INSIDE MACINTOSH

QuickDraw GX Typography

Addison-Wesley Publishing Company

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408-996-1010

ISBN 0-201-40679-9
1 2 3 4 5 6 7 8 9-CRW-9897969594
First Printing, June 1994

The paper used in this book meets the EPA standards for recycled fiber.

Library of Congress Cataloging-in-Publication Data

Inside Macintosh. QuickDraw GX typography / [by Apple Computer, Inc.].
p. cm.
Includes index.
ISBN 0-201-40679-9
1. Macintosh (Computer)—Programming. 2. Computer graphics.
3. QuickDraw GX. I. Apple Computer, Inc. II. Title: QuickDraw GX typography
QA76.8.M3I56235 1994
006.6'765—dc20 94-17334
CIP

Contents

Figures, Tables, and Listings

Preface — About This Book

Chapter 1 — Introduction to QuickDraw GX Typography 1-1

Chapter 2 — Typographic Shapes 2-1

Chapter 5 Layout Shapes 5-1

Chapter 6 Typographic Styles 6-1

About This Book

QuickDraw GX is an integrated, object-based approach to graphics programming on Macintosh computers. This book, *Inside Macintosh: QuickDraw GX Typography*, describes the QuickDraw GX typographic shapes that display text and shows you how to create and manipulate those shapes.

For application programming purposes, QuickDraw GX augments the capabilities of some of the Macintosh system software managers documented in other parts of *Inside Macintosh*. It supplements the chapters "QuickDraw Text" and "Font Manager," as well as parts of the chapter "Script Manager" and the appendix "International Resources" in *Inside Macintosh: Text*. QuickDraw GX and other Macintosh managers coexist without conflict, however, and you can use both in the same program. Furthermore, for tasks outside the scope of QuickDraw GX you need to use other parts of the system software. For example, for multilingual word breaks and line breaks, you need to use the Macintosh Text Utilities.

Before you read this book, you should already be familiar with *Inside Macintosh: QuickDraw GX Objects*. Figure P-1 on page xxii shows the suggested reading order for the QuickDraw GX books. A pictorial overview of *Inside Macintosh*, including the QuickDraw GX suite of books, appears at the back of this book.

What to Read

This book is for all QuickDraw GX programmers. You can read the chapters in any order, except that the first chapter introduces concepts that the others build on:

Chapter 1, "Introduction to QuickDraw GX Typography," provides an overview of typography and QuickDraw GX and describes how text is stored, measured, and displayed. Read this chapter first.

Chapter 2, "Typographic Shapes," describes how to create and use QuickDraw GX shapes for the text you draw.

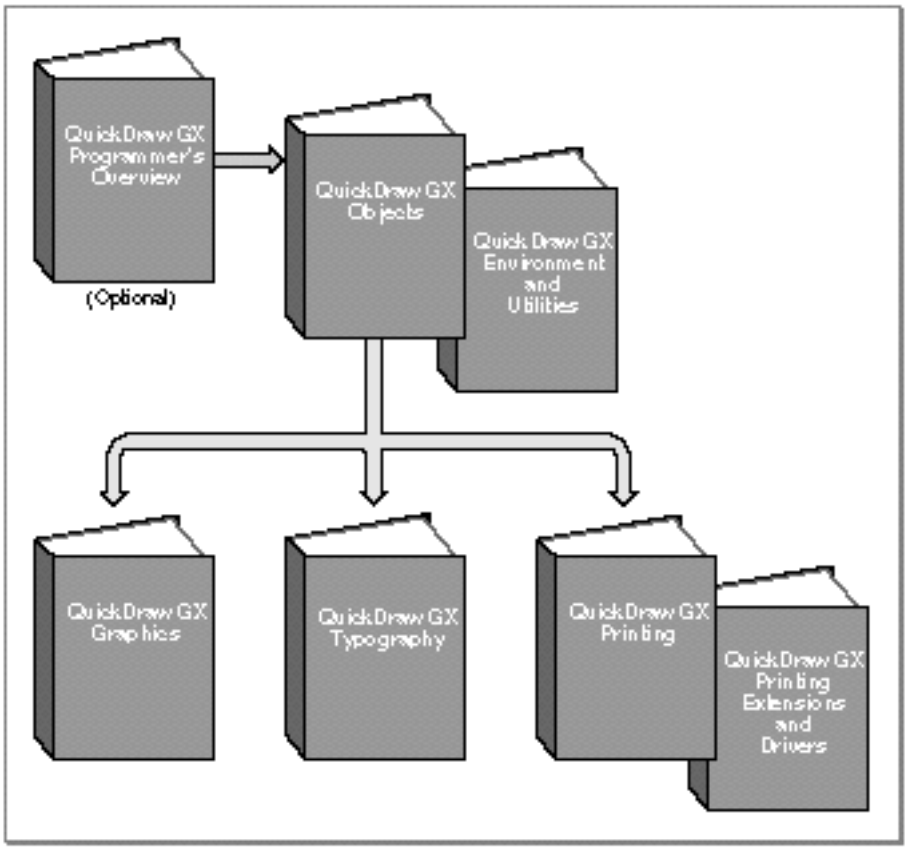
Chapter 3, "Text Shapes," describes how to create and use QuickDraw GX text shapes.

Chapter 4, "Glyph Shapes," describes how to create and use QuickDraw GX glyph shapes.

Chapter 5, "Layout Shapes," describes how to create and use QuickDraw GX layout shapes.

Chapter 6, "Typographic Styles," describes how to use the typographic properties of QuickDraw GX style objects.

Figure P-1 Roadmap to the QuickDraw GX suite of books



Chapter 7, “Font Objects,” describes the properties of QuickDraw GX font objects and how to use them.

Chapter 8, “Layout Styles,” describes how to use the layout shape-related properties of QuickDraw GX style objects.

Chapter 9, “Layout Line Control,” describes how to measure and control lines for layout shapes.

Chapter 10, “Layout Carets, Highlighting, and Hit-Testing,” describes how to use carets, how to highlight, and how to hit-test layout shapes.

Chapter Organization

Most chapters in this book follow a standard general structure. For example, the chapter "Layout Shapes" contains these major sections:

- "About Layout Shapes." This section provides an overview of the properties of layout shapes.
- "Using Layout Shapes." This section describes how you can create and manipulate layout shape objects using QuickDraw GX. It describes how to use the most common functions, gives related user interface information, provides code samples, and supplies additional information.
- "Layout Shapes Reference." This section provides a complete reference for layout shape objects by describing the constants, data types, and functions that you use with layout shapes. Each function description follows a standard format, which gives the function declaration; a description of every parameter; the function result, if any; and a list of errors, warnings, and notices. Most function descriptions give additional information about using the function and include cross-references to related information elsewhere.
- "Summary of Layout Shapes." This section shows the C interface for the constants, data types, and functions associated with layout shapes.

Conventions Used in This Book

This book uses various conventions to present certain types of information.

Special Fonts

All code listings, reserved words, and names of data structures, constants, fields, parameters, and functions are shown in Courier (`this is Courier`).

When new terms are introduced, they are in **boldface.** These terms are also defined in the glossary.

Types of Notes

There are several types of notes used in this book.

Note
A note formatted like this contains information that is interesting but possibly not essential to an understanding of the main text. The wording in the title may say something more descriptive than just “Note”; for example, “Terminology Note.” u

IMPORTANT
A note like this contains information that is especially important. (An example appears on page 7-13.) s

Numerical Formats

Hexadecimal numbers are shown in this format: 0x0008.

The numerical values of constants are shown in decimal, unless the constants are flag or mask elements that can be summed, in which case they are shown in hexadecimal.

Type Definitions for Enumerations

Enumeration declarations in this book are commonly followed by a type definition that is not strictly part of the enumeration. You can use the type to specify one of the enumerated values for a parameter or field. The type name is usually the singular of the enumeration name, as in the following example:

```
enum gxFontPlatforms {
   gxGlyphPlatform = -1,
   gxNoPlatform,
   gxUnicodePlatform,
   gxMacintoshPlatform,
   gxReservedPlatform,
   gxMicrosoftPlatform,
   gxCustomPlatform,
};
typedef long gxFontPlatform;
```

Illustrations

This book uses several conventions in its illustrations.

In illustrations that show object properties, properties that are object references are in italics.

In order to focus attention on the key part of some drawings, other parts are printed in gray, rather than in black.

Objects in diagrams, whether shown with their properties or without, are represented by distinctive icons, such as:

See, for example, Figure 2-4 in Chapter 2.

Development Environment

The QuickDraw GX functions described in this book are available using C interfaces. How you access these functions depends on the development environment you are using.

Code listings in this book are shown in ANSI C. They suggest methods of using various functions and illustrate techniques for accomplishing particular tasks. Although most code listings have been compiled and tested, Apple Computer, Inc., does not intend for you to use these code samples in your applications.

Developer Products and Support

APDA is Apple's worldwide source for over three hundred development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the quarterly *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most of our products. APDA offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 1-716-871-6555 (International)
Fax	1-716-871-6511
AppleLink	APDA
America Online	APDAorder
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

CHAPTER 1

Introduction to QuickDraw GX Typography

Contents

This chapter introduces the text-handling capabilities of QuickDraw GX and defines important typographic terms related to text. If you are developing a QuickDraw GX application that uses text, read this chapter before reading any other parts of this book.

This chapter assumes that you have read the book *Inside Macintosh: QuickDraw GX Objects* and that you know what shape objects and style objects are.

This chapter starts by outlining the different components that make up text. It then describes

- how text is measured and stored
- how you can arrange text on a display device
- how you can adjust the text in various ways by affecting the text direction, kerning, alignment, justification, and line breaks
- how your application can draw, highlight, and hit-test text

Typography and QuickDraw GX

Text has traditionally been defined as the written representation of spoken language. QuickDraw GX extends this definition by treating text as both text and graphics and by allowing you to use special typographic features to generate and manipulate fully editable, text-related shapes.

Because each line of text is a QuickDraw GX shape, you can modify it as you would any other graphic shape, yet the shape still maintains its identity and editability as a text line. With QuickDraw GX, you can use one or more of the typographic shapes described in this book for simple word-processing tasks as well as for laying out more complex, typographically sophisticated text lines.

Typographic shapes have the same fundamental structure as other shapes. The geometry of a typographic shape contains its characters, just as a rectangle shape's geometry contains the points that make up the upper-left and lower-right corners of the rectangle. There are three kinds of typographic shapes, each with specific characteristics and properties:

- A **text shape** consists of a string of one or more characters or glyphs, all to be displayed in the same font with the same typestyle.
- A **glyph shape** consists of one or more characters or glyphs, each of which can be independently located, rotated, sized, and styled.
- A **layout shape** consists of a line of text that may be in multiple languages, and which may be displayed with multiple writing directions (including vertical), with ligatures and other contextual forms, and with other sophisticated formatting and stylistic properties.

The three shapes are described as a group in the chapter "Typographic Shapes" and individually in the chapters "Text Shapes," "Glyph Shapes," and "Layout Shapes."

Characters, Glyphs, and Fonts

A writing system's alphabet, numbers, punctuation, and other writing marks consist of characters. A **character** is a symbolic representation of an element of a writing system; it is the concept of, for example, "lowercase a" or "the number 3." It is an abstract object, defined by custom in its own language.

As soon as you write a character, however, it is no longer abstract but concrete. The exact shape by which a character is represented is called a **glyph.** The "characters" that QuickDraw GX places on the screen are really glyphs.

Glyphs and characters do not necessarily have a one-to-one correspondence. For example, a single character may be represented by one or more glyphs (the character "lowercase i" could be represented by the combination of glyphs "ı" and "·"), and a single glyph can represent two or more characters (the single glyph "fi" could represent the two characters "f" and "i").

Context—where the glyph appears in a line of text—also affects which glyph represents a character. Figure 1-1 shows examples of such contextual forms in a Roman font. Different forms of a glyph are used according to whether the glyph stands alone or occurs at the beginning of a word, occurs at the end of a word, or forms part of a new glyph, as in a ligature.

A **font** is a collection of glyphs, all of similar design, that constitute one way to represent the characters of one or more language.

Figure 1-1 Contextual forms in a Roman font

Fonts usually have some element of design consistency, such as the shape of the ovals (known as the **counter**), the design of the stem, stroke thickness, or the use of **serifs**, which are the fine lines stemming from the upper and lower ends of the main strokes of a letter. Figure 1-2 shows some of the elements of glyphs that indicate they are members of the same family.

Figure 1-2 Elements that distinguish glyphs of a Roman font

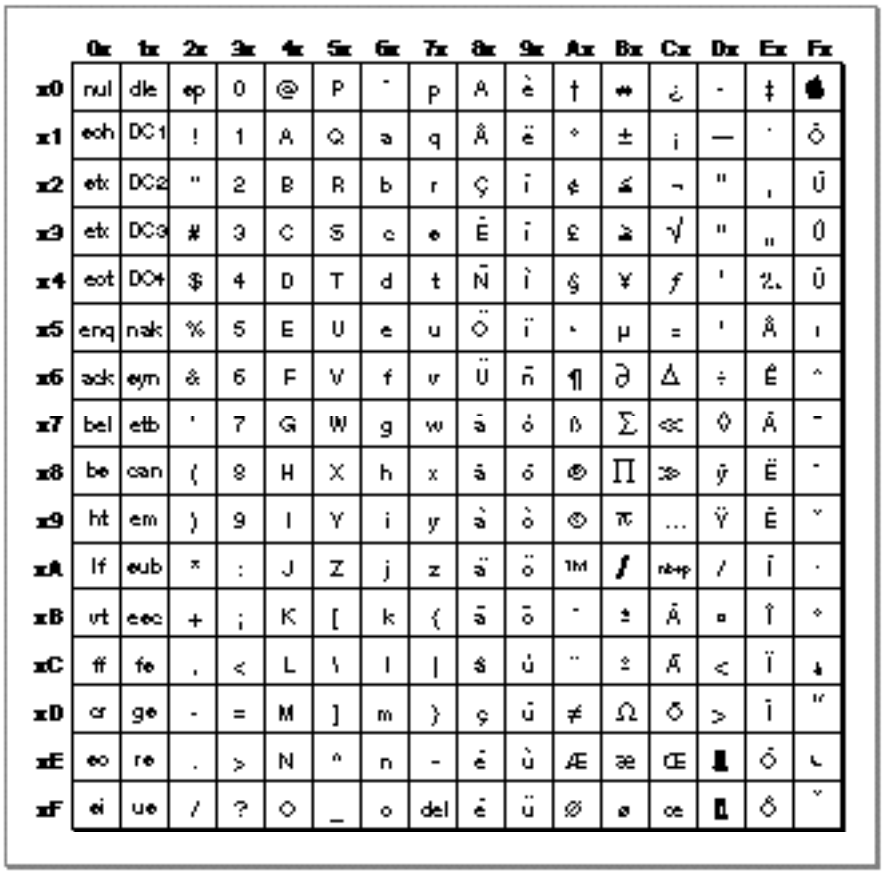
A font always has a full name—for example, Geneva Regular or Times Bold. The full name determines which family the font belongs to and what typestyle it represents. (Typestyles are discussed on page 1-10.) The font Geneva Italic, for example, shares many characteristics with Geneva Regular, but all of the glyphs slant at a certain angle. Though different, these fonts are part of the same font family. A **font family** is a group of fonts that share certain characteristics and have a common family name. Each font family has its own name, such as “New York,” “Geneva,” or “Symbol.” Several fonts may have the same family names (such as Geneva, Geneva Bold, and so on) but are stored separately—these fonts are still part of the same font family.

Note
QuickDraw GX does not use the `'FOND'` resource to determine what fonts are part of which font family. It uses information in the naming table of each font—a table that every QuickDraw GX font has. u

Encodings

For Roman fonts that have a one-to-one correspondence between glyphs and characters, an application can access the proper glyph using 1-byte **character codes.** As Figure 1-3 shows, Macintosh Roman character codes are hexadecimal numbers from $00 through $FF that represent the characters corresponding to a key or key combination. The figure shows 1-byte character codes; but 2-byte character codes are also used—for example, in Asian fonts, which may have 8,000 or more glyphs.

Figure 1-3 The Standard Roman character set



	0x	1x	2x	3x	4x	5x	6x	7x	8x	9x	Ax	Bx	Cx	Dx	Ex	Fx
x0	nul	dle	ep	0	@	P	`	p	Ä	è	†	∞	¿	-	‡	
x1	soh	DC1	!	1	A	Q	a	q	Å	ë	°	±	¡	—	·	Ò
x2	etx	DC2	"	2	B	R	b	r	Ç	í	¢	≤	¬	"	‚	Ú
x3	etx	DC3	#	3	C	S	c	s	É	ì	£	≥	√	"	„	Û
x4	eot	DC4	$	4	D	T	d	t	Ñ	î	§	¥	ƒ	'	‰	Ù
x5	enq	nak	%	5	E	U	e	u	Ö	ï	•	µ	≈	'	Â	ı
x6	ack	syn	&	6	F	V	f	v	Ü	ñ	¶	∂	Δ	÷	Ê	^
x7	bel	etb	'	7	G	W	g	w	á	ó	ß	∑	«	◊	Á	~
x8	bs	can	(	8	H	X	h	x	à	ò	®	∏	»	ÿ	Ë	¯
x9	ht	em	)	9	I	Y	i	y	â	ô	©	π	…	Ÿ	È	˘
xA	lf	sub	*	:	J	Z	j	z	ä	ö	™	ª	nbsp	/	Í	˙
xB	vt	esc	+	;	K	[	k	{	ã	õ	´	º	Ã	¤	Î	°
xC	ff	fs	,	<	L	\	l	|	å	ú	¨	≠	Õ	<	Ï	¸
xD	cr	gs	-	=	M	]	m	}	ç	ù	Æ	Ω	Œ	>	Ì	˝
xE	so	rs	.	>	N	^	n	~	é	û	æ	æ	œ	fi	Ó	˛
xF	si	us	/	?	O	_	o	del	è	ü	Ø	ø	œ	fl	Ô	ˇ

Fonts also associate each glyph with a 2-byte code called its **glyph code.** Different fonts may have different glyph codes for the same glyphs, and a single font may have several glyph codes associated with a particular character because several glyphs may represent that character. Because the font and general textual context determine which glyph and which glyph codes represent characters, QuickDraw GX transparently handles the details of mapping character codes to the correct glyph codes. Your application does not have to handle the details of obtaining glyphs from a font.

Note

Your application will usually deal with character codes, since those correspond most closely with what the user types. However, QuickDraw GX permits you to deal with glyph codes, if that is more appropriate to your application's functionality and needs. u

Different languages may have different requirements in terms of which glyphs they want from a font. A font contains some number of **character encodings.** Each encoding is an internal conversion table for interpreting a specific character set—that is, a way to map a character code to glyph code for that font.

The reason a font can have multiple encodings is that the requirements for each writing system that the font supports may be different. A **writing system** is a method of depicting words visually. It consists of a character set and a set of rules for displaying, ordering, and formatting the glyphs associated with those characters. Writing systems can differ in line direction, the direction in which their glyphs are read; the size of the character set used to represent the script; and contextual variation (that is, whether a glyph changes according to its position relative to other glyphs). Writing systems have specific requirements for text display, text editing, character set, and fonts. A writing system—for instance, the Roman system—can serve one or several languages, such as French, Italian, and Spanish.

Text Storage

QuickDraw GX stores text in a typographic shape's geometry as a sequence of character codes or glyph codes. The **storage order** is the order in which text is stored. A shape may contain 1-byte or 2-byte codes, or a mixture of both. Using information in the style object, QuickDraw GX determines whether a character code is 1 or 2 bytes. The text stored in a shape is the **source text;** the text displayed is the **display text,** as shown in Figure 1-4.

