Managing Fonts: QuickDraw

(Not Recommended)

Carbon > Text & Fonts

2007-12-11

Contents

Figures, Tables, and Listings

Introduction to Managing Fonts: QuickDraw

Important: QuickDraw is deprecated in Mac OS X version 10.5 and later. Use Core Text instead for Mac OS X v10.5 and later, as described in *Core Text Programming Guide*. For applications that must run on Mac OS X v10.4 and earlier, use Apple Type Services (ATS), as described in *Apple Type Services for Fonts Programming Guide*.

The Font Manager is a QuickDraw-based 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 reads font data from the font files on the system's hard disk and other locations (such as from a PDF, the network, and CD-ROM discs) and creates the bitmap images used to display text. You can use the Font Manager to determine the characteristics of a font, change certain font settings, favor outline fonts over bitmapped fonts, and manipulate fonts in memory. With Mac OS 9 and CarbonLib, additions to the Font Manager API now allow support of more comprehensive font management including: listing, installing, activating, and deactivating font files and retrieving information from the repository of font data.

Font management consists of two tasks: managing font files (installing, activating, and deactivating), and maintaining a repository of font data. In the past, font management relied upon the Finder and the Fonts folder along with specialized code in the Font Manager and Resource Manager to populate and manage the font database. Because the Resource Manager was not designed to handle the sizeable data storage and retrieval needs of professional-level font collections, third-party font management utilities were created to address the limitations inherent in the System suitcase model. Font management utilities have had to manipulate directly system font data, a practice that is not supported in Mac OS X.

Font management capabilities in Mac OS 8 and 9 increased dramatically with the addition of Apple Type Services for Unicode Imaging (ATSUI). ATSUI introduced a text rendering engine and font management model to address the data storage and retrieval needs of professional-level font collections. ATSUI is designed to handle multiple font technologies and file formats. In addition, the Font Manager and QuickDraw have been rewritten to integrate the glyph data cache and font data management system. Developers no longer need to depend upon the Resource Manager to access and modify the font collection. Instead, CarbonLib provides a new Font Management API as part of the Font Manager.

The revised Font Manager supports for the following tasks:

- Enumerating fonts and font families.
- Accessing information on font families, such as name, character encoding, and member fonts.
- Accessing information on fonts, such as font technology, format, name, foundry, and version.
- Accessing font data.
- Creating and managing a Font menu.

Managing Fonts: QuickDraw Concepts

This chapter provides an overview of font management on the Macintosh and includes recent changes in how fonts are managed under Carbon. The information in this chapter will help you 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.

If you are new to font management on the Macintosh, you should read the entire chapter. If you are also new to handling text in Mac OS, you should first read "Introduction to Text on the Macintosh." See

http://developer.apple.com/documentation/mac/Text/Text-13.html

General font-related information and programming suggestions are found in the discussion of font handling in that chapter. You should also be familiar with QuickDraw. See

http://developer.apple.com/documentation/mac/Text/Text-126.html

If you have already worked with the Font Manager and are familiar with fonts, you should read "Font Support in the Mac OS" (page 18) and "How the Font Manager Tracks Changes to the Font Database" (page 27).

If you are writing a font editor, you also need to read the *TrueType Font Format Specification*, and the *OpenType Font Format Specification*. You can find links to these specifications, as well as additional information on TrueType Fonts, on the Apple typography website:

http://developer.apple.com/fonts/

This chapter begins with an overview of the terminology used 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

This chapter then describes

- how font resources are used to store fonts, including information on data–fork and resource–fork fonts
- font and font family references that replace the font ID
- how the Font Manager finds the information your application or QuickDraw requests
- how the Font Manger and QuickDraw work together to create or alter glyph bitmaps for displaying and printing
- how the Font Manager keeps track of changes to the font database

Fonts

This section describes the terminology used throughout the Font Manager documentation to refer to the individual elements of a font, different types of fonts, and the different functions a font can have. If you are already familiar with font terminology on the Macintosh, you can skip this section.

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 operation "+". 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.

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** 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.

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 computer that does not use the Mac OS. Different regions of the world require different glyphs for their typography, which make it impossible for any one standard to be complete.

Character-encoding schemes were developed to manage the assignment of different glyphs to character codes in different fonts. An encoding scheme 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 described in the style mapping table. See *Inside Macintosh: Text* located on the Apple developer website:

http://developer.apple.com/documentation/mac/Text/Text-2.html

The Font Manager handles 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 type face, 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 in the Mac OS 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 created 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. The Font Manager can generate thousands of point sizes from the same outline font. For example, 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**. 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.

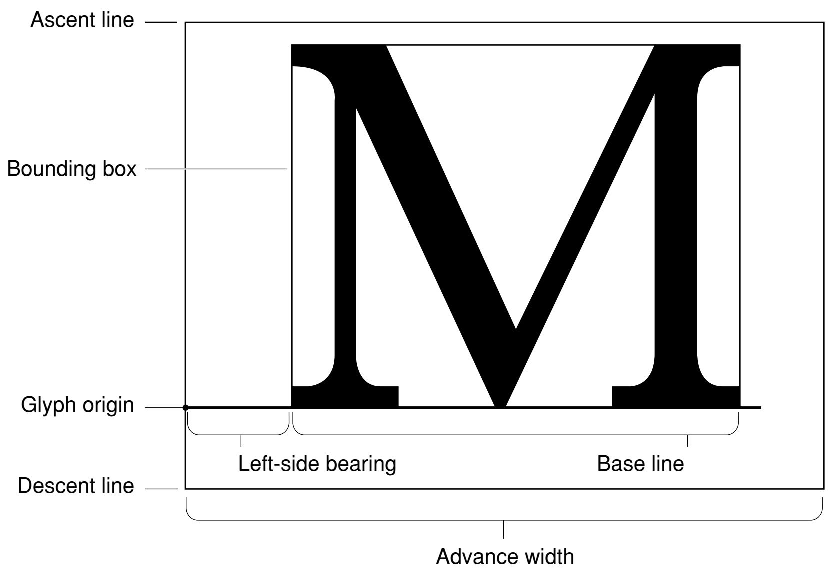
A font has a **font name**, which is stored as a string such as "Geneva" or "New York". The font name is usually the same name as the typeface from which it was derived. If a font is not in plain style, its style becomes part of the font's name and distinguishes it from the plain style of that font: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

A **font family reference** is a reference to an opaque structure that represents a collection of fonts with the same design characteristics. You can use the font family reference to refer to font families while you application is running. However, if you need to refer a font or font family every time your application launches (that is, across reboots), you should refer to the font or font family by name, and not by the font family reference.

Font Measurements

Font designers use specific terms for the measurements of different parts of a glyph, whether outline or bitmapped. Figure 2-1 shows the terms used for the most frequently used measurements.

Figure 1-1 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 2-1, the **bounding box** of a glyph is the smallest rectangle that entirely encloses the pixels of the glyph. 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.

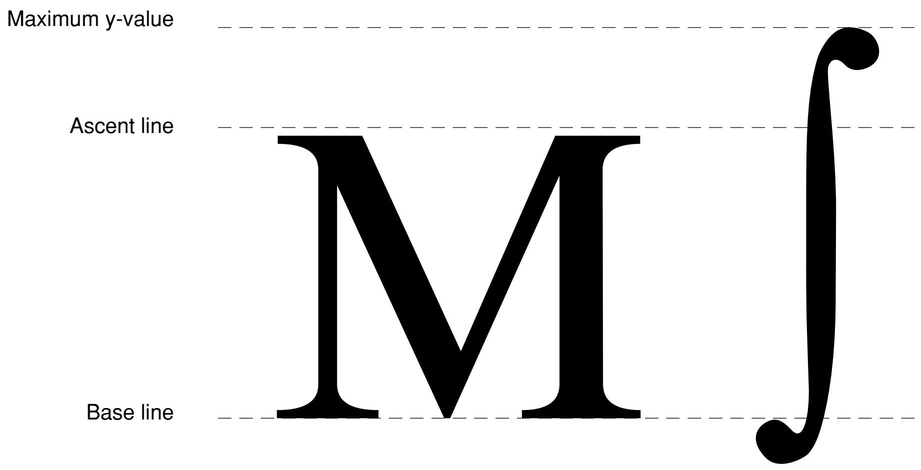
If all of the glyph images in the font were superimposed using a common glyph origin, the smallest rectangle that would enclose the resulting image is the **font rectangle**.

