dispose();
fontSans.Dispose();
fontSerif.Dispose();
```

Sample: GenericFonts

GenericFonts is a sample that shows the creation of generic fonts in various sizes and styles. Figure 16-3 shows the output of this program for three 10-point fonts. This program lists the name of the actual font selected, and so you may notice that both the serif and the sans-serif fonts have the same face-name, Tahoma. This does not reflect a bug in our program or a bug in the Compact Framework, but rather occurs because Tahoma – a sans-serif font – is the closest match when either a serif or a sans-serif font is requested on a Pocket PC 2002. This is an artifact of font mapping, which we discuss earlier in this chapter. Listing 16-2 contains a portion of the code used to create this output. [\[Comment 17.41\]](#)

Figure 16-3 Output of GenericFonts for three 10-point fonts [\[Comment 17.42\]](#)



Listing 16-2 A fragment of GenericFonts.cs

```
private void
DisplaySerifFont(Graphics g, ref float x, ref float y)
{
    // Create brush for standard text color.
    Brush brText = new SolidBrush(SystemColors.WindowText);

    // Create Serif bold m_ptSize pt font.
    Font font = new Font(FontFamily.GenericSerif, m_ptSize,
        FontStyle.Regular);
    string str = "Serif: ";
    x = 10;
    g.DrawString(str, font, brText, x, y);
    SizeF sizeString = g.MeasureString(str, font);
```

```
x += sizeString.Width;

// Draw with Serif bold m_ptSize pt font.
str = "Regular (" + font.Name + ")";
g.DrawString(str, font, brText, x, y);
sizeString = g.MeasureString(str, font);
y += sizeString.Height;
font.Dispose();

// Create Serif italic m_ptSize pt font.
font = new Font(FontFamily.GenericSerif, m_ptSize,
    FontStyle.Italic);
str = "Italic";
g.DrawString(str, font, brText, x, y);
sizeString = g.MeasureString(str, font);
y += sizeString.Height;
font.Dispose();

// Create Serif regular m_ptSize pt font.
font = new Font(FontFamily.GenericSerif, m_ptSize,
    FontStyle.Bold);
str = "Bold";
g.DrawString(str, font, brText, x, y);
sizeString = g.MeasureString(str, font);
y += sizeString.Height;
font.Dispose();

// Create Serif strikethrough m_ptSize pt font.
font = new Font(FontFamily.GenericSerif, m_ptSize,
    FontStyle.Strikeout);
str = "Strikeout";
g.DrawString(str, font, brText, x, y);
sizeString = g.MeasureString(str, font);
y += sizeString.Height;
font.Dispose();

// Create Serif underline m_ptSize pt font.
font = new Font(FontFamily.GenericSerif, m_ptSize,
    FontStyle.Underline);
str = "Underline";
g.DrawString(str, font, brText, x, y);
sizeString = g.MeasureString(str, font);
y += sizeString.Height;
font.Dispose();

// Add spacing between text blocks.
y += sizeString.Height / 2;
}

private void
FormMain_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    float x = 10;
    float y = 10;
```

```

//
//  GenericMonospace Font Styles
//
if (mitemFontMono.Checked)
{
    DisplayMonoFont(g, ref x, ref y);
}

//
//  GenericSansSerif Font Styles
//
if (mitemFontSans.Checked)
{
    DisplaySansSerifFont(g, ref x, ref y);
}

//
//  GenericSerif Font Styles
//
if (mitemFontSerif.Checked)
{
    DisplaySerifFont(g, ref x, ref y);
}
}

```

Creating Named Fonts

Font selection by font family is easy, but – as shown in the request for the serif font – it is not very precise. An alternative is to create fonts using the font name. This requires you to know for this, you must know the name of the font you wish to create. On a Pocket PC, that means you have a choice of three TrueType fonts: [\[Comment 17.45\]](#)

- Bookdings
- Courier New
- Tahoma

Armed with this information, here is a constructor to create a font: [\[Comment 17.46\]](#)

```

public Font(
    string familyName,
    float emSize,
    FontStyle style);

```

Figure 16-4 shows the output from the `NamedFonts` sample, which creates three fonts by name, and displays one line of text for each font. The three fonts are the ones found on a Pocket PC. The code for the `Paint` event handler appears in listing 16-3. Two of the fonts create reasonable output, but the output of the third font – the font named `Bookdings` – does not seem quite right. The `DrawString` call which used that font was given the string '12 Point Bookdings', but that is not what is displayed in figure 16-4. [\[Comment 17.47\]](#)

The output seems wrong because the `Bookdings` font does not contain regular text characters, or "glyphs" in the language of font specialists. This font is used to draw various

Listing 16-3 The Paint event handler from NamedFonts.cs