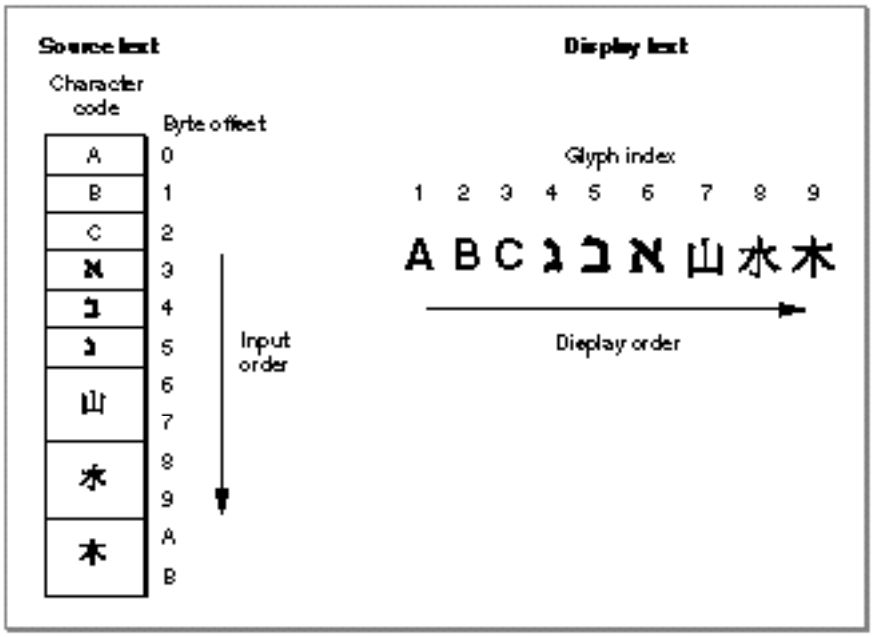
Display order is the left-to-right (or top-to-bottom) order in which glyphs are drawn. For text shapes, and by default for glyph shapes, storage order and display order are the same. For layout shapes, however, QuickDraw GX expects you to store your text in **input order,** which is the "logical" order, or the order in which the characters, *not* glyphs, would be read or pronounced in the language of the text. Because text of different languages may be read from left to right, right to left, or top to bottom, the input order is not necessarily the same as the display order of the text when it is drawn. Your application needs to differentiate between the order in which the character codes are stored in the shape and the order in which the corresponding glyphs are displayed.

Figure 1-4 shows Hebrew glyphs that are stored one way and displayed another way in a layout shape.

Note

In Figure 1-4 and throughout this book, text in computer memory is drawn as a vertical table of codes, representing sequential (downward) storage of text characters in a buffer. Some diagrams also include byte offsets in the buffer, and even miniature representations of the characters themselves in a given language. u

Figure 1-4 Input order and display order



As shown in Figure 1-4, the character codes that make up the text are numbered using zero-based **offsets.** Therefore, the first character code in the figure has an offset of 0.

QuickDraw GX uses a different numbering scheme to index the glyphs that are actually displayed. The **glyph index,** which gives the glyph's position in the display order, always starts at 1. Therefore, the offset of the character code and the index of the corresponding glyph may be different. Also, each glyph has a single index, even if its character code is 2 bytes long (as in the case of Chinese characters). In Figure 1-4, the character code offset of the uppercase "A" is 0, but the glyph's index is 1. Likewise, the last Chinese glyph begins at character code offset A but has an index of 9.

For layout shapes, QuickDraw GX provides functions to map back and forth from byte offsets to glyph indexes. For text and glyph shapes, such mapping is not needed because with those shapes a character offset corresponds one-to-one with a glyph index.

Text Measurements

Most users use point size to specify the size of the glyphs in a document. **Point size** indicates the size of a font's glyphs as measured from the baseline of one line of text to the baseline of the next line of single-spaced text; in the United States, point size is measured in **typographic points,** and there are 72.27 points per inch. However, QuickDraw GX and the PostScript™ language both define 1 point to be exactly $^{1}/_{72}$ of an inch. QuickDraw GX permits fractional point sizes.

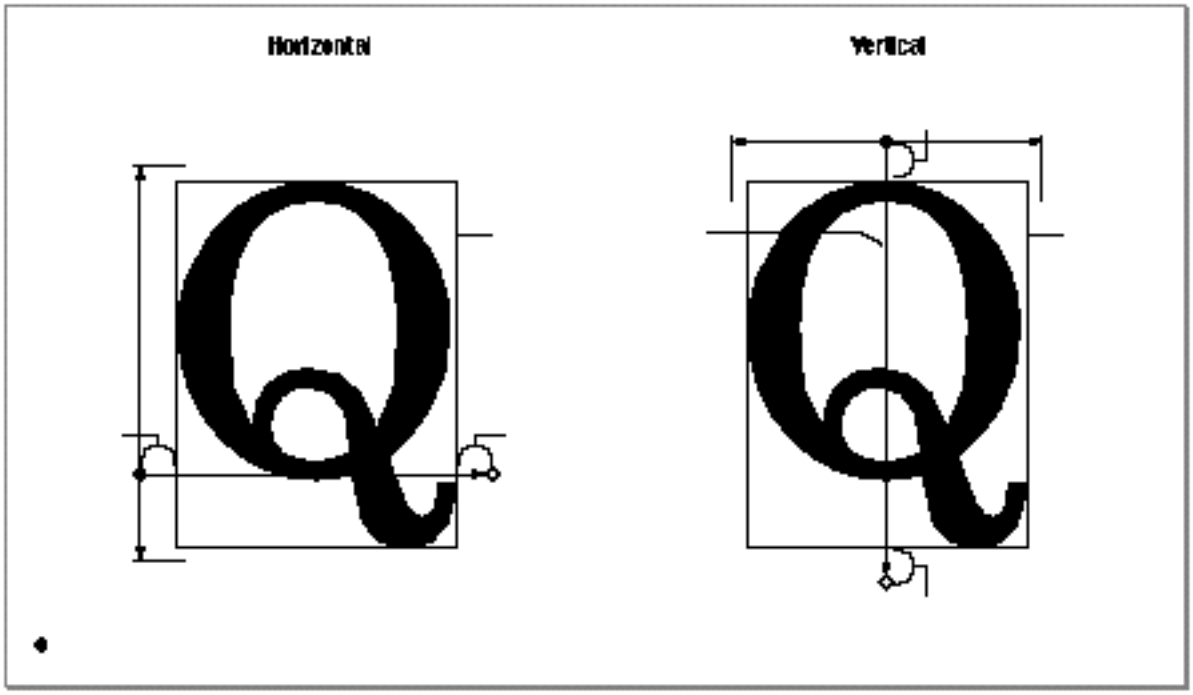
Although point size is a useful measure of the size of text, you may wish to use more exact measurements for greater control over placement of the glyphs on the display device.

Font designers use a special vocabulary for the measurements of different parts of a glyph. Figure 1-5 shows the terms describing the most frequently used measurements.

The **bounding box** of a glyph is the smallest rectangle that entirely encloses the drawn parts of the glyph. The **glyph origin** is the point that QuickDraw GX uses to position the glyph when drawing. In Figure 1-5, notice that there is some space between the glyph origin and the edge of the bounding box: this space is the glyph's **left-side bearing.** The left-side bearing value can be negative, which decreases the space between adjacent characters. The **right-side bearing** is space on the right side of the glyph; this value may or may not be equal to the value of the left-side bearing. The **advance width** is the full horizontal width of the glyph as measured from its origin to the origin of the next glyph on the line, including the side bearings on both sides.

Most glyphs in Roman fonts appear to sit astride the **baseline**, an imaginary horizontal line. The **ascent** is a distance above the baseline, chosen by the font's designer and the same for all glyphs in a font, that often corresponds approximately to the tops of the uppercase letters in a Roman font. Uppercase letters are chosen because, among the regularly used glyphs in a font, they are generally the tallest. The **descent** is a distance below the baseline that usually corresponds to the bottoms of the descenders (the "tails" on glyphs such as "p" or "g"). The descent line is the same distance from the baseline for all glyphs in the font, whether or not they have descenders. The sum of ascent plus descent marks the **line height** of a font.

Figure 1-5 Terms for glyph measurements



For vertical text, font designers may use additional measurements. The **top-side bearing** is the space between the top of the glyph and the top edge of the bounding box. The **bottom-side bearing** is the distance from the bottom of the bounding box to the origin of the next glyph. For vertical text, the **advance height** is the sum of the top-side bearing, the bounding-box height, and the bottom-side bearing.

These metrics are useful if, for example, you want to display a horizontal font vertically. Likewise, vertical fonts such as Kanji may also have horizontal metrics.

Typestyles

Glyphs can be differentiated not only by font but by typestyle. A **typestyle** is a specific variation in the appearance of a glyph that can be applied consistently to all the glyphs in a font family. Some of the typical typestyles available on the Macintosh computer include plain, bold, italic, underline, outline, shadow, condensed, and extended. Other styles that may be available are Demibold, Extra Condensed, or Antique.

Font Variations and Instances

A **font variation** is a setting along a particular variation axis. Font variations allow your application to produce a range of typestyles algorithmically.

Each **variation axis** has a name identifying the typestyle that the axis represents (such as weight or width), a tag to represent that name (such as `'wght'`), a set of maximum and minimum values for the axis, and the default value of the axis. The weight axis, for example, governs the possible values for the weight of the font; the minimum value may produce the lightest appearance of that font, the maximum value the boldest. The default value is the position along the variation axis value at which that font falls normally.

Because the axis is created by the font designer, font variations can be optimized for their particular font. Figure 1-6 shows a range of possible weights for a glyph, from the minimum weight to the maximum weight.

A **font instance** is a set of named variations identified by the font designer that matches specific values along the available variation axes and associates those values with a name. For example, suppose a font has the variation axis `'wght'` with a minimum value of 0.0, a default of 0.5, and a maximum value of 1.0. The corresponding font instance might have the name “Demibold” with a value along that variation axis of 0.8.

Figure 1-6 Font variations along a variation axis

In Figure 1-6, the variation axis value of the glyph at the far right could represent the named instance “Extra Bold,” whereas the glyph at the far left could represent the named instance “Light.” The other values represented in the figure could likewise have instance names.

Font variations and font instances give your application the ability to provide whatever typestyles the font designer has decided to include with the font.

Text Faces

If the desired typestyle is not available as a separate font and cannot be produced as a font variation, your application can generate a **text face,** which is an algorithmic way of producing typestyles. You can use text faces to produce bold, italic, condensed, and other typical Macintosh typestyles. You can also use them to create unusual typestyles not supported anywhere else; for example, you can create a “sunburst” text face, as shown in Figure 1-7.

Figure 1-7 An example of an unusual text face

Laying Out Text

When you arrange text on a display device, QuickDraw GX offers a simple approach: you can put the information into one of the typographic shapes and then display it. Depending on the shape type, QuickDraw GX automatically handles many aspects of the text display. However, QuickDraw GX also offers you the ability to adjust the text in various ways, to gain greater control over the presentation and arrangement of the text. For example, you can affect

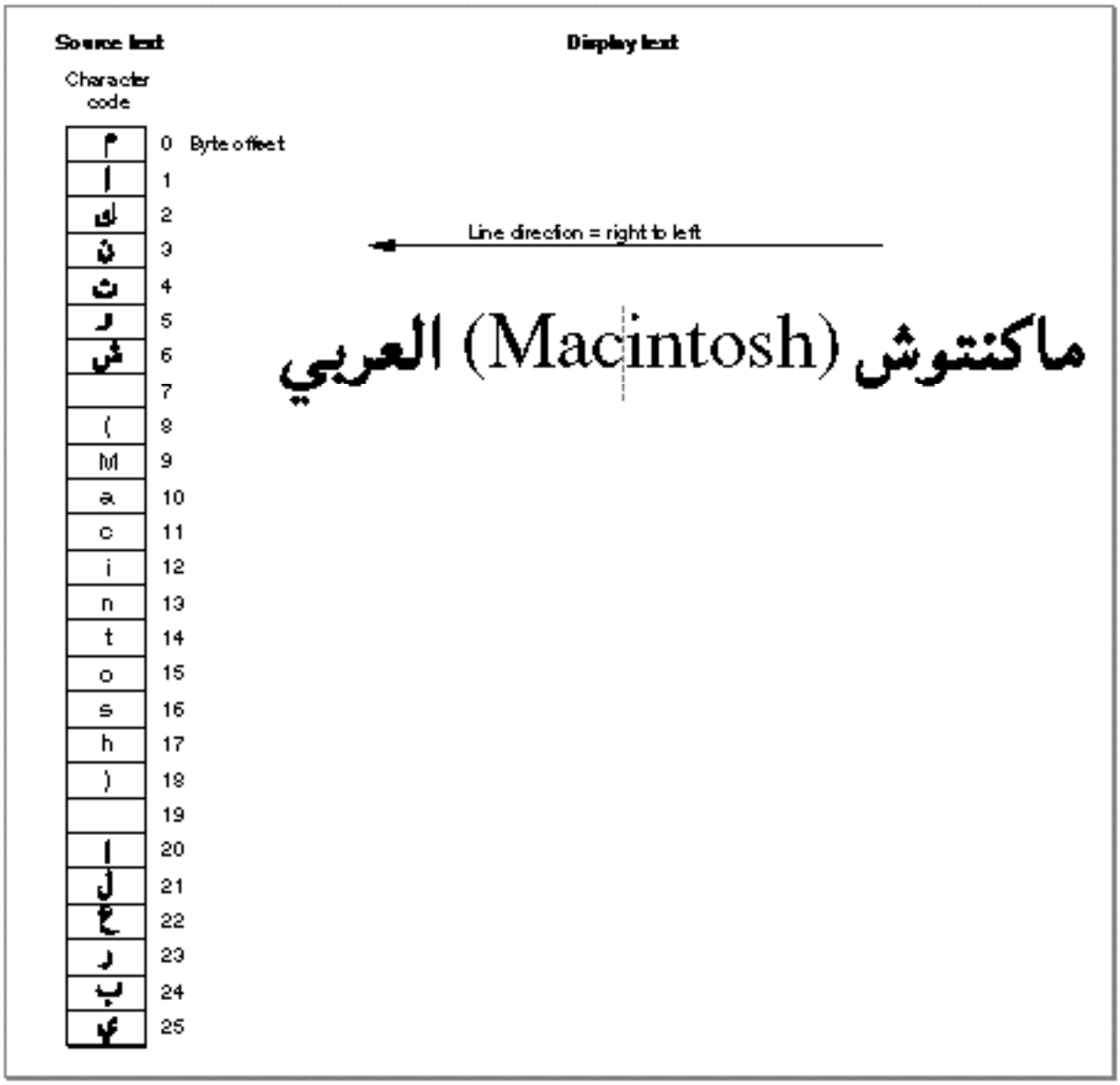
- text direction and baselines
- text runs, style runs, and direction runs
- contextual forms and ligatures
- alignment and justification
- kerning and tracking
- line breaks

This section describes some of the general concepts governing how you can affect the text presented by your application. Actual implementation of these concepts is described in various chapters in this book.

Text Direction and Baselines

Text direction consists of text orientation (horizontal or vertical) and the direction in which the text is read. Text in your application can be oriented in three common directions: horizontally, left to right; horizontally, right to left; and vertically, top to bottom. QuickDraw GX allows your application, for example, to draw lines of text in multiple directions, as shown in Figure 1-8.

Figure 1-8 Multi-direction text



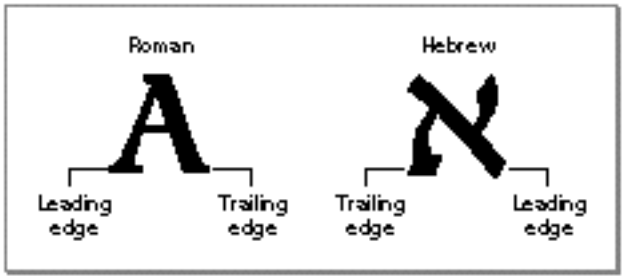
The text in Figure 1-8 contains a mixture of English and Arabic, with the primary direction being right to left. The figure also shows the character codes and byte offsets of the source text.

Leading Edges and Trailing Edges

Because text has a direction, the concept of which glyph comes “first” in a line of text cannot always be limited to the visual terms “left” and “right.” The **leading edge** is defined as the edge of a glyph you first encounter—such as the left foot of a Roman glyph—when you first read the text that includes that glyph. The **trailing edge** is the edge of a glyph encountered last.

Figure 1-9 shows how the concepts of leading edge and trailing edge change depending on the characteristics of the glyph. In the first example—a Roman glyph—the leading edge is on the left, because the reader encounters that side first. In the second example, the leading edge of the Hebrew glyph is on the right for the same reason. For more information, see the chapter “Layout Carets, Highlighting, and Hit-testing” in this book.

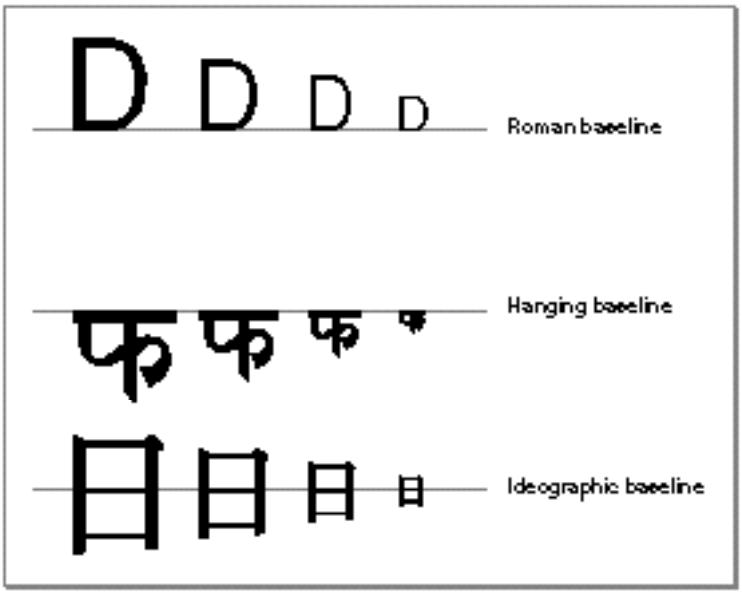
Figure 1-9 Leading edges and trailing edges



Baselines

A **baseline** is an imaginary line that coincides with some point in a font—for example, the bottom, middle, or top of each glyph. The baseline of a glyph defines the position of the glyph with respect to other glyphs at different point sizes when all the glyphs are aligned. It represents a stable platform from which glyphs of different sizes and different writing systems grow proportionally, as shown in Figure 1-10.

Figure 1-10 Baselines for different sizes of a glyph and for different writing systems



Note that, depending on the writing system, the baseline may be above, below, or through the center of each glyph.

QuickDraw GX provides your application with capabilities for using multiple baselines. For more information, see the chapter “Layout Line Control” in this book.

Various baselines can also be used to create special effects, such as drop capitals. (A **drop capital** is an initial capital letter that is much larger than surrounding glyphs and embedded in them.) Figure 1-11 is an example of drop capitals formed solely on the basis of the baselines in the font. The default baseline for this text is the Roman baseline for 18-point type. The hanging baseline of the drop capitals aligns with the hanging baseline of the regular text, creating the effect shown.

Figure 1-11 Drop capitals

Text Runs, Style Runs, and Direction Runs

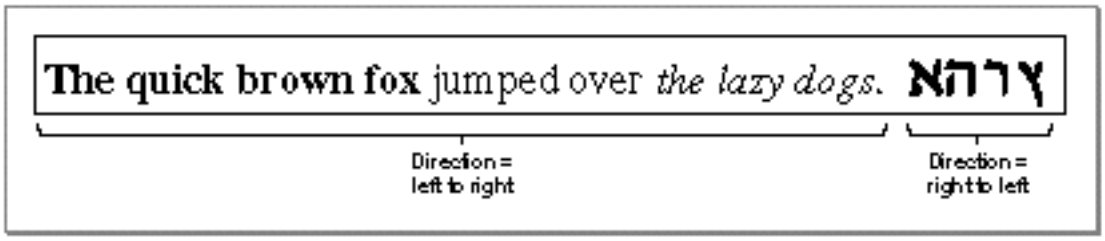
In any segment of contiguous text, certain parts stand out as belonging together, because the glyphs share a certain font, typestyle, or direction. For the purposes of referring to individual segments of text, you can think of sequences of glyphs that are contiguous in memory and share a set of common attributes as **runs.**

The complete text in a shape is one **text run.** (A shape has only one text run internally, although for layout shapes your application can provide multiple text runs.) A sequence of glyphs continuous in memory that share the same style object is a **style run.** As Figure 1-12 shows, a text run can be subdivided into several style runs.

Figure 1-12 Three style runs in a line of text

A sequence of contiguous glyphs that share the same text direction is a **direction run.** As with text runs and style runs, the number of direction runs does not necessarily correlate to the number of style runs available, as shown in Figure 1-13.

Figure 1-13 Two direction runs in a line of text



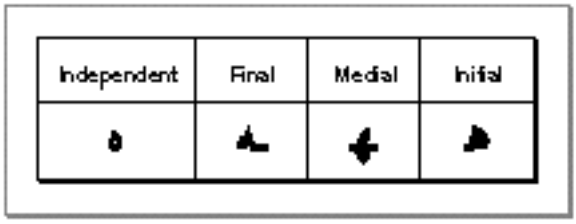
For more information on using text runs, style runs, and direction runs, see the chapters “Layout Shapes” and “Glyph Shapes” in this book.

Contextual Forms and Ligatures

A glyph's position next to other glyphs or its position in a word or a line of text may determine its appearance. For some writing systems, such as Roman, alternate glyphs are used for aesthetic reasons; in other writing systems, use of alternate forms is required.