The glyphs of a **fixed-width font** all have the same advance width. Fixed-width fonts are also know 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".

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 aligns approximately with the tops of the uppercase letters in the font, because these are 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 aligns with the bottoms of descenders (the tails on the glyphs such as "p" or "g"), and it is 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 2-2, 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" (page 41) for details.

Figure 1-2 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 1/72 of an inch, so there are exactly 72 points per inch on the Macintosh.

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.

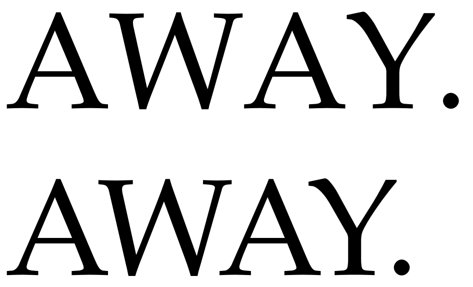
There is not a strict typographical standard for defining a point size. It is often, but not always, the sum of the ascent, descent, and leading values for a font. Point size is used by a font designer to indicate the size of a font relative to other fonts in the same family. Glyphs from fonts with the 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 could affect the alignment and spacing of text, so you need to take it into account when your application lays out text. You may need to determine the actual height of the text that you are displaying by using the QuickDraw function `MeasureText` 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` function for both outline and bitmapped fonts. QuickDraw returns similar information in the `GetFontInfo` function. 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 is overlaps another glyph. The period in the top portion of Figure 2-3 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.

Figure 1-3 Unkerned text (top) and kerned text (bottom)



Font Rendering

This section describes how TrueType outline fonts are rendered by a font rendering engine. Although PostScript fonts are rendered using different mathematical algorithms, the two rendering methods are similar in that both use mathematical equations to estimate contours. For information on PostScript font rendering, see the *PostScript Language Reference Manual* by Adobe Systems, Inc., available through Addison-Wesley Publishing Company.

TrueType outline fonts are stored in an outline font (`'sfnt'`) resource as a collection of outline points. (Do not 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.) A font rendering engine 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 curvature of the line between the on-curve points. Two consecutive on-curve points define a straight line. To draw a curve, the font rendering engine needs a third point that is off the curve and between the two on-curve points.

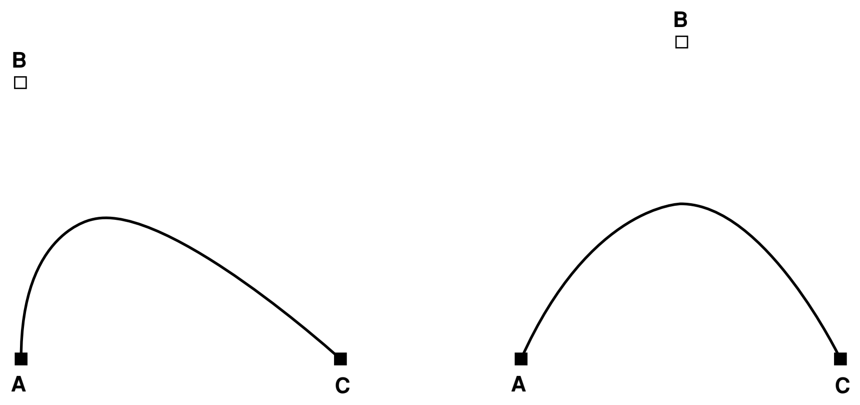
A font rendering engine uses this parametric Bézier equation to draw the curves of the glyph from an outline font:

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

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 s an off-curve point.

Figure 2-4 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.

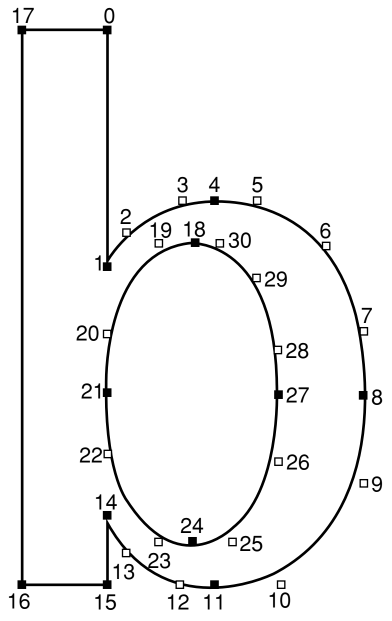
Figure 1-4 The effect of an off-curve point on two Bézier curves



■ = on-curve point
□ = off-curve point

A font designer can use any number of outline points to create a glyph outline. These points must be numbered consecutively along the contours of the glyph, because a font rendering engine draws lines on curves sequentially. This process produces a glyph such as the lower case “b” in Figure 2-5.

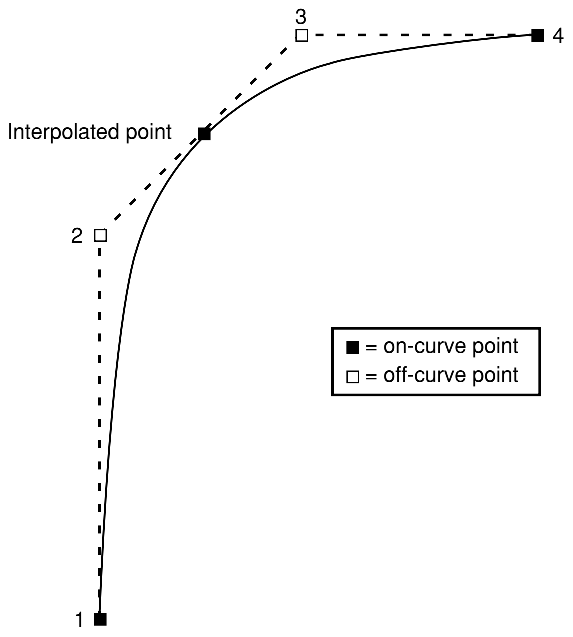
Figure 1-5 An outline with points on and off the curve



■ = on-curve point
□ = off-curve point

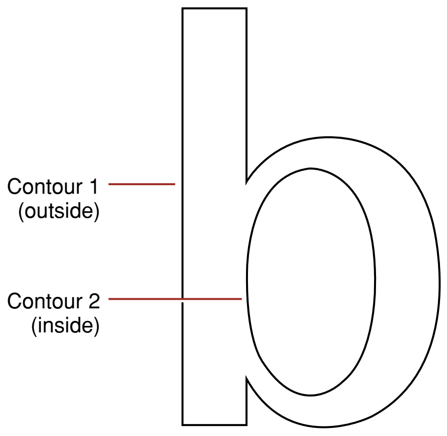
There are several groups of points in Figure 2-5 that include two consecutive off-curve points. For instance, points 2 and 3 are both off-curve. In this case, the font rendering engine interpolates an on-curve point midway between the two off-curve points, thereby defining two Bézier curves, as shown in Figure 2-6. Note that this additional on-curve point is used for creation of the glyph only; the font rendering engine does not alter the outline font resource's list of points.

Figure 1-6 A curve with consecutive off-curve points



When the font rendering engine 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 2-5, the font rendering engine 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 2-7.

Figure 1-7 A glyph from an outline font



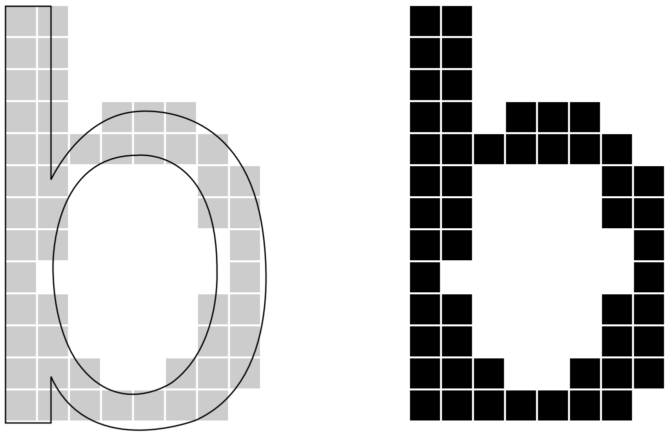
At this stage the glyph does not have a fixed point size. Remember that point size is measured as the distance from the base line of one line of text to the base line of the next line of single-spaced text. The font rendering engine has the measurements of the outline relative to the base line and ascent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

The font rendering engine 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 rendering engine 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 rendering engine sets the bit for the bitmap.

