The Font Manager is a collection of routines and data structures that you can use to manage the fonts your application uses to display and print text. The Font Manager takes care of reading font data from font resources and creating the bitmap images that QuickDraw uses to display text.

This chapter describes how your application can use the Font Manager to find specific fonts and to get font display information, such as the size of the letters, the amount of space between letters, and how sizing and spacing change if the user decides to apply a style such as bold or italic. It also describes how the Font Manager keeps track of fonts and font families.

You need to read this chapter if you are designing a font or if your application uses different font families or allows the user to choose from a variety of fonts. Two types of fonts can be used on the Macintosh computer: bitmapped fonts and TrueType outline fonts. Your application should be able to handle both types. The information in this chapter about outline fonts applies only to TrueType fonts on the Macintosh, and not to other kinds of outline fonts or to TrueType fonts on any other platform.

Almost half of the information in this chapter describes the tables that make up the resources that are used to define fonts on the Macintosh. Unless you are writing an application, such as a font editor, that needs access to these details, you can skip over most of the material in the "The Bitmapped Font ('NFNT') Resource," "The Outline Font ('sfnt') Resource," and "The Font Family ('FOND') Resource" sections of the Reference portion of the chapter.

Before reading this chapter, read the chapter "Introduction to Text on the Macintosh" in this book. General font-related information and programming suggestions are found in the discussion of font handling in that chapter. You should also be familiar with the information in the chapter "QuickDraw Text" in this book. If you are writing a font editor for TrueType fonts, you also need to read the *TrueType Font Format Specification*, available from APDA.

This chapter begins with an overview of the terminology used throughout *Inside Macintosh* to describe fonts and basic Font Manager concepts, including

- characters, character codes, and glyphs
- bitmapped and outline fonts
- font families, font names, and font IDs
- system and application font usage
- font measurements such as left-side bearing, advance width, base line, leading, and kerning

The chapter then describes

- how font resources are used to store fonts
- how the Font Manager finds the information your application or QuickDraw requests
- how the Font Manager and QuickDraw work together to create or alter glyph bitmaps for displaying and printing
- how to use the Font Manager routines to manipulate information about fonts
- the data structures and font resources used by the Font Manager

About Fonts

This section describes the terminology used throughout this chapter to refer to the individual elements of a font, different types of fonts, and the different functions a font can have. Even if you are already familiar with the basic terminology of fonts and typography, you need to know the specific Font Manager concepts described in this section in order to understand the functions of all the Font Manager resources, data structures, and routines.

Characters, Character Codes, and Glyphs

The smallest element in any character set is a **character,** which is a symbol that represents the concept of, for example, a lowercase “b”, the number “2”, or the arithmetic operator “+”. You do not ever see a character on a display device. What you actually see on a display device is a **glyph,** the visual representation of the character. One glyph can represent one character, such as a lowercase “b”; more than one character, such as the “fi” ligature, which is a single glyph that could represent two characters; or a nonprinting character, such as the space character.

When you want to print or display text, you generally refer to characters rather than glyphs. The Font Manager identifies an individual character by a **character code** and provides the glyph for that character to QuickDraw. Character codes for most character sets are single byte values between $00 and $FF; however, the character codes for some large character sets, such as the Japanese character set, are two bytes long. A font designer must supply a **missing-character glyph**—usually an empty rectangle (□)—for characters that are not included in the font. QuickDraw displays this glyph whenever the user presses a key for a character that is not in the font. The Font Manager does not use the missing-character glyph for nonprinting characters, such as the space character, that are included in the `'FONT'`, bitmapped font, and outline font resources.

Although most fonts assign the same glyphs to character code values $00 to $7F, there are differences in which glyphs are assigned to the remaining character codes. For example, the glyph assigned to byte value $F0 ( ) in the Apple Standard Roman character set is not typically included in a font defined for a non-Apple software system. And different regions of the world require different glyphs for their typography, which makes it impossible for any one standard to be complete.

The **character-encoding** scheme was developed to manage the assignment of different glyphs to character codes in different fonts. It names each character and then maps that name into a character code in each font. PostScript fonts that use an encoding scheme that differs from the standard Apple encoding scheme can specify their glyph assignments in the encoding table of the font family resource, which is described in the section "The Style-Mapping Table," beginning on page 4-99. This table contains a collection of assignments of glyph names to character codes. For example, the PostScript name of the character "ñ" is "ntilde"; a font designer can specify in the encoding table that this character is assigned to character code $B9.

The Font Manager uses two types of glyphs: bitmapped glyphs and glyphs from outline fonts. A **bitmapped glyph** is a bitmap—a collection of bits arranged in rows and columns—designed at a fixed point size for a particular display device, such as a monitor or a printer. For example, after deciding that a glyph for a screen font should be so many pixels tall and so many pixels wide, a font designer carefully chooses the individual pixels that constitute the bitmapped glyph. A pixel is the smallest dot the screen can display. The font stores the bitmapped glyph as a picture for the display device.

A glyph from an outline font is a model of how a glyph should look. A font designer uses lines and curves rather than pixels to draw the glyph. The outline, a mathematical description of a glyph from an outline font, has no designated point size or display device characteristic (such as the size of a pixel) attached to it. The Font Manager uses the outline as a pattern to create bitmaps at any size for any display device.

Kinds of Fonts

Each glyph has some characteristics that distinguish it from other glyphs that represent the same character: for example, the shape of the oval, the design of the stem, or whether or not the glyph has a serif. If all the glyphs for a particular character set share certain characteristics, they form a typeface, which is a distinctly designed collection of glyphs. Each typeface has its own name, such as New York, Geneva, or Symbol. The same typeface can be used with different hardware, such as typesetting machines, monitors, or laser printers.

A **style** is a specific variation in the appearance of a glyph that can be applied consistently to all the glyphs in a typeface. Styles available on the Macintosh computer include plain, bold, italic, underline, outline, shadow, condensed, and extended. QuickDraw can add styles such as bold or italic to bitmaps, or a font designer can design a font in a specific style (for instance, Courier Bold).

A **font** refers to a complete set of glyphs in a specific typeface and style—and, in the case of bitmapped fonts, a specific size. **Bitmapped fonts** are fonts of the bitmapped font (`'NFNT'`) resource type or `'FONT'` resource type that provide an individual bitmap for each glyph in each size and style. Courier plain 10-point, Courier bold 10-point, and Courier plain 12-point, for example, are considered three different fonts. If the user requests a font that is not available in a particular size, QuickDraw can alter a bitmapped font at a different size to create the required glyphs. However, this generated bitmap often appears to be irregular in some way.

Outline fonts are fonts of the outline font (`'sfnt'`) resource type that consist of glyphs in a particular typeface and style with no size restriction. TrueType outline fonts are outline fonts that use the Apple TrueType format. The Font Manager can generate thousands of point sizes from the same TrueType outline font: a single outline Courier font can produce Courier 10-point, Courier 12-point, and Courier 200-point.

Identifying Fonts

When multiple fonts of the same typeface are present in system software, the Font Manager groups them into **font families** of the font family (`'FOND'`) resource type. Each font in a font family can be bitmapped or outline. Bitmapped fonts in the same family can be different styles or sizes. For example, an outline plain font for Geneva and two bitmapped fonts for Geneva plain 12-point and Geneva italic 12-point might make up one font family named Geneva, to which a user could subsequently add other sizes or styles.

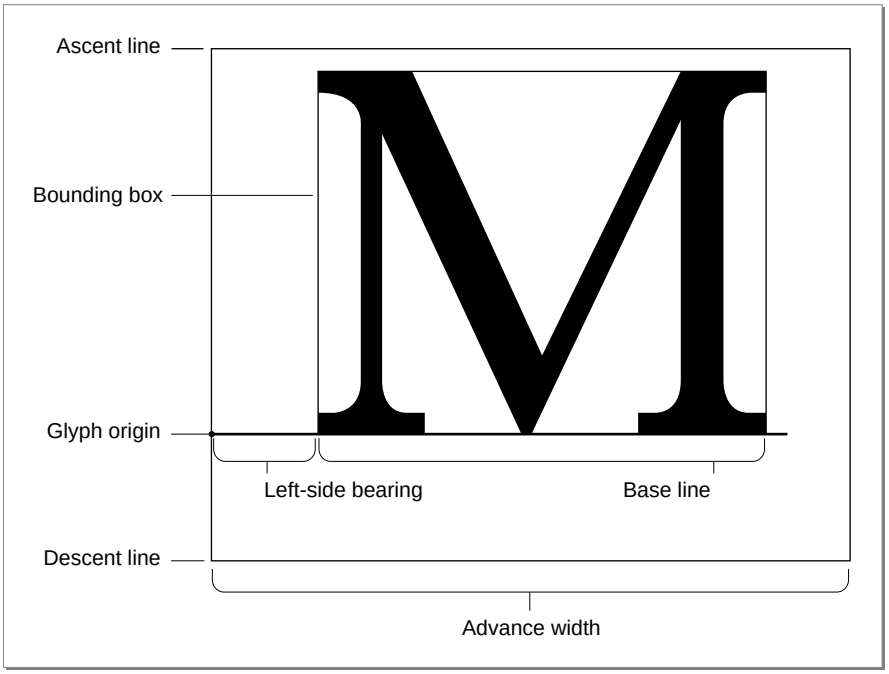
A font has a **font name,** which is stored as a string such as “Geneva” or “New York”. The font name is usually the same name as the typeface from which it was derived. If a font is not in plain style, its style becomes part of the font’s name and distinguishes it from the plain style of that font: for example, “Palatino” and “Palatino Bold”.

A **font family ID** is the resource ID for a font family. Because there are so many font families available for the Macintosh, many families have the same font ID. Therefore, to avoid confusion, when your application stores font references in a document, it should refer to fonts by name and not by number.

Font Measurements

Font designers use specific terms for the measurements of different parts of a glyph, whether outline or bitmapped. Figure 4-1 shows the terms used for the most frequently used measurements.

Figure 4-1 Terms for font measurements



Note
The terms given here are based on the characteristics of the Roman script system, which is associated with most European languages and uses fonts that are meant to be read from left to right. Some other script systems use different definitions for some of these terms. However, QuickDraw always draws glyphs using the glyph origin and advance width measurement, even if the font is read from right to left. ◆

As shown in Figure 4-1, the bounding box of a glyph is the smallest rectangle that entirely encloses the pixels of the bitmap. The **glyph origin** is where QuickDraw begins drawing the glyph. Notice that there is some white space between the glyph origin and the visible beginning of the glyph: this is the left-side bearing of the glyph. The left-side bearing value can be negative, which lessens the spacing between adjacent characters. The **advance width** is the full horizontal measurement of the glyph as measured from its glyph origin to the glyph origin of the next glyph on the line, including the white space on both sides.

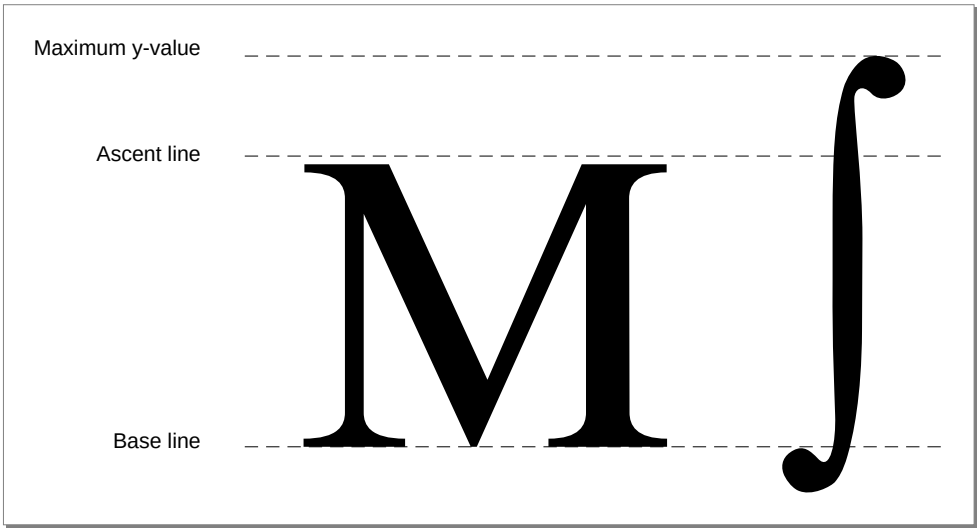
If all of the glyph images in the font were superimposed using a common glyph origin, the smallest rectangle that would enclose the resulting image is the **font rectangle.**

The glyphs of a **fixed-width font** all have the same advance width. Fixed-width fonts are also known as **monospaced fonts.** In Courier, a fixed-width font, the uppercase "M" has the same width as the lowercase "i". In a **proportional font,** different glyphs may have different widths, so the uppercase "M" is wider than the lowercase "i". For example, the proportionally spaced text "iMaGe" has a different appearance from the fixed-width version of the same string "`iMaGe`".

Most glyphs in a font appear to sit on the **base line,** an imaginary horizontal line. The **ascent line** is an imaginary horizontal line chosen by the font's designer that corresponds approximately with the tops of the uppercase letters in the font, because these are generally the tallest commonly used glyphs in a font. The ascent line is the same distance from the base line for all glyphs in the font. The **descent line** is an imaginary horizontal line that usually corresponds with the bottoms of descenders (the tails on glyphs like "p" or "g"), and it's the same distance from the base line for every glyph in the font. The ascent and descent lines are part of the font designer's recommendations about line spacing as measured from base line to base line. All of these lines are horizontal because Roman text is read from left to right, in a straight horizontal line.

For bitmapped fonts, the ascent line marks the maximum y-value and the descent line marks the minimum y-value used for the font. The y-value is the location on the vertical axis of each indicated line: the minimum y-value is the lowest location on the vertical axis and the maximum y-value is the highest location on the vertical axis. For outline fonts, a font designer can create individual glyphs that extend above the ascent line or below the descent line. The integral sign in Figure 4-2, for example, is much taller than the uppercase "M". In this case, the maximum y-value is more important than the ascent line for determining the proper line spacing for a line containing both of these glyphs. You can have the Font Manager reduce such oversized glyphs so that they fit between the ascent and descent lines. See "Preserving the Shapes of Glyphs," which begins on page 4-35, for details.

Figure 4-2 The ascent line and maximum y-value



Font size (or *point size*) indicates the size of a font's glyphs as measured from the base line of one line of text to the base line of the next line of single-spaced text. In the United States, font size is traditionally measured in **points**, and there are 72.27 traditional points per inch. However, QuickDraw and the PostScript language define 1 point to be $\frac{1}{72}$ of an inch, so there are exactly 72 points per inch on the Macintosh.

Previously, the Font Manager required fonts to be less than or equal to 127 points in size, but this restriction no longer applies to any type of font. All bitmaps must fit on the QuickDraw coordinate plane; on a 72-dpi display device, fonts have an upper size limit of 32,767 points.

There is no strict typographical standard for defining a point size: it is often, but not always, the sum of the ascent, descent, and leading values for a font. Point size is used by a font designer to indicate the size of a font relative to other fonts in the same family. Glyphs from fonts with the same point size are not necessarily of the same height. This means that a 12-point font can exceed the measurement of 12 points from the base line of one line of text to the base line of the next.

Note
The Font Manager does not force fonts that are specified as having a certain point size to be of that size. This can have an impact when laying out text in your application, so you need to take it into account. You may need to determine the actual height of the text that you are displaying by using the QuickDraw routine `MeasureText` (which is described in the chapter "QuickDraw Text" in this book) rather than relying on the point size of the font. ◆

Leading (pronounced "LED-ing") is the amount of blank vertical space between the descent line of one line of text drawn using a font and the ascent line of the next line of single-spaced text drawn in the same font. The Font Manager returns the font's suggested leading, which is in pixels, in the `FontMetrics` procedure for both outline and bitmapped fonts. QuickDraw returns similar information in the `GetFontInfo` procedure. Although the designer specifies a recommended leading value for each font, you can always change that value if you need more or less space between the lines of text in your application. The **line spacing** for a font can be calculated by adding the value of the leading to the distance from the ascent line to the descent line of a single line of text.

Although each glyph has a specific advance width and left-side bearing measurement assigned to it, you can change the amount of white space that appears between glyphs. **Kerning** is the process of drawing part of a glyph so that it overlaps another glyph. The period in the top portion of Figure 4-3 stands apart from the uppercase "Y". In the bottom portion of the figure, the word and the period have been kerned: the period has been moved under the right arm of the "Y" and the glyphs of the word are closer. Kerning data—the distances by which pairs of specified glyphs should be moved closer together—is stored in the kerning tables of the different font resources. The kerning table of the outline font resource is described on page 4-84. The kerning table of the font family resource is described on page 4-106.

Figure 4-3 Unkerned text (top) and kerned text (bottom)

About Font Resources

This section provides a general description of the resources used for font management on the Macintosh, including

- an overview of each of the font resource types
- a brief history of the evolution of font resource use on the Macintosh
- information about font family IDs

Font Resource Types

Although the display of different fonts has always been an important aspect of using the Macintosh computer, the need for increased font availability and flexibility has expanded significantly since the introduction of the Macintosh. The built-in font management software has increased in power and complexity to accommodate the expanded needs of users. There used to be only one font resource type, of type `'FONT'`, but it is now out of date. There are now three additional font resource types.

- Bitmapped font (`'NFNT'`) resources describe bitmapped fonts. These bitmapped font resources have an identical structure to the earlier `'FONT'` resources, which they have replaced, but the bitmapped font resources add a more flexible font ID number scheme. This chapter assumes that you are working with bitmapped font resources rather than `'FONT'` resources. The fields of the bitmapped font resource are described in the section "The Bitmapped Font ('NFNT') Resource," which begins on page 4-66.
- Outline font (`'sfnt'`) resources describe outline fonts. The fields and tables of the outline font resource are described in the section "The Outline Font ('sfnt') Resource," which begins on page 4-72.
- Font family (`'FOND'`) resources describe font families, including information such as which fonts are included in the family and the recommended width for a glyph at a given point size. The fields and tables of the font family resource are described in the section "The Font Family ('FOND') Resource," which begins on page 4-90.

Each font that you use is represented by either a bitmapped font or outline font resource (or a `'FONT'` resource, in some cases), and each is part of a font family. A single font family can contain a mixture of bitmapped and outline fonts. The font association table in the font family resource refers to the font resources that the family includes.

Handles to font resources, found in data structures such as the global width table and the `FMOutput` record, can point to either kind of font.

A Brief History of Font Resource Use

The use of font resources has evolved considerably since the early days of the Macintosh computer. Knowing how the changes have evolved can help you understand their use in current software.

The earliest versions of Macintosh system software stored and created all font data in `'FONT'` resources. Font families were created by storing a unique family ID in bits 7–14 of the resource ID of each font in the family. To name a font family, a designer included a `'FONT'` resource with a point size of zero. This method severely restricted the range of both family IDs and point sizes.

With the introduction of the 128K ROM, there were two new resource types: the font family (`'FOND'`) resource, which stores size-independent information for a font family, and the bitmapped font (`'NFNT'`) resource, which has the same internal format as the `'FONT'` resource, but can use any resource ID. Each font family resource names the family and contains a font association table, which consists of a number of individual font entries. Each entry contains a word for the font style, a word for the font size, and a

word for the 'FONT' resource ID or bitmapped font resource ID of the font. This new scheme expanded the range of both font sizes and font family IDs to allow values from 0 to 32,767.

When TrueType outline font support was added for System 7, a new font resource type was added: the outline font resource, the internal format of which is substantially different from that of the bitmapped font resources.

Note
Because of the way that 'FONT' resources were originally constructed, a 'FONT' resource can exist independently of a font family resource. This is not true of bitmapped font and outline font resources, each of which must be associated with a font family resource. ◆

Font Family IDs

Several of the Font Manager routines and data structures make use of a font ID value, which is actually a font family ID value. The valid values for font family IDs, like the resource IDs of all script-specific resources, are subdivided into ranges for each script system, with half of the total range allocated for Roman font families and 512 IDs allocated for each other script system. The ranges for each non-Roman script system are listed in the appendix "International Resources" in this book.

The system software keeps track of two font family IDs. It uses the **system font** for drawing items such as system menus and system dialog boxes. The **application font** is the font that your application will use for text unless specified otherwise by you or the user. In Roman script systems, the system font is Chicago, the system font size is 12 points, and the application font is Geneva. In other script systems, the system and application fonts are defined in the international resources. The Script Manager variables smScriptSysFond, smScriptAppFond, smScriptSysFondSize, and smScriptAppFondSize, which define the system and application fonts and font sizes for each script system, are described in the "Script Manager" chapter in this book.

Font family ID values 0 and 1 are reserved. The system software always maps the system font to font 0 and the application font to font 1. The Roman font family ID range is itself subdivided as shown in Table 4-1.

Table 4-1 Subdivisions of Roman font family IDs

ID range	Use
2–255	Mostly older font families that use the 'FONT' resource numbering method. Do not use these IDs.
256–1023	Reserved numbers that should not be used for family IDs. The Font/DA Mover program uses this range of IDs to resolve font conflicts.
1024–16382	Commercial fonts. This is the range of IDs that all Roman font families should use.
16383	Reserved. Do not use.

The Font Manager defines constants for the system font and the application font, as well as for several of the older font IDs in the range from 2 to 255. These constants are presented here; however, you need to use the older font ID constants with caution, since most of them have become obsolete.

```
CONST
   systemFont = 0;      {the system font}
   applFont = 1;        {the application font}
   newYork = 2;         {hard-coded New York font ID}
   geneva = 3;          {hard-coded Geneva font ID}
   monaco = 4;          {hard-coded Monaco font ID}
   venice = 5;          {hard-coded Venice font ID}
   london = 6;          {hard-coded London font ID}
   athens = 7;          {hard-coded Athens font ID}
   sanFran = 8;         {hard-coded San Francisco font ID}
   toronto = 9;         {hard-coded Toronto font ID}
   cairo = 11;          {hard-coded Cairo font ID}
   losAngeles = 12;     {hard-coded Los Angeles font ID}
   times = 20;          {hard-coded Times Roman font ID}
   helvetica = 21;      {hard-coded Helvetica font ID}
   courier = 22;        {hard-coded Courier font ID}
   symbol = 23;         {hard-coded Symbol font ID}
   mobile = 24;         {hard-coded Mobile font ID}
```

The Script Manager provides functions that allow you to determine which script a font belongs to. For more information, see the chapter "Script Manager" in this book.

Restrictions on the Use of 'FONT' Resources

Since some older applications only work with `'FONT'` resources, you might need to create a `'FONT'` resource to retain compatibility. If this is necessary, you need to follow these restrictions on `'FONT'` resources that are part of a font family.

- The font name and family name must be identical.
- The font ID and font family ID must be identical.
- The resource ID of the font must equal the number produced by concatenating the font ID times 128 with the font size. Remember that fonts stored in `'FONT'` resources are restricted to a point size of less than 128 and to a font ID in the range 0 to 255. The resource ID is computed by the following formula:

```
resourceID := (font ID * 128) + font size;
```

These restrictions ensure that both the 64K ROM found in older Macintosh computers and the newer 128K ROM versions of the Font Manager will associate the font family ID and point size with the proper corresponding `'FONT'` resource ID, whether or not there is a family resource.

Font Resource Tables

The Font Manager takes care of the details of how fonts are stored in resources, reading the resource files when required and building internal representations of the data stored in them. The Font Manager provides routines to interact with fonts, meeting the font-manipulation needs of most application developers.

However, if you are developing an application that requires you to work directly with font resource data, you may need to understand how the font data is stored in resource files. Each font resource consists of a number of tables, each of which has a specific structure. Some of these tables are described in the section "Font Manager Reference" beginning on page 4-39, while others are described in the *TrueType Font Format Specification*.

About the Font Manager

QuickDraw draws text to the screen and, sometimes, to a printer. For its purposes, the glyphs that make up text are simply little images that make up a large, albeit well-ordered, image. QuickDraw uses size information, such as height and width, as it might use that information when arranging any graphic image.

The Font Manager, by contrast, keeps track of detailed font information such as the glyphs' character codes, whether fonts are fixed-width or proportional, and which fonts are related to each other by name.

When QuickDraw needs to draw some text in a particular font, it sends a request for that font to the Font Manager. The Font Manager finds the font or the closest match to it that is available, and sends the font back to QuickDraw with some information that QuickDraw uses for stylistic variations and layout.

Note
Although the terms *glyph* and *character code* have different meanings, QuickDraw routines and data structure fields often use the word *character* for both. Review the purpose of the routine or data structure you're using before deciding whether it handles character codes or glyphs. ◆

How QuickDraw Requests a Font

When your application calls a QuickDraw routine that does anything with text (for example, `DrawText` or `TextFont`), QuickDraw gets information from the Font Manager about the font specified in the current graphics port record and the individual glyphs of that font. The Font Manager performs any necessary calculations and returns the requested information to QuickDraw.

QuickDraw makes its request for font information using a font input record (of data type `FMInput`), which is described on page 4-40. This record contains the font family ID, the size, the style, and the scaling factors of the font request.

QuickDraw makes a font request by filling in a font input record and calling the `FMSwapFont` function. If your application needs to make a font request in the same way that QuickDraw does, you can call `FMSwapFont`. Since responding to a font request can be a lot of work, `FMSwapFont` has been optimized to return as quickly as possible if the request is for the same font as was most recently requested. Building the global width table, which is described in "How the Font Manager Calculates Glyph Widths" on page 4-23, is one of the more time-consuming tasks in this process, which is why the Font Manager maintains a cache of up to 12 width tables.

The Font Manager looks for the font family resource of the requested font and from that determines information about which font it can use to meet the request. If necessary, the Font Manager calls the Resource Manager to read the font.

For certain types of devices, such as a screen or the ImageWriter printer, the Font Manager uses the font characterization table from the device driver to determine any additional information that QuickDraw may need. The font characterization table contains information about the dots per vertical inch and dots per horizontal inch for that device, along with information about the different styles that the device can produce. Non-QuickDraw devices, such as the LaserWriter printer, return an error when the Font Manager requests their font characterization table.

▲ **WARNING**
Never assume that the font resource is a bitmapped font resource or outline font resource. If you need to read information from the resource, you should first call the Resource Manager `GetResInfo` procedure with the handle to the resource. The `GetResInfo` procedure is described in the Resource Manager chapter in *Inside Macintosh: More Macintosh Toolbox*. ▲

How the Font Manager Responds to a Font Request

The Font Manager returns the needed information to QuickDraw in a font output record (of data type `FMOutput`), which is described on page 4-41. This record contains a handle to the font resource that the `FMInput` record requested, information on how different stylistic variations affect the display of the font's glyphs, and the scaling factors.

When the Font Manager gets a request for a font in a font input record, it attempts to find a font family resource for the requested font family by following these steps:

1. The Font Manager checks the global variable `LastFOND`, which contains a handle to the last font family resource used.
2. If the last font family resource used is not the one requested, the Font Manager checks its memory cache, in which it keeps the last 12 width tables used.
3. If the font family resource is not in the cache, the Font Manager calls the Resource Manager `GetResource` function to get the resource.