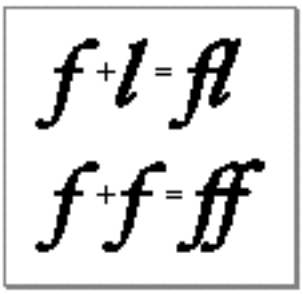
A **contextual form** is an alternate form of a glyph that is chosen depending on the glyph's placement in a certain context, such as a certain word or line. Some contextual forms of initial and final forms of glyphs in a Roman font are shown in Figure 1-1 on page 1-4. Other writing systems, such as Arabic, require different contextual forms of glyphs according to where they appear. Figure 1-14 shows the forms of the Arabic letter "ha" that appear alone and at the beginning, middle, or end of a word. The same character code is used for each case; QuickDraw GX finds the appropriate glyph code.

Figure 1-14 Contextual forms of the Arabic letter "ha"



Ligatures are two or more glyphs combined to form a single new glyph (whereas contextual forms are variations on the shape of one glyph). In the Roman writing system, ligatures are generally an optional aesthetic refinement; in other writing systems, special ligatures are required when certain glyphs appear next to one another. Some examples of ligatures used in a Roman font are shown in Figure 1-15.

Figure 1-15 Examples of Roman ligatures



In general, the font contains all of the information needed to determine when your application should use the appropriate contextual forms and ligatures. If your application allows alternate forms of glyphs to be used, QuickDraw GX does the substitution for you.

Only the layout shape substitutes ligatures and contextual forms for the character codes stored in the shape when the shape is displayed, if you request that behavior. The text and glyph shapes do not perform contextual substitutions.

Alignment and Justification

Once you have the text, you can arrange it in the text area. The **text area** is the space on the display device in which the text should fit. The left, right, top, and bottom sides of that area are the **margins.**

How you arrange the text depends on the effect you want to achieve. There are two primary methods of arranging text: alignment and justification.

Alignment is the process of placing text in relation to one or both margins. You can set the alignment in the style object for glyph and text shapes, but not for layout shapes, which use a different mechanism to align lines of text.

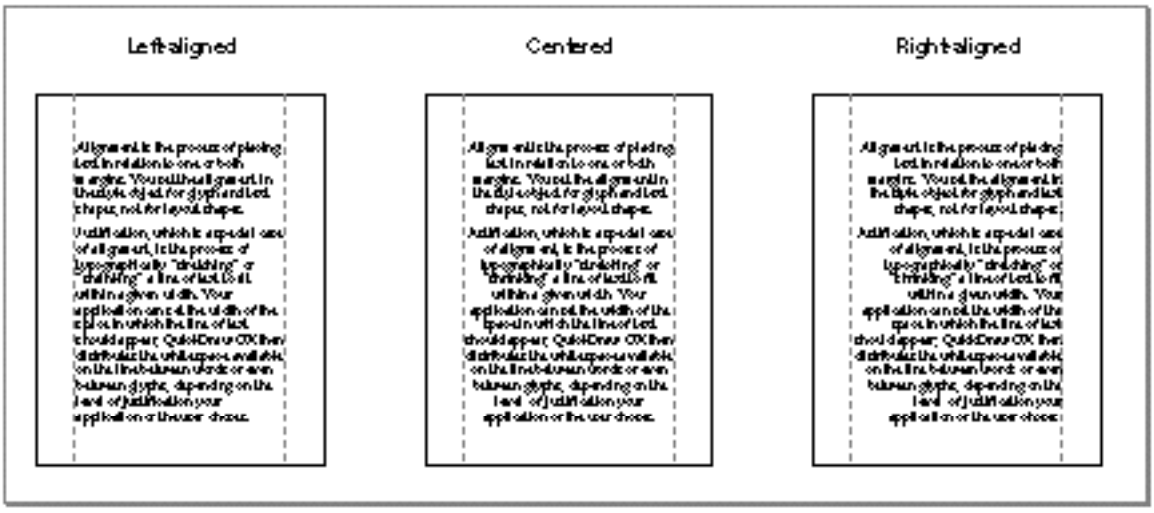
Figure 1-16 shows left, right, and center alignment of text.

Justification is the process of typographically "stretching" or "shrinking" a line of text to fit within a given width. Your application can set the width of the space in which the line of text should appear; QuickDraw GX then distributes the white space available on the line between words or even between glyphs, depending on the level of justification chosen.

For the layout shape, there are other means of aligning or justifying a line—for example, stretching a glyph or decomposing a ligature. QuickDraw GX can also handle complex justification such as that used in Arabic writing systems.

For more information about justification, see the chapter "Layout Line Control" in this book.

Figure 1-16 Different kinds of alignment



Kerning and Tracking

Kerning is an adjustment to the normal spacing between two or more specific glyphs. A **kerning pair** consists of two adjacent glyphs such that the position of the second glyph is changed with respect to the first. The font designer determines which glyphs participate in kerning and in what context. Any adjustments to glyph positions are specified relative to the point size of the glyphs. Kerning usually improves the apparent letter-spacing between glyphs that "fit together" naturally.

Figure 1-17 shows how glyphs are positioned differently with and without kerning. Note that the phrase is shorter when kerning is applied than when not.

Figure 1-17 Glyphs with and without kerning

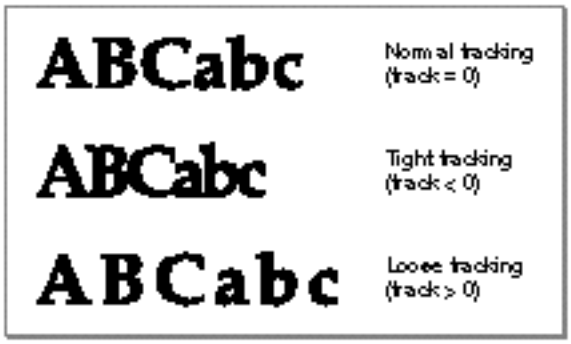
Cross-stream kerning allows the automatic movement of glyphs perpendicular to the line orientation of the text. (For example, when QuickDraw GX applies cross-stream kerning to horizontal text, the automatic movement is vertical; this feature is required for writing systems such as Taliq, which is used in the Urdu language.)

When your application lays out text, it has the option of using the interglyph spacing specified by the font designer or altering the spacing slightly in order to achieve a tighter fit between letters and improve the look of a line of text.

Your application can also use tracking. In **tracking,** space is adjusted between all glyphs in the run. You can increase or decrease interglyph spacing by using a **track setting,** which is a value that specifies the relative tightness or looseness of interglyph spacing. Positive track settings result in an increase in the looseness of all glyphs in the run. Negative track settings result in a increase in the tightness of all glyphs. Normal tracking, tight tracking, and loose tracking are shown in Figure 1-18.

For more information about kerning and tracking, see the chapter "Layout Styles" in this book.

Figure 1-18 Normal, tight, and loose tracking by the selection of track setting



Special Font Features

Some special features are available in certain fonts. You can increase the control a user has over the presentation of text in a document if you provide access to these features when they are available in a font; the font provides the functionality for using these features. Table 1-1 shows some of the currently defined features.

Table 1-1 Some special font features for layout shapes

Feature	Description
Ligatures	Permits selection from different ranges of ligatures.
Cursive connections	Controls the level of cursive connection in the font. This feature is used in fonts, such as cursive Roman fonts or Arabic fonts, in which glyphs are connected to each other.
Vertical substitution	Specifies that glyphs need to change their appearance in vertical runs of text.
Smart swashes	Controls contextual swash substitution, such as substituting a final glyph when a particular glyph appears as the end of a word.
Vertical position	Controls superscripts, subscripts, and ordinal forms.
Fractions	Governs selection and generation of fractions.
Overlapping glyphs	Prevents the collision of long tails on glyphs with the descenders of other glyphs.
Typographic extras	Allows fine typographic effects, such as the automatic conversion of two adjacent hyphens to an em dash.
Ornament sets	Governs nonletter ornament sets of glyphs.
Style options	Allows the font designer to group together collections of noncontextual substitutions into named sets.
Character shape	Specifies the use, with Chinese fonts, of the traditional or simplified character forms.

Some of these features, such as typographic extras, are fancy elements that provide the user with alternate forms of glyphs or other ornamental designs. Other features are contextual and absolutely necessary for using that font correctly—for example, cursive connectors for Arabic. (The fonts that require these features include them.)

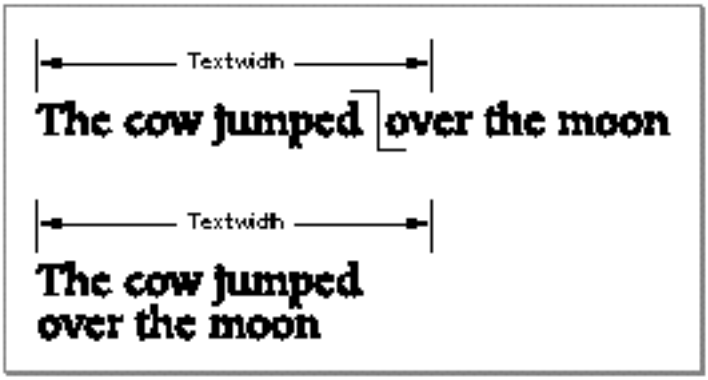
Only the layout shape allows you to take advantage of special typographic features in a font. For more information, see the chapters “Layout Shapes” and “Layout Styles” in this book.

Line Breaking

A typographic shape is designed to represent one line of text. Your application determines how wide the text area for a line should be. At times, a user enters a line of text that does not fit neatly in the given text area and overlaps one of the margins. When this happens, you break the line of text and wrap the text onto the next line.

Figure 1-19 shows a line break made on the basis of a simple algorithm: the application backs up in the source text of the shape to the trailing edge of the last white space and then carries over all of the text following that white space to the next line.

Figure 1-19 Determining where to break a line



Your application can devise more complex algorithms, such as breaking a word at an appropriate hyphenation point, if possible.

QuickDraw GX leaves the final decision about where to break the line up to your application. However, when you use the layout shape, QuickDraw GX provides you with a set of functions designed to help you determine the best place to break a line. For more information, see the chapter “Layout Line Control” in this book.

Drawing, Highlighting, and Hit-Testing Text

After you have laid out the text exactly as you want, you can draw it. If you want the user to interact with the text, you also need to support carets, highlighting, and hit-testing.

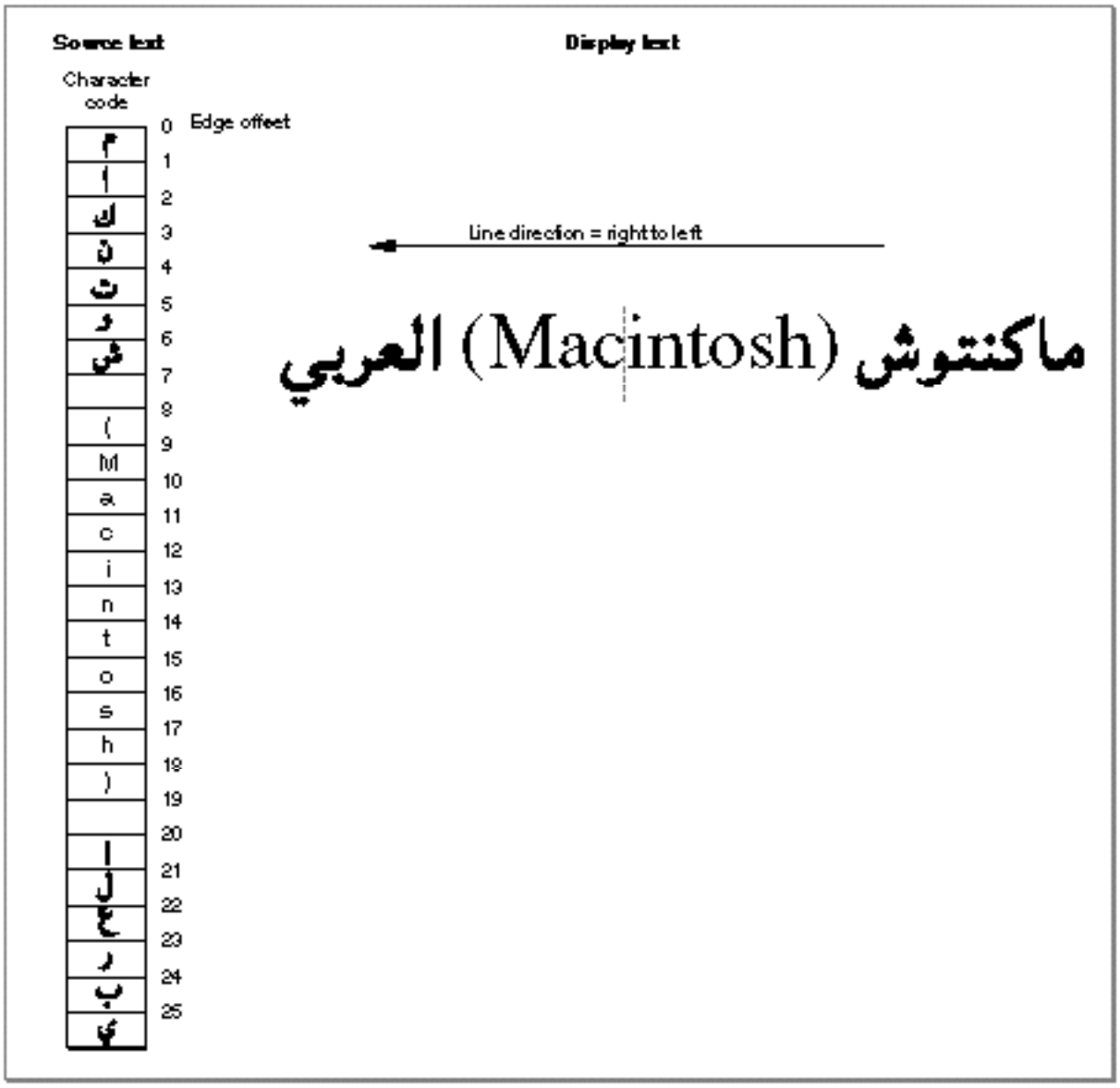
Drawing text using QuickDraw GX is simple. You draw the typographic shapes exactly as you might any other type of shape. There are no special requirements for drawing text; a single call, `GXDrawShape`, is all you need.

Carets

A **caret** is a single line that appears at the position in the text where the user can insert the next character. Carets indicate where the user can next add text. Figure 1-20 shows how a caret appears between glyphs of a word; if the user were to add new text at this point, the corresponding glyphs would appear between the “c” and the “i” of the word.

Remember that the glyphs in a line of text are numbered using a 1-based indexing scheme. (See “Text Storage” on page 1-7.) However, when you place a caret between glyphs, you need to be able to relate it to the insertion point: a point between byte offsets in the source text. The **edge offset** is a byte offset between character codes in the source text that corresponds to the caret location between glyphs. In Figure 1-20, the edge offset is 12; the glyphs on either side have indexes of 12 and 13.

Figure 1-20 Caret position



Highlighting

Highlighting is the display of text in inverse video or with a colored background. Figure 1-21 shows highlighting in some Arabic text.

Figure 1-21 Highlighting

The characters in memory corresponding to the glyphs that are highlighted make up the **selection range,** which indicates where the next editing operation is to occur. The characters in a selection range are always contiguous in memory, but their corresponding glyphs are not necessarily so onscreen.

If the selection range crosses a **direction boundary**—a point at which the direction of the displayed text changes—you may get **discontiguous highlighting,** as shown in Figure 1-22.

Figure 1-22 Discontiguous highlighting

QuickDraw GX provides functions that create carets, contiguous highlighting, and discontiguous highlighting for a layout shape. If you use one of the other types of shapes, you must create your own carets and highlighting. For more information, see the chapter “Layout Carets, Highlighting, and Hit-testing” in this book.

Hit-Testing

Caret handling and highlighting require you to convert from the edge offset to the screen position. You must also be able to convert from screen position to edge offset. For example, if the user clicks the mouse button while the cursor is in displayed text, you need to determine the offset in your text buffer equivalent to that mouse-down event. You can then use that information to set the insertion point or selection range.

QuickDraw GX does most of the work of **hit-testing,** or converting a location within the line into the corresponding edge offset that corresponds to that location in the original string. Figure 1-23 shows a simple example of hit-testing. The user clicks on a glyph near its boundary with another glyph. QuickDraw GX translates the location of this mouse click into a point in the view port and to an offset in the source text that marks the equivalent boundary between character codes. It takes into account the glyph edge (the leading edge or trailing edge) nearest to which the click occurred and the most appropriate place to display the caret.

Figure 1-23 Hit-testing

QuickDraw GX provides functions that perform hit-testing on any type of shape, and these functions can be used on any typographic shape. In general, if you are using layout shapes, you should use the layout hit-testing function, which provides more information about complex situations, such as when the shape contains multi-directional text (for example, English and Hebrew).

For more information, see the chapter “Layout Carets, Highlighting, and Hit-Testing” in this book.

CHAPTER 2

Typographic Shapes

Contents

This chapter gives an overview of the typographic shape objects and the functions you can use to manipulate them. Read this chapter if you create or use any kind of QuickDraw GX typographic shapes.

Before reading this chapter, you should be familiar with the information in the chapter "Introduction to QuickDraw GX Typography" in this book. You also need to be familiar with the concept of objects as described in the book *Inside Macintosh: QuickDraw GX Objects*.

This chapter introduces the QuickDraw GX typographic shape object, explains how it relates to the concept of a QuickDraw GX shape object, and describes typographic shapes and their properties. It then shows how to use QuickDraw GX functions to

- hit-test typographic shapes
- measure typographic shapes
- convert typographic shapes to other types of shapes
- flatten typographic shapes
- use functions described in *Inside Macintosh: QuickDraw GX Objects* and *Inside Macintosh: QuickDraw GX Graphics* on typographic shapes

About Typographic Shapes

Typographic shapes are the QuickDraw GX shapes that display text: text shapes, glyph shapes, and layout shapes. In general, they act like other QuickDraw GX shapes, such as paths, polygons, or pictures. The geometry of the shape holds the unique characteristics of the typographic shape.

Each of the three shape types has different capabilities that make them suited for different uses. The needs of your application determine which type of shape you should use.

Types of Typographic Shapes

Text shapes contain the text you want to draw, one style to draw the text in, and a position at which to start drawing the text. They provide the simplest way for your application to draw text. Figure 2-1 shows how a text shape looks when drawn.

Figure 2-1 Text shape

Note
In this book, the term *text* always refers to displayed writing. When referring to the text shape, this book always uses the term *text shape.* u

Glyph shapes allow you to draw text in several styles with independently positioned glyphs. You can give each glyph in the shape its own absolute position and draw each glyph at a different angle. In Figure 2-2, the glyph shape has two styles: most of the sentence is in regular Hoefler Text Italic, and the word “offices” is in bold. In addition, the shape is angled so that the first half of the sentence is at a 90-degree angle to the second half.

Figure 2-2 Glyph shape

Note
Do not confuse the term *glyph* with the shape type *glyph shape*, which is one of the typographic shapes. A glyph is a single unit of a font, and a glyph shape is a QuickDraw GX shape type that, when drawn, contains one or more glyphs. When referring to the shape, this book always uses the phrase *glyph shape.* u

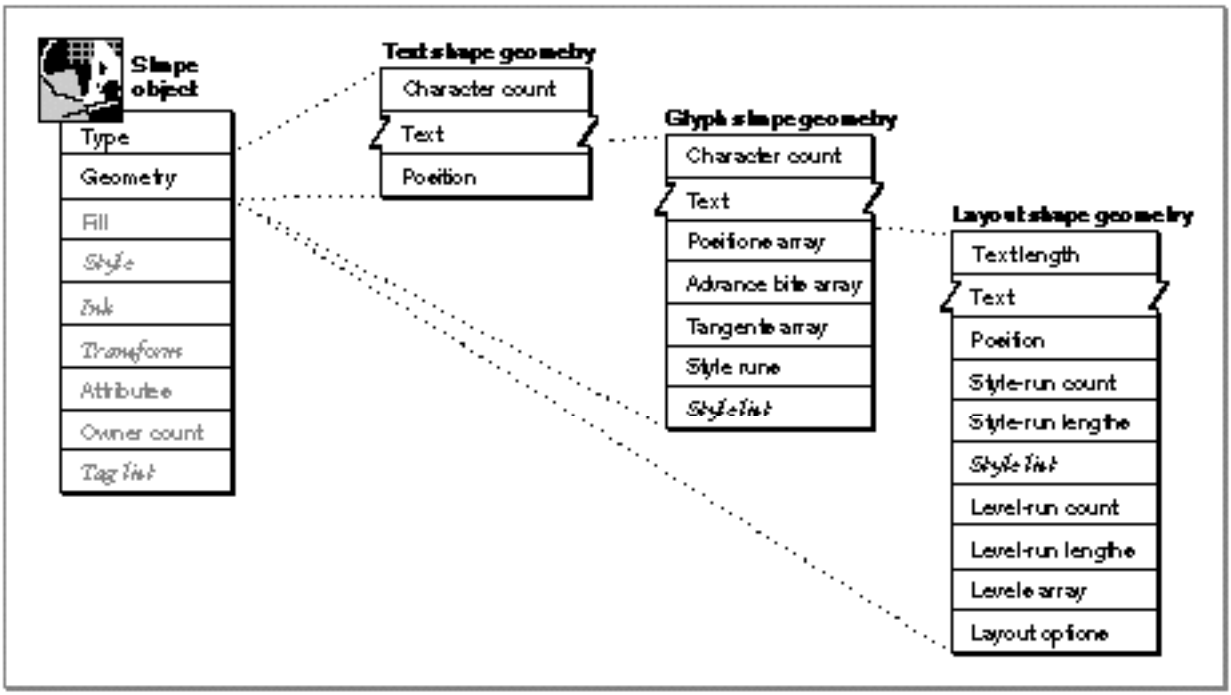
Layout shapes display text in a typographically sophisticated manner. These shapes use more information from the font than other shapes do in order to display text correctly. Correct display might require kerning the text, drawing the glyphs in a left-to-right or right-to-left direction, or choosing different versions of glyphs depending on the glyph’s position in the text. For example, a layout shape containing Arabic text automatically draws the text from right to left and joins the glyphs together as required by the Arabic script. Figure 2-3 shows a layout shape with the same text as Figure 2-2. However, the layout shape uses information in the font to pick the best glyphs for display. Note that the uppercase “T” that begins the sentence is a variant form of the glyph that appears in Figure 2-2, because the “T” in Figure 2-3 is in a ligature. Also notice several other ligatures.