Because there are two contours for the glyph in Figure 2-7 the font rendering engine 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, such as the Å glyph, have three or more contours.

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 2-8 that the outline has produced an unattractive bitmap. There are gaps and blocky areas that would not be found in the high-resolution version of the same glyph.

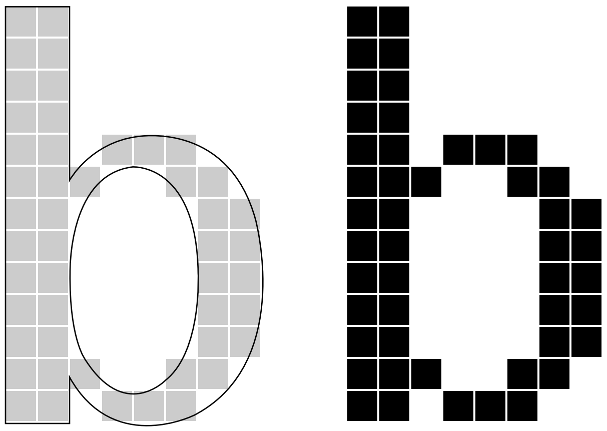
Figure 1-8 An unmodified glyph from an outline font at a small point size



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 resources 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 2-9 is the same one depicted in Figure 2-8, except that the font rendering engine has applied the instructions to the figure and produces 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 rendering engine 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 *TrueType Font Format Specification* (available through http://developer.apple.com/fonts/).

In the case of the Mac OS, 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 1-9 An instructed glyph from an outline font



Font Support in the Mac OS

Prior to the release of Mac OS 8.5, the Font Manager supported only bitmapped fonts and TrueType outline fonts whose data were stored only in the resource fork. With the release of Mac OS 8.5, Apple began to provide support for TrueType and OpenType fonts that are stored in data–fork files. As a result, you can use the Font Manager to access both resource–fork and data–fork fonts. There are many new Font Manager functions you can use to assure your application supports both kinds of fonts.

Data–Fork Fonts

There are currently two types of data–fork fonts: TrueType and OpenType. TrueType data–fork fonts originated on the Windows platform. These fonts store data in an `'sfnt'` structure in a file with the extension `.TTF`. In the past, to use these fonts in the Mac OS, a user would first have to use a utility to convert the font as a resource-based suitcase. Starting with Mac OS 8.5, this is no longer necessary.

TrueType collections, or `.TTC` files, contain several `'sfnt'` font structures organized with a simple directory scheme. This organization allows the individual fonts to share complete tables among each other. Until now, these fonts have also not been usable on the Macintosh. TrueType collections have the file type `'ttcf'`.

OpenType represents new naming and packaging for fonts. Adobe, in conjunction with Microsoft, have defined a data–fork-based, `'sfnt'` structured font file to contain PostScript font data. The `'sfnt'` structure is in a file with the extension `.OTF`. The glyph data itself is stored in a new format called CFF, or Compact Font Format. The structure was designed such that these fonts behave the same as TrueType fonts on the Windows platform. These fonts are also designed to work in the Mac OS.

An OpenType font file contains data, in table format, that comprises either a TrueType or a PostScript outline font. Rasterizers use combinations of data from the tables contained in the font to render the TrueType or PostScript glyph outlines. Some of this supporting data is used no matter which outline format is used; some of the supporting data is specific to either TrueType or PostScript.

The Finder in Mac OS 8.5 was enhanced to support identification and autorouting of data–fork font files. Individual font files (`.TTF` and `.OTF`) and collection files (`.TTC`). are assigned file types `'sfnt'` and `'ttcf'`, respectively, by Internet Config when the fonts are first brought onto the system. When dropped into the System Folder, the Finder assigns appropriate icons to the files and automatically routes them to the Fonts folder.

Currently, there is no support for double-clicking a data–fork font file. They behave most like the individual TrueType files (as opposed to suitcases) and as such, they cannot be renamed or opened.

Resource–Fork Fonts

The Font Manager takes care of the details of how fonts are stored in resources, reading the resource fields when required and building **internal representations** of the data stored in them.

There are three types of resources associated with resource–fork fonts:

- Bitmapped font (`'NFNT'`) resources describe bitmapped fonts. These bitmapped font resources have an identical structure to the earlier `'FONT'` resources, which they replace, but the bitmapped font resources add a more flexible numbering scheme.
- Outline font (`'sfnt'`) resources describe outline fonts.
- 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.

Note: `'FONT'` resources that are not referenced by a `'FOND'` resource are not supported. `'FONT'` resources are not supported in Mac OS X and should be converted to `'NFNT'` resources.

Font File Formats

There are a variety of font file formats you should be aware of when working with the Font Manager. The font file formats can have different implications for how the fonts are enumerated and font data is accessed in Mac OS X versus in Mac OS 8 and 9. The most important file formats include the following:

- Suitcase. There are resource–fork and data–fork suitcases. If you want to use a suitcase in Mac OS 9, you should package it as a traditional resource–fork suitcase. Data–fork suitcases (`.dfont`) are supported only in Mac OS X.
- Multiple-file resource-based PostScript fonts. Outline data is stored in a separate file from the font suitcase. Mac OS 8 and 9 supports plain and Multiple Master LWFN-class fonts. (CID fonts not packaged in an `'sfnt'` structure, that is, "naked" CID and OCF are not currently supported in Mac OS X.)
- Data–fork fonts. These are fonts that consist of a single data–fork file that contains `'sfnt'` font data structures. Typically, there is only one `'sfnt'` per file, as with OpenType and Windows TrueType fonts, but there may be more, in which case you would have a TrueType collection file. There is no suitcase associated with the font, so there is no `'FOND'` resource or associated bitmapped fonts. This means that in Mac OS 9 (with or without CarbonLib) these fonts do not belong to any font family and cannot be used with the Font Manger and QuickDraw. (See "Compatibility Issues" (page 20) for information on associating a `FOND` resource with a data–fork font.) In Mac OS X, the system synthesizes font family information to allow the fonts to be accessible to the entire system.

Font Family and Font References

The font family ID and font ID have been replaced by the font family reference and font reference.

A **font family reference** (`FMFontFamily`) represents a collection of fonts with the same design characteristics. It replaces the standard QuickDraw font identifier and may be used with all QuickDraw functions, including `GetFontName` and `TextFont`. A font family reference does not imply a particular **script system**, nor is the character encoding of a font family determined by an arithmetic mapping of its value.

Note: The Font Manager uses opaque data types to store information about fonts and font families. You cannot access a font family or font object directly. Instead, you must use the accessor functions provided in the Font Manager API.

A **font family** is a collection of fonts, each of which is identified by a **font reference** (`FMFont`) that maps to a single **font object** registered with the font database.

Each font object maps to one or more combinations of a font family reference and a standard QuickDraw style. This is roughly equivalent to the information stored in the font association table of the `'FOND'` resource handle, with the exception of a descriptor for point size. Since each font object represents the entire array of point sizes for a given font, a font family reference and style together fully specify any given font object. A **font family instance** (`FMFontFamilyInstance`) is a data structure that contains a font family reference, font style, and font size.

The system software always maps the **system font** to `sysFont` and the **application font** to `applFont`. The system software uses the system font for drawing items such as system menus and system dialogs. The application font is the font that your application uses for text unless specified otherwise by you or the user. The fonts assigned as the system and application fonts depend on the Appearance Manager settings as well as the Script Manager variables `smScriptSysFond`, `smScriptAppFond`, `smScriptSysFondSize`, and `smScriptAppFondSize`.