If the font family resource is available, the Font Manager looks in the font family resource for the ID of the appropriate font resource to match the request. If a font family resource isn't available, the Font Manager follows these steps:

1. The Font Manager looks for a `'FONT'` resource, since such resources can exist without being associated with a font family resource.
2. If it can't find a `'FONT'` resource, the Font Manager looks for the application font.
3. If it can't find the application font, the Font Manager looks for a **neighborhood base font**, which is the font with the lowest font ID for that script system. For fonts numbered below 16384, this is font 0. For fonts above 16384, the Font Manager looks for the nearest font resource that is a multiple of 512 and less than the specified font value.
4. If it can't find a neighborhood font, the Font Manager gets the system font.
5. If it can't find the system font, the Font Manager always uses Chicago 12.

When responding to a font request, the Font Manager first looks for a font family resource of the specified size. It then looks for the stylistic variation that was requested. It does this by assigning weights to the various styles (for example, a weight of 8 for italic, 4 for bold) and then choosing the font whose style weight most closely matches the weight of the requested style.

If the Font Manager cannot find the exact font style that QuickDraw has requested, it uses the closest font style that it does find for that font and QuickDraw then applies the correct style to that font. For example, if an italic version of the requested font cannot be found, the Font Manager returns the plain version of the font and QuickDraw will slant the characters as it draws them. The QuickDraw styles are given in the QuickDraw data type `Style`, which includes the values `bold`, `italic`, `underline`, `outline`, `shadow`, `condense`, and `extend`.

With the additional complication of having both outline and bitmapped fonts available, this process can sometimes produce results other than those that you expected. The Font Manager can be set to favor either outline or bitmapped fonts when both are available to meet a request, as described in "Favoring Outline or Bitmapped Fonts" on page 4-35. The following scenario is one example of how the font that is selected can be a surprise:

1. You have specified that bitmapped fonts are to be preferred over outline fonts when both are available in a specific size.
2. The system software on which your application is running has the bitmap font Times 12 and the outline fonts Times, Times Italic, and Times Bold.
3. The user requests Times Bold 12.
4. The Font Manager chooses the bitmapped version of Times 12 and QuickDraw algorithmically smears it to create the bold effect.

There's not much that you can do about such situations except to be aware that telling the Font Manager to prefer one kind of a font over another has implications beyond what you might initially expect.

How the Font Manager Scales Fonts

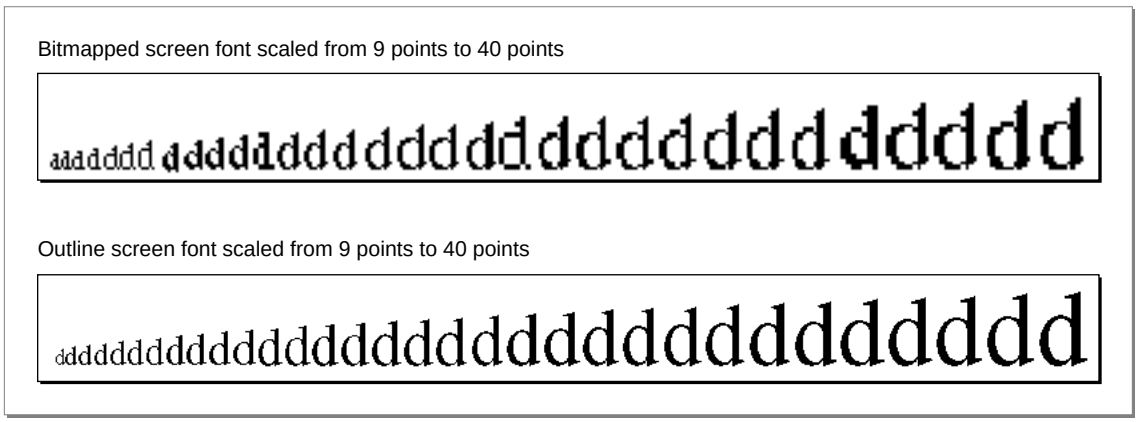
Font scaling is the process of changing a glyph from one size or shape to another. The Font Manager and QuickDraw can scale bitmapped and outline fonts in three ways: changing a glyph's point size, modifying the glyph (but not its point size) for display on a different device, and altering the shape of the glyph.

For bitmapped fonts, the Font Manager does not actually perform scaling of the glyph bitmaps. Instead, the Font Manager finds an appropriate font and computes the horizontal and vertical scaling factors that QuickDraw must apply to scale the bitmaps. QuickDraw performs all modifications of bitmapped font glyphs.

The simplest form of scaling occurs when the Font Manager returns scaling factors for QuickDraw to change a glyph from one point size to another on the same display device. If the glyph is bitmapped and the requested font size is not available, there are certain rules the Font Manager follows to create a new bitmapped glyph from an existing one (see "The Scaling Process for a Bitmapped Font" on page 4-22). If the glyph is from an outline font, the Font Manager uses the outline for that glyph to create a bitmap.

Figure 4-4 shows how the Font Manager and QuickDraw scale a bitmapped font and an outline font from 9 points to 40 points for screen display. The sizes of the bitmapped fonts available to the Font Manager to create all 32 sizes were 9, 10, 12, 14, 18, and 24 points. A single glyph outline produces a smoother bitmap in all point sizes.

Figure 4-4 A comparison of scaled bitmapped and outline fonts



The Font Manager produces better results by scaling glyphs from outline fonts, because it changes the font's original outline to the new size or shape, and then makes the bitmap. Outlines give better results than bitmaps when scaled, because the outlines are intended for use at all point sizes, whereas the bitmaps are not.

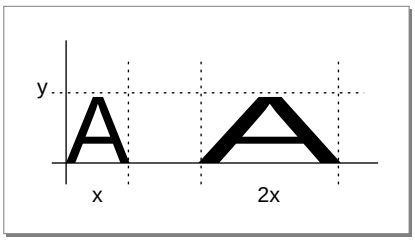
The Font Manager also determines that a glyph must be scaled when moving it from one device to another device with a different resolution: for instance, from the screen to a printer. A bitmap that is 72 pixels high on a 72-dpi screen measures one inch, but on a 144-dpi printer it measures a half inch. In order to print a figure the same size as the original screen bitmap, QuickDraw needs a bitmap twice the size of the original. If there are no bitmaps available in twice the point size of the bitmap that appears on the screen, the Font Manager returns the proper scaling factors, and QuickDraw scales the original bitmap to twice its original size in order to draw it on the printer.

With some QuickDraw calls, your application can also use the Font Manager to explicitly scale a glyph by stretching or shrinking it, which changes the glyph from a familiar point size to something a little stranger—for example, a glyph that is 12 points high but as wide as a whole page of text. Your application tells the Font Manager how to scale a glyph using **font scaling factors,** which are represented as proportions or fractions that indicate how the Font Manager should scale the glyph in the vertical and horizontal directions. The ratio given by the font scaling factors determines whether the glyph grows or shrinks: if the ratio is greater than one, the glyph increases in size, and if it is less than one, the glyph decreases in size. If the font scaling factors are 1-to-1 (1/1) for both horizontal and vertical scaling, the glyph does not change size.

In some circumstances, the Font Manager finds a font and returns different scaling factors to QuickDraw. The scaling factors in a QuickDraw font request tell the Font Manager how much QuickDraw wants to scale the font, and the scaling factors returned by the Font Manager tell QuickDraw how much to actually scale the glyphs before drawing them.

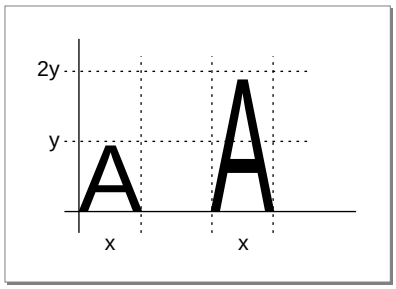
In Figure 4-5, the font scaling factors are 2/1 in the horizontal direction and 1/1 in the vertical direction. The glyph stays the same height, but grows twice as large in width.

Figure 4-5 A glyph stretched horizontally



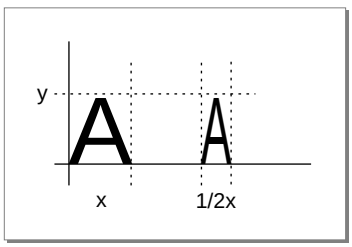
In Figure 4-6, the font scaling factors are 2/1 in the vertical direction and 1/1 in the horizontal direction. The glyph stays the same width, but grows to twice its original height.

Figure 4-6 A glyph stretched vertically



In Figure 4-7, the font scaling factors are 1/1 in the vertical direction and 1/2 in the horizontal direction. The glyph stays the same height but retains only half its width.

Figure 4-7 A glyph condensed horizontally



If the font scaling factors are 2/1 in both directions and the font is an outline font, then the Font Manager computes the size of the glyph as twice the specified size and QuickDraw draws the glyph. With bitmapped fonts, QuickDraw first looks for a bitmap at twice the size of the original before redrawing the glyph at the new point size.

Many routines use the value of the font scaling factors in order to calculate the best measurements for text in the current graphics port record. You can find the current horizontal and vertical scaling factors in the global variables `FScaleHFact` and `FScaleVFact`. The exact value of the font output scaling factors can be found by multiplying the value of the global width table's `hOutput` and `vOutput` fields by the values of the `hFactor` and `vFactor` fields, also of the global width table, respectively. The description of the global width table begins on page 4-43.

The Scaling Process for a Bitmapped Font

Although the Font Manager does not scale the glyph bitmaps of a bitmapped font, it does compute the scaling factors that QuickDraw uses to perform the scaling. The Font Manager computes scaling factors other than 1/1 when the exact point size requested is not available. Font scaling is the default behavior; however, you can disable it, as described below. When the Font Manager cannot find the proper bitmapped font that QuickDraw has requested and font scaling is enabled, it uses the following procedure:

1. The Font Manager looks for a font of the same font family that is twice the size of the font requested. If it finds that font, the Font Manager computes and returns to QuickDraw factors to scale it down to the requested size.
2. The Font Manager looks for a font of the same font family that is half the size of the font requested. If it finds that font, the Font Manager computes and returns to QuickDraw factors to scale it up to the requested size.
3. The Font Manager looks for a font of the same font family that is the next larger size of the font requested. If it finds that font, the Font Manager computes and returns to QuickDraw factors to scale it down to the requested size.
4. The Font Manager looks for a font of the same font family that is the next smaller size of the font requested. If it finds that font, the Font Manager computes and returns to QuickDraw factors to scale it up to the requested size.
5. If the Font Manager cannot find any size of that font family, it returns the application font, system font, or neighborhood base font, as described in the section "How the Font Manager Responds to a Font Request" beginning on page 4-17. The Font Manager computes and returns to QuickDraw the factors to scale that font to the requested size.

You can disable the scaling of bitmapped fonts in your programs by calling the `SetFScaleDisable` procedure. When the Font Manager cannot find the proper bitmapped font that QuickDraw has requested and font scaling is disabled, the Font Manager looks for a different font to substitute instead of scaling. The `SetFScaleDisable` procedure is described on page 4-59.

With scaling disabled, the Font Manager looks for a font with characters with the correct width, which may mean that their height is smaller than the requested size. The Font Manager returns this font and returns scaling factors of 1/1, so that QuickDraw does not scale the bitmaps. QuickDraw draws the smaller font, the widths of which produce the spacing of the requested font. This is faster than font scaling and accurately mirrors the word spacing and line breaks that the document will have when printed, especially if fractional character widths are used. Disabling and enabling of font scaling are described in the section "Using Fractional Glyph Widths and Font Scaling" on page 4-38.

Note
A font request made with scaling disabled does not necessarily return the same result as an identical request with scaling enabled. The widths are sure to be the same only if fractional widths are enabled, the font does not have a glyph-width table, and the font is a member of a family record with a family character-width table. Fractional widths and width tables are discussed in "How the Font Manager Calculates Glyph Widths" on page 4-23. ◆

The Scaling Process for an Outline Font

The Font Manager always scales an outline font in order to produce a bitmapped glyph in the requested size, regardless of whether font scaling for bitmapped fonts is enabled or disabled. An outline font is considered to be the model for all possible point sizes, so the Font Manager is not scaling it from one "real" size to a "created" size, the way it does with a bitmapped font; it is drawing the outline in the requested point size, so that it can then create the bitmapped glyph.

How the Font Manager Calculates Glyph Widths

Integer glyph widths are measurements of a glyph's width that are in whole pixels. Fractional glyph widths are measurements that can include fractions of a pixel. For instance, instead of a glyph measuring exactly 5 pixels across, it may be 5.5 pixels across. Fractional glyph widths allow the sizes of glyphs as stored by the Font Manager to be closer in proportion to the original glyphs of the font than integer widths allow. Fractional widths also make it possible for high-resolution printers to print with better spacing.

You can enable or disable the use of fractional glyph widths in your application, as described in "Using Fractional Glyph Widths and Font Scaling" on page 4-38. As a default, fractional widths are disabled to retain compatibility with older applications.

When using fractional glyph widths, the Font Manager stores the locations of glyphs more accurately than any actual screen can display: since screen glyphs are made up of whole pixels, QuickDraw cannot draw a glyph that is 5.5 pixels wide. The placement of glyphs on the screen matches the eventual placement of glyphs on a page printed by the high-resolution printers more closely, but the spacing between glyphs and words is uneven as QuickDraw rounds off the fractional parts. The extent of the distortion that is visible on the screen depends on the font point size relative to the resolution of the screen.

The Font Manager communicates fractional glyph widths to QuickDraw through the **global width table,** which is a data structure that is allocated in the system heap. The Font Manager fills in this table by accessing data from one of several places:

- Integer glyph widths are taken from the width/offset table of the bitmapped font resource and the horizontal device metrics table of the outline font resource.
- Fractional glyph widths are taken from the glyph-width table in the bitmapped font resource, the horizontal metrics table in the outline font resource, and the family glyph-width table in the font family resource.

The Font Manager looks for width data in the following sequence:

1. For a bitmapped font, it first looks for a font glyph-width table in the font record, which is the record used to represent in memory the data in a bitmapped font resource. For an outline font, it first looks for data in the horizontal metrics table. The width table for bitmapped fonts is described in the section "The Bitmapped Font ('NFNT') Resource," which begins on page 4-66. The width table for outline fonts is described in "The Horizontal Device Metrics Table" on page 4-78.
2. If it doesn't find this table, the Font Manager looks in the font family record for a family glyph-width table. The font family record is used to represent in memory the data in a font family resource. This is described in "The Family Glyph-Width Table" on page 4-98.
3. If the Font Manager doesn't find a family glyph-width table, it derives the global character widths from the integer widths contained in the width/offset table in the bitmapped font record, as described in "The Bitmapped Font ('NFNT') Resource" on page 4-66.

Note

If you need to use different widths than those returned by the global width table, you should change the values in the global width table only. You should never change any values in the font resources themselves. ◆

To use fractional glyph widths effectively, your application must get accurate widths when laying out text. Your application should obtain glyph widths either from the QuickDraw procedure `MeasureText` or by looking in the global width table. The `MeasureText` procedure is described in the chapter "QuickDraw Text" in this book. You can get a handle to the global width table by calling the `FontMetrics` procedure, which is described on page 4-54.

Synthetic Fonts

You may want your application to handle fonts that have a font depth greater than the normal 1-bit depth. (The font depth is the number of bits per pixel; it is specified in bits 2 and 3 of the `fontType` field of the bitmapped font resource, which is described beginning on page 4-70. The Font Manager supports font depths of 1, 2, 4, and 8 bits.) An advantage of using fonts with a larger font depth is that the Font Manager draws bitmapped fonts to the screen considerably faster if the font depth matches the screen depth specified by the user in the Monitors control panel.

The Font Manager can create a **synthetic font** from a `'FONT'` or bitmapped font resource (but not from an outline font resource) by expanding the 1-bit font into a font that matches the current screen depth. The Font Manager creates and maintains synthetic fonts internally, for performance reasons. However, if there is not enough memory to support synthetic fonts, the Font Manager displays a font at 1-bit depth, no matter what the current screen depth is. Font manufacturers can specify that the Font Manager should not expand a font by setting bit 14 of the `fontType` field of the bitmapped font resource.

How the Font Manager Renders Outline Fonts

Outline fonts are stored in an outline font (`'sfnt'`) resource as a collection of outline points. (Don't confuse these outline points with the points that determine point size, or the `Point` data type, which specifies a location in the QuickDraw coordinate plane.) The Font Manager calculates lines and curves between the points, sets the bits that make the bitmap, and then sends the bitmap to QuickDraw for display.

There are two types of outline points: **on-curve points** define the endpoints of lines, and **off-curve points** determine the curve of the line between the on-curve points. Two consecutive on-curve points define a straight line. To draw a curve, the Font Manager needs a third point that is off the curve and between the two on-curve points.

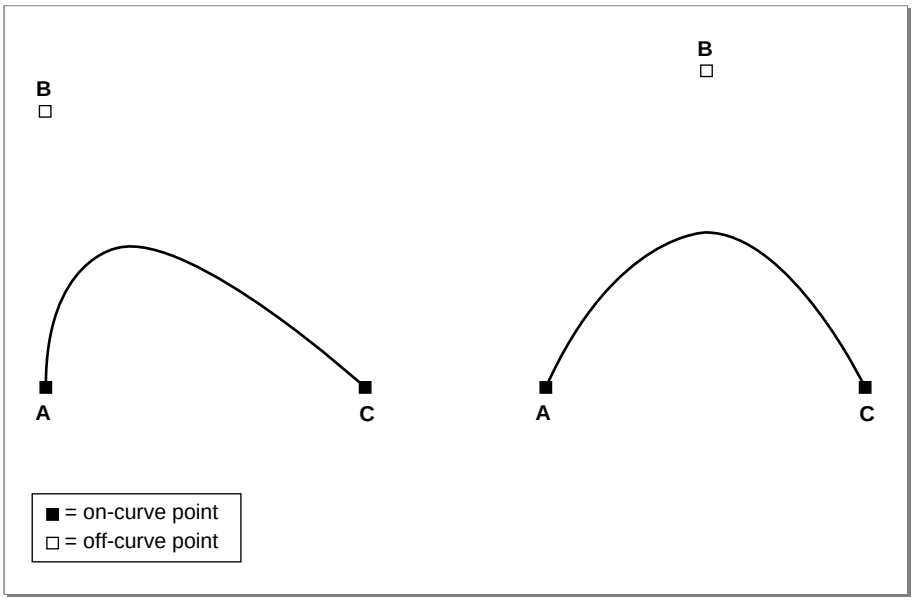
The Font Manager uses this parametric Bézier equation to draw the curves of the glyph from an outline font:

$$F(t) = (1 - t)^2 * A + 2t(1 - t) * B + t^2 * C$$

where `t` ranges between 0 and 1 as the curve moves from point A to point C. A and C are on-curve points; B is an off-curve point.

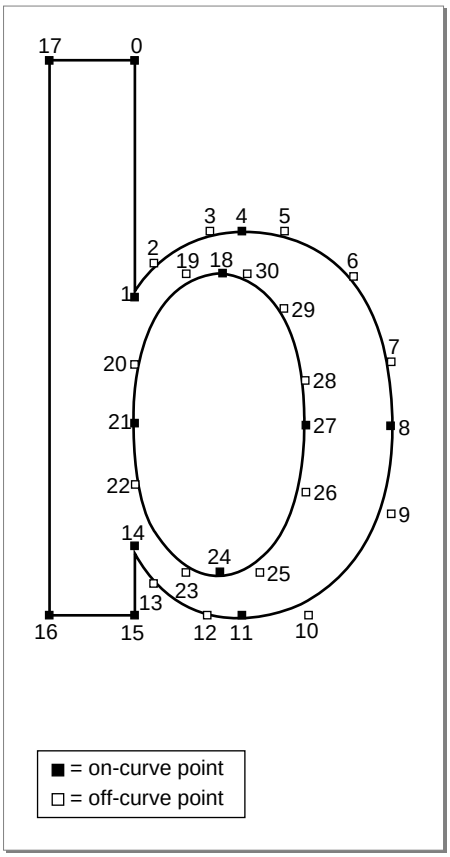
Figure 4-8 shows two Bézier curves. The positions of on-curve points A and C remain constant, while off-curve point B shifts. The curve changes in relation to the position of point B.

Figure 4-8 The effect of an off-curve point on two Bézier curves



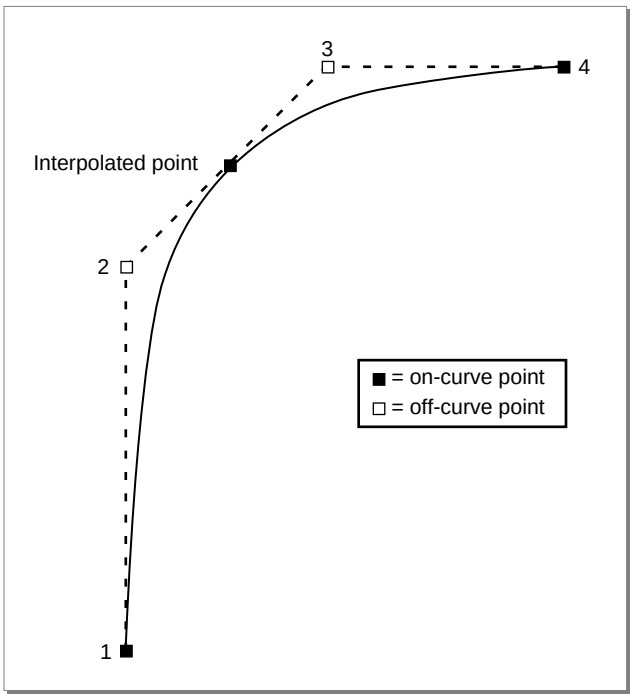
A font designer can use any number of outline points to create a glyph outline. These points must be numbered in a logical order, because the Font Manager draws lines and curves sequentially. This process produces a glyph such as the lowercase “b” in Figure 4-9.

Figure 4-9 An outline with points on and off the curve



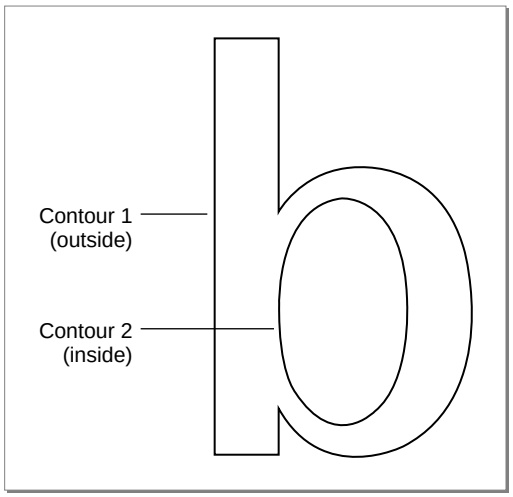
There are several groups of points in Figure 4-9 that include two consecutive off-curve points. For instance, points 2 and 3 are both off-curve. In this case, the Font Manager interpolates an on-curve point midway between the two off-curve points, thereby defining two Bézier curves, as shown in Figure 4-10. Note that this additional on-curve point is used for creation of the glyph only; the Font Manager does not alter the outline font resource's list of points.

Figure 4-10 A curve with consecutive off-curve points



When the Font Manager has finished drawing a closed loop, it has completed one contour of the outline. The font designer groups the points in the outline font resource into contours. In Figure 4-9, the Font Manager draws the first contour in the glyph from point 0 to point 17, and the second contour from point 18 to the end, creating the glyph in Figure 4-11.

Figure 4-11 A glyph from an outline font



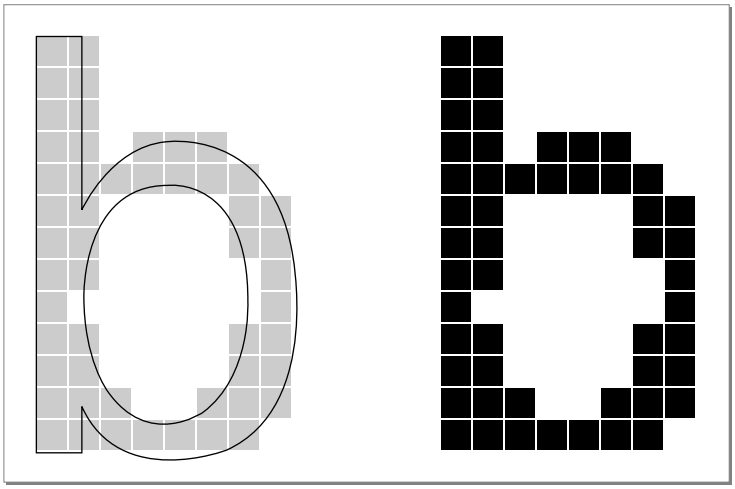
At this stage, the glyph does not have a fixed point size. Remember that point size is measured as the distance from the base line of one line of text to the base line of the next line of single-spaced text. Because the Font Manager has the measurements of the outline relative to the base line and ascent line, it can correlate the measurements with the requested point size and calculate how large the outline should be for that point size.

The Font Manager uses the contours to determine the boundaries of the bitmap for this glyph when it is displayed. For example, the Macintosh computer's screen is a grid made of pixels. The Font Manager fits the glyph, scaled for the correct size, to this grid. If the center of one section of this grid—comparable to a pixel or a printer dot—falls on a contour or within two contours, the Font Manager sets this bit for the bitmap.

Because there are two contours for the glyph in Figure 4-11, the Font Manager begins with pixels at the boundary marked by contour 1 and stops when it gets to contour 2. Some glyphs need only one contour, such as the uppercase "I" in some fonts. Others have three or more contours, such as the ✍ glyph from the ITC Zapf Dingbats font.

If the pixels (or dots) are tiny in proportion to the outline (when resolution is high or the point size of the glyph is large), they fill out the outline smoothly, and any pixels that jut out from the contours are not noticeable. If the display device has a low resolution or the point size is small, the pixels are large in relation to the outline. You can see in Figure 4-12 that the outline has produced an unattractive bitmap. There are gaps and blocky areas that would not be found in the high-resolution versions of the same glyph.

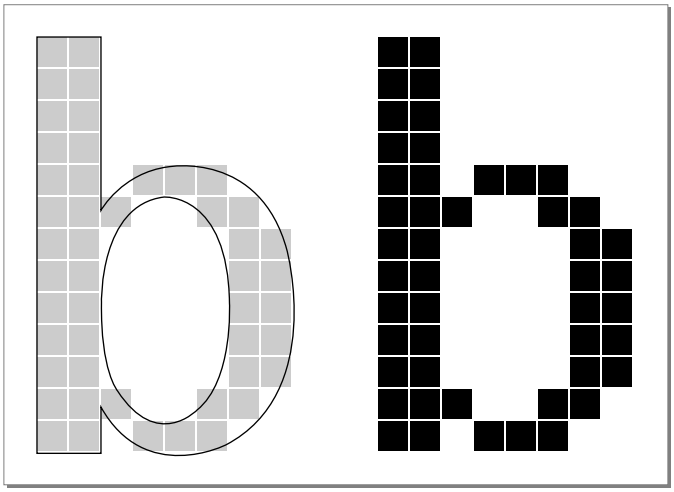
Figure 4-12 An unmodified glyph from an outline font at a small point size



Because the size of the pixels or dots used by the display device cannot change, the outline should adapt in order to produce a better bitmap. To achieve this end, font designers include instructions in the outline font resource that indicate how to change the shape of the outline under various conditions, such as low resolution or small point size. The lowercase "b" outline in Figure 4-13 is the same one depicted in Figure 4-12, except that the Font Manager has applied the instructions to the figure and produced a better bitmapped glyph. These instructions are equivalent to "move these points here" or "change the angle formed by these points." A font designer includes programs consisting of these instructions in certain outline font resource tables, where the Font Manager finds them and executes them under specified conditions. Most applications do not need to use instructions; however, if you want to know more about them, see the book *TrueType Font Format Specification*.