Figure 2-3 Layout shape

Typographic Shape Structure

Typographic shapes are organized like other shapes: they have the same shape object structure and have associated style, ink, and transform objects. Figure 2-4 shows the organization of a typographic shape, including the contents of its geometry.

Figure 2-4 Geometry of a typographic shape



The geometry of a typographic shape contains the following elements in common:

- **Character count.** For text and glyph shapes only, the number of characters in the text array (byte count for layout shapes).
- **Text.** For glyph and layout shapes only, an array of character codes, glyph codes, or both character and glyph codes. For text shapes only, an array of character codes or glyph codes.
- **Position.** The starting position of the typographic shape in geometry space.

Note

The shape type property specifies whether the shape is a text, glyph, or layout shape. u

Typographic Shape Attributes

Typographic shapes, like other shapes, use shape attributes. The two shape attributes that apply most frequently to typographic shapes are the `gxMapTransformShape` and `gxIgnorePlatformShape` shape attributes.

- The `gxMapTransformShape` shape attribute applies transformation operations—for example, rotating, scaling, or skewing—to the shape's transform object rather than changing the information in the shape object's geometry. This attribute is set by default for layout shapes.
- The `gxIgnorePlatformShape` shape attribute indicates that QuickDraw GX should treat the codes in the geometry of this shape as glyph codes rather than as character codes. This attribute overrides information in the style object or style run arrays about the platform, script, and language used for individual style runs (explained in the chapter "Typographic Styles" in this book). The only time your application needs to set this attribute is if it needs to specify glyph codes directly—for example, if it is a font editor.

Default Characteristics of a Typographic Shape

The default settings for all typographic shapes are:

- Geometry: depends on the particular typographic shape. See the chapters "Text Shapes," "Glyph Shapes," and "Layout Shapes" in this book for specifics about the geometry of each of these shapes.
- Fill: winding fill. The valid fill types for typographic shapes are: winding, even-odd, and no fill.
- Style: the same default style object as other QuickDraw GX shapes. See the chapter "Typographic Styles" for more information about style properties, such as text attributes, that are specific to typographic shapes.
- Ink: the same default ink object as other shapes. Thus, the entire shape can only have one color, and all glyphs in the shape must be of the same color.
- Transform: the same default transform object as other shapes. The default layout shape has the `gxMapTransformShape` attribute set, whereas the other two typographic shapes, text and glyph shapes, do not. If the `gxMapTransformShape` attribute is set,

any transformations that you apply to the shape are in fact applied to the mapping of the transform object. If the attribute is not set, the transformations are applied to the shape's geometry.

- Shape attributes: none. The exception is the layout shape, which has the `gxMapTransformShape` attribute set.
- Owner count: 1.
- Tags: none.

Typographic Shapes and the Style Object

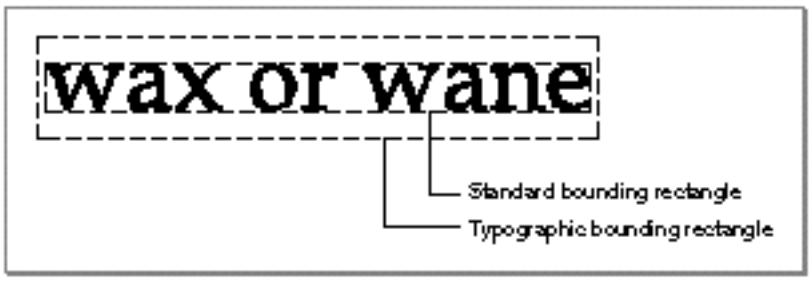
The style object associated with a typographic shape contains the font, the text size in points, and other information that determines the characteristics of a shape when it is displayed. (For more information about the parts of a style object that apply to typographic shapes, see the chapter "Typographic Styles" in this book.)

The Standard and Typographic Bounding Rectangles

Every QuickDraw GX shape has a standard bounding rectangle, which is the smallest rectangle that completely encloses the filled or framed parts of the shape. However, because of the height differences between glyphs—for example, between a small glyph, such as a lowercase "e", and a taller, larger glyph, such an uppercase "M", or even between glyphs of different fonts and point sizes—the standard bounding rectangle may not be sufficient for your application's purposes. Therefore, you can also use the **typographic bounding rectangle,** which is the smallest rectangle that encloses the full span of the glyphs from the ascent line to the descent line.

Figure 2-5 shows an example of how the typographic bounding rectangle and standard bounding rectangle relate. The two rectangles are markedly different because the text has no ascenders or descenders. Whereas the standard bounding rectangle encloses just the black bits of the shape, the typographic bounding rectangle takes into account the ascent and descent lines for the shape. If the text in the figure includes glyphs with ascenders and descenders, the typographic bounding rectangle doesn't change, but the standard bounding rectangle does.

Figure 2-5 Standard bounding rectangle and typographic bounding rectangle



Using Typographic Shapes

This section describes the basic method of creating a typographic shape and some ways you might manipulate that shape. You can manipulate all three types of shapes in these ways, unless otherwise noted. For detailed information on using a specific typographic shape, look in this book for the chapter that describes that shape.

Because typographic shapes act like other types of shapes, you can use many of the functions described in *Inside Macintosh: QuickDraw GX Objects* on any of the typographic shapes. You can also use many of the functions described in *Inside Macintosh: QuickDraw GX Graphics* on the typographic shapes; these functions are mentioned in this section. This chapter assumes that you are either familiar with these functions already or have access to these other books—the function descriptions are not repeated here.

This section describes how you can

- position typographic shapes
- hit-test typographic shapes
- measure typographic shapes
- edit typographic shapes
- convert typographic shapes to other shape types
- insert part of a typographic shape into another shape
- flatten typographic shapes

Positioning Typographic Shapes

The initial position of your typographic shape is specified when the shape is created. In some cases, this will suffice. In other cases, you may wish to actually move the shape.

To move a shape to a specified position, use the `GXMoveShapeTo` function. This function positions the shape at a specified point and either changes the shape's transform object to reflect the move or changes the geometry of the shape, depending on whether the `gxMapTransformShape` shape attribute is set. For example, if you want to move the glyph origin of a typographic shape to the point (100.0,50.0), make this call:

```
GXMoveShapeTo(myGlyphShape, ff(100), ff(50));
```

The `GXMoveShapeTo` function can move a shape a specified distance rather than to a specified point, if the map transform bit is set and the shape has the identity transform.

You can also use the `GXMoveShape` function to move a shape a specified distance from its original position. This function moves the shape horizontally and vertically by the distances you specify. It either changes the shape's transform object to reflect the move or

changes the geometry of the shape, depending on whether the `gxMapTransformShape` attribute is set. For example, if you want to move the glyph origin of a typographic shape down and to the right by 30 points, make this call

```
GXMoveShape(myTextShape, ff(30), ff(30));
```

The `GXMoveShapeTo` and `GXMoveShape` functions are described in *Inside Macintosh: QuickDraw GX Objects*.

Hit-Testing Typographic Shapes

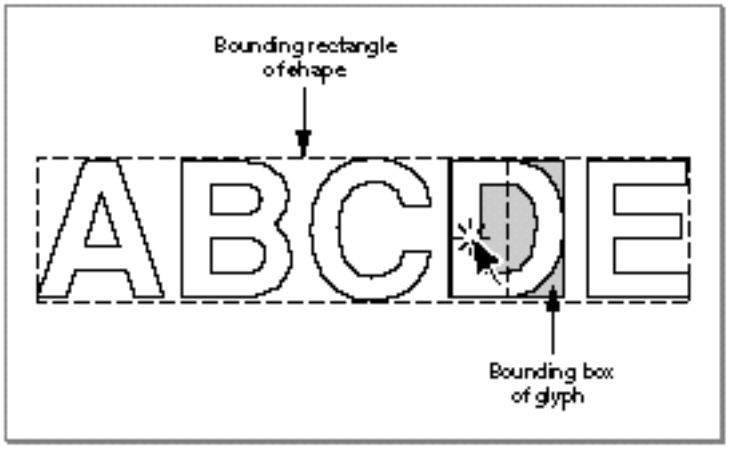
Hit-testing is a way of determining where the user clicked. To hit-test a typographic shape, QuickDraw GX provides two functions: `GXHitTestShape` and `GXHitTestLayout`.

Using GXHitTestShape

You can use the `GXHitTestShape` function to hit-test any kind of shape. This function takes parameters specifying the shape and the point where the user clicked, and it returns a `gxHitTestInfo` structure, which contains information about the specified point: its location in the shape and its distance from the bounding box of the glyph, depending on the hit-test parameters.

Figure 2-6 shows an example of hit-testing a typographic shape.

Figure 2-6 Hit-testing a typographic shape



The `GXHitTestShape` function and the `gxHitTestInfo` structure are described in *Inside Macintosh: QuickDraw GX Objects*. Functions specifically designed to hit-test layout shapes are described in the chapter “Layout Carets, Highlighting, and Hit-Testing” in this book.

Using GXHitTestLayout

You can use the `GXHitTestLayout` function to convert a view port location (representing, for example, the position of a mouse-down event) into an edge offset in the source text of a layout shape. The `GXHitTestLayout` function determines which part of which glyph in the display text of a layout shape is closest to a particular location.

The `GXHitTestLayout` function is described in the chapter "Layout Carets, Highlighting, and Hit-Testing" in this book.

Measuring Typographic Shapes

You can measure typographic shapes exactly as you would any other type of shape: using, for example, the `GXGetShapeArea` function. This function returns the area covered by the shape's standard bounding rectangle. You can also measure a typographic shape in different ways: you can get the bounding boxes of the individual glyphs in the shape or the typographic bounding rectangle of the shape.

This section describes how you can

- get the area of a typographic shape
- get and set the standard bounding rectangle
- get the font measurements from a typographic shape
- get the typographic bounding rectangle

Functions specific to the layout shape that measure line lengths and line spans are described in the chapter "Layout Line Control" in this book.

Getting the Area of a Typographic Shape

To get the area of the shape in square points, you can use the `GXGetShapeArea` function. For example, if you want the total area of a layout shape, send the function a reference to the shape, an index of 0, and a reference to a variable that will hold the result of the function:

```
GXGetShapeArea(myLayoutShape, 0, &theArea);
```

The `GXGetShapeArea` function converts a copy of the shape into a path before computing the area. The index can be 0, to refer to all paths, or an index to a specific contour. Because contours don't necessarily have a one-to-one correspondence to characters or glyphs, it is rarely useful to specify a nonzero index for typographic shapes.

The function measures the shape area as defined in the shape's geometry; it does not consider transformations to the shape made by the shape's transform object.

The `GXGetShapeArea` function is described in *Inside Macintosh: QuickDraw GX Graphics*.

Getting and Setting the Standard Bounding Rectangle

To get the bounding rectangle of the shape, which is the smallest rectangle that completely encloses the filled part of the shape, you can use the `GXGetShapeBounds` function, which is described in *Inside Macintosh: QuickDraw GX Graphics*.

You can change the size of a typographic shape using the `GXSetShapeBounds` function, which is also described in *Inside Macintosh: QuickDraw GX Graphics*. By changing the bounding rectangle of the shape, you can change the width and height of the glyphs in the shape.

Figure 2-7 shows how decreasing the bounding rectangle can affect the size of the resulting shape. As with other shape types, the `gxMapTransformShape` attribute of the source shape determines how the function changes the bounding rectangle. If this attribute is set, the function does not alter the shape's geometry directly; if it is not, the function changes the geometry of the source shape to fit the new bounding rectangle.

Figure 2-7 Effects of the `GXSetShapeBounds` function

Getting the Font Measurements From a Typographic Shape

To get the glyph origins, bounding boxes, and left-side bearings of the glyphs in the typographic shape, you can use the `GXGetGlyphMetrics` function, which is described on page 2-24.

Getting the Typographic Bounding Rectangle

To get the typographic bounding rectangle of the shape, you can use the `GXGetShapeTypographicBounds` function, which is described on page 2-26. The function returns a rectangle that completely encloses the shape, from the highest ascent line to the lowest descent line in the shape. (Ascent lines and descent lines are described in the chapter "Introduction to QuickDraw GX Typography" in this book.

You cannot set the typographic bounding rectangle, because the measurements of the glyphs in the shape are determined by the font, not by QuickDraw GX.

Editing Typographic Shapes

In general, you should use the `GXSet`*ShapeType* and `GXSet`*ShapeType*`Parts` functions to edit typographic shapes. Each shape—text, glyph, or layout—has its own specific function, such as `GXSetTextParts`. These functions are more efficient than `GXSet`*ShapeType* functions because you don't have to replace all of the information in a shape at once, as you do with the `GXSet`*ShapeType* functions. Instead, you can replace specific parts of the shape, such as inserting text into the shape's text array, quickly. For example, use `GXSetLayoutParts` to append a single character to an existing layout shape.

Converting Typographic Shapes

You can convert one typographic shape to its primitive form or to a geometric shape, bitmap, or picture. This section describes these operations.

Converting a Typographic Shape to Its Primitive Form

You can use the `GXPrimitiveShape` function, described in *Inside Macintosh: QuickDraw GX Graphics*, to convert any of the typographic shapes to their primitive forms. The primitive form of a typographic shape is a glyph shape, unless the typographic shape has one or more text faces, in which case its primitive form is a path shape.

The primitive glyph shape contains simple styles in its style run; none of the style run entries are `nil`, and the styles do not contain tags, caps, dashes, patterns, joins, font variations, text faces, or any of the layout style features, such as run controls, justification overrides, glyph substitutions, run features, or kerning adjustments. (Font variations and text faces are discussed in the chapter "Typographic Styles" in this book. Caps, dashes, patterns, and joins are discussed in the chapter "Geometric Styles" in *Inside Macintosh: QuickDraw GX Graphics*.)

The `GXPrimitiveShape` function replaces any `nil` styles in the style list with a reference to the shape's style object.

Converting Typographic Shapes to Other Shape Types

You can change any type of typographic shape to a geometric shape, bitmap shape, picture shape, an empty shape, or a full shape, with the varying results shown in Table 2-1.

You cannot, however, change other types of shapes to typographic shapes. If you try to convert the data from a geometric shape, a bitmap shape, or a picture shape to a typographic shape, QuickDraw GX always posts the error `new_shape_contains_invalid_data`.

Table 2-1 Results of converting typographic shapes to other types of shapes

New shape type	Result
Point	A point equal to the upper-left corner of the bounding rectangle of the original shape.
Line	A line from the upper-left corner to the lower-right corner of the bounding rectangle of the original shape.
Curve	A point equal to the glyph origin for the original shape.
Rectangle	The bounding rectangle of the original shape.
Polygon	A polygon of the text from the original shape.
Path	A path shape that traces the text from the original shape.
Bitmap	A bitmap of the text from the original shape.
Picture	A picture of the text from the original shape.
Empty	An empty shape.
Full	A full shape.

You can change any type of typographic shape into another typographic shape with the varying results shown in Table 2-2.

Table 2-2 Converting a typographic shape to another typographic shape

Source shape	Target shape	Result
Glyph shape	Text shape	The source text is gathered together and set as the text shape's text, but the new text shape gets the default style (which may not match the style information of the original glyph shape). None of the tangent, positioning, or advance bit information is preserved (except for the initial position, which is set as the position of the text shape).
Glyph shape	Layout shape	The source text and style run information is converted. Any specified tangents, advance bits, or positions are not included in the resulting layout shape (except for the initial position, which is set as the position of the layout shape).
Text shape	Glyph shape	A one-run glyph shape using the text shape's text and the default style is the result.
Text shape	Layout shape	A one-run layout using the text shape's text and the default style is the result.

continued

Table 2-2 Converting a typographic shape to another typographic shape (continued)

Source shape	Target shape	Result
Layout shape	Glyph shape	This conversion can be done in one of two ways. If `GXSetShapeType` is used, the resulting glyph shape has the same geometry as the layout shape, with none of the layout effects. If `GXPrimitiveShape` is used, the resulting glyph shape has all the layout effects.
Layout shape	Text shape	The source text is gathered together and set as the text shape's text, but the new text shape gets the default style (which may not match the style information of the original glyph shape). None of the layout effects are preserved.

Inserting Part of a Typographic Shape Into Another Shape

You can use the `GXGetShapeParts` and `GXSetShapeParts` functions, described in the chapter "Geometric Shapes" in *Inside Macintosh: QuickDraw GX Graphics*, to insert parts of one typographic shape into another typographic shape. You can also use the `GXSet`*`ShapeType`*`Parts` functions described in the chapters "Text Shapes," "Glyph Shapes," and "Layout Shapes" in this book.

Note

When you use these functions with typographic shapes, a "shape part" consists of one or more characters or glyphs and any styles that apply to those characters or glyphs. These functions do not use the hit-testing `gxShapeParts` enumeration, described on page 2-23. u

For example, suppose a text shape contains "abcd" and a glyph shape contains "efgh". The following code fragment inserts the first three glyphs from the text shape ("abc") into the middle of the glyph shape (after "f"):

```
GXGetShapeParts(myTextShape, 1, 3, myInsertShape);
GXSetShapeParts(myGlyphsShape, 2, 0, myInsertShape,
myEditShapeFlags);
```

The result of this code would be a glyph shape that reads "efabcgh".

If you want to put part of a geometric shape, a full shape, or a picture shape into a glyph or layout shape, the `GXSetShapeParts` function converts the part to be inserted to the empty shape. If you want to insert the part into a text shape, the function converts both shapes to path shapes. Table 2-3 lists the behavior for most of the general operations involving the `GXSetShapeParts` function.

Table 2-3 Setting the shape parts of various types of shapes

Source shape	Target shape	Action
Typographic	Point, line	Changes a typographic shape to a polygon shape.
Typographic	Curve, path	Changes a typographic shape to a path shape.
Typographic	Rectangle	Posts the error `rectangle_cannot_be_inserted_into`.
Typographic	Bitmap	Posts the warning `shape_operator_may_not_be_a_bitmap`.
Typographic	Picture	Replaces the specified shape or shapes in the picture with the parts of the source shape.
Glyph, layout	Text	Changes a text shape to a glyph or layout shape.
Geometry, full, picture	Glyph, layout	Converts the source shape to an empty shape and posts the warning `shape_does_not_contain_text`.
Bitmap	Glyph, layout	Posts the warning `shape_operator_may_not_be_a_bitmap`.
Geometry, full, picture	Text	Changes both shapes to path shapes.

Note

The layout shape has its own functions for getting and setting shape parts. The functions `GXSetLayoutShapeParts` and `GXGetLayoutShapeParts` are described in the chapter "Layout Shapes" in this book. The `GXGetTextParts` and the `GXSetTextParts` functions are described in the chapter "Text Shapes" in this book. The `GXGetGlyphParts` and the `GXSetGlyphParts` functions are described in the chapter "Glyph Shapes" in this book. u

Flattening Typographic Shapes

You can use the `GXFlattenShape` function (described in *Inside Macintosh: QuickDraw GX Objects*) on typographic shapes exactly as you would on any other type of shape. Flattening a shape, however, does not flatten the fonts that are in that shape. It does flatten all the style objects in the style list that are part of the geometry of a glyph or layout shape.

When you flatten a shape that contains fonts, QuickDraw GX creates a **flat font list.** This list specifies which fonts were used in the shape, which glyphs in that font were used in the shape, or both. Each entry in the flat font list has the tag `'flst'`.