Compatibility Issues

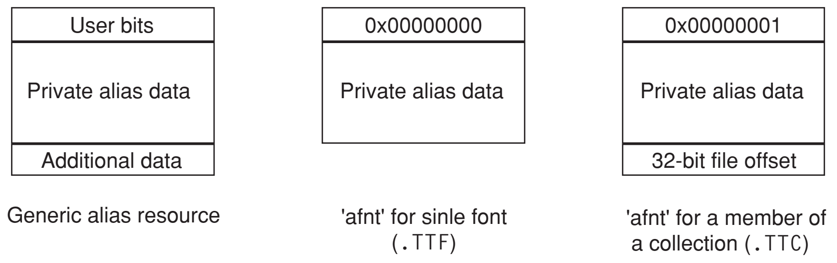
You should read this section only if you need to parse a font association table for a `'FOND'` resource.

The Font Manager cannot make use of any font for which there is no `'FOND'` resource, as that resource forms the basis for accessing and processing fonts. Data–fork fonts do not have a `'FOND'` resource and so are not usable as is by the Font Manager in Mac OS 8 and 9 or the Classic environment, but they are usable in Mac OS X. It is not possible right now for the Font Manager to synthesize a `'FOND'` resource, but there is a work-around the Font Manager uses to connect a `'FOND'` resource to a data fork. It involves the creation of an `'afnt'`, or font alias, resource.

The Font Manager uses the `'FOND'` font association table to choose from among the available bitmap sizes in a font, or to elect to use an outline font. Each entry in the font association table contains a size field to indicate the bitmap size, with the special value of zero denoting an outline font (`'sfnt'` resource). The Font Manager uses the entry's resource ID to access the appropriate type of resource using the Resource Manager.

Starting with Mac OS 8.5, Apple defined the size (-1) to indicate that an outline font is stored in a data fork. The font association table entry's resource ID is `'afnt'`. An `'afnt'` resource is simply an alias record that has the structure shown in Figure 2-10. For more information on alias records, see the *Alias Manager Reference* in Carbon File Management Documentation.

Figure 1-10 An alias record



Apple currently reserves all of the user bits and has defined the least significant:

1. A value of all zeros for the user bits indicates a simple, single–font data fork. The alias data can be passed to the Alias Manager for resolution. The target font is then expected to begin at offset zero in the file.

2. If the least significant byte of the user bit word is 1, then an offset into the data–fork file is present following the private alias data in the resource. This is used for targeting a member of a TrueType collection file. Thus, an `'afnt'` resource targets a single font.

Starting with Mac OS 9, a font management utility can activate data for fonts by simply making a `'FOND'` and associated `'afnt'` resources visible to the Font Manager. The data for files themselves can reside outside the Fonts folder. When creating the alias that goes in the `'afnt'` resource, do not make a "minimal" alias. Ideally, the alias should be relative to the temporary file (or the suitcase file, if a suitcase is created). Keeping the data–fork file out of the Fonts folder is efficient since the system opens the data–fork file only when needed. Although you can reference data forks on network volumes, it is not recommended because the network connection could close unexpectedly in midstream. Data–fork files can also be stored on and accessed from CDs.

How the Font Manager Works

The Font Manager works in conjunction with QuickDraw. QuickDraw treats the glyphs that make up text as small images that make up a larger, coherent image. It uses size information, such as height and width, the same way it users similar information when arranging an image that does not contain text. 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 draws 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 provides QuickDraw with the necessary information to lay out text and implement the appropriate stylistic variations.

How QuickDraw Requests a Font

When your application calls a QuickDraw function, QuickDraw gets information from the Font Manager about the font specified in the current graphics port 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 structure. This structure contains the font family reference, the size, the style, and the scaling factors of the font request.

QuickDraw makes a font request by filling in a font input structure (`FMInput`) and calling the function `FMSwapFont`. If your application needs to make a font request in the same way that QuickDraw does, you can call `FMSwapFont`. The `FMSwapFont` function has been optimized to return as quickly as possible if the request is for a recently requested font. Building the global width tables, which is described in "How the Font Manager Calculates Glyph Widths" (page 26), is one of the more time-consuming tasks in this process, which is why the Font Manager maintains a cache of width tables.

The Font Manager looks for the font family of the requested font and from that determines information about which font it can use to meet the request.

How the Font Manager Responds to a Font Request

The Font Manager returns information to QuickDraw in a font output structure (`FMOutput`). This structure contains a handle to the font resource that the `FMInput` structure 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 structure, it attempts to find a font family resource for the requested font family. If the font family resource is available, the Font Manager looks in the font family resource for the font family reference of the appropriate font resource to match the request. If a font family resource is not available, the Font Manager takes additional steps until it finds a suitable substitute.

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 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 slants the characters as it draws them. The **QuickDraw styles** include the values`bold`, `italic`, `underline`, `outline`, `shadow`, `condense`, and `extend`.

With the additional complication of having both bitmapped and outline fonts available, this process can sometimes produce results other than those that you expect. The Font Manager can be set to favor either outline or bitmapped fonts when both are available to meet a request. (For information on how to do this, see "Favoring Outline or Bitmapped Fonts" (page 40)). 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 is not much that you can do about such situations except to be aware that setting the Font Manager to prefer one kind of font over another has implications beyond what you might expect.

How the Font Manager Scales Fonts

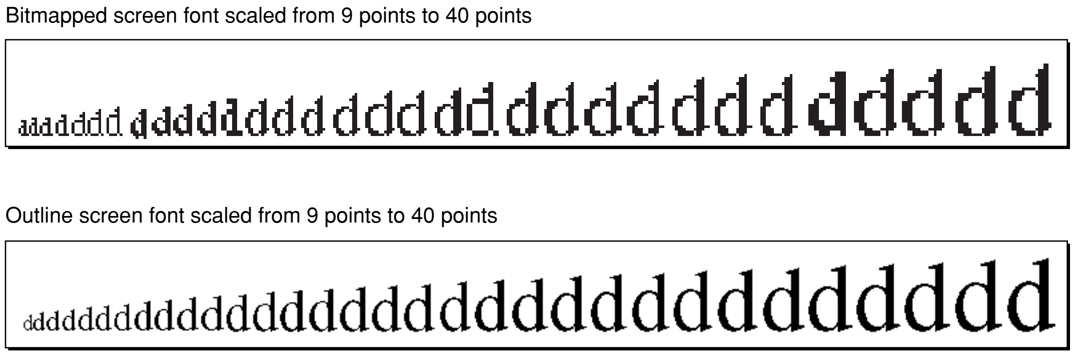
Font scaling is the process of changing a glyph from one size or shape to another. The Font Manager and QuickDraw can scale bitmapped and outline fonts in three ways: changing a glyph's point size, modifying the glyph (but not its point size) for display on a different device, and altering the shape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

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" (page 25)). If the glyph is from an outline font, the Font Manager uses the outline for that glyph to create a bitmap (see "The Scaling Process for an Outline Font" (page 25)).

Figure 2-11 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.

Figure 1-11 A comparison of scaled bitmapped and outline fonts


Bitmapped screen font scaled from 9 points to 40 points
Outline screen font scaled from 9 points to 40 points

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 creates the bitmap. Outlines give better results than bitmaps when scaled, because the outlines are intended for use at all point sizes, whereas 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.

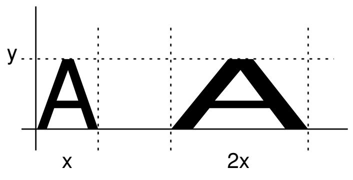
With some QuickDraw calls, your application can also use the Font Manager to explicitly scale a glyph by stretching or shrinking it. For example, you can use the Font Manager to scale a glyph that is normally 12 points to one that is 12 points high but stretched to the width of the entire page of text. 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 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.

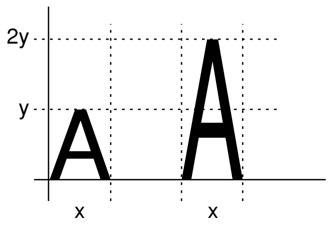
In Figure 2-12, the font scaling factors are 2/1 in the horizontal direction and 1/1 in the vertical direction. The glyph stays the same height, but grows twice as large in width.

Figure 1-12 A glyph stretched horizontally



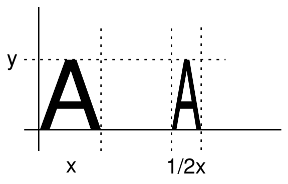
In Figure 2-13, the font scaling factors are 2/1 in the vertical direction and 1/1 in the horizontal direction. The glyph stays the same width, but grows to twice its original height.