```
private void
FormMain_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    float x = 10;
    float y = 10;

    Font font1 = new Font("Tahoma", 14, FontStyle.Regular);
    Font font2 = new Font("Courier New", 10, FontStyle.Regular);
    Font font3 = new Font("Bookdings", 12, FontStyle.Regular);

    Brush brText = new SolidBrush(SystemColors.WindowText);

    g.DrawString("14 Point Tahoma", font1, brText, x, y);
    SizeF sizeX = g.MeasureString("X", font1);
    y += sizeX.Height;

    g.DrawString("10 Point Courier New", font2, brText, x, y);
    sizeX = g.MeasureString("X", font2);
    y += sizeX.Height;

    g.DrawString("12 Point Bookdings", font3, brText, x, y);

    // Cleanup
    font1.Dispose();
    font2.Dispose();
    font3.Dispose();
    brText.Dispose();
}
```

Listing 16-4 Setting the font to Bookdings at runtime for button controls [\[Comment 17.54\]](#)

```
private void FormMain_Load(object sender, System.EventArgs e)
{
    Font fontBookDings = new Font("BookDings", 14, FontStyle.Bold);

    // [ << ] button
    cmdRewind.Font = fontBookDings;
    cmdRewind.Text = "2";

    // [ < ] button
    cmdBack.Font = fontBookDings;
    cmdBack.Text = "3";

    // [ || ] button
    cmdPause.Font = fontBookDings;
    cmdPause.Text = "0";

    // [ > ] button
```

```
cmdNext.Font = fontBookDings;  
cmdNext.Text = "4";  
  
// [ >> ] button  
cmdForward.Font = fontBookDings;  
cmdForward.Text = "7";  
  
fontBookDings.Dispose();  
}
```

Enumerating Fonts

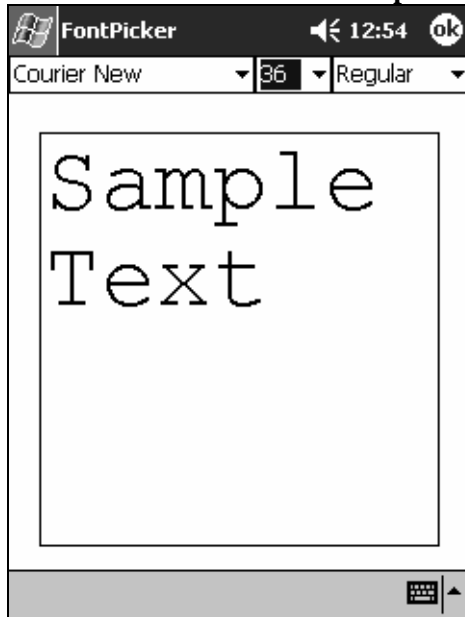
To successfully create a named font, the desired font must be present. The Compact Framework does not provide a means to get a list of available fonts like the Desktop Framework does⁵, so we rely instead on the Win32 approach, known as font enumeration. We are able to enumerate fonts by calling the `EnumFontFamilies` function. There is a complication, however, which requires more work to call this function from a Compact Framework program. This function requires a callback function – essentially a pointer to a function – which is then called for each available font family. While the Desktop Framework supports callback functions, the Compact Framework does not. [\[Comment 17.55\]](#)

What we must do, instead, is put the actual font enumeration code within a Win32 DLL, and then call the DLL from our Compact Framework program. To illustrate this, we present two samples. The first, `FontPicker`, is a Compact Framework program which lets the user pick an available font by face name, size, and style. To help reuse this code, we put the salient parts of the font collection in a separate class: `YaoDurant.Drawing.FontCollection`. The second sample, `FontList`, is a Win32 DLL that does the actual work of enumerating the fonts, and returning the font face names in a manner which is compatible with the Compact Framework. [\[Comment 17.56\]](#)

Sample: FontPicker

Figure 16-6 shows the `FontPicker` sample, with its three `ComboBox` controls for specifying the desired face name from available fonts, the desired size, and the font style. A `TextBox` displays a sample of the resulting font. Listing 16-5 shows the `FontCollection` class, which provides the managed code portion of the font enumeration process. [\[Comment 17.57\]](#)

⁵ In the Desktop Framework, the `Families` property of the `FontFamily` class does the trick.

Figure 16-6 FontPicker with a sample of an available font [\[Comment 17.58\]](#)**Listing 16-5** The FontCollection class [\[Comment 17.59\]](#)