To flatten the fonts in the shape and include them along with the flattened shape, you can use the entries in the flat font list.

For more information about flattening shapes, see the chapter “Shape Objects” of *Inside Macintosh: QuickDraw GX Objects* and the chapter “QuickDraw GX Stream Format” of *Inside Macintosh: QuickDraw GX Environment and Utilities.*

Applying Functions Described Elsewhere to Typographic Shapes

QuickDraw GX provides only a small number of functions that apply exclusively to typographic shapes. However, most of the QuickDraw GX functions that apply to other types of shapes can also be applied to typographic shapes.

The next four sections give an overview of these functions and their effects on typographic shapes.

- “Shape-Related Functions,” lists functions that operate on typographic shape objects and geometric operations that work for typographic shape geometries.
- “Style-Related Functions” on page 2-20 discusses how style-related functions affect the drawing of typographic shapes.
- “Ink- and Color-Related Functions” on page 2-20 discusses how the transfer mode of a typographic shape’s ink object is used to draw a typographic shape.
- “Transform- and View-Related Functions” on page 2-20 lists the functions that allow you to map and clip a typographic shape as well as set its hit-test parameters and its view-port list. This section also includes the functions that manipulate the typographic shape associated with a view device object.

Shape-Related Functions

You can apply all of the functions described in the “Shape Objects” chapter of *Inside Macintosh: QuickDraw GX Objects* to typographic shapes. These functions allow you to:

- manipulate the shape object that represents the typographic shape; for example, copy, clone, cache, compare, and dispose of the typographic shape
- set the geometry, shape type, shape fill, and shape attributes of the typographic shape
- change the style, ink, and transform objects that are associated with the typographic shape
- manipulate the typographic shape’s tags and owner count

Table 2-4 gives a partial list of the functions from the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects.* You should be aware of the results these functions have when you apply them to typographic shapes. All other general shape functions should act on typographic shapes exactly as they do on any other type of shape.

Table 2-4 Selected effects of shape-related functions that you can apply to typographic shapes

Function name	Action taken
`GXCopyToShape`	Makes a copy of the typographic shape. It does not copy the styles in the styles list (for glyph and layout shapes).
`GXCopyDeepToShape`	Makes a copy of the typographic shape, including copies of the style in the styles list (for glyph and layout shapes).
`GXGetShapeStyle`	Returns the style object associated with the typographic shape.

You can also apply certain geometric shape functions to typographic shapes. Table 2-5 gives a selected list of the functions from the chapter "Geometric Shapes" in *Inside Macintosh: QuickDraw GX Graphics*. You should be aware of the effects of these functions have when you apply them to typographic shapes.

Table 2-5 Geometric shape functions that you can apply to typographic shapes

Function name	Action taken
`GXCountShapeContours`	For text and layout shapes, the function returns the number of bytes. For glyph shapes, it returns the number of characters.
`GXCountShapePoints`	Returns 1 for text shapes. For glyph and layout shapes, the `contour` parameter specifies the style run about which you want information (or, if you pass 0, the entire shape). For glyph shapes, the function returns the number of characters in the specified style run or in the shape. For layout shapes, the function returns the number of bytes in the specified style run or in the shape.
`GXGetShapeIndex`	Posts the warning `graphic_type_does_not_have_multiple_contours` and returns 0.
`GXGetShapePoints`	For text and layout shapes, the function always returns 1. For glyph shapes, it returns the positions array of the shape and the number of entries in the array.
`GXSetShapePoints`	For text and layout shapes, sets the initial glyph position of the shape. For glyph shapes, the function sets elements in the positions array.

You can also apply certain geometric operation functions to typographic shapes. Table 2-6 gives partial list of the functions from the chapter "Geometric Operations" in *Inside Macintosh: QuickDraw GX Graphics*. You should be aware of the effects these functions have when you apply them to typographic shapes. All other functions in this chapter should act on, or have no meaning for, typographic shapes.

Table 2-6 Geometric operations that you can apply to typographic shapes

Function name	Action taken
`GXBreakShape`	Converts text and layout shapes into glyph shapes. On glyph shapes, the function uses the `index` parameter as a character index and splits the style run at that index into two runs.
`GXContainsBoundsShape`	Returns `true` if the rectangle indicated in the `container` parameter contains the bounding rectangle of the untransformed typographic shape; returns `false` otherwise.
`GXContainsShape`	Treats both parameters of the function as though they were path shapes. The function returns `true` if the `container` parameter is equal to or larger than the area of the `test` parameter.
`GXGetShapeDirection`	Posts the error `illegal_type_for_shape`.
`GXReverseShape`	Posts the warning `contour_out_of_range`.
`GXReduceShape`	Posts the warning `graphic_type_cannot_be_reduced`.
`GXSimplifyShape`	Posts the notice `shape_already_in_simple_form`.
`GXPrimitiveShape`	Creates the primitive form of the typographic shape.
`GXGetShapeLength`	Posts the warning `shape_does_not_have_length`.
`GXShapeLengthToPoint`	For text and glyph shapes, posts the warning `shape_does_not_have_length`.
`GXGetShapeArea`	Converts layout shapes to glyph shapes before continuing. For text and glyph shapes, the function returns the weighted center of all of the bounding boxes of all glyphs (excluding glyphs without contours, such as the space glyph).
`GXGetShapeCenter`	Returns the point that is the center of the shape's standard bounding rectangle.
`GXGetShapeBounds`	Returns the bounds of the black bits area of the specified glyph in the shape or the bounds of all the glyphs in the shape if you specify an index of 0. For layout shapes, the function returns the bounds as if the layout shape had been converted to a glyph shape by `GXPrimitiveShape`.

continued

Table 2-6 Geometric operations that you can apply to typographic shapes (continued)

Function name	Action taken
`GXSetShapeBounds`	If the new bounds is a translation of the old bounds or if the scaling is square, changes the shape type to a glyph shape and changes the tangents array of the shape. Keep in mind that if the `gxMapTransformShape` attribute is set, this function changes the typographic shape's transform mapping; otherwise, it changes the geometry. If the scaling is not square, the function changes the shape to a path shape.
`GXInsetShape`	Moves the on-curve control points of each glyph by the amount you specify. If you specify very small values for the `inset` parameter, you can use the function to make the glyphs of the shape thicker or thinner, creating a bold or thin look for the glyphs. However, large inset values are discouraged, because they may distort the glyphs beyond recognition.
`GXTouchesBoundsShape`	Treats the typographic shape as if it were a path shape, although the function does not change the shape's actual type. The function returns `true` if the rectangle indicated in the `target` parameter intersects the bounding rectangle of the path shape described by the untransformed typographic shape; returns `false` otherwise.
`GXTouchesShape`	Returns `true` if two shapes intersect. One or both of the shapes may be typographic shapes.
`GXInvertShape`	Posts the warning `shape_cannot_be_inverted`.

`GXIntersectShape`
`GXUnionShape`
`GXDifferenceShape`
`GXReverseDifferenceShape`
`GXExcludeShape`
	If the typographic shape is the shape specified by the `target` parameter, these functions convert the shape to a path shape in most cases. Otherwise, these functions act exactly as they do for other shape types.

The other functions described in the chapter "Geometric Operations" are primarily designed for the geometric, bitmap, and picture shape types. These functions and the errors and warnings that they post are described in *Inside Macintosh: QuickDraw GX Graphics*.

Style-Related Functions

Functions and attributes related to how typographic shapes use the style object are described in the chapter “Typographic Styles” in this book.

Although you can use certain functions described in *Inside Macintosh: QuickDraw GX Graphics* (such as `GXSetShapePen`, `GXSetShapeDash`, and so on) to set the other properties of a typographic shape’s style object and other corresponding functions (`GXGetShapePen`, `GXGetShapeDash`, and so on) to examine those properties, QuickDraw GX ignores these properties when drawing a typographic shape. It does use them, however, when these properties are part of a text face’s style. (See the chapter “Typographic Styles” for more information on how the style object can use geometric properties.)

Ink- and Color-Related Functions

You can use only a single ink object when drawing a typographic shape; thus, the entire shape must share the same color. QuickDraw GX considers the transfer mode of the ink object and applies it when drawing a typographic shape.

Transform- and View-Related Functions

You can apply all of the transform-related functions to typographic shapes. If the `gxMapTransformShape` shape attribute is set, all transformations are applied to the mapping matrix in the shape’s transform object. If the attribute is not set, the transformations are applied to the shape’s geometry. (The layout shape, by default, has the `gxMapTransformShape` shape attribute set; the text and glyph shapes do not.)

Table 2-7 gives a partial list of the functions from the “Transform Objects” chapter of *Inside Macintosh: QuickDraw GX Objects*. You should be aware of the effects these functions have when you apply them to typographic shapes. All other functions should act on typographic shapes exactly as they do on any other type of shape

Typographic Shapes Reference

This section describes constants, data types, and functions that are useful for all three typographic shapes: text shapes, glyph shapes, and layout shapes.

Each shape type also has its own constants, data types, and functions, in addition to the ones described in this chapter. See the chapters “Text Shapes,” “Glyph Shapes,” and “Layout Shapes” in this book.

Constants and Data Types

This section describes how those parts of the `gxShapeAttributes` enumeration and the `gxShapeParts` enumeration that apply specifically to typographic shapes. The `gxShapeAttributes` enumeration is described in more detail in the "Shape Objects" chapter in *Inside Macintosh: QuickDraw GX Objects*; the `gxShapeParts` enumeration is described in the "Transform Objects" chapter of *Inside Macintosh: QuickDraw GX Objects*.

Shape Attributes

Each shape object has a set of shape attributes. **Shape attributes** are a group of flags that modify the behavior of the shape object. The shape attributes are defined as follows:

```
enum gxShapeAttributes{
   gxNoAttributes,
   gxDirectShape           = 0x0001,
   gxRemoteShape           = 0x0002,
   gxCachedShape           = 0x0004,
   gxLockedShape           = 0x0008,
   gxGroupShape            = 0x0010,
   gxMapTransformShape     = 0x0020,
   gxUniqueItemsShape      = 0x0040,
   gxIgnorePlatformShape   = 0x0080,
   gxNoMetricsGridShape    = 0x0100,
   gxDiskShape             = 0x0200,
   gxMemoryShape           = 0x0400
};

typedef long gxShapeAttribute;
```

The following constants are of particular use for typographic shapes:

Constant descriptions

`gxMapTransformShape`
: Indicates that any transforms or mappings you apply to the shape are applied to the shape's transform object. This bit is set by default for layout shapes. If this bit is not set, transforms and mappings are applied to the shape's geometry.

`gxIgnorePlatformShape`
: Indicates that the typographic shape contains only glyph codes, which are 16-bit long values. This value overrides the current setting of the platform value of the style object. Setting this attribute is equivalent to setting the platform of all the styles attached to a shape to `gxGlyphPlatform`.

`gxNoMetricsGridShape`
: Indicates that QuickDraw GX is not to use hints that may be provided by a font. Set this attribute if you intend to manipulate

text as a shape. The hinting can affect a shape's geometry, which may be undesirable if you want to perform other operations, such as scaling, on the shape.

All other values in the `gxShapeAttributes` enumeration are described in the chapter "Shape Objects" in *Inside Macintosh: QuickDraw GX Objects*. Font platforms are described in the chapter "Font Objects" in this book.

Shape Parts

To determine exactly where the user clicked on a glyph, use the values of the `gxShapeParts` enumeration.

```
enum gxShapeParts {/* parts of a gxShape in hit-testing: */
   gxNoPart             = 0,     /* (in order of evaluation) */
   gxBoundsPart         = 0x0001,
   gxGeometryPart       = 0x0002,
   gxPenPart            = 0x0004,
   gxCornerPointPart    = 0x0008,
   gxControlPointPart   = 0x0010,
   gxEdgePart           = 0x0020,
   gxJoinPart           = 0x0040,
   gxStartCapPart       = 0x0080,
   gxEndCapPart         = 0x0100,
   gxDashPart           = 0x0200,
   gxPatternPart        = 0x0400,
   gxGlyphBoundsPart    = gxJoinPart,
   gxGlyphFirstPart     = gxStartCapPart,
   gxGlyphLastPart      = gxEndCapPart,
   gxSideBearingPart    = gxDashPart,
   gxAnyPart            = gxBoundsPart | gxGeometryPart |
      gxPenPart | gxCornerPointPart | gxControlPointPart |
      gxEdgePart | gxJoinPart | gxStartCapPart | gxEndCapPart |
      gxDashPart | gxPatternPart
};

typedef long gxShapePart;
```

The following constants are of particular use for typographic shapes.

Constant type descriptions

`gxBoundsPart` Indicates that the user clicked within the bounds rectangle that surrounds the shape. You can get this rectangle using the `GXGetShapeBounds` function, described in *Inside Macintosh: QuickDraw GX Graphics*.

`gxCornerPointPart`
: Indicates that the user clicked on the glyph advance starting pen position.

`gxControlPointPart`
: Indicates that the user clicked on the character advance ending pen position.

`gxEdgePart`
: Indicates that the user clicked on the line defined by the advance vector (starting pen position to ending pen position).

`gxGlyphBoundsPart`
: Indicates that the user clicked in the bounding box of the glyph.

`gxGlyphFirstPart`
: Indicates that the user clicked on the left or top side of the glyph (depending on the rotation of the shape).

`gxGlyphLastPart`
: Indicates that the user clicked on the right or bottom side of the glyph (depending on the rotation of the shape).

`gxSideBearingPart`
: Indicates that the user clicked in the side bearing of the glyph. You can use this value in combination with either the `gxGlyphFirstPart` or `gxGlyphLastPart` values to determine whether the user clicked on the left or right (top or bottom) side of the glyph.

All other values in the `gxShapeParts` enumeration are described in the "Transform Objects" chapter of *Inside Macintosh: QuickDraw GX Objects*.

Functions

This section describes the functions that allow you to get the measurements of the advance widths, bounding boxes, and side bearings of any of the typographic shapes.

Measuring Typographic Shapes

The `GXGetGlyphMetrics` function returns the metrics of the glyphs produced by the shape. The measurements for these glyphs may change depending on the font, text size, platform value, text direction (horizontal or vertical), and other variables.

The `GXGetShapeTypographicBounds` function returns the typographic bounding rectangle. (See the section "The Standard and Typographic Bounding Rectangles" on page 2-7 for more information about the typographic bounding rectangle.)

GXGetGlyphMetrics

You can use the `GXGetGlyphMetrics` function to determine the metrics of the glyphs in any typographic shape.

```
long GXGetGlyphMetrics(gxShape source, gxPoint glyphOrigins[],
                       gxRectangle boundingBoxes[],
                       gxPoint sideBearings[]);
```

`source` A reference to the typographic shape (text, glyph, or layout) whose metrics you want to determine.

`glyphOrigins`
An array of point structures. On return, the array contains the glyph origins, as points in the view port, for the glyphs in the shape. The array always contains one entry *more* than the number of glyphs in the shape. The last entry is the position of the end of the advance width of the final glyph in the shape. This array is optional; you may pass `nil` for this parameter.

`boundingBoxes`
An array of rectangle structures. On return, the array specifies the bounding boxes for the black portion of each glyph; there is one entry in the array for each glyph in the shape. The bounding boxes are relative to the origin of the shape, not to the beginning of the glyph that the bounding box describes. This array is optional; you may pass `nil` for this parameter.

`sideBearings`
An array of point structures. On return, the array contains the vectors along the advance between the pen position and the glyph's bounding box; there is one entry in the array for each glyph in the shape. This array is optional; you may pass `nil` for this parameter.

function result The number of glyphs in the shape.

DESCRIPTION

The `GXGetGlyphMetrics` function returns the number of glyphs in the shape. The function also returns the various metrics that describe the text, glyph, and layout shapes. Note that the glyph metrics returned by the `GXGetGlyphMetrics` function are device metrics, which are specific to the device on which you are rendering the shape, and not ideal metrics, which are device-independent.

The glyph origins array has one more entry than the number of glyphs in the shape. Be aware that the first entry in the array may not correspond to the given starting position. For example, suppose the layout shape "office" has its position set to (100.0,100.0) and uses hanging punctuation. The entry in the advance bits array for the open quotation mark could be (92.0,100.0) and the entry for the "o" could be (99.0,100.0), because of the

optical alignment of the glyph. (Hanging glyphs and optical alignment are described in the chapter “Layout Styles” in this book.)

The final position in the glyph origins array is the position of the end of the advance width of the final glyph. For example, if the origin of the final glyph is (50.0,50.0), and the glyph is 10 points wide, the last entry in the array will be (60.0,50.0).

ERRORS, WARNINGS, AND NOTICES

Errors

`illegal_type_for_shape (if not typographic)`	(debugging version)
`shape_is_nil`	

GXGetShapeTypographicBounds

You can use the `GXGetShapeTypographicBounds` function to get the typographic bounding rectangle of any typographic shape.

```
gxRectangle *GXGetShapeTypographicBounds(gxShape source,
                                         gxRectangle *rect);
```

`source` A reference to the shape whose typographic bounding rectangle you want.

`rect` A pointer to a rectangle structure. On return, the structure contains the typographic bounding rectangle of the source shape.

function result A pointer to the typographic bounding rectangle of the typographic shape.

DESCRIPTION

The `GXGetShapeTypographicBounds` function returns the typographic bounding rectangle of text, glyph, and layout shapes. The left edge of the rectangle is at the starting position of the first glyph, and the right edge of the rectangle is at the advance width of the final glyph. The height of the bounding rectangle is the distance from the ascent line to the descent line.

For shape types other than typographic shapes, the function posts an error.

ERRORS, WARNINGS, AND NOTICES

Errors

`illegal_type_for_shape`	(debugging version)
`shape_is_nil`	
`parameter_is_nil`	(debugging version)

Summary of Typographic Shapes

Constants and Data Types

Shape Attributes

```
enum gxShapeAttributes{
   gxNoAttributes,
   gxDirectShape           = 0x0001,
   gxRemoteShape           = 0x0002,
   gxCachedShape           = 0x0004,
   gxLockedShape           = 0x0008,
   gxGroupShape            = 0x0010,
   gxMapTransformShape     = 0x0020,
   gxUniqueItemsShape      = 0x0040,
   gxIgnorePlatformShape   = 0x0080,
   gxNoMetricsGridShape    = 0x0100,
   gxDiskShape             = 0x0200,
   gxMemoryShape           = 0x0400
};

typedef long gxShapeAttribute;
```

Shape Parts

```
enum gxShapeParts {/* parts of a gxShape in hit-testing: */
   gxNoPart                = 0,     /* (in order of evaluation) */
   gxBoundsPart            = 0x0001,
   gxGeometryPart          = 0x0002,
   gxPenPart               = 0x0004,
   gxCornerPointPart       = 0x0008,
   gxControlPointPart      = 0x0010,
   gxEdgePart              = 0x0020,
   gxJoinPart              = 0x0040,
   gxStartCapPart          = 0x0080,
   gxEndCapPart            = 0x0100,
   gxDashPart              = 0x0200,
   gxPatternPart           = 0x0400,
   gxGlyphBoundsPart       = gxJoinPart,
   gxGlyphFirstPart        = gxStartCapPart,
```

```
   gxGlyphLastPart         = gxEndCapPart,
   gxSideBearingPart       = gxDashPart,
   gxAnyPart               = gxBoundsPart | gxGeometryPart |
      gxPenPart | gxCornerPointPart | gxControlPointPart |
      gxEdgePart | gxJoinPart | gxStartCapPart | gxEndCapPart |
      gxDashPart | gxPatternPart
};

typedef long gxShapePart;
```

Functions

Measuring Typographic Shapes

```
long GXGetGlyphMetrics      (gxShape source, gxPoint glyphOrigins[],
                             gxRectangle boundingBoxes[],
                             gxPoint sideBearings[]);
gxRectangle *GXGetShapeTypographicBounds
                            (gxShape source, gxRectangle *rect);
```

CHAPTER 3

Text Shapes

Contents

This chapter defines the QuickDraw GX text shape, describes what it contains, and tells how you can create one. Read this chapter if your application has simple text requirements, such as displaying text in a single font, text size, and typestyle.

Before reading this chapter, you should be familiar with the information in *Inside Macintosh: QuickDraw GX Objects*. You should also be familiar with the information in the chapters "Introduction to QuickDraw GX Typography" and "Typographic Shapes" in this book.