Once the Font Manager has produced the outline according to the design and instructions, it creates a bitmap and sends the bitmap to QuickDraw, which draws it on the screen. The Font Manager then saves the bitmapped glyph in memory (caches it) and uses it the next time the user requests this glyph in this font at this point size.

Figure 4-13 An instructed glyph from an outline font



Using the Font Manager

You can use the Font Manager to take full advantage of the information that fonts contain about their widths and scaling possibilities and present this information to the user. The Font Manager provides routines that give your application control over selecting fonts and measuring the individual glyphs of the font. It also helps you to handle the coexistence of bitmapped and outline versions of fonts.

This section describes how to use the capabilities of the Font Manager in your program to handle tasks, including

- initializing the Font Manager
- adding font names and sizes to the Font menu
- storing font names in your documents
- getting font measurement information
- setting the Font Manager to favor outline or bitmapped fonts
- preserving or scaling the shapes of glyphs
- using the font tables
- getting the system or application font ID
- enabling and disabling font scaling and fractional width use

To initialize the Font Manager, you must call the `InitFonts` procedure. Before calling `InitFonts`, you need to initialize QuickDraw by calling the `InitGraf` procedure, which is described in *Inside Macintosh: Imaging*.

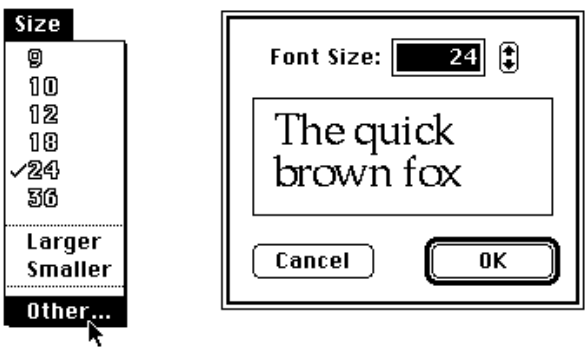
Adding Font Sizes and Names to the Menu

When you use the Menu Manager to add font sizes to a menu, make sure that you construct the menu so that it displays appropriate sizes for both bitmapped and outline fonts. Keep the following guidelines in mind:

- Support all possible font sizes. The maximum point size on the QuickDraw coordinate plane is 32,767 points.
- Provide a short list of the most useful font sizes. For the menu that your application uses to display font sizes, you shouldn't predefine a static list of sizes available to the user or allow the default to be every possible font size, because outline fonts can produce thousands of sizes.
- Provide a method of increasing or decreasing the font size by one point at a time. You can add Larger and Smaller commands, which make choosing slightly different sizes for outline fonts easier for the user. Also, the user should be able to choose any possible point size at any time in a simple manner.
- Place a check next to the current size.
- Display available font sizes in outline style. For a bitmapped font, the `RealFont` function returns `TRUE` if the font is available in the requested point size and `FALSE` if the font is not; you can thereby determine which bitmapped fonts are available. For outline fonts, the `RealFont` function returns `TRUE` for almost any size. The font's designer may decide that there is a lower limit to the point sizes at which the font looks acceptable. The `RealFont` function returns `FALSE` for an outline font if the size requested is smaller than this lower limit.

Figure 4-14 shows one possible method for accomplishing these goals in a menu.

Figure 4-14 A sample Size menu and font size dialog box



To create a menu that displays font names, use the `AddResMenu` procedure. This procedure ensures that any changes to the Font Manager do not affect your application

and that the menu that displays font names is not dependent on how fonts are stored in system software. The `AddResMenu` procedure is documented in the chapter "Menu Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

Storing a Font Name in a Document

When presenting a font to a user, you should always refer to a font by name rather than by font family ID; this prevents several problems that can arise if you use the font family ID. One problem with identifying fonts by font family ID rather than by name is the plethora of font families available for the Macintosh computer. Many share the same font family ID, and even though the font the user wants is present in the System file, another font with the same ID may appear in a font menu. Another problem is that one font family may have different IDs on different computer systems, so that when the application opens the document using this font family on a different computer system, it can't find the proper font, even though it is available, and substitutes another.

If you've stored the name of the font in the document, you can find its font family ID by calling the `GetFNum` procedure, which is described on page 4-52. However, if the font isn't present in the system software when the user opens the document, `GetFNum` returns 0 for the ID. Since 0 is also the system font ID (or the neighborhood base font for the active script system), you need to double-check the name of the font from the document against the name of the system font, as illustrated in Listing 4-1.

Listing 4-1 Checking a font name against the system font name

```
FUNCTION MyGetFontNumber(fontName: Str255;
                         VAR fontNum: Integer): Boolean;
      {MyGetFontNumber returns in the fontNum parameter the number for }
      { the font with the given font name. If there's no such font, }
      { it returns FALSE.}

VAR
   systemFontName: Str255;

BEGIN
   GetFNum(fontName, fontNum);
   IF fontNum = 0 THEN
      BEGIN {either the font was not found, or it is the system font}
      GetFontName(0, systemFontName);
      GetFontNumber := EqualString(fontName, systemFontName, FALSE, FALSE);
      END{ if theNum was not 0, the font is available }
   ELSE
      GetFontNumber := TRUE;
END;
```

Storing a font's name rather than its ID is a more reliable method of finding a font, because the name, unlike the font family ID, does not change from one computer system to another. You may also want to store the checksum of a font (the sum of the values of the bytes in the font data) with its name, to be sure that the version of the font is the same on different computer systems. Listing 4-2 on page 4-76 provides a function for computing a checksum.

If the font versions are different—that is, if the checksums don't match—you should offer users the option of substituting for the font temporarily (until they can find the proper version of the font) or permanently (with another font that is currently available).

If you are developing software for use with non-Roman fonts and the font is not found (by a function such as `MyGetFontNumber` above), you can use the neighborhood base font rather than the system font. The neighborhood base font is the lowest font ID for a particular script.

Getting Font Measurement Information

You sometimes need to get font measurement information for the text font in the current graphics port. The Font Manager provides two routines for this purpose: `FontMetrics` and `OutlineMetrics`. In addition, QuickDraw provides font measurement information in the `GetFontInfo` procedure. You can use this information when arranging the glyphs of one font or several fonts on a line or to calculate adjustments needed when font size or style changes.

The `FontMetrics` procedure can be used on any kind of font, whether bitmapped or outline. It returns the ascent and descent measurements, the width of the largest glyph in the font, and the leading measurements. The `FontMetrics` procedure returns these measurements in a font metrics record (of data type `FMetricRec`), which allows fractional widths, whereas QuickDraw's `GetFontInfo` procedure returns a font information record (of data type `FontInfo`), which uses integer widths. In addition to these four measurements, the font metrics record includes a handle to the global width table, which in turn contains a handle to the font family resource for the current text font. The `GetFontInfo` procedure and the font information record are described in the chapter "QuickDraw Text" in this book. The global width table is described on page 4-36. The `FontMetrics` procedure and the font metrics record are described on page 4-54.

The `OutlineMetrics` function returns measurements for glyphs to be displayed in an outline font. The function returns an error if the text font in the current graphics port is any other kind of font. These measurements include the maximum y-values, minimum y-values, advance widths, left-side bearings, and bounding boxes. (For the definitions of these terms, see the section "About Fonts," which begins on page 4-6.) The `OutlineMetrics` function is described beginning on page 4-56.

For a font of a non-Roman script system that uses an associated font, the font measurements reflect combined values from the current font and the associated font. This is to accommodate the script system's automatic display of Roman characters in the

associated font instead of the current font. See the discussion of associated fonts in the chapter "Introduction to Text on the Macintosh" in this book.

Favoring Outline or Bitmapped Fonts

When a document uses a font that is available as both an outline font and a bitmapped font, the Font Manager has to decide which kind of font to use. Its default behavior is to use the bitmapped font when your application opens the document. This behavior avoids problems with documents that were created on a computer system on which outline fonts were not available. See "How the Font Manager Responds to a Font Request" on page 4-17 for more information.

You can change this default behavior by calling the `SetOutlinePreferred` procedure. If you call `SetOutlinePreferred` with the `outlinePreferred` parameter set to `TRUE`, the Font Manager chooses outline fonts over bitmapped fonts when both are available.

The `GetOutlinePreferred` function returns a Boolean value that indicates which kind of font the Font Manager has been set to favor. You should call this function and save the value that it returns with your documents. Then, when the user opens a document in your application, you can call `SetOutlinePreferred` with that value to ensure that the same fonts are used.

If only one kind of font is available, the Font Manager chooses that kind of font to use in the document, no matter which kind of font is favored. You can determine whether the font being used in the current graphics port is an outline font by using the `IsOutline` function, which is described on page 4-61.

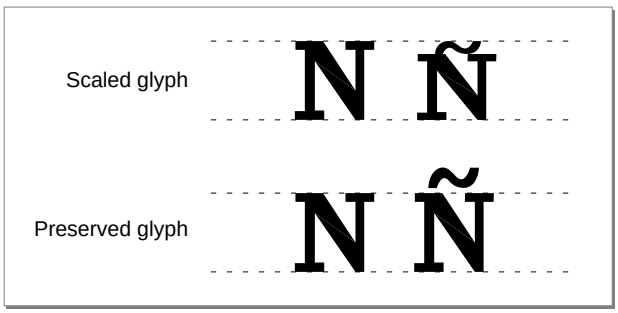
Preserving the Shapes of Glyphs

Most glyphs in an alphabetic font fit between the ascent line and the descent line, which roughly mark (respectively) the tops of the lowercase ascenders and the bottoms of the descenders. Bitmapped fonts always fit between the ascent line and descent line. One aim of outline fonts is to provide glyphs that are more accurate renditions of the original typeface design, and there are glyphs in some typefaces that exceed the ascent or descent line (or both). An example of this type of glyph is an uppercase letter with a diacritical mark: "N" with a tilde produces "Ñ". Many languages use glyphs that extend beyond the ascent line or descent line.

However, these glyphs may disturb the line spacing in a line or a paragraph. The glyph that exceeds the ascent line on one line may cross the descent line of the line above it, where it may overwrite a glyph that has a descender. You can determine whether glyphs from outline fonts exceed the ascent and descent lines by using the `OutlineMetrics` function. `OutlineMetrics` returns the maximum and minimum y-values for whatever glyphs you choose. You can get the values of the ascent and descent lines using the `FontMetrics` procedure. If a glyph's maximum or minimum y-value is greater than, respectively, the ascent or descent line, you can opt for one of two paths of action: you can change the way that your application handles line spacing to accommodate the glyph, or you can change the height of the glyph.

The Font Manager's default behavior is to change the height of the glyph, providing compatibility with bitmapped fonts, which are scaled between the ascent and descent lines. Figure 4-15 shows the difference between an "Ñ" scaled to fit in the same amount of space as an "N" and a preserved "Ñ". The tilde on the preserved "Ñ" clearly exceeds the ascent line.

Figure 4-15 The difference between a scaled glyph and a preserved glyph



You can change this default behavior by calling the `SetPreserveGlyph` procedure. If you call `SetPreserveGlyph` with the `preserveGlyph` parameter set to `TRUE`, the Font Manager preserves the shape of the glyph intended by the font's designer.

The `GetPreserveGlyph` function returns a Boolean value that indicates whether or not the Font Manager has been set to preserve the shapes of glyphs from outline fonts. You should call this function and save the value that it returns with your documents. Then, when the user opens a document in your application, you can call `SetPreserveGlyph` with that value to ensure that glyphs are scaled appropriately.

Using Width Tables

When the Font Manager responds to a request to make a font available, the font resource is loaded into memory and the Font Manager allocates memory for various tables that are needed to use the font. To make font usage more efficient, the Font Manager maintains a cache of the tables for the most recently used fonts, so that it does not have to reread the resources and rebuild the tables more often than necessary. The Font Manager can cache the tables for up to 12 fonts. For outline fonts, the cached information includes the width tables and any bitmaps that have been created from the outlines.

The global width table contains the widths of all the glyphs of one font. If you are measuring text to be displayed on the screen, you can use the QuickDraw procedure `MeasureText` to determine glyph widths; however, if you are printing text and need to determine glyph widths, you have to use widths from the global width table. The `OutlineMetrics` function returns the individual widths of glyphs for an outline font

and the `FontMetrics` procedure returns the width of the largest glyph in a bitmapped font. You can also directly access the global width table, which is defined by the `WidthTable` data type. This data type is described in the section "The Global Width Table" beginning on page 4-43.

To use the global width table, you can get a handle to it from the `FontMetrics` procedure, or you can use the handle stored in the global variable `WidthTabHandle`. The global variable `WidthPtr` contains a pointer to the global width table; however, this variable is reliable only immediately after a call to `FMSwapFont`. Like all pointers to data in handles, `WidthPtr` may become invalid after a call to the Memory Manager. In general, use the `WidthTabHandle` global variable instead of the `WidthPtr` global variable.

The global variable `WidthListHand` is a handle to a list of up to 12 handles to recently used width tables. You can scan this list and look for width tables that match the font family ID, size, and style of the font you wish to measure. If you reach a width-table handle that contains –1 in the `tabFont`, `fID`, and `aFID` fields, that width table is invalid. When you reach a handle that is equal to `NIL`, you have reached the end of the list.

IMPORTANT
Do not use the values from the global width table if your application is running on a computer on which non-Roman script systems are installed. You can check to see if a non-Roman script system is present by calling the `GetScriptManagerVariable` function with a selector of `smEnabled`; if the function returns a value greater than 0, at least one non-Roman script system is present and you need to call `MeasureText` to measure text that is displayed on the screen. Measuring text from a non-Roman script system for printing is handled by the printer driver. ▲

If your application directly manipulates data in a font resource (for example, if your application edits fonts), you may need to flush the Font Manager's cache, so that the cached information reflects any changes that your application makes. The `FlushFonts` function, which is described on page 4-66, erases all of the Font Manager's caches. The Font Manager then rebuilds the cache as new fonts are called into use again.

Normally, fonts are purgeable, which means that the space used for each font's resource information can be released from memory. You can temporarily prevent a font from being purged by locking it with a call to the `SetFontLock` procedure. A subsequent call to `SetFontLock` unlocks the font, allowing the Font Manager to purge it from memory. The `SetFontLock` procedure is described on page 4-65.

If you are calculating the amount of extra width that is added to a glyph as a result of adding a font style, you can choose how you want the Font Manager to determine the extra pixels needed. It can find the information in the style property field of the font family resource or from the style extra field in the Font Manager's internal tables. If the value of the global variable `FDevDisable` is 0, the Font Manager uses the style extra value from its internal tables; if `FDevDisable` is any other value, the Font Manager uses the value from the style property field, which is described on page 4-93.

Getting the System or Application Font ID

When your application does not allow the user to change the font, your application has to tell the Font Manager to use either the system font or the application font. You do this by passing either the `systemFont` constant or the `applFont` constant to the `TextFont` procedure, which is described in the chapter "QuickDraw Text" in this book. The Font Manager maps fonts with other resource IDs to these values, as described in the chapter "Script Manager" in this book.

If you need to know the true font family ID of the system font, you can call the `GetSysFont` function, which checks the global variable `SysFontFam` and returns that resource ID. Similarly, if you want to know the true font family ID of the application font, you can call the `GetAppFont` function or check the global variable `ApFontID`.

The global variable `SysFontSize` contains the point size of the current system font. If you call the `TextSize` procedure (which is described in the chapter "QuickDraw Text" in this book) with a value of 0, the default application font size is used. You can find the default system font size value by calling the `GetDefFontSize` function. If the default system font point size is set to 0, the Font Manager uses 12 as its value.

You can read more about the system and application fonts in the chapter "Introduction to Text on the Macintosh" in this book.

Using Fractional Glyph Widths and Font Scaling

Using fractional glyph widths allows the Font Manager to place glyphs on the screen in a manner that closely matches the eventual placement of glyphs on a page printer by high-resolution printers. (See "How the Font Manager Calculates Glyph Widths" on page 4-23.)

You can enable the use of fractional glyph widths with the `SetFractEnable` procedure. If you set the parameter `fractEnable` to `TRUE`, the Font Manager uses fractional glyph widths. If you set it to `FALSE`, the Font Manager uses integer glyph widths. The Font Manager sets the global variable `FractEnable` to `FALSE` by default. You can find out whether the Font Manager has used fractional widths in the calculations for the global width table or other tables by checking the value of the `UsedFWidths` global variable; if the value is nonzero, the Font Manager used fractional widths.

When a bitmapped font is not available in a specific size, the Font Manager can compute scaling factors for QuickDraw to use to create a bitmap of the requested size. You can set the Font Manager to compute scaling factors for bitmapped fonts by using the `SetFScaleDisable` procedure, which sets the value of the `FScaleDisable` global variable. If you set the `fontScaleDisable` parameter of this procedure to `TRUE`, the Font Manager disables font scaling.

When font scaling is disabled, the Font Manager responds to a request for a font size that is not available by returning a bitmapped font with the requested widths, which may mean that their height is smaller than the requested size. If you set it to `FALSE`, the Font Manager computes scaling factors for bitmapped fonts and QuickDraw scales the glyph bitmaps. The Font Manager sets the global variable `FScaleDisable` to `FALSE` by

default. If the value of this global variable is FALSE, scaling is enabled. (See "The Scaling Process for a Bitmapped Font" on page 4-22.) If scaling is enabled, you can get the current horizontal and vertical scaling factors from the global variable FScaleHFact and FScaleVFact, respectively.

The Font Manager always scales an outline font, regardless of the value of the FScaleDisable global variable.

Fractional glyph widths and font scaling are also described in the chapter "QuickDraw Text" in this book.

Font Manager Reference

This section describes the data structures, routines, and resources provided by the Font Manager.

The "Data Structures" section shows the Pascal data structures used by the bitmapped font resource, the font family resource, and the Font Manager routines. Many, but not all of the tables in these resources have corresponding high-level data structures and detailed descriptions of the tables in each resource type are found in the sections dedicated to each resource.

The "Routines" section describes the routines you can use to get information about the font in the current graphics port record—such as its name, ID, and measurements for layout—or to get a handle to a specific font.

The resources sections describe the resources used by the Font Manager: the bitmapped font ('NFNT') resource, the outline font ('sfnt') resource, and the font family ('FOND') resource. You only need to understand most of the information in this section if you are writing an appliction, such as a font editor, that works directly with font resource data.

Equivalent declarations in the C language for the data structures and routines presented here can be found in the "Summary of the Font Manager" section at the end of this chapter.

Data Structures

This section describes the data structures that you use to provide information to the Font Manager.

You use the font input record to request a font that matches the specified characteristics. The actual characteristics of the font that the Font Manager chooses for the request are returned in a font output record.

You use the global width table record to find the widths of all glyphs in a font.

You use the font record to access the contents of a bitmapped font (`'NFNT'`) resource and a font family record to access the contents of a font family (`'FOND'`) resource. The font family resource includes a number of other tables, each of which has a corresponding data structure, including the font association table record, the bounding-box table record, the family glyph-width table record, the style-mapping table record, and the family kerning table record.

Although some of the resource tables have corresponding data types, many of them do not. If you need to define a data type for a table that does not yet have one defined for it, the resources sections contain pictures of each table, including the length of each table element.

The Font Input Record

The font input record, of data type `FMInput`, is used by QuickDraw to request a font from the Font Manager, as described in the section "How QuickDraw Requests a Font" on page 4-16. You can also use this data type to request a font with the `FMSwapFont` function, which is described on page 4-64.

```
TYPE FMInput =
PACKED RECORD
   family:     Integer;    {font family ID}
   size:       Integer;    {requested point size}
   face:       Style;      {requested font style}
   needBits:   Boolean;    {if bitmaps need to be constructed}
   device:     Integer;    {device driver ID}
   numer:      Point;      {scaling factor numerators}
   denom:      Point;      {scaling factor denominators}
END;
```

Field descriptions

`family`	The font family ID of the requested font.
`size`	The point size of the requested font.
`face`	The requested font style. The defined QuickDraw styles are bold, italic, underline, outline, shadow, condense, and extend.
`needBits`	Indicates whether QuickDraw draws the glyphs. If QuickDraw does not draw the glyphs, as is the case for measurement routines such as `MeasureText`, then the glyph bitmaps do not have to be read or constructed. If QuickDraw draws the glyphs and the font is contained in a bitmapped font resource, all of the information describing the font, including the bit image, is read into memory.
`device`	The high-order byte contains the device driver reference number. The low-order byte is reserved.

`numer` The numerators of the vertical and horizontal scaling factors. (For more information about font scaling, see "How the Font Manager Scales Fonts" on page 4-19.) The `numer` field is of type `Point` and contains two integers: the first is the numerator of the ratio for vertical scaling and the second is the numerator of the ratio for horizontal scaling.

`denom` The denominators of the vertical and horizontal scaling factors. (For more information about font scaling, see "How the Font Manager Scales Fonts" on page 4-19.) The `denom` field is of type `Point` and contains two integers: the first is the denominator of the ratio for vertical scaling and the second is the denominator of the ratio for horizontal scaling.

The Font Output Record

The font output record, of data type `FMOutput`, contains a handle to a font and information about font measurements. It is filled in by the Font Manager upon responding to a font request. You can request a font using the `FMSwapFont` function, which is described on page 4-64.

```
TYPE FMOutput =
PACKED RECORD
   errNum:     Integer;    {reserved for internal use}
   fontHandle: Handle;     {handle to font}
   bold:       Byte;       {for drawing of bold style}
   italic:     Byte;       {for drawing of italic style}
   ulOffset:   Byte;       {for drawing of underline style}
   ulShadow:   Byte;       {for drawing of underline shadow style}
   ulThick:    Byte;       {for drawing of underline thickness}
   shadow:     Byte;       {for drawing of shadow style}
   extra:      SignedByte; {# of pixels added for styles}
   ascent:     Byte;       {ascent measurement of font}
   descent:    Byte;       {descent measurement of font}
   widMax:     Byte;       {maximum width of glyphs in font}
   leading:    SignedByte; {leading value for font}
   fOutCurStyle:
               Byte;       {actual output font style}
   numer:      Point;      {scaling factor numerators}
   denom:      Point;      {scaling factor denominators}
END;
```

Field descriptions

`errNum` Reserved for use by Apple Computer, Inc.

`fontHandle`	A handle to the font resource requested by the font input record, which may either be a bitmapped font or outline font resource. The bitmapped font is described in the section "The Bitmapped Font ('NFNT') Resource," which begins on page 4-66. The outline font is described in the section "The Outline Font ('sfnt') Resource," which begins on page 4-72.
`bold`	Modifies how QuickDraw applies the bold style on the screen and on raster printers. Other display devices may handle styles differently.
`italic`	Modifies how QuickDraw applies the italic style on the screen and on raster printers. Other display devices may handle styles differently.
`ulOffset`	Modifies how QuickDraw applies the underline style on the screen and on raster printers. Other display devices may handle styles differently.
`ulShadow`	Modifies how QuickDraw applies the underline shadow style on the screen and on raster printers. Other display devices may handle styles differently.
`ulThick`	Modifies how QuickDraw applies the thickness of the underline style on the screen and on raster printers. Other display devices may handle styles differently.
`shadow`	Modifies how QuickDraw applies the shadow style on the screen and on raster printers. Other display devices may handle styles differently.
`extra`	The number of pixels by which the styles have widened each glyph.
`ascent`	The ascent measurement of the font. Any algorithmic styles or stretching that may be applied to the font are not taken into account for this value.
`descent`	The descent measurement of the font. Any algorithmic styles or stretching that may be applied to the font are not taken into account for this value.
`widMax`	The maximum width of the font. Any algorithmic styles or stretching that may be applied to the font are not taken into account for this value.
`leading`	The leading assigned to the font. Any algorithmic styles or stretching that may be applied to the font are not taken into account for this value.
`fOutCurStyle`	The actual style being made available for QuickDraw's text drawing, as opposed to the requested style.
`numer`	The numerators of the vertical and horizontal scaling factors. (For more information about font scaling, see "How the Font Manager Scales Fonts" on page 4-19.) The `numer` field is of type `Point` and contains two integers: the first is the numerator of the ratio for vertical scaling and the second is the numerator of the ratio for horizontal scaling.

`denom` The denominators of the vertical and horizontal scaling factors. (For more information about font scaling, see "How the Font Manager Scales Fonts" on page 4-19.) The `denom` field is of type `Point` and contains two integers: the first is the denominator of the ratio for vertical scaling and the second is the denominator of the ratio for horizontal scaling.

The `bold`, `italic`, `ulOffset`, `ulShadow`, `ulThick`, and `shadow` values are all used to communicate to QuickDraw how to modify the way it renders each stylistic variation. Each byte value is taken from the font characterization table of the printer driver and is used by QuickDraw when it draws to a screen or raster printer.

The `ascent`, `descent`, `widMax`, and `leading` values can all be different in this record than the corresponding values in the `FontInfo` record that is produced by the `GetFontInfo` function in QuickDraw. This is because `GetFontInfo` takes into account any algorithmic styles or stretching that QuickDraw performs, while the Font Manager routines do not.

The `numer` and `denom` values are used to designate how font scaling is to be done. The values for these fields in the font output record can be different than the values specified in the font input record. For more information about font scaling, see the section "How the Font Manager Scales Fonts," which begins on page 4-19.

The Global Width Table

The global width table record, of data type `WidthTable`, contains the widths of all the glyphs of one font. The font family, point size, and style of this font are specified in this table. Your application should use the widths found in the global width table for placement of glyphs and words both on the screen and on the printed page. You can use the `FontMetrics` procedure, described on page 4-54, to get a handle to the global width table. However, you should not assume that the table is the same size as shown in the record declaration; it may be larger because of some private system-specific information that is attached to it.

```
Type WidthTable =
PACKED RECORD
   tabData: ARRAY [1..256] OF Fixed;
                           {character widths}
   tabFont:    Handle;     {font record used to build table}
   sExtra:     LongInt;    {extra line spacing}
   style:      LongInt;    {extra line spacing due to style}
   fID:        Integer;    {font family ID}
   fSize:      Integer;    {font size request}
   face:       Integer;    {style (face) request}
   device:     Integer;    {device requested}
   inNumer:    Point;      {scale factors requested}
   inDenom:    Point;      {scale factors requested}
   aFID:       Integer;    {actual font family ID for table}
```

```
   fHand:      Handle;     {family record used to build up table}
   usedFam:    Boolean;    {used fixed-point family widths}
   aFace:      Byte;       {actual face produced}
   vOutput:    Integer;    {vertical scale output value}
   hOutput:    Integer;    {horizontal scale output value}
   vFactor:    Integer;    {vertical scale output value}
   hFactor:    Integer;    {horizontal scale output value}
   aSize:      Integer;    {size of actual font used}
   tabSize:    Integer;    {total size of table}
END;
```

Field descriptions

`tabData` The widths for the glyphs in the font, in standard 32-bit fixed-point format. If a glyph is missing in the font, its entry contains the width of the missing-character glyph.