```
// YaoDurant.Drawing.FontCollection.cs - Font enumeration
// wrapper class for FONTLIST.DLL.
//
// Code from _Programming the .NET Compact Framework with C#_
// and _Programming the .NET Compact Framework with VB_
// (c) Copyright 2002-2003 Paul Yao and David Durant.
// All rights reserved.

using System;
using System.Collections;
using System.Runtime.InteropServices;

namespace YaoDurant.Drawing
{
    /// <summary>
    /// Summary description for YaoDurant.
    /// </summary>
    public class FontCollection
    {
        // Constructor.
        public FontCollection()
        {
            IntPtr hFontList = FontList_Create();
            int count = FontList_GetCount(hFontList);
            int i;
            string strFace;
            IntPtr ip;
            for (i = 0; i < count; i++)
            {
                ip = FontList_GetFace(hFontList, i);
            }
        }
    }
}
```

```

        strFace = Marshal.PtrToStringUni(ip);
        m_alFaceNames.Add(strFace);
    }

    FontList_Destroy(hFontList);
}

public void Dispose()
{
    m_alFaceNames.Clear();
}

// P/Invoke declarations for fontlist.dll
[DllImport("fontlist.DLL")]
public static extern IntPtr FontList_Create ();

[DllImport("fontlist.DLL")]
public static extern
int FontList_GetCount (IntPtr hFontList);

[DllImport("fontlist.DLL")]
public static extern
IntPtr FontList_GetFace (IntPtr hFontList, int iFace);

[DllImport("fontlist.DLL")]
public static extern
int FontList_Destroy (IntPtr hFontList);

// Count of available face names.
public int Count
{
    get { return m_alFaceNames.Count; }
}

private ArrayList m_alFaceNames = new ArrayList();

public string this [int index]
{
    get
    {
        if (index < 0 || index >= m_alFaceNames.Count)
            return string.Empty;
        else
            return (string)m_alFaceNames[index];
    }
}

} // class
} // namespace

```

Sample: FontList – Win32 Font Enumeration DLL [\[Comment 17.60\]](#)

We prefer managed code. We call unmanaged (Win32) code only to access features not directly available in managed code. Most of the time, only a few P/Invoke declarations are

needed to connect our managed code to unmanaged functions. Sometimes, though, we must create an intermediary unmanaged code library bridge our managed code to target unmanaged functions. [\[Comment 17.61\]](#)

In the Compact Framework⁶, managed code cannot call unmanaged functions which involve callback functions. The term 'callback function' means that the caller passes a function pointer to the target function. In such cases, the target function uses the function pointer to call back to the caller. Managed code does, in fact, support callback functions. A managed callback function is called a *delegate*. But the Compact Framework only supports managed code-to-managed code delegates; callback functions from unmanaged code back into managed code are not supported. [\[Comment 17.62\]](#)

To solve this problem for font enumeration, we present the `FontList` sample. `FontList` is a dynamic link library which enumerates available fonts, and puts that set of fonts into a linked list. The caller creates, queries, and cleans up the font list through four exported functions: [\[Comment 17.63\]](#)

- `FontList_Create` – Creates the linked list of fonts.
- `FontList_GetCount` – Returns the number of fonts in the list.
- `FontList_GetFace` – Gets the face name of a font at index N.
- `FontList_Destroy` – Free all allocated font list memory.

The complete source listing for `FontList.cpp` appears in listing 16-6. [\[Comment 17.64\]](#)

`FontList` creates a list of font names. A program can take the list of fonts and display them for the user to pick, which we demonstrate in the `FontPicker` sample. Or, a font name can be used to create a font and attach that font to a control. We do this in the `Bookdings` sample, which uses the graphical images from a TrueType font to create push buttons that display non text images. A third use of font names is to create Win32 fonts directly, which is useful to access useful features such as drawing rotated text – which we demonstrate in the `RotateText` sample – or to create ClearType fonts. [\[Comment 17.65\]](#)

Building C++ Projects

You cannot use Visual Studio .NET 2003 to build unmanaged C++ projects for Windows CE, even though this tool can be used to build unmanaged C++ projects for desktop versions of Microsoft Windows. Instead, you need to use one of two special embedded compilers. For Windows CE 3.0 (including Pocket PC and Pocket PC 2002), you must use Embedded Visual C++ 3.0. For Windows CE .NET 4.0 and later, you must use Embedded Visual C++ 4.x. Both tools are available for download from the Microsoft web site. [\[Comment 17.66\]](#)

Listing 16-6 C++ / Win32 font enumeration helper [\[Comment 17.67\]](#)

```
// FontList.cpp : Win32 Font enumeration helper functions.
//
// Code from _Programming the .NET Compact Framework with C#_
// and _Programming the .NET Compact Framework with VB_
// (c) Copyright 2002-2003 Paul Yao and David Durant.
// All rights reserved.