Figure 1-13 A glyph stretched vertically



In Figure 2-14, the 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 1-14 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.

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 down 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 a substitute font, as described in "`How the Font Manager Responds to a Font Request`" (page 22). 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 applications by calling the function `SetFScaleDisable`. 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.

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 Manger returns this font and returns the 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 "Using Fractional Glyph Widths and Font Scaling" (page 41).

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 for 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" (page 41). 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 printer 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**, a data structure that is allocated by the system. 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 resources, 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 data structure, 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" while the width table for outline fonts is described in "The Horizontal Device Metrics Table." Both sections are in *Inside Macintosh: Text* located on the Apple developer documentation website.

2. It the Font Manger does not find this table, it looks in the font family data structure 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" in *Inside Macintosh: Text* located on the Apple developer documentation website.

3. If the Font Manager does not find a family glyph-width table, it derives the global character widths from the integer width contained in the width/offset table in the bitmapped font record as described in "The Bitmapped Font ('NFNT') Resource" in *Inside Macintosh: Text* located on the Apple developer documentation website.s

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 function `MeasureText` or by looking in the global width table. You can get a handle to the global width table by calling the function `FontMetrics`.

How the Font Manager Tracks Changes to the Font Database

The Font Manager uses a database to keep track of available fonts and provides a simple mechanism to track changes to the font database. Any operation that adds, deletes, or modifies one or more font family or font objects triggers an update of a global generation count. Each font family and font object modified during a transaction is tagged with a copy of the generation count.

You can use Font Manager accessor functions (`FMGetGeneration`, `FMGetFontGeneration`, `FMGetFontFamilyGeneration`) to get the current value of the generation count and the generation tag associated with a font family and font object. Then, you can use these values along with Font Manager enumeration functions to identify the changes in the database.

Managing Fonts: QuickDraw Tasks

This chapter provides instructions and code samples for the most common tasks you can accomplish with the Font Manager such as enumerating fonts, getting font display information, and handling the standard Font menu. It also provides information on making existing applications Carbon-compliant.

Several sections show examples of code that is no longer supported under Carbon and equivalent implementations that use Carbon-compliant font access and data management functions. In particular, see "Enumerating Font Families and Fonts" (page 30) and "Handling the Standard Font Menu" (page 35).

The code samples assume you are developing your application in Mac OS using CarbonLib. All code samples are in C.

Initializing the Font Manager

You do not need to initialize the Font Manager in Carbon. As a result, the function `InitFonts` is not supported in Carbon.

Accessing Font Data

You should no longer use the Resource Manager routines to access the contents of the data stored in the font **suitcase files** of standard Mac OS resource fork–based fonts, including `'FOND'`, `'FONT'`, `'NFNT'`, `'sfnt'`, and `'typ1'` resources.

To access other font resource data, you should use the function `FMGetFontContainer`. This function gets the location of the file that contains the data. You then pass the file reference or specification to the Resource Manager or File Manager to obtain the information you need. You should not assume that the **resource identifier** of a font family resource determines the QuickDraw identifier of the font family obtained with `GetResInfo`.

To access the individual `'sfnt'` tables that you need, you should use the function `FMGetFontTable`. This function gets the requested table and copies it into a caller-provided buffer. You may want to first use the function `FMGetFontTableDirectory` to get the `'sfnt'` directory and parse it to determine which tables are available.

Enumerating Font Families and Fonts

There are two categories of iterators available to you: font family iterators and font iterators. A font family iterator enumerates fonts that are associated with a font family. When you enumerate font families, you also use a font family instance iterator to access all instances of a particular font family.

A font iterator enumerates outline fonts without regard to family. You must use a font iterator to enumerate fonts that are not associated with a font family, such as a data–fork font in Mac OS 9. In Mac OS X, every font is a member of at least one font family.

Platform Considerations

In theory, you can use Font Manager iterators to enumerate all fonts available on the system. In practice, there are a number of issues you need to consider when enumerating fonts and font families.

You cannot use a font iterator to enumerate the bitmapped `'FONT'`/`'NFNT'` font resources traditionally used with QuickDraw-based applications. Instead, you must use a font family instance iterator followed by a font family iterator to obtain each of the outline and bitmapped font resources that comprise the font family. If the font family instance iterator returns an invalid font object, then you can assume the particular style and size for the font family reference is a bitmapped font.

In Mac OS X, the font family iterator and font family instance iterator have been extended to support non–resource–based fonts, such as the Windows TrueType (`'.ttf'`/`'.ttc'`) and OpenType PostScript (`'.otf'`) fonts. Note that in Mac OS 9, you can still gain access to these fonts through a standard font iterator or font object data access function of the Font Manager.

In Mac OS 9 (with or without CarbonLib) data-fork fonts do not belong to any family and the operating system does not synthesize a font family reference, so you cannot enumerate these fonts with a font family iterator.

Enumerating Font Family Data

To enumerate font family data, do the following:

- Set up a filter if you want to restrict iteration to specific criteria. This is optional.
- Create iterators. To access both the font family and individual font data, you need to create a font family iterator and a font instance iterator.

- Get font and font family information.
- Dispose of the iterators.

Listing 3-1 shows a function `MyGetFontInformation` that iterates through the fonts registered with the system. Your application would need to add code that does something with the font information it retrieves. Error-handling code has been omitted to make the sample function more readable. Following this listing is a detailed explanation for each line of code that has a numbered comment.

Listing 2-1 Getting font family and font instance information

```
void MyGetFontInformation (WindowRef window, FMGeneration myGeneration)
{
     FMFontFamilyIterator                       myFontFamilyIterator;
     FMFontFamily                               myFontFamily;
     OSStatus                                   status1 = 0;
     OSStatus                                   status2 = 0;
     FMFontFamilyInstanceIterator               myInstanceIterator;
     FMFont                                     myFont;
     FMFontStyle                                myStyle;
     FMFontSize                                 mySize;
     Str255                                     myFontString;
     FMFilter                                   myFilter;

     myFilter.format = kFMCurrentFilterFormat;                             // 1
     myFilter.selector = kFMGenerationFilterSelector;                      // 2
     myFilter.filter.generationFilter = myGeneration;                      // 3

    FMCreateFontFamilyIterator (&myFilter, NULL, kFMGlobalIterationScope,
                        &myFontFamilyIterator);                            // 4
    while (status1 == 0)
    {
        status1 = FMGetNextFontFamily (&myFontFamilyIterator,
                                       &myFontFamily);
        // Your code here.                                                 // 5

        FMCreateFontFamilyInstanceIterator (myFontFamily,
                                   &myInstanceIterator);                   // 6
        while (status2 == 0)
            {
                status2 = FMGetNextFontFamilyInstance (&myInstanceIterator,
                                &myFont, &myStyle, &mySize);
                // Your code here.                                         // 7
            }
        status2 = 0;
        FMDisposeFontFamilyInstanceIterator (&myInstanceIterator);         // 8
    }
    FMDisposeFontFamilyIterator (&myFontFamilyIterator);                   // 9
}
```

Here's what the code does:

1. Sets filter options. You can use filter options to restrict the iteration to specified criteria. If you don't want to restrict iteration, you don't need to set filter options. Pass `NULL` instead. The current filter format is the only one you can specify right now.

2. Specifies to use a generation filter.
3. Restricts iteration to fonts of a specified generation.
4. Creates a font family iterator. The global scope options specifies to iterate through all fonts registered with the system. Use `kFMLocalIterationScope` to restrict the iteration only to the fonts accessible to your application.
5. Insert code that does something with the font family information. For example, you can get the string associated with the font family by calling the function `FMGetFontFamilyName (myFontFamily, myFontString);`
6. Creates a font family instance iterator.
7. Insert code that does something with the font family instance information.
8. Disposes of the contents of the iterator before creating another the next time through the loop.
9. Disposes of the font family iterator.

Enumerating Font Data

If you want to enumerate outline fonts without regard to family, you can use a font iterator. Listing 3-2 shows a `MyGetFontObjects` function that iterates through fonts accessible to your application, and gets font format information for each font. Error-handling code has been omitted to make the sample function more readable. Following this listing is a detailed explanation for each line of code that has a numbered comment.