`tabFont` A handle to the font resource used to build this table.

`sExtra` The average number of pixels by which QuickDraw widens each space in a line of text.

`style` The average number of pixels by which QuickDraw widens a line of text after applying a style.

`fID` The font family ID of the font represented by this table. This is the ID that was used in the request to build the table. It may be different from the ID of the font family that was used, which is indicated by the `aFID` field.

`fSize` The point size that was originally requested for the font represented by this table. The actual size used is specified in the `aSize` field.

`face` The font style that was originally requested for the font represented by this table. The actual style used is specified in the `aFace` field.

`device` The device ID of the device on which these widths may be used.

`inNumer` The numerators of the vertical and horizontal scaling factors. The `numer` field is of type `Point` and contains two integers: the first is the numerator of the ratio for vertical scaling and the second is the numerator of the ratio for horizontal scaling.

`inDenom` The denominators of the vertical and horizontal scaling factors. The `denom` field is of type `Point` and contains two integers: the first is the denominator of the ratio for vertical scaling and the second is the denominator of the ratio for horizontal scaling.

`aFID` The font family ID of the font family actually used to build this table. If the Font Manager could not find the font requested, this value may be different from the value of the `fID` field.

`fHand` The handle to the font family resource used to build this table.

`usedFam` Set to `TRUE` if the fixed-point family glyph widths were used rather than integer glyph widths.

`aFace` The font style of the font whose widths are contained in this table.

`vOutput`	The factor by which glyphs are to be expanded vertically in the current graphics port. This is a 16-bit fixed-point number, with the integer part in the high-order byte and a fractional part in the low-order byte.
`hOutput`	The factor by which glyphs are to be expanded horizontally in the current graphics port. This is a 16-bit fixed-point number, with the integer part in the high-order byte and a fractional part in the low-order byte.
`vFactor`	The factor by which widths of the chosen font, after a style has been applied, have been increased vertically in the current graphics port. This is a 16-bit fixed-point number, with the integer part in the high-order byte and a fractional part in the low-order byte. The value of the `vFactor` field is not used by the Font Manager.
`hFactor`	The factor by which widths of the chosen font, after a style has been applied, have been increased horizontally in the current graphics port. This is a 16-bit fixed-point number, with the integer part in the high-order byte and a fractional part in the low-order byte.
`aSize`	The size of the font actually used to build this table. Both the point size and the font used to build this table may be different from the requested point size and font. If font scaling is disabled, the Font Manager may use a size different from the size requested and add more or less space to approximate the appearance of the font requested. See "The Scaling Process for a Bitmapped Font" on page 4-22 for more information.
`tabSize`	The total size of the global width table.

Multiplying the values of the `hOutput` and `vOutput` fields by the values of the `hFactor` and `vFactor` fields, respectively, gives the font scaling. (Because the value of the `vFactor` field is ignored, the Font Manager multiplies the value of the `vOutput` field by 1.) The product of the value of the `hOutput` field and an entry in the global width table is the scaled width for that glyph.

The Font Manager gathers data for the global width table from one of three data structures:

1. The Font Manager looks in the font resource for a table that stores fractional glyph widths. For bitmapped fonts, the Font Manager uses the glyph-width table of the bitmapped font resource (described on page 4-70). For outline fonts, the Font Manager uses the advance width and left-side bearing values in the horizontal metrics table of the outline font (described on page 4-83). In both cases, the values are stored in 16-bit fixed format, with the integer part in the high-order byte and the fractional part in the low-order byte.
2. If there is no glyph-width table in the font resource, the Font Manager looks for the font family's glyph-width table in the font family resource, which contains fractional widths for a hypothetical 1-point font. The Font Manager calculates the actual values by multiplying these widths by the requested font size. The font family's glyph-width table is described in "The Family Glyph-Width Table" on page 4-98.
3. If there is no glyph-width table in the font family resource, and if the font is contained in a bitmapped font resource, the Font Manager derives the glyph widths from the

integer widths contained in the glyph-width table of the bitmapped font resource, which is described on page 4-70. There is no corresponding table for the outline font resource.

Your application should obtain glyph widths either from the global width table or from the QuickDraw procedure `MeasureText`. The `MeasureText` procedure works only with text to be displayed on the screen, not with text to be printed. You can get the individual widths of glyphs of an outline font using the `OutlineMetrics` function. The `FontMetrics` procedure returns only the width of the largest glyph in a font contained in a bitmapped font resource.

IMPORTANT

Do not use the values from the global width table if your application is running on a computer on which non-Roman script systems are installed. You can check to see if a non-Roman script system is present by calling the `GetScriptManagerVariable` function with a selector of `smEnabled`; if the function returns a value greater than 0, at least one non-Roman script system is present and you need to call `MeasureText` to measure text that is displayed on the screen. Measuring text from a non-Roman script system for printing is handled by the printer driver. ▲

For more information about the `MeasureText` procedure, see the chapter "QuickDraw Text" in this book. The `FontMetrics` procedure is described on page 4-54 and the `OutlineMetrics` function is described on page 4-56.

The Font Record

The font record, of data type `FontRec`, describes the format of the bitmapped font (`'NFNT'`) resource (and, likewise, the `'FONT'` resource). It is shown here as a guide to the format of the resource. The font record is not used directly by any Font Manager routines.

```
TYPE FontRec =
RECORD
   fontType:   Integer;    {font type}
   firstChar:  Integer;    {character code of first glyph}
   lastChar:   Integer;    {character code of last glyph}
   widMax:     Integer;    {maximum glyph width}
   kernMax:    Integer;    {maximum glyph kern}
   nDescent:   Integer;    {negative of descent}
   fRectWidth: Integer;    {width of font rectangle}
   fRectHeight:Integer;    {height of font rectangle}
   owTLoc:     Integer;    {offset to width/offset table}
   ascent:     Integer;    {maximum ascent measurement}
   descent:    Integer;    {maximum descent measurement}
   leading:    Integer;    {leading measurement}
   rowWords:   Integer;    {row width of bit image in 16-bit wds}
END;
```

The fields of the font record are described in the section "The Bitmapped Font ('NFNT') Resource," beginning on page 4-66.

The Font Family Record

The font family record, of data type `FamRec`, describes the format of the font family (`'FOND'`) resource. It is shown here as a guide to the format of the resource. The font family record is not used directly by any Font Manager routines.

```
TYPE FamRec =
RECORD
   ffFlags:    Integer;    {flags for family}
   ffFamID:    Integer;    {family ID number}
   ffFirstChar:Integer;    {ASCII code of first character}
   ffLastChar: Integer;    {ASCII code of last character}
   ffAscent:   Integer;    {maximum ascent for 1-pt font}
   ffDescent:  Integer;    {maximum descent for 1-pt font}
   ffLeading:  Integer;    {maximum leading for 1-pt font}
   ffWidMax:   Integer;    {maximum glyph width for 1-pt font}
   ffWTabOff:  LongInt;    {offset to family glyph-width table}
   ffKernOff:  LongInt;    {offset to kerning table}
   ffStylOff:  LongInt;    {offset to style-mapping table}
   ffProperty: ARRAY [1..9] OF Integer;
                           {style properties info}
   ffIntl:     ARRAY [1..2] OF Integer;
                           {for international use}
   ffVersion:  Integer;    {version number}
END;
```

The fields of the font family record are described in the section "The Font Family ('FOND') Resource," beginning on page 4-90.

The Font Association Table Record

The font association table record, which is part of the font family resource, maps a point size and style to a specific font that is part of the family. The table record, of data type `FontAssoc`, consists of a count of the entries in the table and is followed by the entry records.

```
TYPE FontAssoc =
RECORD
   numAssoc:   Integer;    {number of entries - 1}
   {entries:   ARRAY[0..n] of AsscEntry;}
END;
```

Each entry in the font association table is a font association entry record, of data type `AsscEntry`.

```
TYPE AsscEntry =
RECORD
   fontSize:   Integer;    {point size of font}
   fontStyle:  Integer;    {style of font}
   fontID:     Integer;    {font resource ID}
END;
```

The fields of the font association table and font association table entry record are described in the section "The Font Association Table," beginning on page 4-95.

The Family Glyph-Width Table Record

The font family glyph-width table record, which is part of the font family resource, is used to specify glyph widths for the font family on a per-style basis. The table record, of data type `WidTable`, consists of a count of the entries in the table and is followed by the entry records.

```
TYPE WidTable =
RECORD
   numWidths:  Integer;    {number of entries - 1}
END;
```

Each entry in the family glyph-width table is a family glyph-width table entry record, of data type `WidEntry`, which specifies a style and a variable length array of glyph-width values.

```
TYPE WidEntry =
RECORD
   widStyle:   Integer;    {style code}
   {widths:    ARRAY[0..n] of Fixed;}
END;
```

The fields of the family glyph-width table and family glyph-width table entry records are described in the section "The Family Glyph-Width Table," beginning on page 4-98.

The Style-Mapping Table Record

The style-mapping table record, which is part of the font family resource, provides information that is used by printer drivers to implement font styles. Each font family can have its own character encoding and its own set of font suffix names for style designations. Each style of a font has its own name, typically created by adding a style suffix to the base name of the font, as described in the section "The Style-Mapping Table" beginning on page 4-99. The table record, of data type `StyleTable`, provides information about the font class and is followed by the font name suffix subtable and the font glyph-encoding subtable.

```
TYPE StyleTable =
RECORD
   fontClass:  Integer;       {font class of this font family}
   offset:     LongInt;       {offset to glyph-encoding subtable}
   reserved:   LongInt;       {reserved}
   indexes:    PACKED ARRAY [0..47] OF SignedByte;
                              {indexes into the font suffix name }
                              { table that follows this table}
END;
```

The font suffix name subtable record, of data type `NameTable`, contains the base name and suffixes for a font family.

```
TYPE NameTable =
RECORD
   stringCount:   Integer;    {string count}
   baseFontName:  Str255;     {base font name}
   {suffix strings}           {strings}
END;
```

The fields of the style-mapping table and font suffix name subtable are described in the section "The Style-Mapping Table," beginning on page 4-99.

The Font Family Kerning Table Record

The font family kerning table record, which is part of the font family resource, contains a number of kerning subtable entries, with different subtables for different stylistic variations. The table record, of data type `KernTable`, consists of a count of the entries in the table and is followed by the entry records.

```
TYPE KernTable =
RECORD
   numKerns:   Integer;    {number of subtable entries}
   {kernPairs: ARRAY[0..n] of KernEntry}
END;
```

Each kerning subtable record entry, of data type `KernEntry`, contains kerning pair records for a specific stylistic variation of the font family. It is followed by the kerning pair records.

```
TYPE KernEntry =
RECORD
   kernStyle:  Integer; {kern style}
   kernLength: Integer; {entry length}
   {kernRec:   ARRAY[0..n] of kernPair}
                        {the kerning data records}
END;
```

Each kerning pair record, of data type `KernPair`, specifies a kerning value for a pair of glyphs. Each glyph in the pair is specified by its ASCII character code.

```
TYPE KernPair =
RECORD
   kernFirst:  CHAR;    {Code of 1st character of kerned pair}
   kernSecond: CHAR;    {Code of 2nd character of kerned pair}
   kernWidth:  Integer; {kerning value in 1pt fixed format}
END;
```

The fields of the kerning table, kerning subtable entry, and kerning pair records are described in the section "The Font Family Kerning Table," beginning on page 4-106.

Routines

This section describes the routines you use to initialize the Font Manager and to get information about a font, such as its name, ID, or measurements. It also describes the routines you use to get a handle to a font and to control aspects of the way the Font Manager manipulates fonts, such as font scaling and fractional widths.

▲ **WARNING**
Do not change any data in a font or in any of the font data structures or global variables (except where expressly noted). ▲

Initializing the Font Manager

Typically, the Font Manager has already been initialized when your application opens. However, you should call the `InitFonts` procedure before you call any Font Manager or QuickDraw text routines, just to be sure.

InitFonts

The `InitFonts` procedure initializes the Font Manager.

```
PROCEDURE InitFonts;
```

DESCRIPTION

If the system font isn't already in memory, the `InitFonts` procedure reads it into memory. Call this procedure once, after calling the `InitGrafs` procedure and before calling any other Font Manager routines or any Toolbox routine that calls the Font Manager.

ASSEMBLY–LANGUAGE INFORMATION

The trap macro for the `InitFonts` procedure is

Trap macro

```
_InitFonts
```

Getting Font Information

The Font Manager provides three routines that allow you to get basic information about a font. The `GetFontName` procedure gets the name of a font family with a specified ID, and the `GetFNum` procedure gets the font family ID for a font with a specified name. The `RealFont` function tells you whether a font is available in a specific point size.

GetFontName

The `GetFontName` procedure gets the name of a font family that has a specified family ID number.

```
PROCEDURE GetFontName (familyID: Integer; VAR theName: Str255);
```

`familyID` The font family ID.

`theName` On output, this parameter contains the font family name for the font family specified in `familyID`.

DESCRIPTION

Given a font family ID, the `GetFontName` procedure returns, in the parameter `theName`, the name of the font family. If the font specified in the `familyID` parameter does not exist, `theName` contains an empty string.

ASSEMBLY–LANGUAGE INFORMATION

The trap macro for the `GetFontName` procedure is

Trap macro

`_GetFontName`

GetFNum

The `GetFNum` procedure gets the font family ID for a specified font family name.

```
PROCEDURE GetFNum (theName: Str255; VAR familyID: Integer);
```

`theName`	The font family name.
`familyID`	On output, this parameter contains the font family ID for the font family specified in `theName`.

DESCRIPTION

Given a font name, the `GetFNum` procedure returns, in the `familyID` parameter, the font family ID for the font family. If the font specified in the parameter `theName` does not exist, `familyID` contains 0.

ASSEMBLY–LANGUAGE INFORMATION

The trap macro for the `GetFNum` procedure is

Trap macro

`_GetFNum`

RealFont

The `RealFont` function determines whether a font is available or is intended for use in a specified size.

```
FUNCTION RealFont (fontNum: Integer; size: Integer): Boolean;
```

`fontNum`	The font family ID.
`size`	The font size requested.

DESCRIPTION

The `RealFont` function returns `TRUE` if the requested size of a font is available. `RealFont` first checks for a bitmapped font from the specified family. If one is not available, `RealFont` checks next for an outline font. If neither kind of font is available, `RealFont` returns `FALSE`.

If an outline font exists for the requested font family, `RealFont` normally considers the font to be available in any requested size; however, the font designer can include instructions in the font that outlines should not be used at certain point sizes, in which case the `RealFont` function will consider the font unavailable and return `FALSE`. The Font Manager determines whether the size is valid by testing the value of the smallest readable size element of the font family header table, which is described in "The Font Header Table," beginning on page 4-79.

ASSEMBLY–LANGUAGE INFORMATION

The trap macro for the `RealFont` function is

Trap macro

`_RealFont`

Using the Current, System, and Application Fonts

The `GetDefFontSize`, `GetSysFont`, and `GetAppFont` functions return the current values of the global variables that contain the default size of the system font, the ID number of the system font, and the ID number of the application font.

GetDefFontSize

The `GetDefFontSize` function determines the default size of the system font.

```
FUNCTION GetDefFontSize: Integer;
```

DESCRIPTION

The `GetDefFontSize` function returns the current value of the global variable `SysFontSize` if that value is not 0. If the value of `SysFontSize` is 0, `GetDefFontSize` returns 12 as the default font size.

At system startup, the value of `SysFontSize` is set to 0.

SEE ALSO

You can determine the preferred size for either the system font or the application font of any enabled script system by calling the `GetScriptManagerVariable` function. See the chapter "Script Manager" in this book.

GetSysFont

The `GetSysFont` function determines the font family ID of the current system font.

```
FUNCTION GetSysFont: Integer;
```

DESCRIPTION

The `GetSysFont` function returns the current value of the global variable `SysFontFam`, which is the font family ID of the current system font. This is the font family ID that has been mapped to 0 by the system software.

GetAppFont

The `GetAppFont` function returns the font family ID of the current application font.

```
FUNCTION GetAppFont: Integer;
```

DESCRIPTION

The `GetAppFont` function returns the current value of the global variable `ApFontID`, which is the font family ID of the current application font. This is the font family ID that has been mapped to 1 by the system software.

Getting the Characteristics of a Font

The `FontMetrics` procedure and the `OutlineMetrics` function both return font measurement information. The `FontMetrics` procedure returns the ascent and descent measurements, width of the largest glyph, and leading measurements for either a bitmapped or an outline font. The `OutlineMetrics` function returns measurements for text to be written in an outline font.

FontMetrics

The `FontMetrics` procedure gets fractional measurements for the font, size, and style specified in the current graphics port.

```
PROCEDURE FontMetrics (VAR theMetrics: FMetricRec);
```

`theMetrics`
A font metrics record that contains the font measurement information, in fractional values.

DESCRIPTION

The `FontMetrics` procedure returns measurements for the ascent, descent, leading, and width of the largest glyph in the font for the font, size, and style specified in the current graphics port. `FontMetrics` returns this information in a font metrics record.

The font metrics record (of data type `FMetricRec`) contains a handle to the global width table, which in turn contains a handle to the associated font family resource for the current font (the font in the current graphics port). It also contains the values of four measurements for the current font.

```
Type FMetricRec =
RECORD
   ascent: Fixed;    {baseline to top}
   descent: Fixed;   {baseline to bottom}
   leading: Fixed;   {leading between lines}
   widMax: Fixed;    {maximum glyph width}
   wTabHandle: Handle; {handle to global width table}
END;
```

Field descriptions

`ascent`	The measurement from the baseline to the ascent line of the font.
`descent`	The measurement from the baseline to the descent line of the font.
`leading`	The measurement from the descent line to the ascent line below it.
`widMax`	The width of the largest glyph in the font.
`wTabHandle`	A handle to the global width table.

You can determine the line height, in pixels, by adding the values of the `ascent`, `descent`, and `leading` fields of the font metrics record.

The `FontMetrics` procedure is similar to QuickDraw's `GetFontInfo` procedure, except that `FontMetrics` returns fractional values for greater accuracy in high-resolution printing. `FontMetrics` also does not take into account any additional widths that are added by QuickDraw when it applies styles to the glyphs in a font.

ASSEMBLY–LANGUAGE INFORMATION

The trap macro for the `FontMetrics` procedure is

Trap macro

```
_FontMetrics
```

SEE ALSO

The `GetFontInfo` procedure is described in the chapter "QuickDraw Text" in this book.

OutlineMetrics

The `OutlineMetrics` function gets font measurements for a block of text to be drawn in a specified outline font.

```
FUNCTION OutlineMetrics (byteCount: Integer; textPtr: UNIV Ptr;
                         numer,denom: Point;
                         VAR yMax: Integer; VAR yMin: Integer;
                         awArray: FixedPtr; lsbArray: FixedPtr;
                         boundsArray: RectPtr): OSErr;
```

`byteCount` The number of bytes in the block of text that you want measured.

`textPtr` A pointer to the block of text that you want measured.

`numer` The numerators of the vertical and horizontal scaling factors. The `numer` parameter is of type `Point`, and contains two integers: the first is the numerator of the ratio for vertical scaling, and the second is the numerator of the ratio for horizontal scaling. The Font Manager applies these scaling factors to the current font in order to calculate the measurements for glyphs in the block of text.

`denom` The denominators of the vertical and horizontal scaling factors. The `denom` parameter is of type `Point`, and contains two integers: the first is the denominator of the ratio for vertical scaling, and the second is the denominator of the ratio for horizontal scaling. The Font Manager applies these scaling factors to the current font in order to calculate the measurements for glyphs in the block of text.

`yMax` On output, this is the maximum y-value for the text. Pass `NIL` in this parameter if you don't want this value returned.

`yMin` On output, this is the minimum y-value for the text. Pass `NIL` in this parameter if you don't want this value returned.

`awArray` A pointer to an array that, on output, is filled with the advance width measurements for the glyphs being measured. These measurements are in pixels, based on the point size and font scaling factors of the current font. There is an entry in this array for each glyph that is being measured.

The `awArray` parameter is of type `FixedPtr`. The `FixedPtr` data type is a pointer to an array, and each entry in the array is of type `Fixed`, which is 4 bytes in length. Multiply `byteCount` by 4 to calculate the memory you need in bytes.

If the `FractEnable` global variable has been set to `TRUE` through the `SetFractEnable` procedure, the values in `awArray` have fractional character widths. If `FractEnable` has been set to `FALSE`, the Font Manager returns integer values for the advance widths, with 0 in the decimal part of the values.

`lsbArray` A pointer to an array that is, on output, filled with the left-side bearing measurements for the glyphs being measured. The measurements are in pixels, based on the point size of the current font. There is an entry in this array for each glyph that is being measured.

The `lsbArray` parameter is of type `FixedPtr`. The `FixedPtr` data type is a pointer to an array, and each entry in the array is of type `Fixed`, which is 4 bytes in length. Multiply `byteCount` by 4 to calculate the memory you need in bytes.

Left-side bearing values are not rounded.

`boundsArray`

A pointer to an array that is, on output, filled with the bounding boxes for the glyphs being measured. Bounding boxes are the smallest rectangles that fit around the pixels of the glyph. There is an entry in this array for each glyph that is being measured.

The coordinate system used to describe the bounding boxes is in pixel units, centered at the glyph origin, and with a vertical positive direction upwards, which is the opposite of the QuickDraw vertical orientation.

The `boundsArray` parameter is of type `RectPtr`. The `RectPtr` data type is a pointer to QuickDraw's `Rect` data type, which is 8 bytes in length. Multiply `byteCount` by 8 to calculate the memory you need in bytes. Allocate the memory needed for the array and pass a pointer to the array in the `boundsArray` parameter.

DESCRIPTION

The `OutlineMetrics` function computes the maximum y-value, minimum y-value, advance widths, left-side bearings, and bounding boxes for a block of text. It uses the font, size, and style specified in the current graphics port. You can use these measurements when laying out text. You may need to adjust line spacing to accommodate exceptionally large glyphs.

The `OutlineMetrics` function works for outline fonts only and is the preferred method for measuring text that is drawn with an outline font.

When you are using `OutlineMetrics` to compute advance width values, left-side bearing values, or bounding boxes, you need to bear in mind that when a text block contains 2-byte characters, not every byte in the `awArray`, `lsbArray`, and `boundsArray` structures is used. Each of these arrays is indexed by the glyph index; thus, if you have five characters in a string and two of them are 2-byte characters, only the first five entries in each array contains a value. Call the `CharByte` function (described in the chapter "Script Manager" in this book) to determine how many characters there are in the text block, and ignore the unused array entries (which occur at the end of each array).

If you don't want `OutlineMetrics` to compute one of these three values, pass `NIL` in the applicable parameter. Otherwise, allocate the amount of memory needed for the array and pass a pointer to it in this parameter.

ASSEMBLY–LANGUAGE INFORMATION

The trap macro and selector for the `OutlineMetrics` procedure are

Trap macro	Selector
`_FontDispatch`	`$7008`

SEE ALSO

The terms used for measuring text, including advance width, left-side bearing, and bounding box, are described in the section "Font Measurements," which begins on page 4-8. Scaling of fonts and the use of the font scaling factors are described in the section "How the Font Manager Scales Fonts," which begins on page 4-19.

Enabling Fractional Glyph Widths

The `SetFractEnable` procedure enables or disables fractional glyph widths. When fractional glyph widths are enabled, the Font Manager can determine the locations of glyphs more accurately than is possible with integer widths, as described in the section "How the Font Manager Calculates Glyph Widths" on page 4-23.

SetFractEnable

The `SetFractEnable` procedure enables or disables fractional glyph widths.

```
PROCEDURE SetFractEnable (fractEnable: Boolean);
```

`fractEnable`
Specifies whether fractional widths or integer widths are to be used to determine glyph measurements. A value of `TRUE` indicates fractional glyph widths; a value of `FALSE` indicates integer glyph widths.

DESCRIPTION

The `SetFractEnable` procedure establishes whether or not the Font Manager provides fractional glyph widths to QuickDraw, which then uses them for advancing the pen during text drawing. If you set the `fractEnable` parameter to `TRUE`, the Font Manager provides fractional glyph widths. If you set it to `FALSE`, the Font Manager provides integer glyph widths.

The `SetFractEnable` procedure assigns the value that you specify in the `fractEnable` parameter to the global variable `FractEnable`.

The Font Manager defaults to integer widths to ensure compatibility with existing applications.

Disabling Font Scaling

The `SetFScaleDisable` procedure enables or disables font scaling of bitmapped glyphs. When font scaling is enabled, the Font Manager can scale a bitmapped glyph that is present in the System file to imitate the appearance of a bitmapped glyph in another point size that is not present. For more information about scaling of bitmapped fonts, see "The Scaling Process for a Bitmapped Font" on page 4-22.

SetFScaleDisable

The `SetFScaleDisable` procedure enables or disables the computation of font scaling factors by the Font Manager for bitmapped glyphs.

```
PROCEDURE SetFScaleDisable (fontScaleDisable: Boolean);
```

`fontScaleDisable`
Specifies whether bitmapped fonts are to be scaled. A value of `TRUE` indicates that font scaling is disabled; a value of `FALSE` indicates that font scaling is enabled.

DESCRIPTION

The `SetFScaleDisable` procedure establishes whether or not the Font Manager computes font scaling factors for bitmapped fonts. If you set the `fontScaleDisable` parameter to `TRUE`, the Font Manager disables font scaling, which means it responds to a request for a font size that is not available by computing font scaling factors of 1/1 and returning a smaller, unscaled bitmapped font with the widths of the requested size. If you set the `fontScaleDisable` parameter to `FALSE`, the Font Manager computes scaling factors for bitmapped fonts.

QuickDraw performs the actual scaling of glyph bitmaps for bitmapped fonts by using the font scaling factors computed and returned by the Font Manager.

As a default, the Font Manager scales fonts to ensure compatibility with existing applications.

ASSEMBLY–LANGUAGE INFORMATION

The trap macro for the `SetFScaleDisable` procedure is

Trap macro

`_SetFScaleDisable`

Favoring Outline Fonts Over Bitmapped Fonts

The `SetOutlinePreferred` procedure causes either outline fonts or bitmapped fonts to be favored when the Font Manager receives a font request. You can use the `GetOutlinePreferred` function to find out whether outline or bitmapped fonts are currently favored. You can use the `IsOutline` function to find out if the font used in the current graphics port is an outline font.