#include "stdafx.h"
```

⁶ By contrast, callback functions from unmanaged to managed code are fully supported in the Desktop Framework. For an example, see Dino Esposito's excellent discussion of Window Hooks in ??????

```

#include "fontlist.h"

BOOL APIENTRY
DllMain( HANDLE hModule,
         DWORD  ul_reason_for_call,
         LPVOID lpReserved)
{
    return TRUE;
}
extern "C"
{

typedef struct __FACENAME
{
    WCHAR FaceName[LF_FACESIZE];
    struct __FACENAME * pNext;
} FACENAME, *LPFACENAME;

typedef struct __HEADER
{
    int signature;
    int count;
    struct __FACENAME * pFirst;
    struct __FACENAME * pLast;
} HEADER, *LPHEADER;

#define HEADER_SIGNATURE 0xf0f0

//-----
//-----

int CALLBACK FontEnumProc(
    const ENUMLOGFONT FAR *lpelf,
    const TEXTMETRIC FAR *lpntm,
    int FontType,
    LPARAM lParam)
{
    LPHEADER pfl = (LPHEADER)lParam;

    // Allocate block to hold facename info.
    LPFACENAME pfnNext = (LPFACENAME)malloc(sizeof(FACENAME));
    wcsncpy(pfnNext->FaceName, lpelf->elfLogFont.lfFaceName);
    pfnNext->pNext = NULL;

    // First time called -- link to header block.
    if (pfl->pFirst == NULL)
    {
        pfl->pFirst = pfnNext;
    }
    else // Link to previous block.
    {
        LPFACENAME pfn = pfl->pLast;
        pfn->pNext = pfnNext;
    }

    // Current block becomes end of list.

```

```

    pfl->pLast = pfnNext;

    pfl->count++; // increment face name count.

    return TRUE;
}

//-----
//-----
HANDLE __cdecl FontList_Create(void)
{
    LPHEADER pfl = (LPHEADER)malloc(sizeof(HEADER));
    if (pfl == NULL)
        return NULL;

    // Init font list.
    pfl->count = 0;
    pfl->pFirst = NULL;
    pfl->pLast = NULL;
    pfl->signature = HEADER_SIGNATURE; 0xf0f0;

    HDC hdc = GetDC(NULL);
    EnumFontFamilies(hdc, NULL, (FONTENUMPROC)FontEnumProc,
        (LPARAM)pfl);
    ReleaseDC(NULL, hdc);

    return (HANDLE)pfl;
}

//-----
//-----
int __cdecl FontList_GetCount(HANDLE hFontList)
{
    LPHEADER pfl = (LPHEADER)hFontList;
    if (pfl->signature != HEADER_SIGNATURE)
        return -1;

    return pfl->count;
}

//-----
//-----
LPTSTR __cdecl FontList_GetFace(HANDLE hFontList, int iFace)
{
    LPHEADER pfl = (LPHEADER)hFontList;
    if (pfl->signature != HEADER_SIGNATURE ||
        pfl->count <= iFace)
        return NULL;

    LPFACENAME pfn = pfl->pFirst;
    if (iFace != 0)
    {
        iFace--;
        for (int i = 0; i <= iFace; i++)
        {
            pfn = pfn->pNext;
        }
    }
}

```

```

    }

    return pfn->FaceName;
}

//-----
//-----
BOOL __cdecl  FontList_Destroy(HANDLE hFontList)
{
    LPHEADER pfl = (LPHEADER)hFontList;
    if (pfl->signature == HEADER_SIGNATURE)
        return NULL;