This chapter discusses the QuickDraw GX text shape, explains how it relates to the concept of a QuickDraw GX shape object, and describes the properties of text shapes. It then shows how to use QuickDraw GX functions to

- create and draw text shapes
- change parts of text shapes

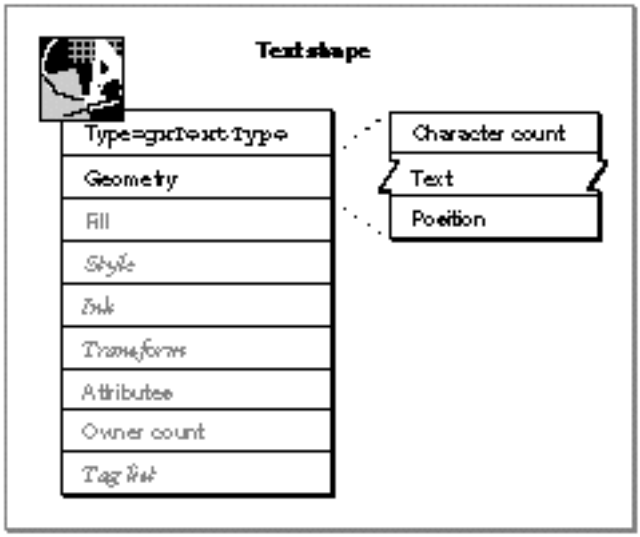
About Text Shapes

A **text shape** is a shape object containing a string of text associated with a single style object. The text shape has the same set of properties as other QuickDraw GX shape objects. Its shape type is `gxTextType`, and its geometry is unique.

The Geometry of a Text Shape

Figure 3-1 shows the properties of the text shape object. Note that, because a text shape is an object and not a data structure, the order of the properties as shown in Figure 3-1 is completely arbitrary.

Figure 3-1 Geometry of a text shape



The geometry of a text shape contains these elements:

- **Character count.** The number of characters in the text array. This number is not necessarily the same as the number of bytes in the text array, because some of the characters may be 16-bit characters.
- **Text.** An array of character codes or glyph codes. Whether each character code is 8-bit or 16-bit depends on both the platform in the text shape's style object and on the actual byte values. If the value of the shape attribute `gxIgnorePlatformShape` is clear, then the text is interpreted as an array of character codes. If it is set, it is interpreted as an array of 16-bit glyph codes. (For more information about platforms, character codes, and glyph codes, see the discussion of encodings in the chapter "Font Objects" in this book.)
- **Position.** A point that marks the glyph origin of the first glyph in the shape.

You use the functions described in this chapter to set the values of the character count, the text, and the position of the text shape.

Because a text shape is a single run, you cannot mix character codes and glyph codes in a text shape; one run can have either character codes or glyph codes—but not both. A text shape always contains one glyph per character, and they are always in the same order. In other words, the character-code index is equal to the glyph index.

Note

The text shape displays only the glyphs the user asks for. For example, if the user types the glyphs "f" and "i", the text shape does not automatically display an "fi" ligature. If you need automatic linguistic processing of glyphs, you should use the layout shape, described in the chapter "Layout Shapes" in this book. u

The Default Text Shape

The default text shape has no text; all of its components have the value 0 or `nil`. Like the default glyph and layout shapes, the default text shape has the `gxWindingFill` type. The `gxWindingFill` type and other shape fills are described in the "Shape Objects" chapter of *Inside Macintosh: QuickDraw GX Objects*.

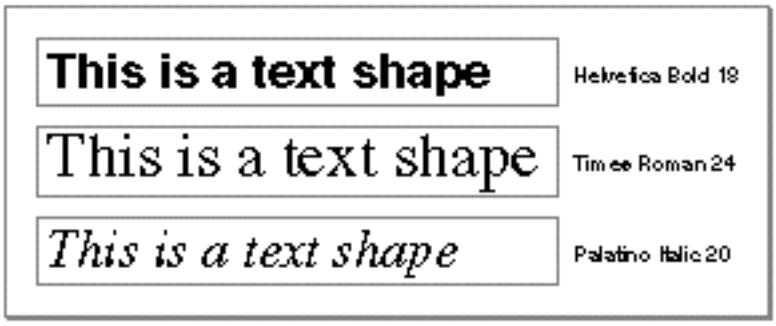
The default text shape has a style object, which is described in detail in the "Typographic Styles" chapter in this book. The default settings for all typographic shapes are described in detail in the chapter "Typographic Shapes" in this book.

The Text Shape and Styles

The style object associated with a text shape contains the font, the text size in points, and other information that determines characteristics of the shape when it is displayed. (For more information about the parts of a style object that apply to typographic shapes, see the chapter "Typographic Styles" in this book.) Because a text shape can have only one style object associated with it, the text of a text shape is displayed in a single style.

Figure 3-2 shows three different examples of a text shape, each with a different style applied to it.

Figure 3-2 Three examples of a text shape, each with a different style applied



Because the text shape has only one style, it is most useful for drawing nonformatted glyphs, such as those used in dialog boxes, terminal emulation programs, or primitive text editors. Because the text shape contains only basic data, it is more efficient where speed is concerned, but it contains less information than the other typographic shapes.

For more complex editors or word processors and for contextual non-Roman text, you should use the layout shape, described in the chapter “Layout Shapes” in this book.

Also, you cannot use a text shape for clipping, dashing, patterns, joins, and start and end caps. Before you can use these features, you must convert the text shape to a glyph shape. You can, however, pattern a text shape by including, in the text shape’s style object, a patterned text face. A text face can also contain joins, start and end caps, dashing, and so on. Text faces are described in the chapter “Typographic Styles” in this book.

In general, you shouldn’t use text shapes and glyph shapes for non-Roman text. For example, if you need to draw text vertically or text from right to left, you should use the layout shape. For more information, see the chapter “Layout Shapes” in this book.

Using Text Shapes

This section describes the functions you use to create and draw text shapes and to change the information in a text shape.

Creating and Drawing a Text Shape

To create a text shape, you can use the `GXNewShape` function, specifying `gxTextType`, and then set properties. Alternatively, you can use the `GXNewText` function, passing it the necessary information. For example, when you create a text shape with `GXNewText`, you must specify its contents and position on the screen.

In Listing 3-1, the value of `myPoint` specifies where QuickDraw GX should display the text shape. The code creates the text shape with the call to `GXNewText`. The text shape starts out with the default style, but the call to `GXSetShapeTextSize` changes the

text size from the default to 120 points. This sample uses the `ff` macro, a shorthand notation for the `IntToFixed` macro. Both versions of the macro are described in the "Mathematics" chapter in *Inside Macintosh: QuickDraw GX Environment and Utilities.*

The `GXNewShape` function is described in the "Shape Objects" chapter in *Inside Macintosh: QuickDraw GX Objects.*

Listing 3-1 Creating a text shape with a nondefault text size

```
gxPoint myPoint = {ff(50), ff(150)};
gxShape myTextShape = GXNewText(4, (unsigned char*)"Wow!",
&myPoint);
GXSetShapeTextSize(myTextShape, ff(120));
```

The function `GXNewText` is described on page 3-8. The function `GXSetShapeTextSize` is described in the chapter "Typographic Styles" in this book.

To draw an existing text shape, use the `GXDrawShape` function.

You can also use the `GXDrawText` function to create, draw, and dispose of a text shape. The function uses the text shape's default style object to determine what the font, text size, and other style attributes of the shape should be. This function is useful if you only need to draw the shape once and if you don't need to set any characteristics of the style used.

```
GXDrawText(myTextLen, myText, myPosition);
```

The `GXDrawText` function is described on page 3-9. The `GXDrawShape` function is described in the "Shape Objects" chapter in *Inside Macintosh: QuickDraw GX Objects.*

Changing Text in a Text Shape

If you want to change all of the text of an existing shape, use the `GXSetText` function, described on page 3-12. This function takes an existing shape, a character count, a string of text, and a position. You can replace the text in the shape, change the position, or both. (If you want to replace only one of these elements, you pass `nil` for the other.)

If you want to change only some of the text of an existing text shape, use the `GXSetTextParts` function, described on page 3-14. The function takes an existing text shape, a glyph index corresponding to a glyph in that shape, a number of characters to be replaced, a number of characters to be added, and the new text to be inserted. You can insert new text while maintaining all the current text, or you can replace or delete existing text from the shape. Table 3-1 lists some of the uses of the `GXSetTextParts` function.

Table 3-1 Changing text in a text shape using the `GXSetTextParts` function

Action	Index	Old character count	New character count	Text
Inserting new text	Glyph index at which new text should start or `gxSelectToEnd`, if you want to insert at the end	0	Length of the new text	Pointer to the new text
Replacing and deleting some text	Glyph index at which the new text should start	Number of glyphs to delete from the original shape or `gxSelectToEnd`	Length of the new text	Pointer to the new text
Replacing all text in the shape	1	`gxSelectToEnd`	Length of the new text	Pointer to the new text
Deleting all text from the shape	1	`gxSelectToEnd`	0	`nil`

For example, suppose you want to change a text shape that reads “The dog” so that it reads “The beast”. The index value, which is the location of the first glyph to replace, is 5. The old character count is 3 (which corresponds to the number of glyphs in the word “dog”), and the new character count is 5 (the number of glyphs in “beast”). You could also set the old character count to the `gxSelectToEnd` constant, because you want to replace all of the text from position 5 to the end of the text in the shape. Listing 3-2 shows how to make this change.

Listing 3-2 Replacing text in a text shape

```
char     *myString = "The dog";
char     *newWord =  "beast";
short    textLength, insertLength;
gxPoint  myPoint = {ff(20), ff(20)};

textLength = strlen(myString);
insertLength = strlen(newWord);
gShape = GXNewText(textLength, (unsigned char *) myString,
                   &myPoint);
GXSetTextParts(gShape, 5, gxSelectToEnd, insertLength,
               (unsigned char *) newWord);
```

Text Shapes Reference

Functions

This section describes the functions that you use for creating, drawing, or changing QuickDraw GX text shapes.

Creating and Drawing Text Shapes

You can create a text shape using the `GXNewText` function. If you simply want to draw some text without creating a new shape, you can use the `GXDrawText` function instead.

GXNewText

You can use the `GXNewText` function to create a text shape.

```
gxShape GXNewText(long charCount, const unsigned char text[],
                  const gxPoint *position);
```

`charCount` The number of character codes in the `text` parameter. For non-Roman scripts, the actual byte length may be up to twice the number of characters. If the value of `charCount` is 0, the text shape is equivalent to the empty shape.

`text` An array of text data. The value of this parameter may be `nil` if the value of `charCount` is 0.

`position` A pointer to the position, in geometry coordinates, of the starting point of the shape. This position is the intersection of the baseline with the left margin of the left-side bearing at the first glyph in the shape. If you pass `nil`, `GXNewText` sets the position to (0.0,0.0).

function result The new text shape.

DESCRIPTION

The `GXNewText` function creates a copy of the default text shape, sets the owner count of the copy to 1, initializes its geometry with the values in the function's parameters, and returns a reference to it as the function result.

The new text shape returned by this function contains references to the same style, ink, and transform objects as the default text shape.

The parameter `charCount` indicates the number of characters present, which may not necessarily equal the number of bytes in the length of the `text` parameter. The interpretation of characters depends on the default text shape's style attribute.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`	
`parameter_is_nil`	(debugging version)
`count_is_less_than_zero`	(debugging version)

SEE ALSO

To create a new glyph shape, use the `GXNewGlyphs` function, described in the chapter "Glyph Shapes" in this book.

To create a new layout shape, use the `GXNewLayout` function, described in the chapter "Layout Shapes" in this book.

For information about the default ink and transform objects, see *Inside Macintosh: QuickDraw GX Objects*.

The default values of the style object for any of the typographic shapes is discussed in the chapter "Typographic Styles" in this book.

You can determine the platform of the style object associated with the default shape using the `GXGetShapeEncoding` function, also described in the chapter "Typographic Styles."

The default text shape is described on page 3-4.

GXDrawText

You can use the `GXDrawText` function to draw a string of text without first creating a text shape.

```
void GXDrawText(long charCount, const unsigned char text[],
                const gxPoint *position);
```

`charCount` The number of characters in the `text` parameter. For non-Roman scripts, the byte length may be up to twice the number of characters.

`text` An array of text data.

`position` A pointer to the position, in geometry coordinates, of the starting point of the shape. This is the intersection of the baseline with the left margin of the left-side bearing at the first glyph in the shape. If you pass `nil`, `GXDrawText` sets the position to (0.0,0.0).

DESCRIPTION

The `GXDrawText` function draws the text string, starting at the point specified by `position`. The `charCount` parameter specifies the number of characters in the text drawn. The default text shape's style object specifies the font, text size, typestyle, baseline direction, and so on.

The parameter `charCount` indicates the number of characters present, which may not necessarily equal the number of bytes in the length of the `text` parameter. The interpretation of characters depends on the text shape's style attribute.

If the value of the `charCount` parameter is 0, the `GXDrawText` function does nothing.

You should use the `GXDrawShape` function if you want to draw the text contained in the `text` parameter several times (by creating a text shape) or if you want to draw the text using a style other than the default style.

ERRORS, WARNINGS, AND NOTICES

Errors

`parameter_is_nil`	(debugging version)

SEE ALSO

To draw an existing text shape, use the `GXDrawShape` function, described in the chapter "Shape Objects" in *Inside Macintosh: QuickDraw GX Objects*.

To draw a line of text that contains glyphs of different fonts, text sizes, or typestyles, use of the `GXDrawGlyphs` function, described in the chapter "Glyph Shapes" in this book.

To draw a line of text using the unique characteristics of the layout shape, see the description of the `GXDrawLayout` function in the chapter "Layout Shapes" in this book.

Manipulating Geometries of Text Shapes

You can obtain the values in a text shape's geometry—the number of character codes it contains, the character codes themselves, or the shape's position—using the `GXGetText` function. You can set any of these values, including replacing all of the text of the shape, by using the `GXSetText` function.

If you want to retrieve part of the text contained in the shape, use the `GXGetTextParts` function. If you want to change only some of the text in a text shape, use the `GXSetTextParts` function.

GXGetText

You can use the `GXGetText` function to return the information in a text shape's geometry, such as its text string or its position.

```
long GXGetText(gxShape source, long *charCount,
               unsigned char text[], gxPoint *position);
```

`source` A reference to the text shape whose character code values you want to change.

`charCount` A pointer to a `long` value. On return, the number of characters in the shape specified by `source`. If you pass `nil` in this parameter, `GXGetText` does not return a value.

`text` A character array. On return, it contains the text string from the source shape. You must allocate the memory for this string. If you pass `nil` in this parameter, `GXGetText` does not return the text string.

`position` A pointer to a point structure. On return, the point is the position, in geometry coordinates, of the text shape. If you pass `nil`, `GXGetText` does not return the position.

function result The number of bytes of text in the text shape.

DESCRIPTION

The `GXGetText` function returns the number of characters, the text string, the position of a text shape, and (in the function result) the byte length of the text shape specified by `source`. The `charCount` parameter may not be equal to the function result; `charCount` indicates the number of characters in the shape, which may not necessarily equal the number of bytes. Call this function twice, once to determine the size of the text array (pass `nil` for `text`), and a second time to fill out the array.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`illegal_type_for_shape` (debugging version)

SEE ALSO

To get information from the geometry of a glyph shape, use the `GXGetGlyphs` function, described in the chapter "Glyph Shapes" in this book.

To get information from the geometry of a layout shape, use the `GXGetLayout` function, described in the chapter "Layout Shapes" in this book.

GXSetText

You can use the GXSetText function to insert a new text string into a text shape, or to change the text shape's position, or both.

```
void GXSetText(gxShape target, long charCount,
               const unsigned char text[],
               const gxPoint *position);
```

target	A reference to the text shape whose character code values you want to change.
charCount	The number of characters to be copied into the new text shape.
text	A character array containing the text to be copied into the text shape.
position	A pointer to a point structure specifying the location of the text shape, in geometry coordinates.

DESCRIPTION

The GXSetText function replaces the geometry of the specified shape with the text in the text parameter and the point in the position parameter. The shape type is set to gxTextType.

If the value of charCount is 0 and the value of text is not nil, the number of bytes in the existing text shape determines the length of the text copied.

The parameter charCount indicates the number of characters present, which may not necessarily equal the number of bytes in the length of the text parameter. The interpretation of characters depends on the text shape's style attribute.

You can shorten the shape by passing nil for the text parameter and a new character count for the text shape. QuickDraw GX changes the contents of the text shape to however many glyphs you specify. You cannot lengthen a text shape using this method, because QuickDraw GX returns the warning new_shape_contains_invalid_data.

If you don't want to change one of the values in the shape, such as the text or the position, set that parameter to nil. If you want to set the value of either the text or the position to nil, pass the value gxSetToNil in that parameter.

ERRORS, WARNINGS, AND NOTICES

Errors

shape_is_nil	
count_is_less_than_zero	(debugging version)
size_of_text_exceeds_implementation_limit	(debugging version)
out_of_memory	(debugging version)

Warnings

shape_access_not_allowed	(debugging version)
new_shape_contains_invalid_data	(debugging version)

Notices (debugging version)
`text_already_set`

SEE ALSO

For more information on how to use the `GXSetText` function, see "Changing Text in a Text Shape" beginning on page 3-6.

You can use the `GXSetTextParts` function, described on page 3-14, to replace part of the text in a text shape or to insert new text into the shape.

You can also change the position of a text shape using the `GXMoveShape` or `GXMoveShapeTo` function, described in the "Transform Objects" chapter in *Inside Macintosh: QuickDraw GX Objects.*

To replace the geometry of a glyph shape with new text, use the `GXSetGlyphs` function, described in the chapter "Glyph Shapes" in this book.

To replace the geometry of a layout shape with new text, use the `GXSetLayout` function, described in the chapter "Layout Shapes" in this book.

To change a shape of another type into a text shape, use the `GXSetShapeType` function, described in the "Shape Objects" chapter of *Inside Macintosh: QuickDraw GX Objects.*

GXGetTextParts

You can use the `GXGetTextParts` function to retrieve part of the text of a text shape.

```
long GXGetTextParts(gxShape source, long index, long charCount,
                    unsigned char text[]);
```

`source`	A reference to the shape from which you retrieve data.
`index`	The index of the first glyph that you want to retrieve. This value must be greater than or equal to 1.
`charCount`	The number of character codes you want to retrieve. This value must be greater than or equal to 1, or it can be `gxSelectToEnd`, which selects all glyphs, beginning at the glyph index specified in `index`.
`text`	A pointer to a character array. On return, the array contains the text retrieved from the source shape.

function result The number of bytes of the retrieved text.

DESCRIPTION

The `GXGetTextParts` function retrieves the specified number of character codes from the shape. The `text` parameter is a pointer to the retrieved text. You can retrieve some or all of the character codes from the shape using this function.

The shape specified in the `source` parameter must be of type `gxTextType`. If it isn't, `GXGetTextParts` posts the error `illegal_type_for_shape`.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`	
`index_is_less_than_one`	(debugging version)
`count_is_less_than_one`	(debugging version)
`illegal_type_for_shape`	(debugging version)

Warnings

`index_out_of_range`
`count_out_of_range`

SEE ALSO

You can use the `GXGetText` function (page 3-11) to retrieve all of the text of a text shape.

To retrieve part of the text from a glyph shape, use the `GXGetGlyphParts` function, described in the chapter "Glyph Shapes" in this book.

To retrieve part of the text from a layout shape, use the `GXGetLayoutParts` function described in the chapter "Layout Shapes" in this book.

GXSetTextParts

You can use the `GXSetTextParts` function to change part of the text of a text shape.

```
void GXSetTextParts(gxShape target, long index, long oldCharCount,
                    long newCharCount, const unsigned char text[]);
```

`target` A reference to the text shape whose character code values you want to change.

`index` The index of the first glyph where the editing operation will begin. If you are deleting text, this is the first glyph deleted; if you are inserting text, the new glyphs will appear before this one. This value must be greater than or equal to 0. A value of `gxSelectToEnd` in the `index` parameter indicates that `GXSetTextParts` should add the new glyphs after the last glyph in the text string. A value of 1 indicates the beginning of the text of the shape.

`oldCharCount`
: The number of glyphs you want to replace. This value can be greater than or equal to 0 (which indicates that you want to insert text), or it can be `gxSelectToEnd` (which is equal to –1 and indicates that you want to replace the text from the position specified by `index` to the end of the available text).

`newCharCount`
: The number of character codes you want to add to the text shape.

`text`
: A pointer to a character array containing the new text that you want to put into the text shape.

DESCRIPTION

The `GXSetTextParts` function replaces the text in the target shape with the text pointed to by the `text` parameter. The function inserts the new text at the glyph index specified by the `index` parameter. If the value of `oldCharCount` is greater than 0, the function replaces a corresponding number of glyphs in the original text string with a number of glyphs from the new text string. The number of new glyphs added is specified by the `newCharCount` parameter. If the value of the `oldCharCount` parameter is 0, the function inserts the new text but does not delete any of the original text.

The target shape must be of type `gxTextType`. If it isn't, `GXSetTextParts` posts the error `illegal_type_for_shape`.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`	
`parameter_out_of_range`	(debugging version)
`inconsistent_parameters`	(debugging version)
`index_is_less_than_zero`	(debugging version)
`count_is_less_than_zero`	(debugging version)
`illegal_type_for_shape`	(debugging version)

Warnings

`index_out_of_range`	
`count_out_of_range`	
`shape_access_not_allowed`	(debugging version)

SEE ALSO

For more information on how to use the `GXSetTextParts` function, see "Changing Text in a Text Shape" beginning on page 3-6.