SetOutlinePreferred

The `SetOutlinePreferred` procedure sets the preference for whether to use bitmapped or outline fonts when both kinds of fonts are available.

```
PROCEDURE SetOutlinePreferred (outlinePreferred: Boolean);
```

`outlinePreferred`
Specifies whether the Font Manager chooses an outline font or a bitmapped font when both are available to fill a font request. A value of `TRUE` indicates an outline font; a value of `FALSE` indicates a bitmapped font.

DESCRIPTION

If an outline font and a bitmapped font are both available for a font request, the default behavior for the Font Manager is to choose the bitmapped font, in order to maintain compatibility with documents that were created on computer systems on which outline fonts were not available. The `SetOutlinePreferred` procedure sets the Font Manager's current preference for either bitmapped or outline fonts when both are available. If you want the Font Manager to choose outline fonts over any bitmapped font counterparts, set the `outlinePreferred` parameter to `TRUE`; if you want it to choose bitmapped fonts, set the `outlinePreferred` parameter to `FALSE`.

If only outline fonts are available, the Font Manager chooses them regardless of the setting of `outlinePreferred`; if only bitmapped fonts are available, they are chosen. The Font Manager chooses bitmapped versus outline fonts on a size basis, before it takes stylistic variations into account, which can lead to unexpected results. For further information, see "How the Font Manager Responds to a Font Request," beginning on page 4-17.

The preference you set is valid only during the current session with your application. The `outlinePreferred` parameter does not set a global variable.

ASSEMBLY–LANGUAGE INFORMATION

The trap macro and routine selector for the `SetOutlinePreferred` procedure are

Trap macro	Routine selector
`_FontDispatch`	`$7001`

GetOutlinePreferred

The `GetOutlinePreferred` function determines whether outline or bitmapped fonts are to be favored when the Font Manager receives a font request.

```
FUNCTION GetOutlinePreferred: Boolean;
```

DESCRIPTION

The `GetOutlinePreferred` function returns the value of the Font Manager's current preference for outline or bitmapped fonts. If `GetOutlinePreferred` returns `TRUE`, then the Font Manager chooses the outline font when both an outline font and a bitmapped font are available for a particular request. If `GetOutlinePreferred` returns `FALSE`, then the Font Manager chooses the bitmapped font when both types are available.

Use the `SetOutlinePreferred` procedure to change this preference.

ASSEMBLY–LANGUAGE INFORMATION

The trap macro and routine selector for the `GetOutlinePreferred` function are

Trap macro	Routine selector
`_FontDispatch`	`$7009`

IsOutline

The `IsOutline` function determines if the Font Manager chooses an outline font for the current graphics port to meet the specified scaling factors.

```
FUNCTION IsOutline (numer: Point; denom: Point): Boolean;
```

`numer` The numerators of the vertical and horizontal scaling factors. The `numer` parameter is of type `Point`, and contains two integers: the first is the numerator of the ratio for vertical scaling, and the second is the numerator of the ratio for horizontal scaling.

`denom` The denominators of the vertical and horizontal scaling factors. The `denom` parameter is of type `Point`, and contains two integers: the first is the denominator of the ratio for vertical scaling, and the second is the denominator of the ratio for horizontal scaling.

DESCRIPTION

The `IsOutline` function returns `TRUE` if the Font Manager would choose an outline font for the current graphics port. The Font Manager uses the font scaling factors specified in the `numer` and `denom` parameters, as well as the current preference (as set by the `SetOutlinePreferred` procedure) to make a decision as to which font to use.

ASSEMBLY–LANGUAGE INFORMATION

The trap macro and routine selector for the `IsOutline` function are

Trap macro	Routine selector
`_FontDispatch`	`$7000`

Scaling Outline Fonts

The `SetPreserveGlyph` procedure determines whether a glyph from an outline font is displayed as designed or whether the Font Manager scales the glyph to fit between the ascent and descent lines. These two behaviors are discussed in "Preserving the Shapes of Glyphs" on page 4-35. You can use the `GetPreserveGlyph` function to find out whether glyphs from outline fonts are to be scaled.

SetPreserveGlyph

The default behavior for the Font Manager is to scale a glyph from an outline font so that it fits between the ascent and descent lines; however, this alters the appearance of the glyph. The `SetPreserveGlyph` procedure changes this behavior temporarily so that the Font Manager does not scale oversized glyphs.

```
PROCEDURE SetPreserveGlyph (preserveGlyph: Boolean);
```

`preserveGlyph`
Specifies whether or not glyphs from an outline font are scaled to fit between the ascent and descent lines.

DESCRIPTION

The `SetPreserveGlyph` procedure establishes how the Font Manager treats glyphs that do not fit between the ascent and descent lines for the current font. If you set the value of the `preserveGlyph` parameter to `TRUE`, the measurements of all glyphs are preserved, which means that your application may have to alter the leading between lines in a document if some of the glyphs extend beyond the ascent or descent lines. If you set the value of the `preserveGlyph` parameter to `FALSE`, all glyphs are scaled to fit between the ascent and descent lines.

You can determine the current behavior of the Font Manager in this regard by calling the `GetPreserveGlyph` function. To ensure that documents have the same appearance whenever they are opened, you need to call `GetPreserveGlyph` and save the value that it returns with your documents and restore it each time a document is displayed by your application.

ASSEMBLY–LANGUAGE INFORMATION

The trap macro and routine selector for the `SetPreserveGlyph` procedure are

Trap macro	Routine selector
`_FontDispatch`	`$700A`

GetPreserveGlyph

The `GetPreserveGlyph` function determines whether the Font Manager preserves the shapes of glyphs from outline fonts.

```
FUNCTION GetPreserveGlyph: Boolean;
```

DESCRIPTION

The `GetPreserveGlyph` function returns a Boolean value indicating whether the Font Manager preserves the shapes of glyphs from outline fonts. Your application can set the value of this variable with the `SetPreserveGlyph` procedure. If `GetPreserveGlyph` returns `TRUE`, the Font Manager preserves glyph shapes; if `GetPreserveGlyph` returns `FALSE`, then the Font Manager scales glyphs to fit between the ascent and descent lines for the font in use.

ASSEMBLY–LANGUAGE INFORMATION

The trap macro and routine selector for the `GetPreserveGlyph` function are

Trap macro	Routine selector
`_FontDispatch`	`$700B`

Accessing Information About a Font

The `FMSwapFont` function gets a handle to a font, and some information about that font. It is used extensively by system software to access fonts.

FMSwapFont

The `FMSwapFont` function returns a handle to a font and information about that font. This function is used by QuickDraw and other parts of the system software to access font handles.

```
FUNCTION FMSwapFont (inRec: FMInput): FMOutPtr;
```

`inRec` A font input record, which contains the font family ID, the style requested, scaling factors, and other information.

DESCRIPTION

The `FMSwapFont` function takes a font request and returns a pointer to a font output record. `FMSwapFont` is the heart of the Font Manager: it does all of the hard work of preparing font data for text measuring and text drawing.

The `inRec` parameter specifies the characteristics of the font that is requested. QuickDraw fills in the fields of the `CurFmInput` global variable and passes that record in this parameter.

The font output record contains a handle to a font resource that fulfills the font request, along with information about the font, such as the ascent, descent, and leading measurements. You supply the `FMSwapFont` function with the font request in the `inRec` parameter, using a font input record, and the Font Manager returns the font handle and the other information in a font output record.

QuickDraw calls the `FMSwapFont` function every time a QuickDraw text routine is used. If you want to call the `FMSwapFont` function in order to get a handle to a font resource or information about that font, you must build a font input record and then use the pointer returned to access the resulting font output record.

You cannot assume that the font resource pointed to by the `fontHandle` field of the font output record returned by this function is of any particular type, such as `'NFNT'` or `'sfnt'`. If you need to access specific information in the font resource, call the Resource Manager procedure `GetResInfo` with the handle returned in the font output record to determine the font resource type.

IMPORTANT

The pointer to the font output record returned as the value of `FMSwapFont` points to a record allocated in low memory by the Font Manager. The same record is reused for each call made to `FMSwapFont`. Do not free the memory allocated for this record. ▲

ASSEMBLY–LANGUAGE INFORMATION

The trap macro for the `FMSwapFont` function is

Trap macro

`_FMSwapFont`

SEE ALSO

For more information about the font input record, see "How QuickDraw Requests a Font" on page 4-16. For more information about the font output record, see "How the Font Manager Responds to a Font Request" on page 4-17. For descriptions of the records themselves, see "The Font Input Record" on page 4-40 and "The Font Output Record" on page 4-41.

The `GetResInfo` procedure is described in the Resource Manager chapter in *Inside Macintosh: More Macintosh Toolbox*.

Handling Fonts in Memory

The Font Manager provides two routines that allow you to manipulate fonts in memory. The `SetFontLock` procedure makes a font resource, which is normally purgeable data in memory, unpurgeable. The `FlushFonts` function erases the Font Manager's memory caches, including resource data and any width tables the Font Manager may have built.

SetFontLock

The `SetFontLock` procedure makes the most recently used font unpurgeable. You can use this procedure when you want a font to remain in memory for the sake of efficiency.

```
PROCEDURE SetFontLock (lockFlag: Boolean);
```

`lockFlag` Specifies whether or not the current font is considered purgeable.

DESCRIPTION

If you set the `lockFlag` parameter to `TRUE`, the `SetFontLock` procedure makes the most recently used font resource unpurgeable, and reads it into memory if it isn't already there. If you set the `lockFlag` parameter to `FALSE`, the `SetFontLock` procedure releases the memory occupied by the most recently used font by calling the `ReleaseResource` procedure.

The font considered to be the most recently used is the one referenced by the font output record in low memory, which is filled in by the `FMSwapFont` function. This is often, but not always, the font in which text has most recently been drawn. Since both QuickDraw and your application program can call `FMSwapFont`, you have to be careful about which

font has most recently been used in a call to that function. To ensure that you are locking the font that you want to lock, explicitly call `FMSwapFont` immediately before calling `SetFontLock`.

ASSEMBLY–LANGUAGE INFORMATION

The trap macro for the `SetFontLock` procedure is

Trap macro
`_SetFontLock`

SEE ALSO

The `ReleaseResource` procedure is described in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.

FlushFonts

The `FlushFonts` function erases the Font Manager's memory caches.

```
FUNCTION FlushFonts: OSErr;
```

DESCRIPTION

The `FlushFonts` function erases all of the Font Manager's memory caches. Your application doesn't need this function unless it directly manipulates data in the outline font resource. Font Manager caches include the width tables, the bitmaps created from the outlines of the outline font resource, the calculations for the outlines, and a small cache of font family resources that have been read into memory.

ASSEMBLY–LANGUAGE INFORMATION

The trap macro and routine selector for the `FlushFonts` function are

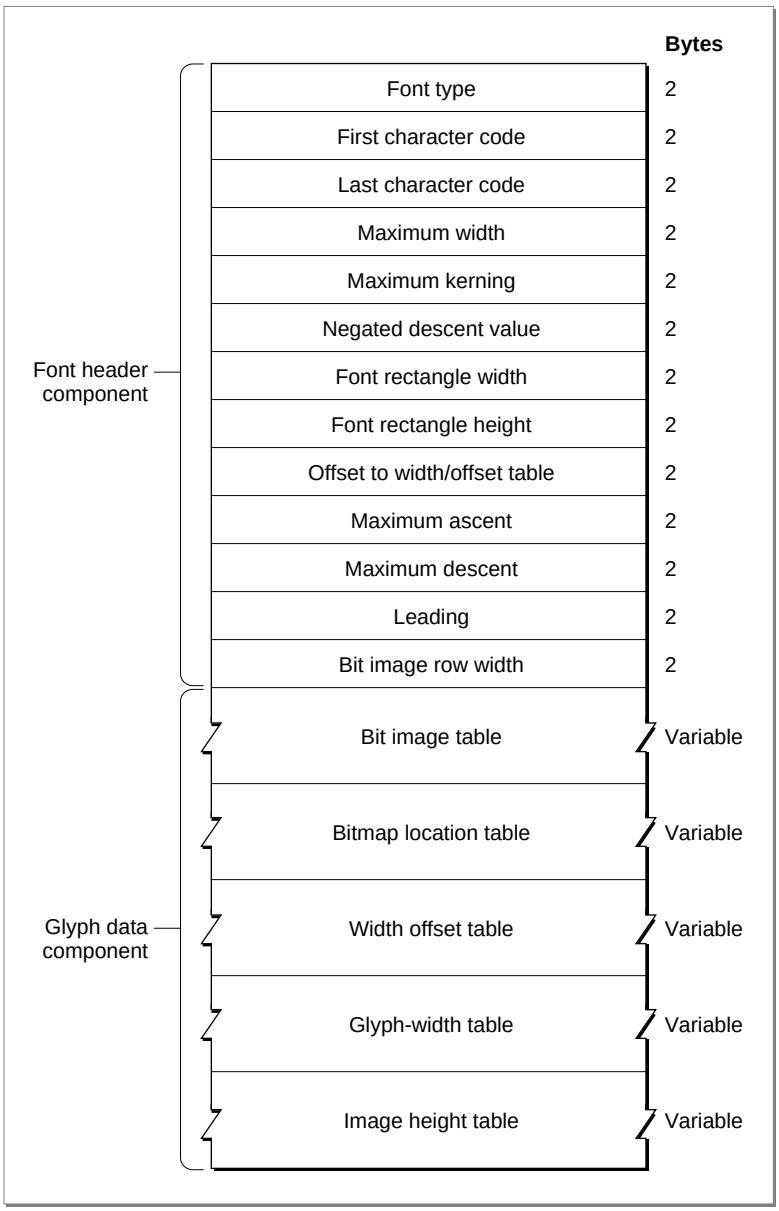
Trap macro	Routine selector
`_FontDispatch`	`$700C`

The Bitmapped Font ('NFNT') Resource

The bitmapped font (`'NFNT'`) resource describes a bitmapped font—a font whose glyphs are represented by bit images. The structure of the bitmapped font resource is identical to that of the older `'FONT'` resource, which can be used for bitmapped fonts as well; however, the bitmapped font resource has a more flexible ID numbering scheme and is preferred over the `'FONT'` resource.

The bitmapped font resource consists of a header component, which describes the font, and a glyph data information component, which contains the definitions of the glyphs in the font. The header component of this resource is represented by the `FontRec` data type, the declaration of which is shown in the section "The Font Record," beginning on page 4-46. The structure of this resource is shown in Figure 4-16.

Figure 4-16 The bitmapped font (`'NFNT'`) resource



The bitmapped font header component consists of the elements listed below, each of which corresponds to a field in the `FontRec` data type.

- Font type. An integer value that is used to specify the general characteristics of the font, such as whether it is fixed-width or proportional, whether the optional image-height and glyph-width tables are attached to the font, and information about the font depth and colors. This value is represented by the `fontType` field in the `FontRec` data type. For the meaning of the bits in this field, see "The Font Type Element" on page 4-70.
- First character code. An integer value that specifies the ASCII character code of the first glyph in the font. This value is represented by the `firstChar` field in the `FontRec` data type.
- Last character code. An integer value that specifies the ASCII character code of the last glyph in the font. This value is represented by the `lastChar` field in the `FontRec` data type.
- Maximum width. An integer value that specifies the maximum width of the widest glyph in the font, in pixels. This value is represented by the `widMax` field in the `FontRec` data type.
- Maximum kerning. An integer value that specifies the distance from the font rectangle's glyph origin to the left edge of the font rectangle, in pixels. If a glyph in the font kerns to the left, the amount is represented as a negative number. If the glyph origin lies on the left edge of the font rectangle, the value of the `kernMax` field is 0. This value is represented by the `kernMax` field in the `FontRec` data type.
- Negated descent value. If this font has very large tables and this value is positive, this value is the high word of the offset to the width/offset table. For more information, see "The Offset to the Width/Offset Table" on page 4-71. If this value is negative, it is the negative of the descent and is not used by the Font Manager. This value is represented by the `nDescent` field in the `FontRec` data type.
- Font rectangle width. An integer value that specifies the width, in pixels, of the image created if all the glyphs in the font were superimposed at their glyph origins. This value is represented by the `fRectWidth` field in the `FontRec` data type.
- Font rectangle height. An integer value that specifies the height, in pixels, of the image created if all the glyphs in the font were superimposed at their glyph origins. This value equals the sum of the maximum ascent and maximum descent measurements for the font. This value is represented by the `fRectHeight` field in the `FontRec` data type.
- Offset to width/offset table. An integer value that specifies the offset to the offset/width table from this point in the font record, in words. If this font has very large tables, this value is only the low word of the offset and the negated descent value is the high word, as explained in the section "The Offset to the Width/Offset Table" on page 4-71. This value is represented by the `owTLoc` field in the `FontRec` data type.

- Maximum ascent. An integer value that specifies the maximum ascent measurement for the entire font, in pixels. The ascent is the distance from the glyph origin to the top of the font rectangle. This value is represented by the `ascent` field in the `FontRec` data type.
- Maximum descent. An integer value that specifies the maximum descent measurement for the entire font, in pixels. The descent is the distance from the glyph origin to the bottom of the font rectangle. This value is represented by the `descent` field in the `FontRec` data type.
- Leading. An integer value that specifies the leading measurement for the entire font, in pixels. Leading is the distance from the descent line of one line of single-spaced text to the ascent line of the next line of text. This value is represented by the `leading` field in the `FontRec` data type.
- Bit image row width. An integer value that specifies the width of the bit image, in words. This is the width of each glyph's bit image as a number of words. This value is represented by the `rowWords` field in the `FontRec` data type.

The glyph data component of the bitmapped font resource consists of five tables that describe the glyphs in the font.

- Bit image table. The bit image of the glyphs in the font. The glyph images of every defined glyph in the font are placed sequentially in order of increasing ASCII code. The bit image is one pixel image with no undefined stretches that has a height given by the value of the font rectangle element and a width given by the value of the bit image row width element. The image is padded at the end with extra pixels to make its length a multiple of 16.
- Bitmap location table. For every glyph in the font, this table contains a word that specifies the bit offset to the location of the bitmap for that glyph in the bit image table. If a glyph is missing from the font, its entry contains the same value for its location as the entry for the next glyph. The missing glyph is the last glyph of the bit image for that font. The last word of the table contains the offset to one bit beyond the end of the bit image. You can determine the image width of each glyph from the bitmap location table by subtracting the bit offset to that glyph from the bit offset to the next glyph in the table.
- Width/offset table. For every glyph in the font, this table contains a word with the glyph offset in the high-order byte and the glyph's width, in integer form, in the low-order byte. The value of the offset, when added to the maximum kerning value for the font, determines the horizontal distance from the glyph origin to the left edge of the bit image of the glyph, in pixels. If this sum is negative, the glyph origin is to the right of the glyph image's left edge, meaning the glyph kerns to the left. If the sum is positive, the origin is to the left of the image's left edge. If the sum equals zero, the glyph origin corresponds with the left edge of the bit image. Missing glyphs are represented by a word value of –1. The last word of this table is also –1, representing the end.

- Glyph-width table. For every glyph in the font, this table contains a word that specifies the glyph's fixed-point glyph width at the given point size and font style, in pixels. The Font Manager gives precedence to the values in this table over those in the font family glyph-width table. There is an unsigned integer in the high-order byte and a fractional part in the low-order byte. This table is optional.
- Image height table. For every glyph in the font, this table contains a word that specifies the image height of the glyph, in pixels. The image height is the height of the glyph image and is less than or equal to the font height. QuickDraw uses the image height for improved character plotting, because it only draws the visible part of the glyph. The high-order byte of the word is the offset from the top of the font rectangle of the first non-blank (or nonwhite) row in the glyph, and the low-order byte is the number of rows that must be drawn. The Font Manager creates this table.

The Font Type Element

The font type element of the bitmapped font resource is represented as the `fontType` field in the `FontRec` data type. This integer field defines the general characteristics of the font and records whether certain tables are present. Its bits are used as follows.

Bit	Meaning
0	This bit is set to 1 if the font resource contains an image height table.
1	This bit is set to 1 if the font resource contains a glyph-width table.
2–3	These two bits define the depth of the font. Each of the four possible values indicates the number of bits (and therefore, the number of colors) used to represent each pixel in the glyph images.

Value	Font depth	Number of colors
0	1-bit	1
1	2-bit	4
2	4-bit	16
3	8-bit	256

Normally the font depth is 0 and the glyphs are specified as monochrome images. If bit 7 of this field is set to 1, a resource of type `'fctb'` with the same ID as the font can optionally be provided to assign RGB colors to specific pixel values.

If this font resource is a member of a font family, the settings of bits 8 and 9 of the `fontStyle` field in this font's association table entry should be the same as the settings of bits 2 and 3 in the `fontType` field. For more information, see "The Font Association Table" on page 4-95.

4–6	Reserved. Should be set to 0.
7	This bit is set to 1 if the font has a font color table (`'fctb'`) resource. The font is for color Macintosh computers only if this bit is set to 1.

Bit	Meaning
8	This bit is set to 1 if the font is a synthetic font, created dynamically from the available font resources in response to a certain color and screen depth combination. The font is for color Macintosh computers only if this bit is set to 1.
9	This bit is set to 1 if the font contains colors other than black. This font is for color Macintosh computers only if this bit is set to 1.
10–11	Reserved. Should be set to 0.
12	Reserved. Should be set to 1.
13	This bit is set to 1 if the font describes a fixed-width font, and is set to 0 if the font describes a proportional font. The Font Manager does not check the setting of this bit.
14	This bit is set to 1 if the font is not to be expanded to match the screen depth. The font is for color Macintosh computers only if this bit is set to 1. This is for some fonts, such as Kanji, which are too large for synthetic fonts to be effective or meaningful, or bitmapped fonts that are larger than 50 points.
15	Reserved. Should be set to 0.

The Offset to the Width/Offset Table

The offset to the width/offset table element of the bitmapped font resource is represented as the `owtLoc` field in the `FontRec` data type. This field defines the offset from the beginning of the resource to the beginning of the width/offset table.

The value of `nDescent`, when positive, is used as the high-order 16 bits in the 32-bit value that is used to store the offset of the width table from the beginning of the resource. To compute the actual offset, the Font Manager uses this computation:

```
actualOffsetWord := BSHL(nDescent, 16) + owTLoc;
```

If the value of `nDescent` is negative, it is still the negative of the descent measurement, as it was in the original usage of these values; however, the Font Manager no longer uses this value.

Note
This field was originally defined as an integer value, because it was not foreseen that this value could exceed 32 KB. The negated descent element, represented in the `nDescent` field of the `FontRec` data type, was created purely for the convenience of the Font Manager. It stored the negative of the value of the descent field, which is always positive by QuickDraw convention. When the depth of fonts increased, the values of the `owTLoc` field had to increase, and the extra bits needed to be stored somewhere. Since the `nDescent` field was created as a convenience, it was a handy place to store more information. ◆

The Outline Font ('sfnt') Resource

The outline font (`'sfnt'`) resource, which describes a TrueType outline font, consists of a sequence of tables that contain the data necessary for drawing the glyphs of the font, measurement information about the font, and any instructions that the font designer might include. These tables can appear in any order in the resource. Some of the tables are required, such as the description of the font's glyphs, and others are optional, such as kerning information. TrueType outline fonts are available on platforms other than the Macintosh computer, and some tables reflect the variety of information needed for these different operating systems. A table directory at the beginning of the outline font resource contains a version number and keys to access the tables.

Note

There are no data type definitions of the outline font resource tables and there are no fields, although the divisions of the tables are referred to as *fields* in this chapter. You must access the data using the routines and data structures that are described in this chapter or write table-specific code. Listing 4-2 beginning on page 4-76 shows how to read the contents of the various tables. ◆

The Font Manager uses some of the tables defined for the outline font resource to construct the font's glyphs or to store the font designer's information about creating bitmaps from the font data. Developers of general-purpose applications do not need these tables; consequently, the internal specifications of these tables are not provided in this chapter, although descriptions of their functions are. The needs of platforms other than the Macintosh computer are also not discussed.

Some of the terms used in descriptions of these tables pertain solely to the font designer's creation of the font. The em square is the imaginary area on which the glyphs of the font are first designed. The term units per em describes the resolution of the grid; the greater the number of units per em, the finer the detail of design that the designer can achieve. Apple's TrueType fonts use a resolution of 2048 units per em. The measurement pixels per em describes the relationship of the point size to the em square; the units per em measurements of the font are translated, using this pixels per em measurement, into bitmaps. The Font Manager handles this translation for you.

Similarly, the instruction set is for the use of the font designer only and cannot be used or altered by the Font Manager routines, and so is not included in this chapter. If you want the complete description of all of the tables in the outline font resource, consult the *TrueType Font Format Specification*.

Each table in the outline font resource is aligned on a longword boundary in memory (long-aligned) and may have been padded when necessary to make it long-aligned. Each table is named with a four character identifier known as its tag name. The only table that does not have a tag name is the font directory table. This table is a guide to the contents of the resource and is mandatory in all outline font resources.

Note
Detailed descriptions of many of the values in the outline font resource tables are found in the *TrueType Font Format Specification* and are not repeated in this chapter. If you are designing a font editor or similar application that requires detailed knowledge of these tables, please refer to that book. ◆

These are the required tables in the outline font resource:

Tag name	Table
(none)	Font directory
`'cmap'`	Character code mapping table
`'glyf'`	Glyph data table
`'head'`	Font header table
`'hhea'`	Horizontal header table
`'hmtx'`	Horizontal metrics table
`'loca'`	Location table
`'maxp'`	Maximum profile table
`'name'`	Font-naming table
`'post'`	PostScript table

Some of the optional tables in the `'sfnt'` resource are

Tag	Table
`'cvt '`	Control-value table
`'fpgm'`	Font program table
`'hdmx'`	Horizontal device metrics table
`'kern'`	Kerning table
`'prep'`	Preprogram (control value program) table

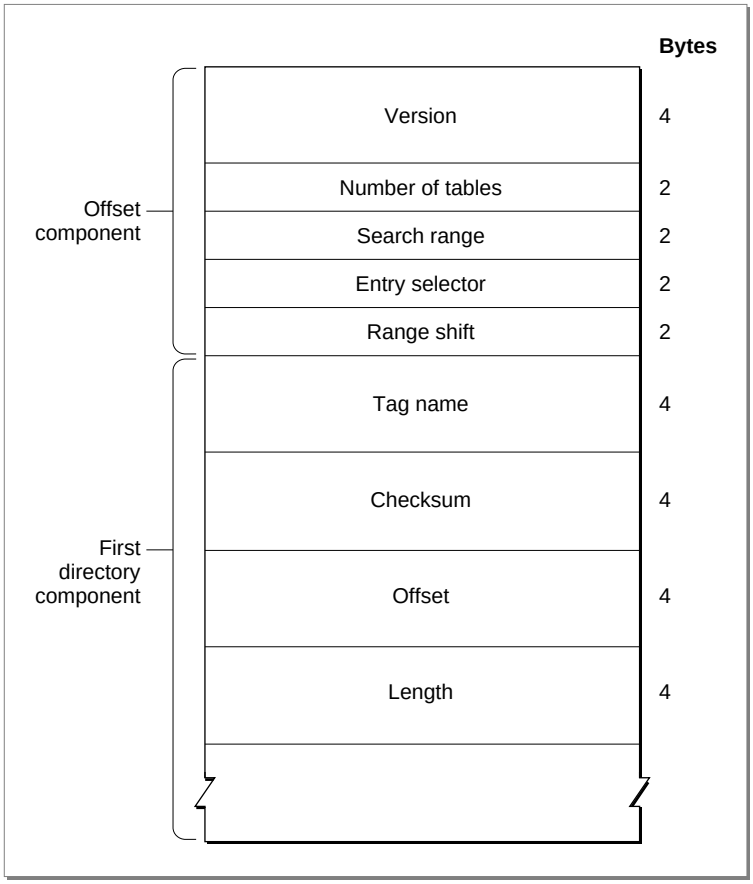
Font designers can define additional tables for the outline font resource to support other platforms where outline fonts are available or to provide for future expansion of a font. Tag names consisting of all lowercase letters are reserved for use by Apple Computer, Inc.