    LPFACENAME pfnFirst = pfl->pFirst;
    LPFACENAME pfnNext;

    int cItems = pfl->count;
    for (int i = 0; i < cItems; i++)
    {
        pfnNext = pfnFirst->pNext;
        free(pfnFirst);
        pfnFirst = pfnNext;
    }

    free(pfl);

    return TRUE;
}
}
}

```

The actual font enumeration function is `EnumFontFamilies`, which we call in the `FontList_Create` function. The callback function is `FontEnumProc`, which creates a structure of type `HEADER` that points to the start and end of the linked list. Each font face name is stored in a `FACENAME` structure. Our callback function only stores the font face name, which is all we need to access the fonts. But the font enumeration function gets far more details about each of the fonts, as represented by the `ENUMLOGFONT` and `TEXTMETRICS` data structures. You can extend the set of details which are stored by adding additional elements to the `FACENAME` structure. [\[Comment 17.68\]](#)

Native Code Fonts

You can access most font features by creating fonts in managed code. There are a few cases, however, when you must go to unmanaged code – meaning P/Invoke-based native calls – to create fonts. The downside to creating fonts in unmanaged code is that such fonts are only useable through additional calls to unmanaged code. Such fonts, for example, cannot be used with managed code drawing functions; nor can such fonts be connected to Compact Framework controls. The following font features are only available using fonts created in unmanaged code: [\[Comment 17.69\]](#)

- **Rotate text** – as illustrated by the `RotateText` sample, you can create TrueType fonts that are rotated at any angle. [\[Comment 17.70\]](#)
- **ClearType™ fonts** – Microsoft developed the ClearType font technology

- to improve the readability of text on LCD screens. [\[Comment 17.71\]](#)
- **Horizontal fitting** – allows you to select fonts based on the character-cell height instead of traditional em height. This is useful for getting the largest text that can be squeezed within a space, such as the largest readable text that can be drawn in a spreadsheet cell. The em-height is the size of upper-case letters, and does not allow for extra space that is needed for text to look good when horizontally challenged. [\[Comment 17.72\]](#)
 - **Vertical fitting** – to create fonts based on desired average character width instead of height. [\[Comment 17.73\]](#)
 - **Alternative character set** – to create fonts for an alternative character set from the default character set. If, for example, you wish to select a font with a Russian or Greek character set on a US-English device, you might have some trouble unless you explicitly request that character set. [\[Comment 17.74\]](#)

Creating fonts in unmanaged code

To create a font in unmanaged code, you populate a LOGFONT structure and call the `CreateFontIndirect` function. The function returns a font handle, which you pass to other unmanaged functions. LOGFONT, the 'logical font' structure, has fourteen members – 5 integers, 8 byte-size flags, and one character array for the font face name. The last member, the character array, causes a complication because of limitations in how P/Invoke works. [\[Comment 17.75\]](#)

The Compact Framework supports passing strings to unmanaged code. But the passing of strings within structures is not supported. Or, to be more precise, the marshalling of strings within a structure is not *automatically* supported. Instead, to create the LOGFONT structure that we need, we have to *manually* assemble the data structure. This is simpler than it sounds, owing to a set of helper functions found in the `Marshal` class⁷. Listing 16-7 shows the creation of fonts in unmanaged code. This code was taken from the `RotateText` sample. [\[Comment 17.76\]](#)

Listing 16-7 Code fragment showing font creation in unmanaged code [\[Comment 17.77\]](#)

```
//
// Helpful GDI support functions
//
[DllImport("coredll.dll")]
public static extern IntPtr GetDC (IntPtr hWnd);
[DllImport("coredll.dll")]
public static extern int ReleaseDC (IntPtr hWnd, IntPtr hdc);
[DllImport("coredll.dll")]
public static extern int GetDeviceCaps (IntPtr hdc, int iIndex);
[DllImport("coredll.dll")]
public static extern IntPtr CreateFontIndirect (IntPtr lplf);

// The logical font structure -- minus the face name.
public struct LOGFONT
{
    public int lfHeight;
    public int lfWidth;
```

⁷ The fully qualified name is `System.Runtime.InteropServices.Marshal`.

Thank You

Thank you for taking the time to read this preview chapter. We hope it has provided you insights and tips to help with your Compact Framework programming project. You can help us create a better book by clicking the comment link at the end of each paragraph, and sending your comments and suggestions on our review web site.

Preview Chapter Text

Our public review site provides the complete table of contents for the Compact Framework book at this link: <http://www.paul Yao.com/cfbook.htm>. That table of contents contains links to all the preview chapters. The preview chapter provides the complete outline of topics covered in a chapter, and also the first section or two from each chapter.

Complete Chapter Text

You can get the complete text for each chapter, available to readers who register at our web site. Registration is simply and easy – we only ask for an email address. To register, click on this link: <http://www.paul Yao.com/ReaderFeedback/Logon.aspx>.

When you register, you can download the available chapters from the full-text Table of Contents, available at this link: <http://www.paul Yao.com/ReaderFeedback/default.aspx>. We notify registered readers of new chapters – and chapter updates – as they become available.