To replace all of the text in a text shape, you can use the `GXSetText` function, described on page 3-12.

To replace glyphs in a glyph shape, use the `GXSetGlyphParts` function, described in the chapter "Glyph Shapes" in this book.

To replace glyphs in a layout shape, use the `GXSetLayoutParts` function, described in the chapter "Layout Shapes" in this book.

Summary of Text Shapes

Functions

Creating and Drawing Text Shapes

```
gxShape GXNewText          (long charCount, const unsigned char text[],
                            const gxPoint *position);
void GXDrawText            (long charCount, const unsigned char text[],
                            const gxPoint *position);
```

Manipulating Geometries of Text Shapes

```
long GXGetText             (gxShape source, long *charCount, unsigned char
                            text[], gxPoint *position)
void GXSetText             (gxShape target, long charCount, const unsigned
                            char text[], const gxPoint *position);
long GXGetTextParts        (gxShape source, long index, long charCount,
                            unsigned char text[]);
void GXSetTextParts        (gxShape target, long index, long oldCharCount,
                            long newCharCount, const unsigned char text[]);
```

CHAPTER 4

Glyph Shapes

Contents

This chapter describes the QuickDraw GX glyph shape and its contents, and tells how to create and use a glyph shape. Read this chapter if you want your application to use a typographic shape as a graphic entity—for example, to dash text along a path or place glyphs in arbitrary positions.

Before reading this chapter, you should be familiar with the information in the chapters "Introduction to QuickDraw GX Typography" and "Typographic Shapes" in this book. You should also be familiar with the information in *Inside Macintosh: QuickDraw GX Objects*.

This chapter discusses the QuickDraw GX glyph shape and describes the properties of glyph shapes. It then shows how to use QuickDraw GX functions to

- create and draw glyph shapes
- change parts of a glyph shape
- set the tangents, positions, and advance bits arrays of a glyph shape

About Glyph Shapes

The **glyph shape** is a typographic shape that allows you to vary the position, font, rotation, and scale of each glyph in a line of text. The glyph shape has the same set of properties as other QuickDraw GX shape objects. Its shape type is `gxGlyphShape`, and its geometry is unique.

A glyph shape is drawn as one or more glyphs. One glyph shape can represent a character, such as "a", or a ligature, such as "fi". Note that the glyph shape, as defined in QuickDraw GX, is a specific type of shape object. In this context, *shape* does not refer to the form of an individual glyph.

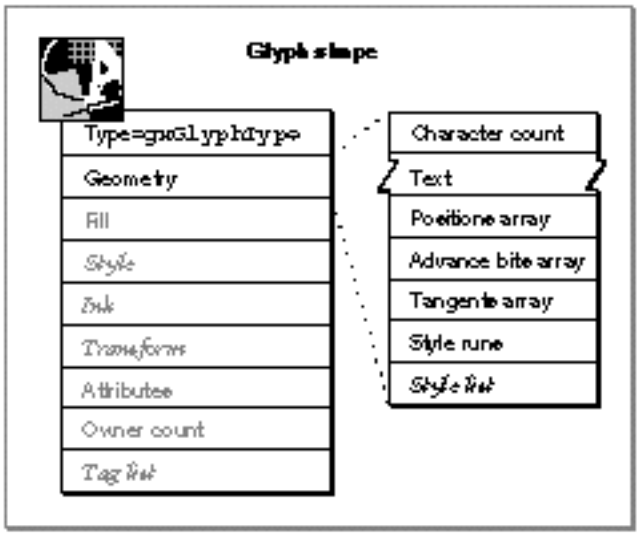
The glyph shape is primarily useful as a graphic entity. Because of its geometry, the glyphs in a glyph shape can be positioned individually, yet still be related to the other glyphs in the shape. This allows you to create a number of graphic effects with the glyph shape that you can't with the text or layout shapes, such as setting text along an arbitrary path—for example, in a circle.

The glyph shape displays only the glyphs for the specific character codes or glyph codes it contains. For example, if you draw a glyph shape that contains adjacent character codes for the glyphs "f" and "i", the glyph shape does not automatically display an "fi" ligature. If you need such automatic linguistic processing of glyphs, you should use the layout shape, described in the chapter "Layout Shapes" in this book.

The Geometry of a Glyph Shape

The geometry of a glyph shape contains seven elements, as shown in Figure 4-1. Note that, because a glyph shape is an object and not a data structure, the order of the properties as shown in Figure 4-1 is completely arbitrary.

Figure 4-1 Geometry of a glyph shape



The geometry of the glyph shape contains the following elements:

- **Character count.** The number of characters in the text array. Note that the number of characters is not necessarily the same as the number of bytes in the text array.
- **Text.** An array of character codes or glyph codes. Whether each character code is 8-bit or 16-bit depends on the platform in the glyph shape's style object and on the actual byte values. If the value of the shape attribute `gxIgnorePlatformShape` is clear, then the text is interpreted as an array of character codes. If it is set, it is interpreted as an array of 16-bit glyph codes. (See the chapter "Font Objects" in this book for more information about platforms, character codes, and glyph codes.)
- **Positions array.** Contains positions for the origin of each character or glyph in the shape. These positions can be relative to the advance width of the previous character or glyph, or they can be absolute positions in geometry coordinates. The first position in the positions array marks the origin of the first character or glyph in the shape.
- **Advance bits array.** Determines whether the points in the positions array are absolute or relative. The advance bits array contains 1 bit for every character or glyph in the shape.
- **Tangents array.** Determines the scaling and orientation of the characters or glyphs in the shape. It contains one entry for each character or glyph in the shape.
- **Style runs array.** Determines how many characters or glyphs in the shape each style applies to. Each entry is the number of characters or glyphs to which the equivalent entry in the style list applies.
- **Style list.** An array of references to the style object attached to the shape. This allows a glyph shape to have more than one style. If there are styles in this array, QuickDraw GX does not use the information in the style object associated with the shape.

You use the functions described in this chapter to set the values in the geometry of the glyph shape.

The Positions and Advance Bits Arrays

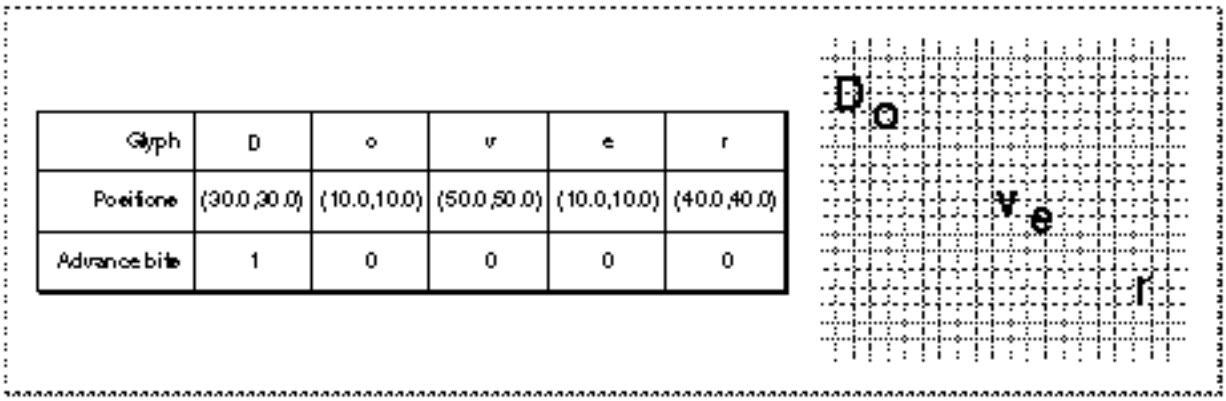
There are two ways of positioning a glyph in a glyph shape: it can be positioned relative to the preceding glyph, or it can have its own position—except for the first glyph, which has to be absolute, not relative—specified as a QuickDraw GX coordinate. Each position is stored in the **positions array** as a point, such as (30.0,50.0).

For every position in the positions array, there is a corresponding bit in the **advance bits array.** A value of 0 for an advance bit indicates a relative position—that is, the position of the glyph at that index is relative to the end of the advance width of the preceding glyph. A value of 1 indicates an **absolute position,** which specifies that the position is absolute in geometry coordinates. For example, suppose the positions array specifies the point (30.0,50.0) for a glyph. If the advance bits array indicates a **relative position,** QuickDraw GX sets the origin of the glyph at 30 points along the x-axis and 50 points along the y-axis from the end of the advance width of the previous glyph. If the advance bits array indicates that the position is absolute, QuickDraw GX draws the glyph at the point (30.0,50.0) in geometry space.

The order of bits in the array moves from high to low, meaning that the highest-order bit in the entire array is for glyph 1. If there are less than 32 bits per glyph shape, the low-order bits are ignored. Because the advance bits are stored in long integers, the array contains a minimum of 32 bits per glyph shape. The value of the first bit is always 1, because the first position in the array is always absolute.

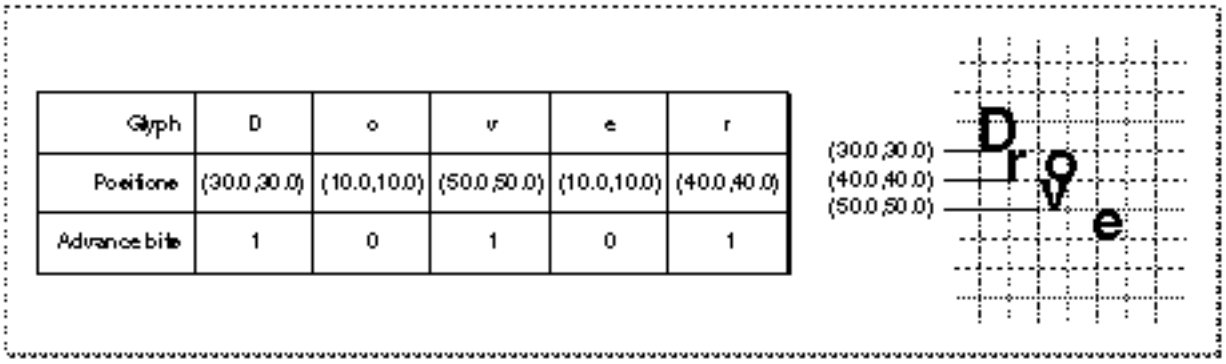
In Figure 4-2, the glyphs in the shape “Dover” have the absolute position (30.0,30.0) and the relative positions (10.0,10.0), (50.0,50.0), (10.0,10.0), and (40.0,40.0). The initial “D” has an absolute position; the glyphs following have relative positions, because their corresponding bits in the advance bits array are 0.

Figure 4-2 The effect of the positions and advance bits arrays on glyph placement



In Figure 4-3, the shape has the same glyphs and the same values in the positions array. However, the first, third, and fifth glyphs now have absolute positioning.

Figure 4-3 The same shape with a new advance bits array



If you don't set positions for the glyph shape explicitly, QuickDraw GX sets the values of all the advance bits (except the first one) and positions to 0 and (0.0,0.0) respectively, resulting in glyphs that line up next to one another, each starting where the previous glyph's advance width ends. It sets the first advance bit to 1, because the first position is always absolute.

However, if you set positions for the shape without specifying the advance bits, QuickDraw GX sets every bit in the advance bits array to 1, because QuickDraw GX interprets all of the positions as absolute positions. If you want some positions to be relative, you must pass the advance bits array with the appropriate bits set to 0.

The Tangents Array

Each glyph in a glyph shape may have an optional **tangent** vector that specifies the scale and angle that QuickDraw GX should apply before drawing that glyph. The tangent vector is represented by a point: the direction of the vector is the direction from the point (0.0,0.0) to the tangent point, and the length of the vector is determined by the length of the line between those two points. The glyph with that tangent is laid out along that vector accordingly, with its glyph origin aligned with the point (0.0,0.0) and the advance width scaled to match the length of the vector. For a 1-unit-long vector, the glyph is "scaled" to 100 percent of its size.

Here are some characteristics of a tangent that you should know about:

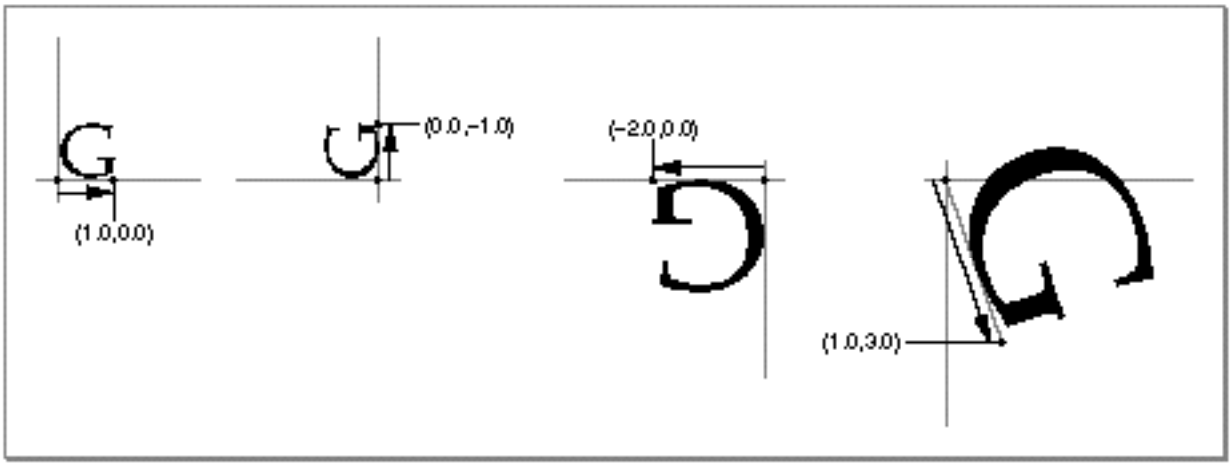
- The tangent vector coordinate system matches the QuickDraw GX coordinate system: values for the tangent are always (x,y), regardless of the baseline of the glyph or glyphs associated with that tangent.
- QuickDraw GX always processes the characters in storage order and always draws the text of a typographic shape from left to right (regardless of the inherent line direction of the text), so the direction of the tangent is the direction of the advance vector.

Figure 4-4 shows the effect of a variety of tangents on a glyph. All of the glyphs shown have the same style, and therefore the same font and text size; they differ only in how the tangents affect their appearance.

In the first example, a glyph has the default tangent vector of (1.0,0.0), which specifies that there is a scaling factor of 1 and no rotation from the natural text direction; a glyph with this tangent vector (typical for Roman text) would draw left to right, and a glyph with the `gxVerticalText` text attribute (typical for Kanji) set in the shape's style would draw top to bottom.

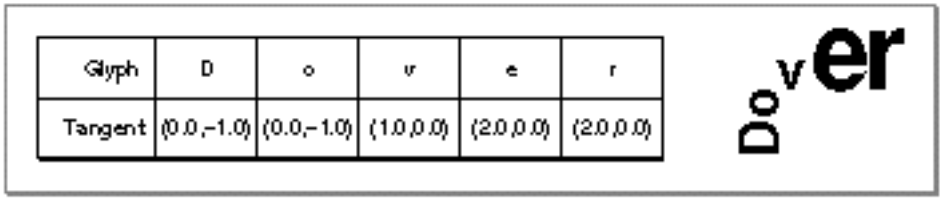
The second example shows a tangent vector of (0.0,-1.0): the glyph is rotated 90 degrees upward from the natural orientation. The third example shows a tangent vector of (–2.0,0.0): notice how the glyph scales to double its normal size to fit the length of the vector. In the fourth example, the glyph follows a tangent of (1.0,3.0). The glyph scales in all directions to fit the tangent.

Figure 4-4 Various tangents



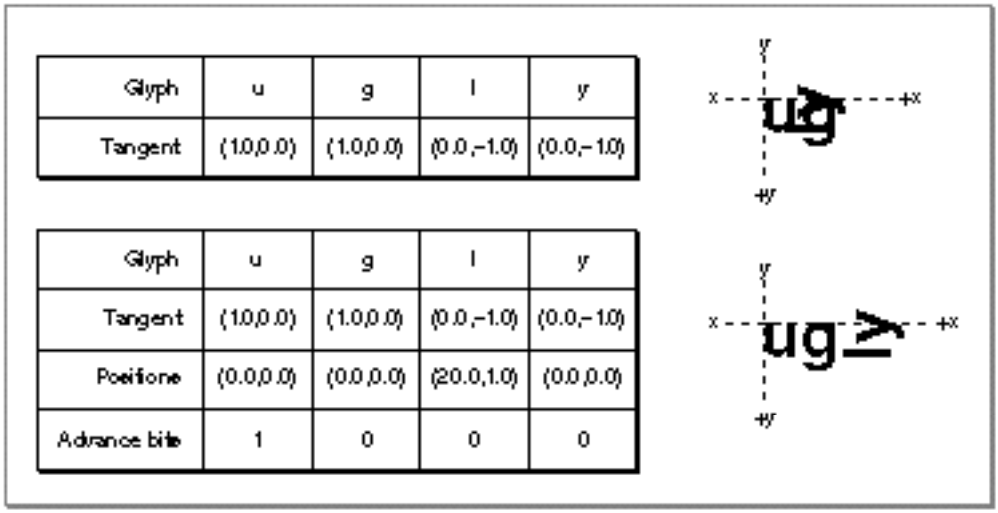
In Figure 4-5, the shape "Dover" has five separate tangents, one for each glyph in the shape. The first two rotate the glyph counterclockwise by 90 degrees off the natural baseline. The last three put the shape back onto the natural baseline: note how the origin of the glyph "v" begins where the advance width of the glyph "o" ends, even though the glyphs have different orientations. The last two tangents are (2.0,0.0), which scale those glyphs to twice their given point size. This shape uses the default positions and advance bits, and a single style for the entire shape, allowing the glyph shape to have glyphs of different sizes in a single style run.

Figure 4-5 The effect of the tangents array on glyph placement



QuickDraw GX begins drawing the glyph origin of a glyph in relation to the end of the advance width of the previous glyph, if there are no other factors affecting the placement of the glyphs in the geometry. In the top part of Figure 4-6, the glyph origin of the glyph “l” begins where the advance width of the glyph “g” ended, and the orientation of the tangents of the glyphs “l” and “y” cause those glyphs to overwrite the previous glyphs. In the bottom part of Figure 4-6, use of the positions and advance bits arrays changes the glyph origins of “l” and “y” so that they are farther away from previous glyphs and avoid overwriting.

Figure 4-6 Tangents used with and without positions



The Style Runs and Style List

Every glyph in a glyph shape must belong to a style run. A **style run** consists of a number of consecutive glyphs that share an associated style object (which contains the font, the typestyle, the text size, and other specific characteristics). By default, QuickDraw GX assigns the style object associated with the shape to all glyphs in a glyph shape. However, each glyph in a glyph shape can be in its own style. The shape’s geometry contains the style list, which is a list of references to style objects. (For more information about what is contained in the style object that applies to typographic shapes, see the chapter “Typographic Styles” in this book.)

Figure 4-7 shows a four-glyph shape with two style runs, each containing two glyphs. The first style run contains the font 24-point Courier Roman for two glyphs, “e” and “r”; the second contains the font 10-point Helvetica® Bold for another two glyphs, “g” and “o”. The style object associated with this shape—that is, the object referenced in the style property of the shape—may contain completely different properties. However, because there are style runs and a style list in the geometry of this glyph shape, QuickDraw GX ignores the information in the “regular” associated style object—the one referenced in the style property of the shape object.

Figure 4-7 The effect of style runs on the appearance of glyphs in a glyph shape

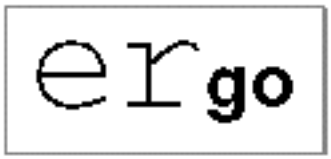


Figure 4-8 shows a glyph shape in which each glyph has its own style. You cannot get the same result using multiple glyph shapes, because the position of each glyph would differ depending on the device characteristics (which affect such values as the advance widths). Each glyph can’t be positioned individually to get the same results over many display devices, because device resolutions may vary.

Figure 4-8 An example of a glyph shape with a style run for each glyph

You can use glyph shapes for clipping, dashing, and patterns only if they are in primitive form. (For more information about primitive forms, see the chapter “Typographic Shapes” in this book.) Remember, however, that any shape you want to convert to primitive form cannot have any styles, whether in the style object or in the style list, that contain caps, dashes, patterns, joins, font variations, or text faces.

To pattern a glyph shape, for example, you must include, in the glyph shape’s style, a text face that is patterned, or that includes joins, start and end caps, dashing, and so on. (Text faces are described in the chapter “Typographic Styles.”)