The Font Directory

The font directory is a guide to the tables in the outline font resource. It provides you with the information that is needed to efficiently find the other parts of the resource. Each table in the resource has a tag name, a checksum, a location that is defined as an offset in bytes from the first byte of the resource, and a length in bytes. To use the data in a table, you first find the table's tag name in the font directory and then access its data starting at the specified location.

The font directory consists of an offset component and a variable length array of directory entries, as shown in Figure 4-17.

Figure 4-17 The font directory



The font directory offset component specifies the number of tables in the resource (and thus in the directory component). It contains several values that you can use to optimize searching through the directory components for a tag name:

- Version. The version number of the font, given as a 32-bit fixed point number. For version 1.0 of any font, this number is $00010000.
- Number of tables. The number of tables in the outline font resource, not counting the font directory or any subtables in the font. This is an unsigned integer value.
- Search range. An unsigned integer value that is used, along with the entry selector and range shift values, to optimize a binary search through the directory.
- Entry selector. An unsigned integer value that is used, along with the search range and range shift values, to optimize a binary search through the directory.
- Range shift. An unsigned integer value that is used, along with the search range and entry selector values, to optimize a binary search through the directory.

The search range, entry selector, and range shift values are used together to construct a binary search through the directory if it is too large for an efficient sequential search. Note, however, that most programs that access kerning data use a linear search and do not make use of these values.

If a font does contain a large number of tables, you can perform a binary search of the directory components. You use the range shift value as the initial position in the directory to examine. Compare the tag name of the component at this position with the one you are searching for. If the target tag name comes before the one you are searching for, search from the beginning of the directory to the range shift position. If the target name comes after the one you are searching for, search from that position to the end of the directory.

The font directory table entries are sorted alphabetically by tag name. Each component consists of the following elements:

- Tag name. The identifying name for this table, such as `'cmap'`.
- Checksum. The checksum for this table, which is the unsigned sum of the long values in the table. This number can be used to verify the integrity of the data in the table.
- Offset. The offset from the beginning of the outline font resource to the beginning of this table, in bytes.
- Length. The length of this table, in bytes.

Listing 4-2 shows a function that determines the checksum of a given table.

Listing 4-2 Calculating the checksum of a given table

```
TYPE
   LongPtr = ^LongInt;

FUNCTION MyCalcTableChecksum (table: LongPtr;
                              lngth: LongInt): LongInt;
VAR
   sum : LongInt;
   mask: LongInt;

BEGIN
   sum := 0;
   WHILE lngth > 0 DO BEGIN
      IF lngth > 3 THEN
         sum := sum + table^
      ELSE BEGIN
         mask := BitShift($FFFFFFFF, 8 * (4 - lngth));
         sum := sum + BitAnd(table^, mask);
         table := LongPtr(ord(table) + 4);
         lngth := lngth - 4;
      END;
   END;
   MyCalcTableChecksum := sum;
END;
```

The Character-Code Mapping Table

The character-code mapping table, with a tag name of `'cmap'`, maps character codes (like ASCII codes) to glyph indexes. The glyph repertoire of an outline font is indexed consecutively from zero to the number of glyphs in the font. The encoding method selected by the font designer depends on the conventions used by the intended platform and sometimes on other platform-specific selectors, such as which script system is in use. A font intended for use on multiple platforms with different conventions requires multiple encoding tables; however, double-byte fonts require various special formats for efficient encoding. As a result, the `'cmap'` table may contain multiple encoding components, one for each supported encoding scheme, often in different formats.

Character codes that do not correspond to any glyph in the font should be mapped to glyph index 0. At this location in the font there should be a special glyph representing a missing character, which typically is a box (□). For more information on requirements for character-to-glyph mapping, see the *TrueType Font Format Specification*.

In the simplest case, the character-code mapping table consists of a header component and only one character-mapping format component, which includes an array of glyph indexes. In other cases, there are several character-mapping components in the table.

The Control-Value Table

The control-value table, with a tag name of `'cvt '`, is an optional table that can be used by fonts that contain instructions. This table contains data (control values) used by the instructions. Each entry in this table is 4 bytes long. The number of values in the table can be computed by dividing the length of the table by 4. The length of the table is found in the directory component for this table in the outline font resource directory.

The font directory is described in "The Font Directory," beginning on page 4-74.

The control-value program, which uses these values, is contained in the preprogram table, which is described on page 4-89.

The Font Program Table

The font program table, with a tag name of `'fpgm'`, is an optional table that contains the font program, a list of instructions that the Font Manager executes once, when it loads the font into memory. The font program is a variable length sequence of bytes that are interpreted by the Font Manager. The length of this table is found in the directory component for this table in the outline font resource directory.

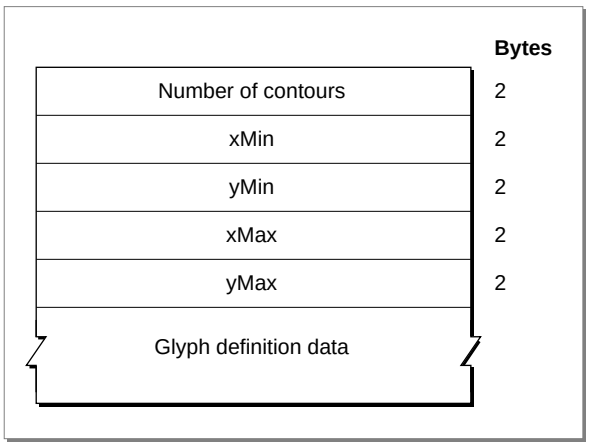
The font directory is described in "The Font Directory," beginning on page 4-74.

The Glyph Data Table

The glyph data table, with a tag name of `'glyf'`, contains the data that defines the appearance of the glyphs in the font: the specification of points that make up the contours of a glyph and the instructions that help change the shape of the glyph under various conditions. Glyphs can be stored in any character-code mapping order, since the location of the data for each is specified separately, through the character-code mapping table, which is described beginning on page 4-76, and the location table, which is described on page 4-84.

The data for each glyph consists of some descriptive information, as shown in Figure 4-18, followed by the actual instructions and coordinate values that define the glyph. The format of the definition data for glyphs is described in the *TrueType Font Format Specification*. Note that the glyph data is compressed.

Figure 4-18 A glyph description



- Number of contours. If this integer value is positive, it specifies the number of closed curves defined in the outline data for the glyph. If it is –1, it indicates that the glyph is composed of other simple glyphs (see the explanation of component glyphs in the section "The Maximum Profile Table" beginning on page 4-84).
- xMin. The left edge of the glyph's bounding box, specified in units per em.
- yMin. The top edge of the glyph's bounding box, specified in units per em.
- xMax. The right edge of the glyph's bounding box, specified in units per em.
- yMax. The bottom edge of the glyph's bounding box, specified in units per em.
- Glyph definition data. The data that defines the appearance of the glyph, as described in the *TrueType Font Format Specification*.

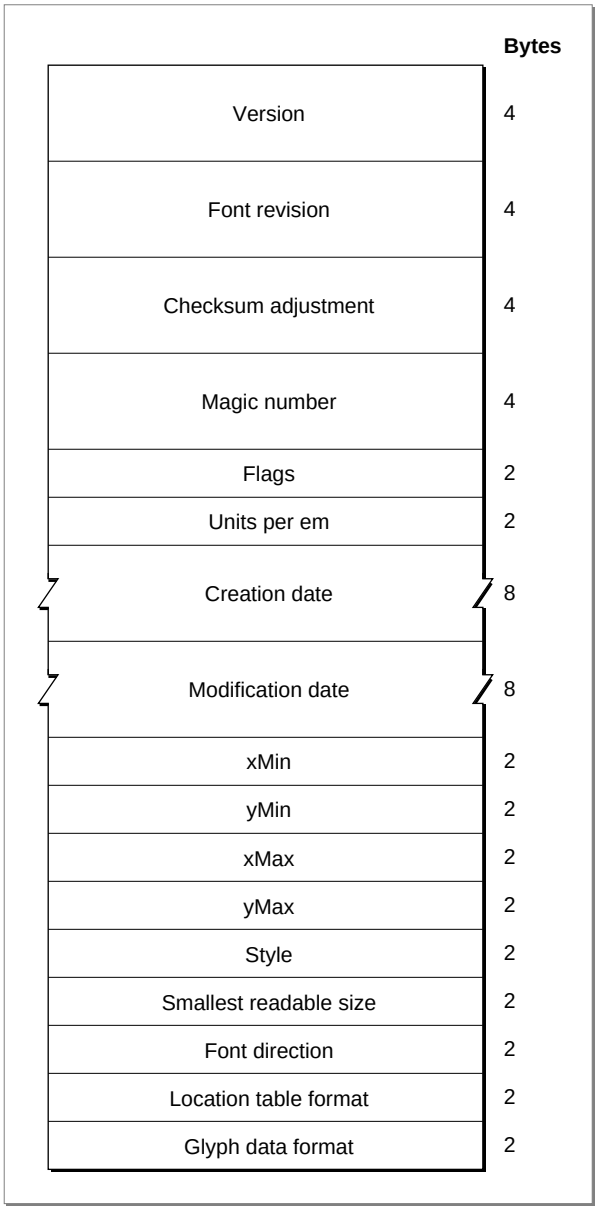
The Horizontal Device Metrics Table

The horizontal device metrics table, with a tag name of `'hdmx'`, is an optional table that stores integer advance widths scaled to pixel-per-em sizes for the Macintosh computer's screen. The horizontal device metrics table is used only for certain screen sizes that are determined by the font's designer and only when fractional widths are disabled (as described in "SetFractEnable" on page 4-58). This table contains fine-tuned integer widths for the glyphs at low pixel-per-em values. These values can be used to reduce the unpleasant consequences of rounding the widths for small point sizes at low resolution.

The Font Header Table

The font header table, with a tag name of `'head'`, is shown in Figure 4-19. This table contains global information about the font: the font version number; creation and modification dates; revision number; basic typographic data that applies to the font as a whole, such as the direction in which the font's glyphs are most likely to be written; and other information about the placement of glyphs in the em square.

Figure 4-19 The font header table



For a complete description of the individual fields in this table, see the *TrueType Font Format Specification*. Application developers may be interested in the following fields:

- Version. The version number of the table, as a fixed-point value. This value is $00010000 if the version number is 1.0.
- Font revision. A fixed-point value set by the font designer.
- Checksum adjustment. The checksum of the font, as an unsigned long integer.
- Units per em. This unsigned integer value represents a power of 2 that ranges from 64 to 16,384. Apple's TrueType fonts use the value 2048.
- Creation date. The date this font was created. This is a long date-time value of data type `LongDateTime`, which is a 64-bit, signed representation of the number of seconds since Jan. 1, 1904.
- Modification date. The date this font was last modified. This is a long date-time value of data type `LongDateTime`, which is a 64-bit, signed representation of the number of seconds since Jan. 1, 1904.
- Smallest readable size. The smallest readable size for the font, in pixels per em. The `RealFont` function, which is described in the section "RealFont" beginning on page 4-52, returns `FALSE` for a TrueType font if the requested size is smaller than this value.
- Location table format. The format of the location table (tag name: `'loca'`), as an signed integer value. The table has two formats: if the value is 0, the table uses the short offset format; if the value is 1, the table uses the long offset format. The location table is described in the section "The Location Table" on page 4-84.

You can use the value of the checksum adjustment element to verify the integrity of the data in the font, to confirm that no data has been changed, or to compare two similar fonts. This value is the designer's checksum value for the font. The checksum value is an unsigned long word that you compute as follows:

1. Set the value of the checksum word to 0 (so that it does not factor into the value that you are computing).
2. Calculate the checksum for each table in the outline font resource and store the table's checksum in the table directory.
3. Now sum the entire font as an unsigned, 32-bit value.
4. Subtract the sum from $B1B0AFBA, which is a magic number for this checksum computation. Store the result.

Listing 4-2 provides an example of a function to compute the checksum of a font. This example includes type declarations for the outline font header information and uses the `MyCalcTableChecksum` function (from Listing 4-2 on page 4-76) to compute the checksum for each table.

Listing 4-3 Calculating the checksum of a font

```
TYPE

DirectoryEntry =
RECORD
   tag: OSType;
   checksum: LongInt;
   offset: LongInt;
   lngth: LongInt;
END;

OffsetTable =
RECORD
   version: LongInt;
   numTables: Integer;
   searchRange: Integer;
   entrySelector: Integer;
   rangeShift: Integer;
   tableDir: ARRAY[1..1] OF DirectoryEntry;
                         { actually 1..numTables }
END;

SfntPtr = ^OffsetTable;
SfntHandle = ^SfntPtr;

HeaderTable =
RECORD
   version: LongInt;
   fontRevision: LongInt;
   checkSumAdjustment: LongInt;
   magicNumber: LongInt;
   flags: Integer;
   unitsPerEm: Integer;
   created: LongDateTime; { defined in Script.p }
   modified: LongDateTime;
   xMin: Integer;
   yMin: Integer;
   xMax: Integer;
   yMax: Integer;
   macStyle: Integer;
```

```
   lowestRecPPEM: Integer;
   fontDirectionHint: Integer;
   indexToLocFormat: Integer;
   glyphDataFormat: Integer;
END;

HeaderTablePtr = ^HeaderTable;


FUNCTION CalcSfntChecksum (sp: SfntPtr): LongInt;
CONST
   checkSumMagic = $B1B0AFBA;
VAR
   i: Integer;
   cs, sum, size: LongInt;
   htp: HeaderTableptr;

BEGIN
   sum := 0;
   FOR i := 1 TO sp^.numTables DO BEGIN
      IF sp^.tableDir[i].tag = 'head' THEN
         BEGIN
            htp := HeaderTablePtr(ord(sp) +
                                  sp^.tableDir[i].offset);
            htp^.checkSumAdjustment := 0;
            cs := CalcTableChecksum(LongPtr(htp),
                                  SizeOf(HeaderTable));
         END
      ELSE
         cs := MyCalcTableChecksum(
                     LongPtr(ord(sp) + sp^.tableDir[i].offset),
                  sp^.tableDir[i].lngth);

      sp^.tableDir[i].checksum := cs;
      sum := sum + cs;
   END;
   size := SizeOf(OffsetTable) +
            (sp^.numTables - 1) * SizeOf(DirectoryEntry);
   sum := sum + CalcTableChecksum(LongPtr(sp), size);
   CalcSfntChecksum := checkSumMagic - sum;
   { to be written into htp^.checkSumAdjustment }
END;
```

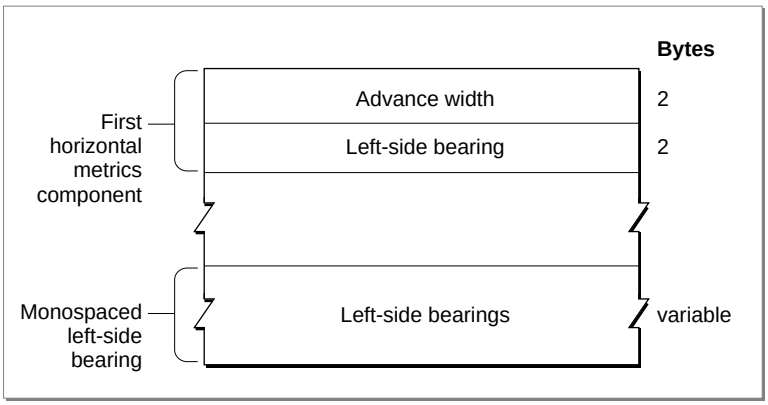
The Horizontal Header Table

The horizontal header table, with a tag name of `'hhea'`, contains information needed to lay out fonts whose glyphs are written horizontally (either left to right or right to left) across a page. This table contains information that pertains to the font as a whole. Information that pertains to specific glyphs is given in the horizontal metrics table, which is described in the next section "The Horizontal Metrics Table."

The Horizontal Metrics Table

The horizontal metrics table, with a tag name of `'hmtx'`, consists of arrays that contain metrics information—the advance widths and left-side bearings—for the horizontal layout of each glyph in the font. The horizontal metrics table structure is shown in Figure 4-20.

Figure 4-20 The horizontal metrics table



The first component of the horizontal metrics table contains two values for each entry: the advance width and the left-side bearing for the associated glyph. The number of value pairs in this component is specified in the number of advance widths element of the horizontal header table, which is described in the preceding section, "The Horizontal Header Table."

The horizontal metrics table may have a second component that is used for fixed-width glyphs. It contains the left-side bearings only. The advance width is the same as the last entry in the first component of this table.

The Kerning Table

The kerning table, with a tag name of `'kern'`, is an optional table that contains the values you can use to adjust the spacing between glyphs in a font. Kerning can be parallel to the flow of text or perpendicular to the flow of text. For example, if you specify perpendicular kerning and if text is normally read horizontally, the glyphs are kerned vertically. Kerning is always applied to pairs of glyphs.

The kerning table in the outline font resource consists of a header and a series of subtables. The *TrueType Font Format Specification* documents a basic set of kerning subtables.

The Location Table

The location table, with a tag name of `'loca'`, stores the offsets to the locations of actual glyph data in the outline font resource relative to the beginning of the glyph table. It provides quick access to the data for a particular glyph. The location table is an array of offset values, one for each glyph in the font, including the 0th or missing character glyph.

Offsets are stored in one of two forms: in the short format, each offset is a 16-bit unsigned integer value that specifies the number of words from the beginning of the glyph data table to the data for the glyph. In the long format, each offset is a 32-bit unsigned integer value that specifies the number of bytes from the beginning of the glyph data table to the data for the glyph. The format that is used for a font is specified in the location table format element of the font header table, whose description begins on page 4-79.

The Maximum Profile Table

The maximum profile table, with a tag name of `'maxp'`, establishes the memory requirements for a font. Most of the information in this table is for the use of the font's designer, a font editor that may alter the makeup of the resource, or the Font Manager itself.

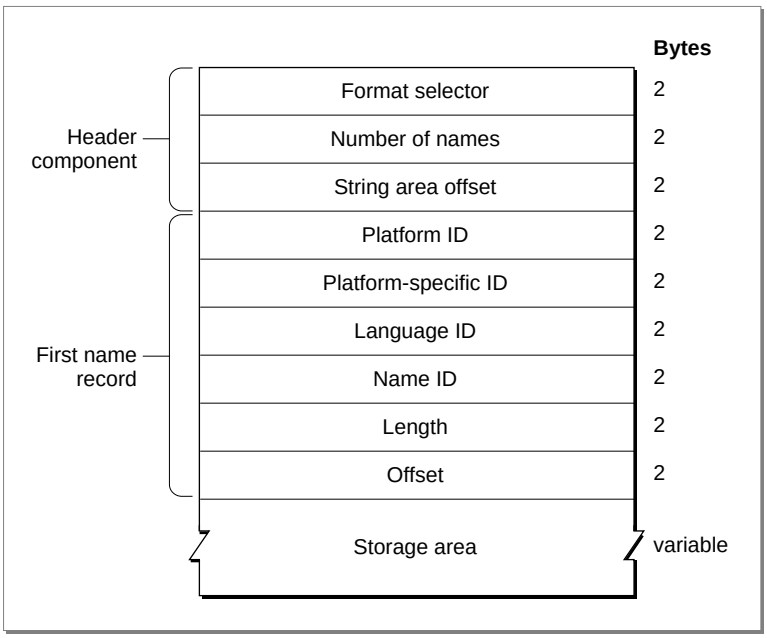
Some of the elements in the maximum profile table refer to simple versus component glyphs. A simple glyph is one that is defined as a single equation, such as an "e". A component glyph is a design that the font designer builds by adding a simple glyph to another equation or by adding two glyphs together. For example, the glyph "ê" can be created as a single entity or as a component glyph: the simple glyph "e" plus the simple glyph "^". In this way, a small set of simple glyphs can create a much larger set of component glyphs. The font designer could also design the glyph "ê" as a simple glyph. However, this leads to separate designs for "â", "î", and so on. Some fonts distributed by Apple use component glyphs and some do not.

For more information about this table, see the *TrueType Font Format Specification*.

The Font Naming Table

The font naming table, with a tag name of `'name'`, is shown in Figure 4-21. This table contains multilingual strings associated with the outline font resource. These strings can represent copyright notices, font names, style names, and so on, and each string is stored in a separate record with some information about what kind of string it is. You may want to provide this information in your application for the user, or you may want to use it to check one version of a font against another.

Figure 4-21 The naming table



The header component of the font naming table consists of the following elements:

- Format selector. The format selector (set to 0). This is an unsigned integer.
- Number of names. The number of name records that follow. This is an unsigned integer.
- String area offset. The offset from the start of the table to the start of string storage, in bytes. This is an unsigned integer.

Each name record contains information about the platform and language of the strings stored in the naming table.

- Platform ID. The platform identifier.
- Platform-specific ID. The platform-specific encoding identifier.
- Language ID. The language identifier.
- Name ID. The name identifier.
- Length. The length of the string, in bytes.
- Offset. The offset from the start of storage area, in bytes.

The storage area at the end of the naming table contains the actual string data.

There is no length limit for the strings contained in a name record, but font designers should not include empty strings (of byte length 0). The Font Manager sorts the entries in the naming table first by platform identifier, next by platform-specific identifier, next by language identifier, and last by name identifier.

To keep the size of this table small, a font designer may make a limited set of name records in a small set of languages, because the font can be localized and the existing strings translated or new strings added. Other parts of the outline font resource that need these strings can refer to them by their index number, and applications that need a particular string can look it up by its platform identifier, language identifier, and font name identifier. Platform IDs are shown in Table 4-2, language identifiers are shown in Table 4-4, and font name identifiers are shown in Table 4-5.

TrueType outline fonts are available on other platforms besides the Macintosh computer, which is why the font designer must specify the platform. There are only four predefined platform identifiers, listed in Table 4-2, and they use values 0 through 3.

Table 4-2 Platform identifiers

ID	Platform	Specific encoding
0	Unicode	Reserved (set to 0)
1	Macintosh	The Script Manager code
2	ISO	ISO encoding
3	Microsoft	Microsoft encoding
240–255	User-defined	Reserved for all nonregistered platforms

When the platform used is the Macintosh computer, the platform-specific identifier names the specific script code for this name record. The script codes defined for the Macintosh system software are listed in the chapter "Script Manager" in this book.

The platform-specific identifier encodings for the ISO platform are listed in Table 4-3.

Table 4-3 ISO platform-specific identifiers

Code	ISO encoding scheme
0	7-bit ASCII
1	ISO 10646
2	ISO 8859-1

The value of the language identifier specifies the language in which a particular string is written. The language identifiers available on the Macintosh platform are listed in Table 4-4.

Table 4-4 ISO language codes

Code	Language	Code	Language
0	English	12	Arabic
1	French	13	Finnish
2	German	14	Greek
3	Italian	15	Icelandic
4	Dutch	16	Maltese
5	Swedish	17	Turkish
6	Spanish	18	Yugoslavian
7	Danish	19	Chinese
8	Portuguese	20	Urdu
9	Norwegian	21	Hindi
10	Hebrew	22	Thai
11	Japanese		

The font name identifier values, listed in Table 4-5, contain the strings with information about the font.

Table 4-5 Font name identifiers

Code	Meaning	Description
0	Copyright notice	The copyright notice of the font—for example, "Copyright Apple Computer, Inc. 1992"
1	Font family name	The font family name, such as "New York"
2	Font style	The style of the font, such as "Bold"
3	Font identification	A unique identification string for the font—for example, "Apple Computer New York Bold version 1.0"
4	Full font name	The font family name combined with the font style name—for example, "New York Bold"
5	Version string	The version of the font, or when it was created—for example, "August 10, 1991, 1.08d21"
6	PostScript name of the font	A name of this font that the PostScript printer driver can recognize—for example, "Times-Bold"
7	Trademark	The trademark notice of the designer
8	Designer	Corporate name of the designer

The full font name for a font family, given in string 4 of Table 4-5, is most often the same as the family name, given in string 1. The default style for a family or the only font in a family should have "Regular" in the font style string. (Font designers use the term "Regular" to denote the plain style for a font, so as to reflect typographic terminology more accurately.) One exception, based on historical convention, is when the full name of a font includes the word "Roman" (e.g. Times Roman). In all other cases, the full name should be made up of the family name and the style name, as in Bookman Bold.

The unique font identification consists of the designer's name, followed by a space serving as a separator, followed by the full name of the font. For example, though there might be many Symbol fonts, the name "Apple Computer Symbol" is unique. The use of unique names allows applications to determine if the current system software has the fonts used in the original document.

The PostScript Table

The PostScript table, with a tag name of `'post'`, contains information needed to use an outline font on a PostScript printer. It contains the PostScript names for all of the glyphs in the font. It also contains memory information needed by the PostScript driver for memory management. The PostScript table consists of a header component and an optional format component, which is used only for two of the possible four PostScript format types.

The header component of the PostScript table contains the memory requirements. PostScript drivers can make better use of the Memory Manager if the virtual memory requirements of an outline font that can be downloaded to the printer are known beforehand. If the font designer does not know the virtual memory requirements, the values for the memory use requirements of this font are set to zero.

The memory use of a downloaded outline font varies depending on whether it is defined as a TrueType or Adobe™ Type 1 font on the printer. You can compute the minimum memory required for a font as follows:

1. Send the PostScript `VMStatus` call to the printer and store the result.
2. Download the font to the printer.
3. Send the `VMStatus` call again.
4. Subtract the first result from the second to calculate the amount of memory that the font requires.

The maximum memory required for a font is computed by adding the maximum run-time memory use to the minimum memory value. The maximum run-time memory use depends on the maximum band size of any bitmap that the outline font scaler might have to create from an outline description.

The Preprogram Table

The preprogram table, with a tag name of `'prep'`, is an optional table that stores the control value program. This is a set of outline font instructions that the Font Manager executes before it creates any glyph and again whenever the user changes the point size, the angle at which the font is being displayed, or the font itself. This table consists of an ordered list of instruction opcodes, each of which is one byte long. Control values for the instructions in the preprogram are found in the `'cvt '` table, which is described in “The Control-Value Table” on page 4-77.