Listing 2-2 Getting information about a font

```
void MyGetFontObjects()
{
    FMFontIterator                  myFontIterator;
    OSStatus                        status = 0;
    FMFont                          myFont;
    FourCharCode                    myFormat;

    status = FMCreateFontIterator (NULL, NULL, kFMLocalIterationScope,
                                  &myFontIterator);                        // 1
    while (status == noErr)
    {
        status = FMGetNextFont (&myFontIterator, &myFont);
        status = FMGetFontFormat (myFont, &myFormat);                      // 2
    }
    status = FMDisposeFontIterator (&myFontIterator);                      // 3
}
```

Here’s what the code does:

1. Creates a font iterator. The first parameter is `NULL` because we are not filtering font information. The second parameter is `NULL` because we are not using a custom filer function. The constant `kFMLocalIterationScope` specifies to apply the iterator only to those fonts accessible to your application. Pass an uninitialized font iterator.

2. Gets the font format. This is one possible thing you could so with the font information. You could replace this line with code that does something else with the font information.

3. Disposes of the contents of the font iterator.

Rewriting Resource-Based Code

The following source code is an example of how to rewrite a standard QuickDraw enumeration of font families and fonts using the new Font Manger functions. Listing 3-3 is the old resource-based code and Listing 3-4 is the equivalent implementation using the new font access and data management functions. You can compare the old and new code samples to see how you can make your already-existing applications Carbon-compliant. Following each listing is a detailed explanation for each line of code that has a numbered comment.

Listing 2-3 Enumerating font families and fonts using resource-based functions

```
SInt16  index;
short   savedResFile;

savedResFile = CurResFile();                                         // 1

index = CountResources (kFONDResourceType);                          // 2

while (index > 0)
    {
        Handle              fontRecHandle;
        Handle              outlineFontHandle;
        AsscEntry           entry;

        fontRecHandle = (FontRecHdl) GetIndResource('FOND', index);
        while (fontRecHandle)
            {
                GetResInfo ((Handle) fontRecHandle, NULL,
                             NULL, rsrcName);                        // 3

                count = ((FontAssoc*)(*fontRecHandle +
                                sizeof (FamRec)))->numAssoc + 1;     // 4
                entry = (AsscEntry*)(*fontRecHandle + sizeof(FamRec) +
                                sizeof(FamRec) + sizeof(FontAssoc));
                while ( count-- > 0 )
                    {
                    if (entry->fontSize == 0)                        // 5
                        {
                            UseResFile (HomeResFile((Handle)
                                                fontRecHandle));     // 6
                            fontHandle = Get1Resource('sfnt',
                                              entry->fontID);
                            if (fontHandle != NULL)
                                            ...
                        }
                    entry++;                                         // 7
                }
                fontRecHandle = (FontFamilyRecordHdl)
                                    GetNextFOND((Handle) fontRecHandle);
            }
        index--;
    }

UseResFile (savedResFile);                                                          // 8
```

Here's what the code does:

1. Calls the Resource Manager function `CurResFile` to get the file reference number of the current resource file. You need to save the file reference number so you can restore it later.
2. Calls the Resource Manager function `CountResources` to obtain the number of font families. This is the number of iterations you need to do.
3. Calls the Resource Manager function `GetResInfo` to get the font family identifier.
4. Gets the values needed to walk through the font association table.
5. Checks for an outline font. A font size of 0 represents an outline font.
6. Gets the resource ID. An `'sfnt'` resource ID is unique only within the font file.
7. Increments `entry` so we can get the next entry of the font association table.
8. Calls the Resource Manager function `UseResFile` to restore the original resource file.

Listing 2-4 Enumerating font families and fonts using Carbon-compliant functions

```
FMFontFamilyIterator            fontFamilyIterator;
FMFontFamilyInstanceIterator    fontFamilyInstanceIterator;
FMFontFamily                    fontFamily;

status = FMCreateFontFamilyInstanceIterator (0,
                            &fontFamilyInstanceIterator);                           // 1
status = FMCreateFontFamilyIterator (NULL, NULL, kFMLocalIterationScope,
                                        &fontFamilyIterator);                       // 2

while ( (status = FMGetNextFontFamily (&fontFamilyIterator, &fontFamily))
                                        == noErr)
    {
        FMFont                          font;
        FMFontStyle                     fontStyle;
        FMFontSize                      fontSize;

        status = FMResetFontFamilyInstanceIterator (fontFamily,
                                &fontFamilyInstanceIterator);                       // 3
        while ( (status =
                FMGetNextFontFamilyInstance(
                        &fontFamilyInstanceIterator, &font,
                            &fontStyle, &fontSize)) == noErr )
        {
                if ( fontSize == 0 )                                                // 4
                    ...
        }
    }
```

```
if (status != noErr && status != kFMIterationCompleted) {                  // 5
    ...
}

FMDisposeFontFamilyIterator (&fontFamilyIterator);                          // 6
FMDisposeFontFamilyInstanceIterator (&fontFamilyInstanceIterator);
```

Here's what the code does:

1. Creates a dummy instance iterator to avoid creating and destroying it in the loop.
2. Creates an iterator to enumerate the font families.
3. Points the instance iterator to the current font family.
4. Checks for a font size of 0. A font size of 0 represents an outline font.
5. Checks to see if there is an error as a result of the iteration. You should add your error-handling code after the `if` statement.
6. Disposes of the iterators.

Converting Between Font and Font Family Data

A **font reference** is an index to a **font object**, which is a specific outline font without regard to font family. A **font family instance** is a reference to a font family paired with a QuickDraw style. At times it can be useful to find a font family of which a font object is a member, or to find the font that is associated with a font family and style. To accomplish these conversions, you can use the functions `FMGetFontFamilyInstanceFromFont` and `FMGetFontFromFontFamilyInstance`.

These functions are not inverses of each other, because a font object can be a member of more than one font family. This means if you call the function `FMGetFontFromFontFamilyInstance` and then call the function `FMGetFontFamilyInstanceFromFont`, you will not necessarily get the **font family reference** you supplied when you called `FMGetFontFromFontFamilyInstance`.

Handling the Standard Font Menu

The functions in the Resource Manager for creating a standard Font menu continue to work with CarbonLib and Mac OS X for resource–fork–based fonts, but you should use the new functions in the Menu Manager that support all font formats available through the new Carbon-compliant functions. Table 3-1 lists the functions you should use to replace the resource-based functions.

Table 2-1 Replacements for resource-based functions

Instead of	Use this
`AppendResMenu`	`CreateStandardFontMenu`

Instead of	Use this
`InsertFontResMenu`	`CreateStandardFontMenu`
`InsertIntlResMenu`	`CreateStandardFontMenu`
`GetMenuItemText` and `GetFNum`	`GetFontFamilyFromMenuSelection`

The Menu Manager provides your application with a standard user interface for soliciting font choices from the user. To create a menu that displays font names, you should use the Menu Manager function `CreateStandardFontMenu`. The `CreateStandardFontMenu` function adds the names of all font objects in the font database as items in the Font menu. This ensures that any changes to the font database do not affect your application and that the display of items in the Font menu is not dependent on how fonts are stored in system software.

The Menu Manager function `GetFontFamilyFromMenuSelection` finds the font family reference and style from menu identifier and menu item number returned by the function `MenuSelect`. For more information on the functions `CreateStandardFontMenu`, `UpdateStandardFontMenu`, and `GetFontFamilyFromMenuSelection`, see the Menu Manager documentation.

The following source code is an example of how to rewrite a standard event handler for the Font menu using the Font Manager data structures `FMFontFamily` and `FMFontStyle`. Listing 3-5 is the resource -based code and Listing 3-6 is the equivalent implementation using the Font Manger data structures.

Listing 2-5 A standard event handler for the Font menu using resource-based functions

```
MenuHandle      menu;
SInt16          menuID;
SInt16          menuItem;
Str255          menuItemText;
...
menuResult = MenuSelect (theEvent->where);
menuID = HiWord (menuResult);
menuItem = LoWord (menuResult);
if ( menuID == kFontMenuResID )
   {
        menu = GetMenuHandle (menuID);
        GetMenuItemText (menu, menuItem, menuItemText);
        if ( status == noErr )
                TextFont (GetFNum (menuItemText));
    }
```

Compare the resource-based code sample in Listing 3-5 with the code sample in Listing 3-6 to see how to make an existing application Carbon-compliant.

Listing 2-6 A standard event handler for the Font menu using Carbon-compliant functions

```
MenuRef         menu;
MenuID          menuID;
MenuItemIndex   menuItem;
FMFontFamily    fontFamily;
FMFontStyle     style;
...
menuResult = MenuSelect (theEvent->where);
menuID = HiWord (menuResult);
```

```
menuItem = LoWord (menuResult);
if ( menuID == kFontMenuResID )
   {
        menu = GetMenuHandle (menuID);
        status = GetFontFamilyFromMenuSelection (menu, menuItem,
                        &fontFamily, &style);
        if ( status == noErr )
            {
                TextFont (fontFamily);
                TextFace (style);
            }
    }
```

Adding Font Sizes 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. You can determine which bitmapped fonts are available by using the function `RealFont`. `RealFont` returns `TRUE` if the font is available in the requested point size and `FALSE` if it is not. For outline fonts, the `RealFont` function returns `TRUE` for almost any size. The font designer may set a lower limit to the point sizes at which an outline font looks acceptable. If the size requested is smaller than this lower limit, the `RealFont` function returns `FALSE`.

Activating and Deactivating Fonts

You use the functions `FMActivateFonts` and `FMDeactivateFonts` to activate or deactivate fonts. Fonts are activated and deactivated in groups defined by their representation in the file system as font suitcase files or other font file formats supported by the Font Manager. In order to activate or deactivate fonts, they must be of a format supported by the Font Manager.

By activating and deactivating fonts, you can control which fonts are available to your users. In a publication environment, font control facilitates handling large sets of fonts. It can minimize font mismatch problems or assure users have access to special fonts needed for their work.

Listing 3-7 shows how to activate and deactivate fonts that are in an application bundle. The code assumes the fonts you want to activate are located in the Resources folder in the application bundle. Following the listing is a detailed explanation for each line of code that has a numbered comment.

Listing 2-7 Code to activate and deactivate fonts in an application bundle

```
CFBundleRef     myAppBundle = NULL;
CFURLRef        myAppResourcesURL = NULL;
FSRef           myResourceDirRef;
FSSpec          myResourceDirSpec;

myAppBundle = CFBundleGetMainBundle (myAppBundle);                    // 1

if (NULL != myAppBundle)
{
    myAppResourcesURL = CFBundleCopyResourcesDirectoryURL (myAppBundle);    // 2
    if (NULL != myAppResourcesURL)
    {
        (void) CFURLGetFSRef (myAppResourcesURL, &myResourceDirRef);     // 3
        status = FSGetCatalogInfo (&myResourceDirRef, kFSCatInfoNone, NULL,
                                NULL, &myResourceDirSpec, NULL);        // 4
        if ( noErr == status )
        {
            status = FMActivateFonts (&myResourceDirSpec, NULL, NULL,
                                kFMLocalActivationContext);             // 5
        }
    }
}
// Your code here to obtain the FSSpec.                                 // 6
status = FMDeactivateFonts (&myResourceDirSpec, NULL, NULL,
                                        kFMDefaultOptions);             // 7
```

Here's what the code does:

1. Calls the Core Foundation Bundle Services function `CFBundleGetMainBundle` to obtain the reference for the application bundle.

2. Calls the Core Foundation Bundle Services function `CFBundleCopyResroucesDirectoryURL` to get the URL of the Resources folder located in the application bundle. The fonts should be stored in this bundle. To activate fonts, you need a pointer to the file specification of the file that contains the font data you want to activate. The next few lines of code covert the URL to an `FSSpec`.

3. Calls the Core Foundation URL Services function `CFURLGetFSRef` to get the file system reference (`FSRef`) associated with the URL of the Resources folder.

4. Calls the File Manager function `FSGetCatalogInfo` to get the `FSRef` associated with the `FSSpec`.

5. Calls the Font Manager function to activate the fonts in the local activation context. This provides the current application with access to the fonts, but other applications do not have access to them. If you want to activate fonts so they are available to other applications, use the constant `kFMGlobalActivationContext` instead of `kFMLocalActivationContext`.

6. Before you can deactivate fonts, you need to obtain the `FSSpec`. You can either follow the steps above to obtain the URL and convert it to an `FSSpec`, or you can write your code so that it caches the `FSSpec` or the URL of the Resources folder located in the application bundle.

7. Calls the Font Manager function `FMDeactivateFonts` to deactivate the fonts. You should deactivate locally activated fonts when you no longer need them. If you've activated the fonts globally you may want to leave them active.

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 reference. If you have stored the name of the font in the document, you can find its font family reference by calling the function `FMGetFontFamilyFromName`. However, if the font is not present in the system software when the user opens the document, the function `FMGetFontFamilyFromName` returns `kFMInvalidFontFamilyErr`.

Storing a font name is the most reliable method of finding a font, because the name does not change from one computer system to another. If you want to make sure that the version of the font is the same on different computer systems, you can use the FontSync function `FNSFontReferenceMatch` to compare the two fonts. See the FontSync documentation for more information.

If the font versions are different, 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).

Getting Font Measurement Information

You sometimes need to get font measurement information for the current font in the current graphics port. The Font Manager provides two functions for this purpose: `FontMetrics` and `OutlineMetrics`. In addition, you can get font measurement information by calling the QuickDraw function `GetFontInfo`. 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 function `FontMetrics` 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` function returns these measurements in a font metrics structure, which allows fractional widths. (By contrast, the QuickDraw function `GetFontInfo` returns integer widths.) In addition to these four measurements, the font metrics structure includes a handle to the global width table, which in turn contains a handle to the font family resource for the current text font.

Listing 3-8 shows a sample function `MyGetFontMetrics`. The function `FontMetrics` checks font metrics for the font associated with the current graphics port. Following this listing is a detailed explanation for each line of code that has a numbered comment.

Listing 2-8 Getting font metrics for the font in the current graphics port

```
void MyGetFontMetrics ()
{
  FMetricRecPtr     myFontMetrics;
  Fixed             myAscentFixed;
  Fixed             myDescentFixed;
  Fixed             myLeadingFixed;
  Fixed             myMaxWidthFixed;
  FontMetrics (&myFontMetrics);                                                          // 1
  myAscentFixed   = myFontMetrics.ascent;
  myDescentFixed  = myFontMetrics.descent;
  myLeadingFixed  = myFontMetrics.leading;
  myMaxWidthFixed = myFontMetrics.widMax;
 // Your code here.                                                                      // 2
}
```

Here's what the code does:

1. Gets the metrics for the font associated with the graphics port.
2. Insert you code here to do something with the metrics information.

The function `OutlineMetrics` returns measurements for glyphs to be displayed in an outline font. The function returns an error if the font in the current graphics port is any other kind of font. (You can call the function `IsOutline` to find out whether the Font Manager would choose an outline font for the current graphics port.) These measurements include the **maximum y-value**, **minimum y-value**, **advance width**, **left-side bearing**, and **bounding box**.

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.

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 must choose 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" (page 22) for more information.

You can change this default behavior by calling the function `SetOutlinePreferred`. 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 the function `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 calling the function `IsOutline`.

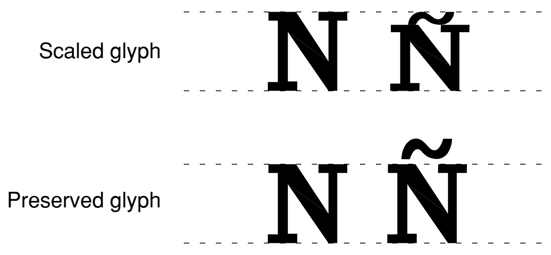
Preserving the Shapes of Glyphs

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 function `OutlineMetrics`. `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 function `FontMetrics`. 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 3-1 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 2-1 The difference between a scaled glyph and a preserved glyph



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

The function `GetPreserveGlyph` 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 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 printed by high-resolution printers. (See “How the Font Manager Calculates Glyph Widths” (page 26)).

You can enable the use of fractional glyph widths with the function `SetFractEnable`. If you set the parameter `fractEnable` to `TRUE`, the Font Manager uses fractional glyph 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 function `SetFScaleDisable`. If you set the `fontScaleDisable` parameter of this function 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 always scales an outline font.