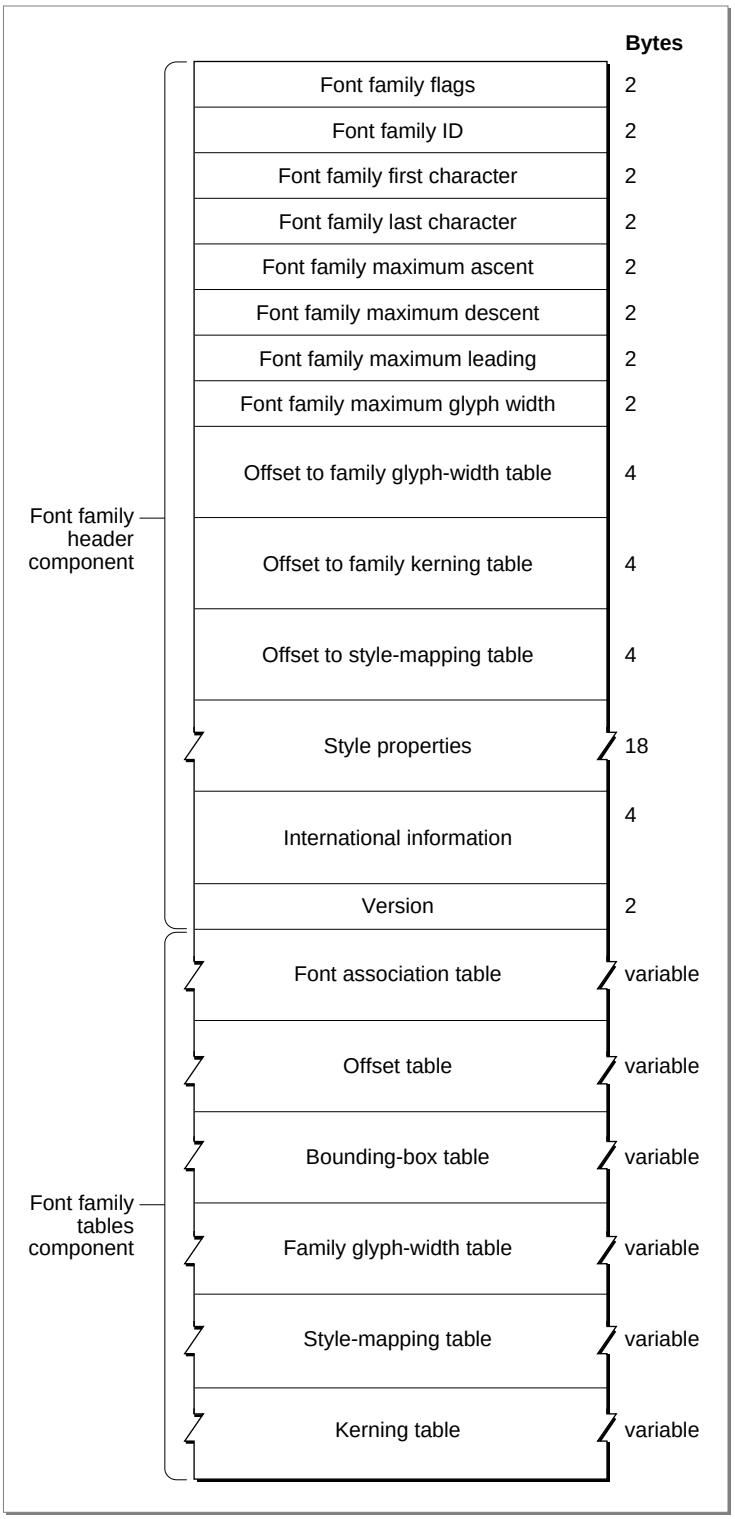
The Font Family ('FOND') Resource

A font family contains references to the fonts (which can be bitmapped font ['NFNT'], outline font ['sfnt'], or 'FONT' resources) that make up the family and information that describes the family as a whole, such as a global width table for each available style.

The font family ('FOND') resource contains general information about the font family, the font association table, and a collection of optional tables: the family glyph-width table, the style-mapping table, the kerning table, the offset table, and the bounding-box table. Several data structures and routines use the font family resource. For example, the global width table can use the font family information to find the recommended glyph widths and the LaserWriter printer driver can use tables that contain information about kerning pairs and mapping of styles to printer fonts.

The font family resource consists of a header component, which contains general information about the font family, and a font family tables component, which consists of the font association table and some number (possibly zero) of the optional tables that provide measurement and naming information about the font family. The header component of this resource is represented by the FamRec data type, the declaration of which is shown in the section "The Font Family Record" on page 4-47. The structure of this resource is shown in Figure 4-22.

Figure 4-22 The font family (`'FOND'`) resource



The header component of the font family resource consists of a number of elements that describe characteristics of the family. Each of the elements in this component is represented by a field in the `FamRec` data type.

- Font family flags. An integer value, the bits of which specify general characteristics of the font family. This value is represented by the `ffFlags` field in the `FamRec` data type. The bits in the `ffFlags` field have the following meanings:

Bit	Meaning
0	This bit is reserved by Apple and should be cleared to 0.
1	This bit is set to 1 if the resource contains a glyph-width table.
2–11	These bits are reserved by Apple and should be cleared to 0.
12	This bit is set to 1 if the font family ignores the value of the `FractEnable` global variable when deciding whether to use fixed-point values for stylistic variations; the value of bit 13 is then the deciding factor. The value of the `FractEnable` global variable is set by the `SetFractEnable` procedure.
13	This bit is set to 1 if the font family should use integer extra width for stylistic variations. If not set, the font family should compute the fixed-point extra width from the family style-mapping table, but only if the `FractEnable` global variable has a value of `TRUE`.
14	This bit is set to 1 if the family fractional-width table is not used, and is cleared to 0 if the table is used.
15	This bit is set to 1 if the font family describes fixed-width fonts, and is cleared to 0 if the font describes proportional fonts.

- Font family ID. An integer value that specifies the `'FOND'` resource ID number for this font family. This value is represented by the `ffFamID` field in the `FamRec` data type.
- Font family first character. An integer value that specifies the ASCII character code of the first glyph in the font family. This value is represented by the `ffFirstChar` field in the `FamRec` data type.
- Font family last character. An integer value that specifies the ASCII character code of the last glyph in the font family. This value is represented by the `ffLastChar` field in the `FamRec` data type.
- Font family maximum ascent. The maximum ascent measurement for a one-point font of the font family. This value is in a 16-bit fixed-point format with an integer part in the high-order 4 bits and a fractional part in the low-order 12 bits. This value is represented by the `ffAscent` field in the `FamRec` data type.
- Font family maximum descent. The maximum descent measurement for a one-point font of the font family. This value is in a 16-bit fixed-point format with an integer part in the high-order 4 bits and a fractional part in the low-order 12 bits. This value is represented by the `ffDescent` field in the `FamRec` data type.

- Font family maximum leading. The maximum leading for a 1-point font of the font family. This value is in a 16-bit fixed-point format with an integer part in the high-order 4 bits and a fractional part in the low-order 12 bits. This value is represented by the `ffLeading` field in the `FamRec` data type.
- Font family maximum glyph width. The maximum glyph width of any glyph in a one-point font of the font family. This value is in a 16-bit fixed-point format with an integer part in the high-order 4 bits and a fractional part in the low-order 12 bits. This value is represented by the `ffWidMax` field in the `FamRec` data type.
- Offset to family glyph-width table. The offset to the family glyph-width table from the beginning of the font family resource to the beginning of the table, in bytes. The family glyph-width table is described in the section "The Family Glyph-Width Table," beginning on page 4-98. This value is represented by the `ffTabOff` field in the `FamRec` data type.
- Offset to family kerning table. The offset to the beginning of the kerning table from the beginning of the `'FOND'` resource, in bytes. The kerning table is described in the section "The Font Family Kerning Table," beginning on page 4-106. This value is represented by the `ffKernOff` field in the `FamRec` data type.
- Offset to family style-mapping table. The offset to the style-mapping table from the beginning of the font family resource to the beginning of the table, in bytes. The style-mapping table is described in the section "The Style-Mapping Table," beginning on page 4-99. This value is represented by the `ffStyleOff` field in the `FamRec` data type.
- Style properties. An array of 9 integers, each indicating the extra width, in pixels, that would be added to the glyphs of a 1-point font in this font family after a stylistic variation has been applied. This value is represented by the `ffProperty` field in the `FamRec` data type, which is an array with nine values. The Font Manager multiplies these values by the requested point size to get the correct width. Each value is in a 16-bit fixed-point format with an integer part in the high-order 4 bits and a fractional part in the low-order 12 bits. If the font with a given stylistic variation already exists as an intrinsic font, the Font Manager ignores the value in the `ffProperty` field for that style. The values in this array are used as follows:

Property index	Meaning
1	Extra width for plain text. Should be set to 0.
2	Extra width for bold text.
3	Extra width for italic text.
4	Extra width for underline text.
5	Extra width for outline text.
6	Extra width for shadow text.
7	Extra width for condensed text.
8	Extra width for extended text.
9	Not used. Should be set to 0.

- International information. An array of 2 integers reserved for internal use by script management software. This value is represented by the `ffIntl` field in the `FamRec` data type.
- Version. An integer value that specifies the version number of the font family resource, which indicates whether certain tables are available. This value is represented by the `ffVersion` field in the `FamRec` data type. Because this field has been used inconsistently in the system software, it is better to analyze the data in the resource itself instead of relying on the version number. The possible values are as follows:

Value	Meaning
$0000	Created by the Macintosh system software. The font family resource will not have the glyph-width tables and the fields will contain 0.
$0001	Original format as designed by the font developer. This font family record probably has the width tables and most of the fields are filled.
$0002	This record may contain the offset and bounding-box tables.
$0003	This record definitely contains the offset and bounding-box tables.

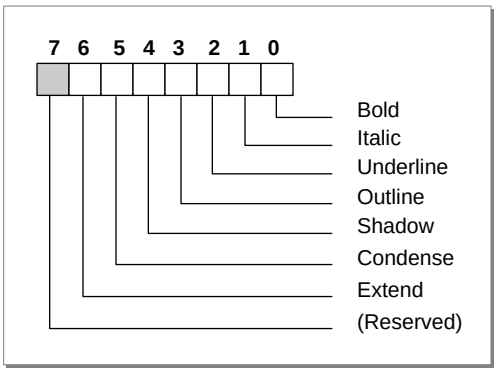
The font family tables component of the font family resource contains a number of tables. The font association table must be included in the resource, but the other tables are all optional. You can determine whether or not the glyph-width, kerning, or style-mapping tables are present by examining the offset value for each. Each offset value is a number of bytes from the beginning of the resource to the table; an offset of 0 means that the table is not present. For example, if the value of the `ffWTabOff` field is greater than 0, the glyph-width table is present in the resource data.

Additional tables, including the bounding-box table, can be added to the font family resource by a font designer. Whenever any table, including the glyph-width, kerning, and style-mapping tables, is included in the resource data, an offset table is included. The offset table contains a long integer offset value for each table that follows it.

The Font Style Code

A number of tables in the font family resource contain information that pertains only to a certain style. Actually, a style can be a combination of styles. The **style code** data type, which is used to represent a style in the tables in this resource, uses a single bit for each of the seven Macintosh character styles. You can set any of these bits to 1 in the style code element of a table to specify the unique style of the font to which that table applies. Although each table that contains a font style code allocates 2 bytes for the value, only the low-order byte of the value is used to specify the style code; the high-order byte is used internally by the Font Manager. The values of the bits in a style code element are shown in Figure 4-23.

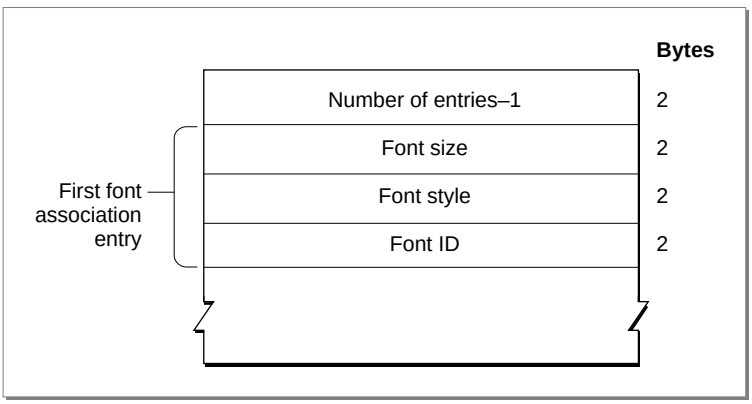
Figure 4-23 Style codes



The Font Association Table

The font association table of the font family resource maps a point size and style into a specific font that is part of the family. This table is represented by the `fontAssoc` field of the font family resource. This table, which is shown in Figure 4-24, matches a given font size and style combination with the resource ID of a `'FONT'`, bitmapped, or outline resource.

Figure 4-24 The font association table



The font association table consists of an integer count and a variable number of font association entries. The table is represented by the `FontAssoc` data type, which is shown on page 4-47.

- Number of entries. An integer value that specifies the number of font association records in this table minus 1. This value is represented by the `numAssoc` field in the `FontAssoc` data type.

Each font association entry is represented by the `AsscEntry` data type, which is shown on page 4-48. The Font Manager looks first for outline font resources, then bitmapped font resources, then `'FONT'` resources. Entries are sorted according to point size, with the smallest sizes coming first in the table. The font size value for outline font resources is 0, so they are always listed first. Plain fonts are sorted before styled fonts. The elements of each entry are:

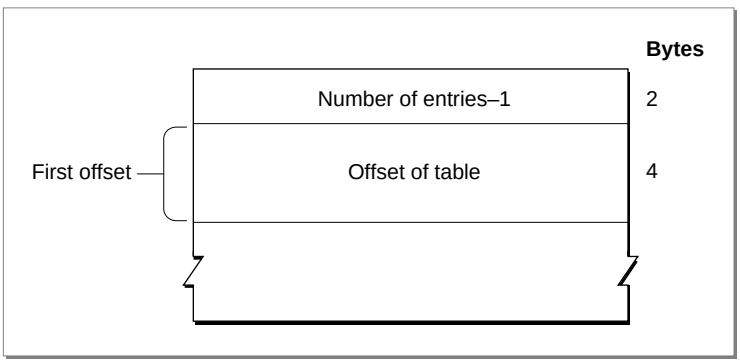
- Font size. This integer value specifies the size of the font in points. This value is represented by the `fontSize` field of the `AsscEntry` data type.
- Font style. This integer value specifies the style code of the entry, as shown in Figure 4-23 on page 4-95. This value is represented by the `fontStyle` field of the `AsscEntry` data type.
- Font ID. This integer value specifies the resource ID of the related `'sfnt'`, `'NFNT'`, or `'FONT'` resource. This value is represented by the `fontID` field of the `AsscEntry` data type.

Note
Bits 8 and 9 of the `fontStyle` field of the font association table entry specify the font depth. They need to contain the same values as bits 2 and 3 of the `fontType` field of the font resource that this entry describes. ◆

The Offset Table

The offset table is an optional table that is included in the font family resource whenever any of the other optional tables are included. This table, which is shown in Figure 4-25, allows the font designer to add more tables to the font family resource.

Figure 4-25 The offset table



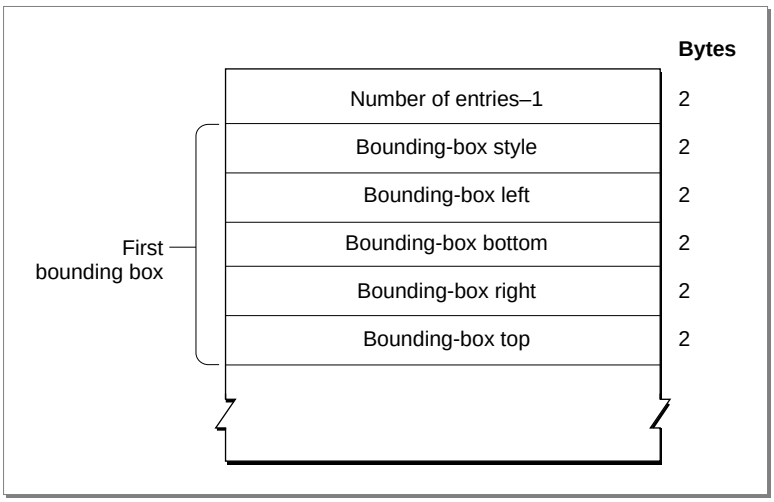
The offset table consists of an integer count and a variable number of table offset values, each of which is 4 bytes long. There is no data type defined for this table.

- Number of entries. An integer value that specifies the number of offset values in this table minus 1.
- Offset of table. A long integer value that specifies the number of bytes from the start of the offset table to the start of the table.

The Bounding-Box Table

The bounding-box table, shown in Figure 4-26, contains the bounding-box measurements for a 1-point font. The bounding boxes used in this table are similar to the font rectangle, since each describes the smallest rectangle that encloses the shape of each glyph in a given font. There are separate bounding-box entries in the table for different styles.

Figure 4-26 The bounding-box table



The bounding-box table consists of an integer count and a variable number of bounding-box entries, each of which is 10 bytes long. There is no data type defined for this table.

- Number of entries. An integer value that specifies the number of bounding-box entries in this table minus 1.

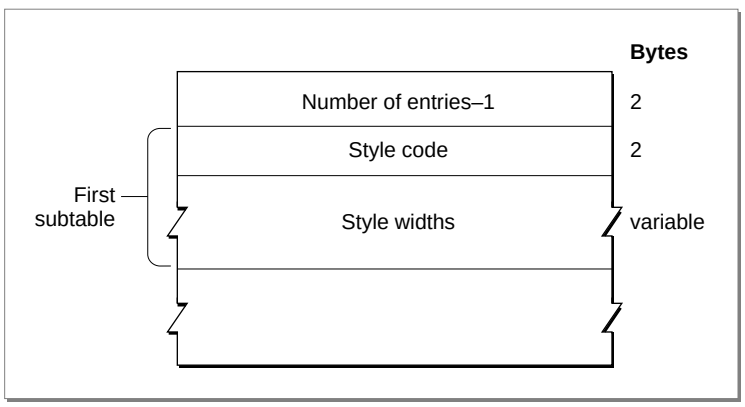
Each bounding-box entry consists of the following elements. There is no data type defined for these entries, each of which is 10 bytes long.

- Bounding-box style. An integer value that specifies the style code for this bounding-box entry. Style codes are shown in Figure 4-23 on page 4-95.
- Bounding-box left. The coordinate value of the left edge of the bounding box, in 16-bit fixed-point format, with an integer part in the high-order 4 bits and a fractional part in the low-order 12 bits.
- Bounding-box bottom. The coordinate value of the bottom edge of the bounding box, in 16-bit fixed-point format, with an integer part in the high-order 4 bits and a fractional part in the low-order 12 bits.
- Bounding-box right. The coordinate value of the right edge of the bounding box, in 16-bit fixed-point format, with an integer part in the high-order 4 bits and a fractional part in the low-order 12 bits.
- Bounding-box top. The coordinate value of the top edge of the bounding box, in 16-bit fixed-point format, with an integer part in the high-order 4 bits and a fractional part in the low-order 12 bits.

The Family Glyph-Width Table

The font family glyph-width table is used to specify glyph widths for the font family on a per-style basis. This table, which is shown in Figure 4-27, can contain a number of glyph-width subtables, with one subtable for each style in the family.

Figure 4-27 The font family glyph-width table



The family glyph-width table consists of an integer count and a variable number of glyph-width subtables. The table is represented by the `WidTable` data type, which is shown on page 4-48.

- Number of entries. An integer value that specifies the number of bounding-box entries in this table minus 1. This value is represented by the `numWidths` field in the `WidTable` data type.

Each glyph-width subtable in the table is represented by the `WidEntry` data type, which is shown on page 4-48. Each subtable consists of the following elements.

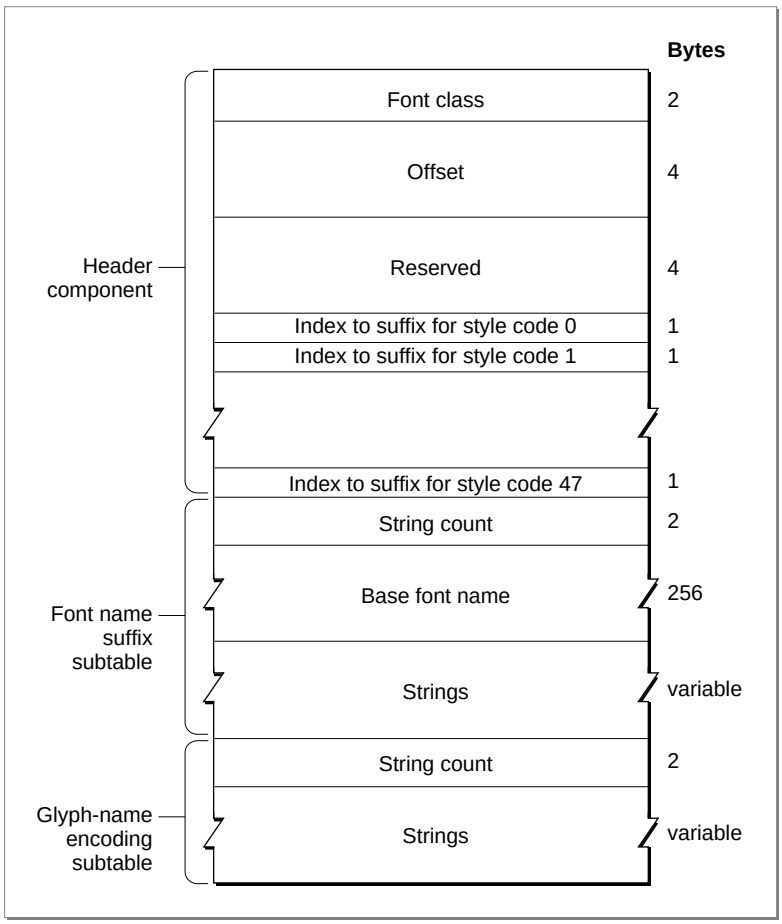
- Style code. An integer value that specifies the style code for this bounding-box entry. Style codes are shown in Figure 4-23 on page 4-95. This value is represented by the `widStyle` field in the `WidEntry` data type.
- Style widths. A variable length array of integer values, with one entry in the array for each glyph in the font. Each width is in 16-bit fixed-point format, with the integer part in the high-order 4 bits and the fractional part in the low-order 12 bits.

The Style-Mapping Table

The printer driver uses *font classes* to differentiate among the different methods of implementing font styles. The style-mapping table provides a flexible way to assign font classes and to specify character-set encodings. The table contains the font class, information about the character-encoding scheme that the font designer used, and a mechanism for obtaining the name of the appropriate printer font. The style-mapping table is primarily used by drivers for high-resolution printers such as the LaserWriter.

The font name suffix subtable and the glyph-encoding subtable that are part of the style-mapping table immediately follow it in the resource data. The font name suffix subtable contains the base font name and the suffixes that can be added to the font family's name to produce a real PostScript name (one that is recognized by the PostScript LaserWriter printer driver). The style-mapping table uses the suffix table to build a font name for a PostScript printer. The glyph-encoding table allows character codes to be mapped to PostScript glyph names. Figure 4-28 shows the structure of the style-mapping table.

Figure 4-28 The style-mapping table



The header component of the style-mapping table contains a list of indexes into the font name suffix subtable, which is described below. The style-mapping table is represented by the `StyleTable` data type, which is shown on page 4-49. The elements of this table are as follows.

- Font class. An integer value that specifies a collection of flags that alert the printer driver to what type of PostScript font this font family is. This value is represented by the `fontClass` field of the `StyleTable` data type. For more information about how these flags are used, see the *LaserWriter Reference* book.

The default font class definition is 0, which has settings that indicate the printer driver should derive the bold, italic, condense, and extend styles from the plain font. Intrinsic fonts are assigned classes (bits 2 through 8) that prevent these derivations from occurring. The meanings of the 16 bits of the `fontClass` word are as follows:

Bit	Meaning
0	This bit is set to 1 if the font name needs coordinating.
1	This bit is set to 1 if the Macintosh vector reencoding scheme is required. Some glyphs in the Apple character set, such as the Apple glyph, do not occur in the standard Adobe character set. This glyph must be mapped in from a font that has it, such as the Symbol font, to a font that does not, like Helvetica.
2	This bit is set to 1 if the font family creates the outline style by changing PaintType, a PostScript variable, to 2.
3	This bit is set to 1 if the font family disallows simulating the outline style by smearing the glyph and whiting out the middle.
4	This bit is set to 1 if the font family does not allow simulation of the bold style by smearing the glyphs.
5	This bit is set to 1 if the font family simulates the bold style by increasing point size.
6	This bit is set to 1 if the font family disallows simulating the italic style.
7	This bit is set to 1 if the font family disallows automatic simulation of the condense style.
8	This bit is set to 1 if the font family disallows automatic simulation of the extend style.
9	This bit is set to 1 if the font family requires reencoding other than Macintosh vector encoding, in which case the glyph-encoding table is present.
10	This bit is set to 1 if the font family should have no additional intercharacter spacing other than the space character.
11–15	Reserved. Should be set to 0.

- Offset. A long integer value that specifies the offset from the start of this table to the glyph-encoding subtable component. This value is represented by the `offset` field of the `StyleTable` data type.
- Reserved. A long integer element reserved for use by Apple Computer, Inc.
- Index to font name suffix subtable. This is an array of 48 integer index values, each of which is a location in the naming table. The value of the first element is an index into the naming table for the string name for style code 0; the value of the forty-eighth element is an index into the naming table for the string name for style code 47. This array is represented by the `indexes` field of the `StyleTable` data type.

The Font Name Suffix Subtable

The font name suffix subtable is part of the style-mapping table. This subtable contains the base font name and the suffixes that can be added to the font family's name to produce a real PostScript name (that is, one that is recognized by the PostScript printer driver). This subtable is represented by the `NameTable` data type, which is described on page 4-49. It consists of the following elements:

- String count. An integer value that specifies the number of strings in the array of suffixes. This value is represented by the `stringCount` field of the `NameTable` data type.
- Base font name. The font family name in a 256 byte long Pascal string. This value is represented by the `baseFontName` field of the `NameTable` data type.
- Strings. A variable length array of Pascal strings, each of which contains the suffixes or numbers specifying which suffixes to put together to produce the real PostScript name. This array is represented by the `strings` field of the `NameTable` data type. This section describes the format of these strings and provides an example of using this subtable.

Each of the strings in the string list contains a sequence of one-byte values, the first of which specifies how many other bytes follow, and each of the following contains an index value. To form the complete name of a font, the base name is concatenated with each of the strings whose index is in the string.

For an example of how this table works, consider the PostScript name of the bold-italic version of the font ExampleFont. Here are the strings of the font name suffix subtable for this font:

Index	Contents
1	`'ExampleFont'`
2	`$02 $09 $0A`
3	`$02 $09 $0B`
4	`$03 $09 $0A $0B`
5	`$02 $09 $0C`
6	`$04 $09 $0C $09 $0A`
7	`$04 $09 $0C $09 $0B`
8	`$05 $09 $0C $09 $0A $0B`
9	`-`
10	`'Bold'`
11	`'Oblique'`
12	`'Narrow'`

QuickDraw has assigned the bold-italic style the number $03; since the base font name is the first entry in this array, you need to access the entry at *i*+1, where *i* is the style value. So, for the bold-italic style, you look at the fourth string. The first byte in this string is $03, which indicates that three string indexes follow.

- The first index is $09, which produces the string `'-'`.
- The second index is $0A, which produces the string `'Bold'`.
- The third index is $0B, which produces the string `'Oblique'`.

By concatenating them together with the base font name, you produce the font name string "ExampleFont-BoldOblique". If the LaserWriter printer driver cannot find the font on the printer, it looks for the font: in version 7.1 and later of system software, the driver looks in the "Fonts" folder; in earlier versions of system software, it first looks in the folder where the driver code is located, then in the System Folder. If the font is there, the driver sends it to the printer. If it is not, the driver sends a QuickDraw bitmap that has already been scaled to the correct size.

Listing 4-4 provides a function for using the style table to build a full PostScript font name.

Listing 4-4 Using the style-mapping table to build a PostScript font name

```
TYPE
   IntegerPtr = ^Integer;
   FamRecPtr = ^FamRec;
   FamRecHdl = ^FamRecPtr;
   StyleTablePtr = ^StyleTable;

FUNCTION MyCompressStyle (aStyle: Style): Integer;
   {A "Set of StyleItem" is mapped into [0..47],}
   {assuming that condense and extend are mutually exclusive}

VAR
   styleCode: Integer;

BEGIN
   styleCode := 0;
   IF bold IN aStyle THEN
      styleCode := styleCode + 1;
   IF italic IN aStyle THEN
      styleCode := styleCode + 2;
   IF outline IN aStyle THEN
      styleCode := styleCode + 4;
   IF shadow IN aStyle THEN
      styleCode := styleCode + 8;
```

```
   IF condense IN aStyle THEN
      styleCode := styleCode + 16
   ELSE IF extend IN aStyle THEN
      styleCode := styleCode + 32;
   MyCompressStyle := styleCode;
END;


FUNCTION MyNthStyleName (index: Integer; q: Ptr): Str255;

VAR
   s: Str255;

BEGIN
   WHILE index > 1 DO
      BEGIN
      q := Ptr(ord(q) + q^ + 1);
                  { assumes q^ = stringlength < 128 ...}
      index := index - 1;
      END;
   BlockMove(q, @s[0], q^ + 1);
                  { assumes q^ = stringlength < 127 ...}
   MyNthStyleName := s;
END;


FUNCTION MyPSFontName(fh: FamRecHdl; aStyle: Style): Str255;
VAR
   stp: StyleTablePtr;
   q: Ptr;     { pointer to Style-name table. }
               { This is not a Pascal structure. }
   PSName, suffixIndices: Str255;
   i, nbOfStrings, offset, whichIndex: Integer;

BEGIN
   PSName := '';
   offset := fh^^.ffStylOff;
   IF offset > 0 THEN
```

```
      BEGIN
      stp := StyleTablePtr(ord(fh^) + offset);
      q := Ptr(ord(stp) + SizeOf(StyleTable));
               { style-name table follows style-mappingTable}
      nbOfStrings := IntegerPtr(q)^;
            { for range checking below }
      q := Ptr(ord(q) + 2);
               { now pointing to basename of font }
      BlockMove(q, @PSName, q^ + 1);
               { basename of font; assumes length < 128 }
      whichIndex := stp^.indexes[MyCompressStyle(aStyle)];
      IF (whichIndex > 1) AND (whichIndex <= nbOfStrings) THEN
         BEGIN
         suffixIndices := MyNthStyleName(whichIndex, q);
         FOR i := 1 TO ord(suffixIndices[0]) DO
            PSName := concat(PSName,
                  MyNthStyleName(ord(suffixIndices[i]), q));
         END
      ELSE { corrupted FOND };
      END
   ELSE { no style mapping table in FOND };
   MyBuildPSFontName := PSName;
END;
```

The Glyph-Name Encoding Subtable

The glyph-name encoding subtable of the style-mapping table allows the font family designer to map 8-bit character codes to PostScript glyph names. This subtable is required when the font family character set is not the Standard Roman character set or the standard Adobe character set. Each entry in this table is a Pascal string, the first byte of which is the character code that is being mapped, and the remaining bytes of which specify the PostScript glyph name.

There is no data type defined to represent the glyph-encoding subtable. The elements of this subtable are as follows:

- String count. An integer value that specifies the number of entries in the encoding subtable.
- Strings. A variable length array of Pascal strings. The first byte of each string is an eight-bit character code, and the remaining bytes are the name of a PostScript glyph. This section beginning on page 4-105, provides an example of using this table.

The following example demonstrates the use of an encoding table in a font resource:

Byte sequence	Use
`$0002`	The number of entries in this encoding table.
`$A8`	The character code of the first character that is being remapped.
`'diamond'`	The name of the PostScript character to be used for character code $A8.
`$A9`	The character code of the second (and last) character that is being remapped.
`'heart'`	The name of the PostScript character to be used for character code $A9.

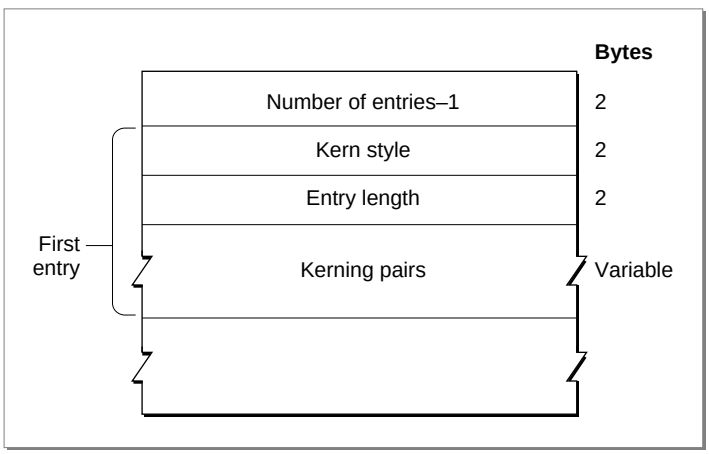
The effect of this table is to assign the PostScript character named diamond to the character code $A8 and to assign the PostScript character named heart to the character code $A9. If either of these character codes has a character assigned to it in the font, that character is replaced by the PostScript character named in the table.

For more information about the font name suffix subtable and the glyph-name encoding table, please see the *LaserWriter Reference*.

The Font Family Kerning Table

The font family kerning table consists of a group of kerning subtable entries. Each subtable contains the measurements of a hypothetical 1-point font of this family with a different stylistic variation. The Font Manager multiplies these measurements by the requested font size. The structure of the font family kerning table is shown in Figure 4-29.

Figure 4-29 The font family kerning table



The font family kerning table is represented by the `KernTable` data type, which is shown on page 4-49. It consists of a count, followed by a variable number of kerning subtable entries.

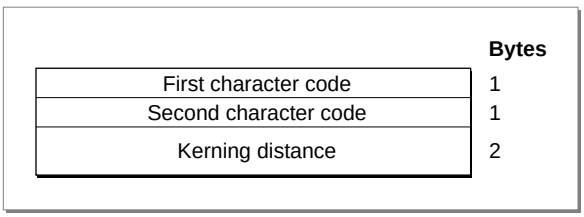
- Number of entries. This is an integer value that specifies the number of kerning subtable entries in this table minus 1. This value is represented by the `numKerns` field of the `KernTable` data type.

Each kerning subtable entry is represented by the `KernEntry` data type, which is described on page 4-50. Each subtable pertains to a specific style code and contains a variable number of kerning pair entries. The style code values are shown in Figure 4-23 on page 4-95. The elements of each subtable entry are as follows:

- Kern style. This is an integer value that specifies the style code to which the kerning information in the subtable pertains. This value is represented by the `kernStyle` field of the `KernEntry` data type.
- Entry length. This is an integer value that specifies the number of bytes in this kerning subtable. This value is represented by the `kernLength` field of the `KernEntry` data type.

Each kerning pair entry specifies a kerning distance in pixels for a pair of glyphs. Each glyph is specified by its character code. The structure of the kerning pair entry is shown in Figure 4-30.

Figure 4-30 A kerning pair entry



Each kerning pair entry is represented by the `KernPair` data type, which is shown on page 4-50. The elements of each entry are as follows:

- First character code. The one-byte character code of the first glyph of the kerning pair. This value is represented by the `kernFirst` field of the `KernPair` data type.
- Second character code. The one-byte character code of the second glyph of the kerning pair. This value is represented by the `kernSecond` field of the `KernPair` data type.
- Kerning distance. The kerning distance, in pixels, for the two glyphs at a point size of 1. This is a 16-bit fixed point value, with the integer part in the high-order 4 bits, and the fractional part in the low-order 12 bits. The Font Manager measures the distance in pixels and then multiplies it by the requested point size. This value is represented by the `kernWidth` field of the `KernPair` data type.

Summary of the Font Manager

Pascal Summary

Constants

```
CONST
   systemFont = 0;
   applFont = 1;
   newYork = 2;
   geneva = 3;
   monaco = 4;
   venice = 5;
   london = 6;
   athens = 7;
   sanFran = 8;
   toronto = 9;
   cairo = 11;
   losAngeles = 12;
   times = 20;
   helvetica = 21;
   courier = 22;
   symbol = 23;
   mobile = 24;
```

Data Types

```
TYPE FMInput =
PACKED RECORD
   family:        Integer;       {font family ID}
   size:          Integer;       {requested point size}
   face:          Style;         {requested font style}
   needBits:      Boolean;       {if bitmaps need to be constructed}
   device:        Integer;       {device driver ID}
   numer:         Point;         {scaling factor numerators}
   denom:         Point;         {scaling factor denominators}
END;
```

```
TYPE FMOutput =
PACKED RECORD
   errNum:        Integer;       {reserved for internal use}
   fontHandle:    Handle;        {handle to font}
   bold:          Byte;          {for drawing of bold style}
   italic:        Byte;          {for drawing of italic style}
   ulOffset:      Byte;          {for drawing of underline style}
   ulShadow:      Byte;          {for drawing of underline shadow style}
   ulThick:       Byte;          {for drawing of underline thickness}
   shadow:        Byte;          {for drawing of shadow style}
   extra:         SignedByte;    {# of pixels added for styles}
   ascent:        Byte;          {ascent measurement of font}
   descent:       Byte;          {descent measurement of font}
   widMax:        Byte;          {maximum width of glyphs in font}
   leading:       SignedByte;    {leading value for font}
   fOutCurStyle:  Byte;          {actual output font style}
   numer:         Point;         {scaling factor numerators}
   denom:         Point;         {scaling factor denominators}
END;

Type WidthTable =
PACKED RECORD
   tabData:       ARRAY [1..256] OF Fixed;
                                 {character widths}
   tabFont:       Handle;        {font record used to build table}
   sExtra:        LongInt;       {extra line spacing}
   style:         LongInt;       {extra line spacing due to style}
   fID:           Integer;       {font family ID}
   fSize:         Integer;       {font size request}
   face:          Integer;       {style (face) request}
   device:        Integer;       {device requested}
   inNumer:       Point;         {scale factors requested}
   inDenom:       Point;         {scale factors requested}
   aFID:          Integer;       {actual font family ID for table}
   fHand:         Handle;        {family record used to build up table}
   usedFam:       Boolean;       {used fixed-point family widths}
   aFace:         Byte;          {actual face produced}
   vOutput:       Integer;       {vertical scale output value}
   hOutput:       Integer;       {horizontal scale output value}
   vFactor:       Integer;       {vertical scale output value}
   hFactor:       Integer;       {horizontal scale output value}
   aSize:         Integer;       {size of actual font used}
   tabSize:       Integer;       {total size of table}
END;
```

```
TYPE FontRec =
RECORD
   fontType:      Integer;       {font type}
   firstChar:     Integer;       {character code of first glyph}
   lastChar:      Integer;       {character code of last glyph}
   widMax:        Integer;       {maximum glyph width}
   kernMax:       Integer;       {negative of maximum glyph kern}
   nDescent:      Integer;       {negative of descent}
   fRectWidth:    Integer;       {width of font rectangle}
   fRectHeight:   Integer;       {height of font rectangle}
   owTLoc:        Integer;       {location of width/offset table}
   ascent:        Integer;       {ascent}
   descent:       Integer;       {descent}
   leading:       Integer;       {leading}
   rowWords:      Integer;       {row width of bit image / 2 }
END;

TYPE FamRec =
RECORD
   ffFlags:       Integer;       {flags for family}
   ffFamID:       Integer;       {family ID number}
   ffFirstChar:   Integer;       {ASCII code of 1st character}
   ffLastChar:    Integer;       {ASCII code of last character}
   ffAscent:      Integer;       {maximum ascent for 1 pt font}
   ffDescent:     Integer;       {maximum descent for 1 pt font}
   ffLeading:     Integer;       {maximum leading for 1 pt font}
   ffWidMax:      Integer;       {maximum widMax for 1 pt font}
   ffWTabOff:     LongInt;       {offset to width table}
   ffKernOff:     LongInt;       {offset to kerning table}
   ffStylOff:     LongInt;       {offset to style-mapping table}
   ffProperty:    ARRAY [1..9] OF Integer;
                                 {style property info}
   ffIntl:        ARRAY [1..2] OF Integer;
                                 {for international use}
   ffVersion:     Integer;       {version number}
END;

TYPE FontAssoc =
RECORD
   numAssoc:      Integer;       {number of entries - 1}
END;
```

```
TYPE AsscEntry =
RECORD
   fontSize:      Integer;       {point size of font}
   fontStyle:     Integer;       {style of font}
   fontID:        Integer;       {font resource ID}
END;

TYPE WidTable =
RECORD
   numWidths:     Integer;       {number of entries - 1}
END;

TYPE WidEntry =
RECORD
   widStyle:      Integer;       {style that entry applies to}
END;

TYPE StyleTable =
RECORD
   fontClass:     Integer;       {the font class of this table}
   offset:        LongInt;       {offset to glyph-encoding subtable}
   reserved:      LongInt;       {reserved}
   indexes:       PACKED ARRAY [0..47] OF SignedByte;
                                 {indexes into the font suffix name }
                                 { table that follows this table}
END;

TYPE NameTable =
RECORD
   stringCount:   Integer;       {number of entries}
   baseFontName:  Str255;        {font family name}
END;

TYPE KernTable =
RECORD
   numKerns:      Integer;    {number of subtable entries}
END;

TYPE KernEntry =
RECORD
   kernStyle:     Integer;    {style the entry applies to}
   kernLength:    Integer;    {length of this entry}
END;
```

```
TYPE KernPair =
RECORD
   kernFirst:     Char;       {Code of 1st character of kerned pair}
   kernSecond:    Char;       {Code of 2nd character of kerned pair}
   kernWidth:     Integer;    {kerning value in 1pt fixed format}
END;

Type FMetricRec =
RECORD
   ascent:        Fixed;         {baseline to top}
   descent:       Fixed;         {baseline to bottom}
   leading:       Fixed;         {leading between lines}
   widMax:        Fixed;         {maximum glyph width}
   wTabHandle:    Handle;        {handle to global width table}
END;
```

Routines

Initializing the Font Manager

```
PROCEDURE InitFonts;
```

Getting Font Information

```
PROCEDURE GetFontName       (familyID: Integer;VAR theName: Str255);
PROCEDURE GetFNum           (theName: Str255;VAR familyID: Integer);
FUNCTION RealFont           (fontNum: Integer;size: Integer): Boolean;
```

Using the Current, System, and Application Fonts

```
FUNCTION GetDefFontSize: Integer;
FUNCTION GetSysFont: Integer;
FUNCTION GetAppFont: Integer;
```

Getting the Characteristics of a Font

```
PROCEDURE FontMetrics       (theMetrics: FMetricRec);
FUNCTION OutlineMetrics     (byteCount: Integer; textPtr: UNIV Ptr;
                             numer: Point; denom: Point; VAR yMax: Integer;
                             VAR yMin: Integer; awArray: FixedPtr;
                             lsbArray: FixedPtr; boundsArray: RectPtr):
                             OSErr;
```

Enabling Fractional Glyph Widths

```
PROCEDURE SetFractEnable    (fractEnable: Boolean);
```

Disabling Font Scaling

```
PROCEDURE SetFScaleDisable  (fscaleDisable: Boolean);
```

Favoring Outline Fonts Over Bitmapped Fonts

```
PROCEDURE SetOutlinePreferred
                            (outlinePreferred: Boolean);
FUNCTION GetOutlinePreferred: Boolean;
FUNCTION IsOutline          (numer,denom: Point) : Boolean;
```

Scaling Outline Fonts

```
PROCEDURE SetPreserveGlyph  (preserveGlyph: Boolean);
FUNCTION GetPreserveGlyph: Boolean;
```

Accessing Information About a Font

```
FUNCTION FMSwapFont         (inRec: FMInput): FMOutPtr;
```

Handling Fonts in Memory

```
PROCEDURE SetFontLock       (lockFlag: Boolean);
FUNCTION FlushFonts: OSErr;
```


C Summary

Constants

```
enum {
   systemFont = 0,
   applFont = 1,
   newYork = 2,
   geneva = 3,
   monaco = 4,
   venice = 5,
   london = 6,
   athens = 7,
   sanFran = 8,
   toronto = 9,
```

```
   cairo = 11,
   losAngeles = 12,
   times = 20,
   helvetica = 21,
   courier = 22,
   symbol = 23,
   mobile = 24,
};
```

Data Types

```
struct FMInput {
   short          family;        /*font family ID*/
   short          size;          /*requested point size*/
   Style          face;          /*requested font style*/
   Boolean        needBits;      /*if bitmaps need to be constructed*/
   short          device;        /*device driver ID*/
   Point          numer;         /*scaling factor numerators*/
   Point          denom;         /*scaling factor denominators*/
};

typedef struct FMInput FMInput;

struct FMOutput {
   short          errNum;        /*reserved for internal use*/
   Handle         fontHandle;    /*handle to font*/
   unsigned char  bold;          /*for drawing of bold style*/
   unsigned char  italic;        /*for drawing of italic style*/
   unsigned char  ulOffset;      /*for drawing of underline style*/
   unsigned char  ulShadow;      /*for drawing of underline shadow style*/
   unsigned char  ulThick;       /*for drawing of underline thickness*/
   unsigned char  shadow;        /*for drawing of shadow style*/
   char           extra;         /*# of pixels added for styles*/
   unsigned char  ascent;        /*ascent measurement of font*/
   unsigned char  descent;       /*descent measurement of font*/
   unsigned char  widMax;        /*maximum width of glyphs in font*/
   char           leading;       /*leading value for font*/
   char           fOutCurStyle;  /*actual output font style*/
   Point          numer;         /*scaling factor numerators*/
   Point          denom;         /*scaling factor denominators*/
};

typedef struct FMOutput FMOutput;
```

```
struct WidthTable {
   Fixed          tabData[256];
                                 /*character widths*/
   Handle         tabFont;       /*font record used to build table*/
   long           sExtra;        /*extra line spacing*/
   long           style;         /*extra line spacing due to style*/
   short          fID;           /*font family ID*/
   short          fSize;         /*font size request*/
   short          face;          /*style (face) request*/
   short          device;        /*device requested*/
   Point          inNumer;       /*scale factors requested*/
   Point          inDenom;       /*scale factors requested*/
   short          aFID;          /*actual font family ID for table*/
   Handle         fHand;         /*family record used to build up table*/
   Boolean        usedFam;       /*used fixed-point family widths*/
   unsigned char  aFace;         /*actual face produced*/
   short          vOutput;       /*vertical scale output value*/
   short          hOutput;       /*horizontal scale output value*/

   short          vFactor;       /*vertical scale output value*/
   short          hFactor;       /*horizontal scale output value*/
   short          aSize;         /*size of actual font used*/
   short          tabSize;       /*total size of table*/
};

typedef struct WidthTable WidthTable;

struct FontRec {
   short          fontType;      /*font type*/
   short          firstChar;     /*character code of first glyph*/
   short          lastChar;      /*character code of last glyph*/
   short          widMax;        /*maximum glyph width*/
   short          kernMax;       /*negative of maximum glyph kern*/
   short          nDescent;      /*negative of descent*/
   short          fRectWidth;    /*width of font rectangle*/
   short          fRectHeight;   /*height of font rectangle*/
   short          owTLoc;        /*location of width/offset table*/
   short          ascent;        /*ascent*/
   short          descent;       /*descent*/
   short          leading;       /*leading*/
   short          rowWords;      /*row width of bit image / 2 */
};

typedef struct FontRec FontRec;
```

```
struct FamRec {
   short          ffFlags;       /*flags for family*/
   short          ffFamID;       /*family ID number*/
   short          ffFirstChar;   /*ASCII code of 1st character*/
   short          ffLastChar;    /*ASCII code of last character*/
   short          ffAscent;      /*maximum ascent for 1 pt font*/
   short          ffDescent;     /*maximum descent for 1 pt font*/
   short          ffLeading;     /*maximum leading for 1 pt font*/
   short          ffWidMax;      /*maximum widMax for 1 pt font*/
   long           ffWTabOff;     /*offset to width table*/
   long           ffKernOff;     /*offset to kerning table*/
   long           ffStylOff;     /*offset to style-mapping table*/
   short          ffProperty[9]; /*style property info*/
   short          ffIntl[2];     /*for international use*/
   short          ffVersion;     /*version number*/
};

typedef struct FamRec FamRec;

struct FontAssoc {
   short          numAssoc;      /*number of entries - 1*/
};

typedef struct FontAssoc FontAssoc;

struct AsscEntry {
   short          fontSize;      /*point size of font*/
   short          fontStyle;     /*style of font*/
   short          fontID;        /*font resource ID*/
};

typedef struct AsscEntry AsscEntry;

struct WidTable {
   short          numWidths;     /*number of entries - 1*/
};

typedef struct WidTable WidTable;

struct WidEntry {
   short          widStyle;      /*style that entry applies to*/
};

typedef struct WidEntry WidEntry;
```

```
struct StyleTable {
   short          fontClass;     /*the font class of this table*/
   long           offset;        /*offset to glyph-encoding subtable*/
   long           reserved;      /*reserved*/
   char           indexes[47];   /*indexes into the font suffix name table*/
};

typedef struct StyleTable StyleTable;

struct NameTable {
   short          stringCount;   /*number of entries*/
   Str255         baseFontName;  /*font family name*/
};

typedef struct NameTable NameTable;

struct KernTable{
   short          numKerns;      /*number of subtable entries*/
};

typedef struct KernTable KernTable;

struct KernEntry {
   short          kernLength;    /* length of this entry*/
   short          kernStyle;     /* style this entry applies to*/
}

typedef struct KernEntry KernEntry;

struct KernPair {
   char        kernFirst;        /*Code of 1st character of kerned pair*/
   char        kernSecond;       /*Code of 2nd character of kerned pair*/
   short       kernWidth;        /*kerning value in 1pt fixed format*/
};

typedef struct KernPair KernPair;

struct FMetricRec {
   Fixed          ascent;        /*baseline to top*/
   Fixed          descent;       /*baseline to bottom*/
   Fixed          leading;       /*leading between lines*/
   Fixed          widMax;        /*maximum glyph width*/
   Handle         wTabHandle;    /*handle to global width table*/
};

typedef struct FMetricRec FMetricRec;
```

Routines

Initializing the Font Manager

```
pascal void InitFonts       (void);
```

Getting Font Information

```
pascal void GetFontName     (short familyID, Str255 theName);
pascal void GetFNum         (ConstStr255Param name, short *familyID);
pascal Boolean RealFont     (short fontNum, short size);
```

Using the Current, System, and Application Fonts

```
pascal short GetDefFontSize (void);
pascal short GetSysFont     (void);
pascal short GetAppFont     (void);
```

Getting the Characteristics of a Font

```
pascal void FontMetrics     (const FMetricRec *theMetrics);
pascal OSErr OutlineMetrics (short byteCount, const void *textPtr,
                             Point numer, Point denom, short *yMax,
                             short *yMin, FixedPtr awArray,
                             FixedPtr lsbArray, RectPtr boundsArray);
```

Enabling Fractional Glyph Widths

```
pascal void SetFractEnable  (Boolean fractEnable);
```

Disabling Font Scaling

```
pascal void SetFScaleDisable(Boolean fscaleDisable);
```

Favoring Outline Fonts Over Bitmapped Fonts

```
pascal void SetOutlinePreferred
                            (Boolean outlinePreferred);
pascal Boolean GetOutlinePreferred
                            (void);
pascal Boolean IsOutline    (Point numer, Point denom);
```

Scaling Outline Fonts

```
pascal void SetPreserveGlyph
                            (Boolean preserveGlyph);
pascal Boolean GetPreserveGlyph
                            (void);
```

Accessing Information About a Font

```
pascal FMOutPtr FMSwapFont  (const FMInput *inRec);
```

Handling Fonts in Memory

```
pascal void SetFontLock     (Boolean lockFlag);
pascal OSErr FlushFonts     (void);
```

Assembly-Language Summary

Trap Macros

Trap Macros with Trap Words

Trap macro name	Trap word
_FMSwapFont	$A901
_FontMetrics	$A835
_GetFNum	$A900
_GetFontName	$A8FF
_InitFonts	$A8FE
_RealFont	$A902
_SetFontLock	$A903
_SetFScaleDisable	$A834

Trap Macros Requiring Routine Selectors

_FontDispatch

Selector	Routine
$7000	IsOutline
$7001	SetOutlinePreferred
$7008	OutlineMetrics
$7009	GetOutlinePreferred
$700A	SetPreserveGlyph
$700B	GetPreserveGlyph
$700C	FlushFonts

Global Variables

ApFontID	Font ID of application font.
CurFMInput	The current QuickDraw FMInput record for FMSwapFont.
FDevDisable	Disables device-defined extra spacing for styles.
FMDefaultSize	The default point size.
FMgrOutRecc	The current FMOutput record from FMSwapFont.
FONDID	The resource ID of the last font family resource used.
FractEnable	If nonzero, fractional widths are enabled.
FScaleDisable	If nonzero, scaling is disabled.
FScaleHFact	The current horizontal scale factor.
FScaleVFact	The current vertical scale factor.
IntlSpec	International software installed if the value of this is greater than zero.
LastFOND	Handle to last family record used.
LastSPExtra	The most recent value of extra spacing for styles.
ROMFont0	Handle to font record for system font.
SynListHandle	Handle to synthetic font list.
SysFontFam	If nonzero, the font ID to use for the system font.
SysFontSiz	If nonzero, the size of the system font.
UsedFWidths	A flag determining whether fractional widths were used for the most recent font request.
WidthListHand	Handle to a list of handles to recently used width tables (referred to in some places as jFontInfo).
WidthPtr	Pointer to global width table.
WidthTabHandle	Handle to global width table.