For more information fractional glyph widths and font scaling, see the QuickDraw documentation.

Document Revision History

This table describes the changes to *Managing Fonts: QuickDraw*.

Date	Notes
2007-12-11	Removed broken link. Deprecated document. Updated document overview and introduction to cite Core Text and ATS as replacement technologies.
2003-12-10	Changed title from *Managing Fonts With the Font Manager* to *Managing Fonts: QuickDraw*.
2002-03-01	Updated information in the section "Platform Considerations" (page 30) and made a few minor changes to clarify some of the concepts. Corrected a few typographical and stylistic errors. Added sample code for activating and deactivating fonts located in an application bundle. Index added.
2002-02-13	Fixed errors in the code shown in Listing 3-7 (page 38).
2000-12-02	Preliminary release of *Managing Fonts With the Font Manager*. The API Reference chapter has been updated to include new Carbon-compliant functions; Concepts and Tasks chapters have been added.

Managing Fonts: QuickDraw Glossary

advance width The full width of a glyph, measured from the glyph origin to the other side of the glyph, including any white space on either side.

application font The default font for use by applications. The application font is defined by each script system.

ascent line An imaginary horizontal line chosen by the font's designer that aligns approximately with the tops of the uppercase letters in the font. See also base line (page 45), descent line (page 45).

base line An imaginary horizontal line that coincides with the bottom of each character in a font, excluding descenders (tails on letters such as p).

bitmapped font A font made up of bitmapped glyphs. Compare outline font (page 47).

bitmapped glyph A bitmap of a character designed for display at a fixed point size for a particular display device.

bounding box The smallest rectangle that entirely encloses the pixels of a bitmapped glyph.

character A symbol standing for a sound, syllable, or notion used in a script; one of the simple elements of a written language, for example, the lowercase letter "a" or the number "1". Compare character code (page 45), glyph (page 46).

character code A value representing a text character. Text is stored in memory as character codes. Each script system's (`'KCHR'`) resource converts the virtual key codes generation by the keyboard or keypad into character codes; each script system's fonts convert the character codes into glyphs for display or printing.

character encoding The organization of the numeric codes that represent the characters of a character set in memory.

derived font A font whose characteristics are partially determined by modifying an intrinsic font. A derived font might be one whose characters are scaled form an intrinsic font to achieve a desired size or are slanted to achieve an italic style.

descent line An imaginary horizontal line that usually aligns with the bottoms of descenders (the tails on the glyphs such as "p" or "g"), and it is the same distance from the base line for every glyph in the font. See also ascent line (page 45).

fixed-width font A font whose characters all have the same width. Compare proportional font (page 47).

font A collection of glyphs that usually have some element of design consistency such as the shapes of the counters, the design of the stem, the stroke thickness, or the use of serifs.

font container A file used to store data for a font.

font depth The number of bits per pixel.

font description A table that contains data that fully describes a font.

font family A group of outline and bitmapped fonts that share certain characteristics and a common family name.

font family container A file used to store data for a font family.

font family reference A reference to an opaque structure that represents a collection of fonts with the same design characteristics. It replaces the font ID and is compatible with Font Manager, QuickDraw, Resource Manager, and Script Manager functions.

font ID (1) A font-family ID. (2) A number that identifies the resource file of a particular individual font, of type `'FONT'`, `'nfnt'`, or `'sfnt'`. The font ID is no longer used in Font Manager functions, as it has been replaced by the font family reference (page 46).

font family instance A font family reference and Quick Draw style that together defines an outline or bitmapped font that is a member of a font family.

filter In the Font Manger, a filter restricts the scope of the enumeration and activation function to the font families and fonts that match a particular technology, font container, or generation tag.

font name (1) The name, such as Geneva or Kyoto, given to a font family to distinguish it from other font families. (2) A set of specific information in a font object about a font, such as its family name, style, copyright date, version, and manufacturer. Some font names are used to build menus in an application, whereas other names are used to identify the font uniquely.

font object A specific outline font without regard to family.

font rectangle The smallest rectangle enclosing all the glyphs in a font if the images are all superimposed over the same glyph origin.

font reference A reference to an opaque structure that represents a font object.

font scaling The process of changing a glyph from one size or shape to another. The Font Manager can scale bitmapped and outline fonts by changing both sizes and shapes of glyphs.

font scaling factors Ratios that indicate how the Font Manager should scale a glyph in the vertical and horizontal directions.

font size The size of the glyphs in a font in points; nominally a measure of the distance from the base line of one line of text to the base line of the next line or single-spaced text.

font technology A methodology used to store and image fonts, such as PostScript or TrueType.

generation count A value used to track changes to the font database. Any operation that adds, deletes, or modifies one or more font family or font references triggers an update of a global generation count.

global width table A data structure that stores information about fractional glyph widths.

glyph The distinct visual representation of a character in a form that a screen or printer can display. A glyph may represent one character (the lowercase a), more than one character (the fi ligature), part of a character (the dot over an i), or a nonprinting character (the space character). See also character (page 45).

glyph origin The point on a base line used as a reference location for drawing a glyph.

kerning The process of drawing part of a glyph so that is overlaps another glyph.

imaging system The system used to render text or graphics.

internal representation The structure and organization used to represent objects, such as font objects, in the operating system.

intrinsic font A font whose characteristics are entirely defined in a `'FONT'` or `'NFNT'` resource. The plain-style font of any family is an intrinsic font. Other styles may or may not be intrinsic. Compare derived font (page 45).

intrinsic style See intrinsic font (page 46).

leading (pronounced "LED-ing") The amount of blank vertical space between the descent line of one line of text and the ascent line of the next line of single-space text. In early typesetting, strips of lead were placed between lines of type for spacing, hence the term. See also line spacing (page 47).

left-side bearing The white space between the glyph origin and the visible beginning of the glyph.

line spacing The vertical distance between two lines of type, measured from base line to base line. For example, 10/12 indicates 10-point type with 12 points base to base (that is, with 2 points of leading).

maximum-y value The highest location on the vertical axis; it corresponds to the tallest glyph in a font

minimum-y value The lowest location on the vertical axis; it corresponds to the bottom of the longest descender in a font.

missing-character glyph The glyph in a font that is drawn when no glyph is defined for a character code in a font.

monospaced font See fixed-width font (page 45).

off-curve point An outline point between two on-curve points that determines the curve of the line between the two on-curve points. A Bézier curve is defined by all three points.

on-curve point One of the outline points that determines the shape of a Bézier curve. Two on-curve points and one off-curve point are required to define the curve.

outline font A font made up of outline glyphs in a particular typeface and style, with no size restriction. The Font Manager can generate thousands of point sizes from the same outline font.

point A unit used to measure font size. Traditionally, there are 72.27 points per inch. However, QuickDraw and the PostScript language define 1 point to be 1/72 of an inch, so there are exactly 72 points per inch in the Mac OS.

proportional font Any font in which different characters have different widths; thus, the space taken up by words having the same number of letters can vary.

QuickDraw style The set of styles supported by QuickDraw—bold, italic, underline, outline, shadow, condense, and extend.

resource Data of any kind stored in a defined format in a file's resource fork and managed by the Resource Manager.

resource fork entries Code or noncode resource entries in a resource fork.

resource identifier An integer that identifies a specific resource of a given type.

script system A collection of software facilities that provides for the representation of a specific writing system. It consists of a set of keyboard resources, a set of international resources, one or more fonts, and possibly a script system extension (1-byte or 2-byte). Scripts systems include Roman, Japanese, Arabic, Traditional Chinese, Simplified Chinese, Hebrew, Greek, Thai, and Korean. Types of script systems include 1-byte simple, 1-byte complex, and 2-byte.

style A visual attribute, other than size, applied as a systematic variation to the plain (unstyled) characteristics of a font glyph. For example bold, italic, underline, outline, shadow, condense, and extend.

system font The font used to display text in menus, dialog boxes, alert boxes, and so forth in a given script system.

suitcase file A traditional packaging mechanism for Mac OS fonts that usually contains bitmapped and outline font data as well as family-wide information, such as font metrics.

table directory A table that contains data for registering fonts with the operating system.

Index