You can convert a glyph shape into a text shape, a layout shape, or any other type of shape by using the `GXSetShapeType` function. For more information about `GXSetShapeType`, see the “Shape Objects” chapter of *Inside Macintosh: QuickDraw GX Objects*.

The Default Glyph Shape

The default glyph shape has no text and no starting position (0,0); all of its components are either zero or `nil`. Like the default text and layout shapes, the default glyph shape has a `gxWindingFill` type.

The default glyph shape has a style and that style is described in the "Typographic Styles" chapter in this book. The default settings for all typographic shapes are described in the chapter "Typographic Shapes" in this book.

Using Glyph Shapes

This section describes how to perform typical operations with the glyph shape. It tells how to

- create and draw a glyph shape
- get information from a glyph shape
- change parts of a glyph shape by modifying the text in the shape and replacing styles and style runs
- manipulate the positions and advance bits arrays to change the position of the shape in the view port
- change the tangent array

Creating and Drawing a Glyph Shape

To create a glyph shape, you use the `GXNewGlyphs` function (described on page 4-22). You can also use the `GXNewShape` function, which is described in the "Shape Objects" chapter of *Inside Macintosh: QuickDraw GX Objects.*

If you create a glyph shape, the shape has one style run (because there is no style list), one absolute position (the starting position of the first glyph of the shape), and the same tangent for each of the glyphs in the shape.

If you want to change the styles in the shape or add new styles to the existing ones, you must determine how many glyphs are in each style run. Listing 4-1 shows how to use the `GXNewGlyphs` function to create a glyph shape with five glyphs in it, with the first two in 30-point New York and the other three in 60-point Geneva.

The code in Listing 4-1 creates the two styles and uses the `GXSetStyleTextSize` and `GXSetStyleFont` functions and the `GXFindFonts` library function to set up each style. It then uses the `GXNewGlyphs` function to create the glyph shape, and positions the shape using the `GXMoveShapeTo` function. (You can also set the position of the shape in the positions array by using `GXSetGlyphPositions` function, described on page 4-33.) It then uses `GXDrawShape` to draw the glyph shape and `GXDisposeShape` to dispose of the shape.

Listing 4-1 Creating a glyph shape with style runs

```
gxShape  myShape;
gxStyle  myStyles[2];
gxFont   newYorkFont = 0, genevaFont = 0;
static const unsigned char myString[] = "glyph";
short    myStyleRuns[2];

/* create two styles */
myStyles[0] = GXNewStyle();
myStyles[1] = GXNewStyle();

/* set up first style as 30-pt.New York*/
GXSetStyleTextSize(myStyles[0], ff(30));
GXFindFonts(0,gxFullFontName,gxMacintoshPlatform,gxRomanScript,
gxEnglishLanguage, 8, “New York”, 1, 1, &newYorkFont);
GXSetStyleFont(myStyles[0], newYorkFont);
myStyleRuns[0] = 2; //length of text in style

/* set up second style as 60-pt.Geneva */
GXFindFonts(0,gxFullFontName,gxMacintoshPlatform,gxRomanScript,
gxEnglishLanguage, 6, “Geneva”, 1, 1, &genevaFont);
GXSetStyleFont(myStyles[1], genevaFont);
GXSetStyleTextSize(myStyles[1], ff(60));
myStyleRuns[1] = 3; //length of text in style

/* create glyph shape using myString and the new styles */
myShape = GXNewGlyphs(sizeof(myString) -1, myString,
                    nil, nil, nil, myStyleRuns, myStyles);

/* move the shape to (50,125)*/
GXMoveShapeTo(myShape, ff(50), ff(125));

GXDrawShape(myShape);

GXDisposeStyle(myStyles[0]);
GXDisposeStyle(myStyles[1]);
GXDisposeShape(myShape);
```

Listing 4-1 produces the output shown in Figure 4-9.

Figure 4-9 A glyph shape with two styles

For more information about `GXSetStyleTextSize` and `GXSetStyleFont` see the chapter "Typographic Styles" in this book. For more information about `GXMoveShapeTo`, `GXDrawShape`, and `GXDisposeShape`, see *Inside Macintosh: QuickDraw GX Objects*.

There are two ways to draw a glyph shape or its equivalent. You can draw an existing glyph shape using the `GXDrawShape` function, described in *Inside Macintosh: QuickDraw GX Objects*. You can also use the `GXDrawGlyphs` function, described on page 4-24, which allows you to draw a glyph shape without first creating the shape.

Getting Information From a Glyph Shape

You can retrieve the information that is in the geometry of a glyph shape—the character or glyph codes, positions, advance bits, tangents, style runs, and style lists—using `GXGetGlyphs`.

Listing 4-2 shows one way of allocating the required space and retrieving all of the glyph shape's data. You call `GXGetGlyphs` twice—once to find out the number of elements in the shape, and once to retrieve the information.The code first calls `GXGetGlyphs` to retrieve the shape's byte count, character count, and style-run count. The code must then assign enough space to the variables to hold that data. The code calls `GXGetGlyphs` again to retrieve the information from the glyph shape.

Listing 4-2 Getting all of the information from a glyph shape

```
long           myByteCount, myCharCount, myRunCount;
unsigned char  *myText;
gxPoint        *myPositions, *myTangents;
long           *myAdvanceBits;
short          *myStyleRunLengths;
gxStyle        *myStyleRunStyles;

/* get the byte count, character count, and style run count */
myByteCount = GXGetGlyphs(myGlyphsShape, &myCharCount, nil, nil,
                          nil, &myRunCount, nil, nil);
```

```
/* calculate space needed, and assign it to variables */
myText = (unsigned char *) NewPtr(myByteCount);
myPositions = (gxPoint *) NewPtr(myCharCount * sizeof(gxPoint));
myAdvanceBits = (long *) NewPtr(((myCharCount + 31) / 32) *
                                sizeof(long);
myTangents = (gxPoint *) NewPtr(myCharCount * sizeof(gxPoint));
myStyleRunLengths = (short *) NewPtr(myRunCount * sizeof(short));
myStyleRunStyles = (gxStyle *) NewPtr(myRunCount *
                                      sizeof(gxStyle));

/* call GXGetGlyphs again to retrieve the information */
GXGetGlyphs(myGlyphsShape, nil, myText, myPositions,
            myAdvanceBits, myTangents, nil, myStyleRunLengths,
            myStyleRunStyles);
```

You can use the `GXGetGlyph`Parts function to retrieve specified glyphs from the source shape. The `GXGetGlyphs` function is described on page 4-26. The `GXGetGlyphParts` function is described on page 4-29.

Changing Parts of a Glyph Shape

You can change any of the information in the geometry of a glyph shape—the text, the style runs, the positions array, the advance bits array, or the tangents array—using the `GXSetGlyphs` or `GXSetGlyphParts` function. The difference between these two functions is that the `GXSetGlyphs` function replaces an entire element of the glyph shape's geometry (for example, the entire positions array), whereas the `GXSetGlyphParts` function allows you to replace a portion of an element (for example, to add some new positions to an existing positions array without changing any of the original data).

To change all or part of the positions array and advance bits array, use the `GXSetGlyphPositions` function, although you must be sure to correlate the data in the new positions array with the values in the advance bits array. The `GXSetGlyphPositions` function is described on page 4-33.

To change the tangents array only, use the `GXSetGlyphTangents` function, which is described on page 4-35.

In each of these functions, if you do not want to change the values for an array, set its parameter to `nil`. If you want to explicitly delete the values for one array and reset its values to the default settings, you must use the constant `gxSetToNil` for that array's parameter.

Changing Text in a Glyph Shape

You can change text in a glyph shape using the `GXSetGlyphs` or `GXSetGlyphParts` function. Use the former if you want to replace all of the text in the shape. (For more information about `GXSetGlyphs`, see page 4-27.)

However, for most editing operations you may want to perform on the shape—whether adding text, deleting text, or replacing text—you should use the `GXSetGlyphParts` function, which is described on page 4-30. Table 4-1 lists some of the parameter settings for changing text in a glyph shape using this function.

Table 4-1 Changing text in a glyph shape using the `GXSetGlyphParts` function

Action	Index	Old character count	New character count	Text
Inserting new text	Glyph index at which new text should start or `gxSelectToEndif` you want to insert at the end	0	Length of the new text	Pointer to the new text
Replacing and deleting some text	Glyph index at which the new text should start	Number of glyphs to delete from the original shape or `gxSelectToEnd`	Length of the new text	Pointer to the new text
Replacing all text in the shape	1	`gxSelectToEnd`	Length of the new text	Pointer to the new text
Deleting all text from the shape	1	`gxSelectToEnd`	0	`nil`

Listing 4-3 shows how to create a glyph shape that reads “Shape”. It then uses the `GXSetGlyphParts` function to insert new text, “hips”, into the existing shape, changing the shape so that it reads “Shipshape”.

Listing 4-3 Inserting text into an existing glyph shape

```
char     *myString = "Shape";
char     *newString = "hips";
short    textLength;
gxShape  myShape;

textLength = strlen(myString);
myShape = GXNewGlyphs(textLength, (unsigned char *)myString,
                      nil, nil, nil, nil, nil);
GXMoveShapeTo(myShape, ff(50), ff(50));
GXDrawShape(myShape);


textLength = strlen(newString);
```

```
GXSetGlyphParts(myShape, 2, 0, textLength,
              (unsigned char *)newString,
              nil, nil, nil, nil, nil);

GXMoveShapeTo(myShape, ff(50), ff(100));
GXDrawShape(myShape);
GXDisposeShape(myShape);
```

Remember that if you add text to a shape in which you have style runs, you must update the runs to reflect the new number of glyphs in the shape.

Changing the Style List and Style Runs Array

When you first create a glyph shape, QuickDraw GX gives it a single style run and the default glyph shape's style. However, you can change the styles of glyphs in the shape when you create the shape, or you can use the `GXSetGlyphParts` function.

Keep in mind that when you change the style list in a glyph shape, you must always pass appropriate values for the style runs as well, even if the style runs are not changing. Likewise, you must always pass values for the style list when you want to change the style runs.

For example, Listing 4-4 shows code that swaps, but doesn't actually change, two typefaces. It initially creates the shape "Birdy" so that the first two glyphs have the New York font and second three glyphs have the Geneva font, and it later reverses the fonts.

The code in Listing 4-4 first creates references to the typefaces New York and Geneva. It uses the `GXNewStyle` function to create the two styles and uses `GXSetStyleTextSize` to set their text sizes, which will not change. The code then uses `GXSetStyleFont` to set the first two glyphs to New York and the remaining three to Geneva, and creates the glyph shape using `GXNewGlyphs`.

The next series of calls swaps the style runs. Note that in the call to `GXSetGlyphParts`, you must reassign the style runs you assigned with `GXNewGlyphs`, even though the style runs are not changing.

Listing 4-4 Changing the style runs of a glyph shape

```
gxShape  myShape;
char     *myString = "Birdy";
short    myStyleRuns[] = {2,3};
gxStyle  myStyles[2];
short    textLength;
gxFont   newYorkFont, genevaFont;

/* create references to typefaces; creates styles and text sizes*/
newYorkFont = FindPNameFont(gxFullFontName, "\pNew York");
genevaFont = FindPNameFont(gxFullFontName, "\pGeneva");
myStyles[0] = GXNewStyle();
```

```
myStyles[1] = GXNewStyle();
GXSetStyleTextSize(myStyles[0], ff(30));
GXSetStyleTextSize(myStyles[1], ff(60));

/* set styles for original shape */
GXSetStyleFont(myStyles[0], newYorkFont);
GXSetStyleFont(myStyles[1], genevaFont);
textLength = strlen(myString);
myShape = GXNewGlyphs(textLength, (unsigned char *)myString,
                    nil, nil, nil, myStyleRuns, myStyles);

/* replace the original style runs with the new ones */
GXSetStyleFont(myStyles[0], genevaFont);
GXSetStyleFont(myStyles[1], newYorkFont);
GXSetGlyphParts(myShape, 1, gxSelectToEnd, textLength,
              nil, nil, nil, nil, myStyleRuns, myStyles);

GXMoveShapeTo(myShape, ff(50), ff(50));
GXDrawShape(myShape);

GXDisposeShape(myShape);
GXDisposeStyle(myStyles[0]);
GXDisposeStyle(myStyles[1]);
```

The `GXSetGlyphParts` function is described on page 4-30; the `GXNewGlyphs` function is described on page 4-22. For more information about the functions `GXSetStyleTextSize` and `GXSetStyleFont`, see the chapter “Typographic Styles” in this book.

Positioning a Glyph Shape

If you want to position the entire glyph shape to start at a particular QuickDraw GX coordinate as the glyph origin for the first glyph in the shape, you can use the `GXMoveShapeTo` function. This function changes the value in the positions array for the first glyph. For example, if the `gxMapTransform` shape attribute is clear, this call to `GXMoveShapeTo` changes the first glyph’s position in the positions array to (30.0,40.0):

```
GXMoveShapeTo(myGlyphsShape, ff(30), ff(40));
```

However, you can explicitly set the positions using the positions array, together with the advance bits array, to position the glyphs of a glyph shape. The positions in the positions array can be absolute or relative, depending on the corresponding bit in the advance bits array for that glyph. A value of 1 indicates that the position is absolute in the geometry; a

value of 0 indicates that the position is relative to the end of the advance width of the previous glyph. There are some other combinations of positions and advance bits arrays to keep in mind:

- If you don't specify the positions or advance bits, QuickDraw GX sets one absolute position and the advance bits array to 0x80000000, which indicates that that first position is absolute.
- If you only specify the positions and do not specify advance bits, QuickDraw GX sets the advance bits array to 0xFFFFFFFF—that is, it sets each of those positions as absolute positions for the shape.
- You cannot set the advance bits to 0x00000000; the first glyph advance must always be absolute.

To place each glyph of the glyph shape at a different location, use the `GXSetGlyphPositions` function to set the positions array of an existing shape, as demonstrated in Listing 4-5. (You can also use the `GXSetGlyphParts` function, a general-purpose function that allows you to change many of the arrays in a glyph shape at the same time. This function is described on page 4-30.)

The listing uses `GXNewGlyphs` to create a basic glyph shape, which includes the default positions array. It then changes the positions and advance bits arrays of the shape using the `GXSetGlyphPositions` function.

Listing 4-5 Setting the positions and advance bits arrays of a glyph shape

```
gxShape  myShape;
gxPoint  positions[] = { {ff(100), ff(100)}, {ff(50), ff(25)},
                         {ff(50), ff(25)}, {ff(50), ff(25)},
                         {ff(50), ff(25)}  };
long     advanceBits[] = {0x80000000};
char     *myString = "Birdy";
short    textLength;

/* create glyph shape, which includes default positions array */
textLength = strlen(myString);
myShape = GXNewGlyphs(textLength, (unsigned char *) myString,
                      nil, nil, nil, nil, nil);
GXSetShapeTextSize(myShape, ff(20));

/* change the shape's positions and advance bits arrays */
GXSetGlyphPositions(myShape, 1, 5, advanceBits, positions);
```

This result of this code is shown in Figure 4-10.

Figure 4-10 A glyph shape with positions and advance bits arrays set

The `GXSetGlyphPositions` function is described on page 4-33; the `GXNewGlyphs` function is described on page 4-22. For more information about `GXSetShapeTextSize`, see the chapter “Typographic Styles” in this book.

Setting the Tangents Arrays

If you want to change the tangents of an existing shape, use the `GXSetGlyphTangents` function, which allows you to replace a single tangent in an existing shape. (You can also use the `GXSetGlyphParts` function, a general-purpose function that is useful if you are replacing much of the information in a glyph shape. This function is described on page 4-30.) For example, the following code draws the third and fifth glyphs of a five-glyph shape at 45-degree angles. (Note that 0.707 represents the square root of 2 divided by 2.)

```
const gxPoint my45DegreeTangent[] = {{fl(0.707), fl(0.707)}};

GXSetGlyphTangents(myShape, 3, 1, my45DegreeTangent);
GXSetGlyphTangents(myShape, 5, 1, my45DegreeTangent);
```

Figure 4-11 shows what happens when this code is applied to an existing shape.

Figure 4-11 A glyph shape with 45-degree angle tangents

Of course, because a tangent represents both angle and scale, you can change both values at the same time. (Note that this example had to explicitly design its tangent so that scaling did not occur.) In Listing 4-6, the main function, `GlyphDemo`, uses the subroutine `CreateGlyphTangents` to create a series of tangents that apply varying scales to the glyphs of a glyph shape, a series of tangents that apply varying angles to the glyphs, and then a series of tangents that combine the scales and angles.

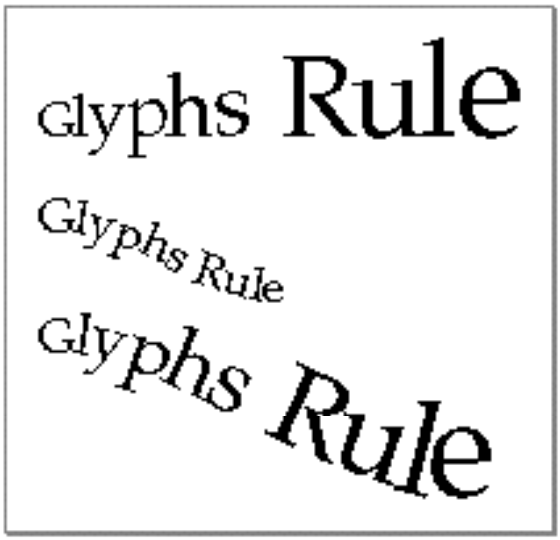
Listing 4-6 Creating a series of tangents with varying angles and scales

```
#include “math routines.h”

static void CreateGlyphTangents(long tangentCount,
                                const Fixed angles[],
                                const Fixed scales[],
                                gxPoint *createdTangents)
{
/* The subroutine uses the angleWalker and scaleWalker variables
for stepping through the various values for angles and scales sent
by the main function. The tangentWalker variable stores the
tangents created as a result of the values given for the angles
and scales.
*/
   register const Fixed *angleWalker = angles;
   register const Fixed *scaleWalker = scales;
   register gxPoint *tangentWalker = createdTangents;
   gxPolar angleScale = {fixed1, 0};

/* If there are angles provided, use the next one in the series.
If there are scales provided, use the next one in the series.
*/
   while (--tangentCount >= 0) {
      if (angleWalker)
         angleScale.angle = *angleWalker++;
      if (scaleWalker)
         angleScale.radius = *scaleWalker++;
      PolarToPoint(&angleScale, tangentWalker++);
   }
}
/* The routine calculates the x and y values of the tangent by
multiplying the scale by the cosine and sine, respectively, of the
angle.
*/

      tangentWalker->x = FractMultiply(x, cos);
      tangentWalker->y = FractMultiply(x, sin);
```

```
      ++tangentWalker;
      --tangentCount;
   }
}

static void GlyphDemo(void)
{
   Fixed angles[11] = { 0, ff(10), ff(15), ff(20), ff(25),
                        ff(30), ff(25), ff(20), ff(15), ff(10),
                        0 };
   Fixed scales[11] = { fl(1.0), fl(1.2), fl(1.4), fl(1.6),
                        fl(1.8), fl(2.0), fl(2.2), fl(2.4),
                        fl(2.6), fl(2.8), fl(3.0) };
   gxPoint tangents[11];
   gxShape myShape;

/* create tangents that scale each glyph of the shape
differently but do not affect the angle.
*/
   CreateGlyphTangents(11, nil, scales, tangents);
   myShape = GXNewGlyphs(11, (unsigned char *)"Glyphs Rule", nil,
                        nil, tangents, nil, nil);
   GXSetShapeTextSize(myShape, ff(24));
   GXMoveShape(myShape, ff(10), ff(30));
   GXDrawShape(myShape);

/* create tangents that draw each glyph of the shape at
different angles but do not affect the scale or the size of the
glyphs.
*/
   CreateGlyphTangents(11, angles, nil, tangents);
   GXSetGlyphs(myShape, 0, nil, nil, nil, tangents, nil, nil);
   GXMoveShape(myShape, 0, ff(120));
   GXDrawShape(myShape);

/* combine the previous two sections: it changes the angle and
scale of each glyph in the shape. */
   CreateGlyphTangents(11, angles, scales, tangents);
   GXSetGlyphs(myShape, 0, nil, nil, nil, tangents, nil, nil);
   GXMoveShape(myShape, 0, ff(150));
   GXDrawShape(myShape);
   GXDisposeShape(myShape);
}
```

The results of Listing 4-6 are shown in Figure 4-12.

Figure 4-12 Varying the angle and scale of individual glyphs using tangents



The `GXSetGlyphTangents` function is described on page 4-35.

If you are trying to calculate the proper coordinates for a tangent that must approximate a particular angle, you can use the `PolarToPoint` function. You can use the `MapPoints` function to take a single mapping that has the rotation and scaling you desire, plus an array of tangents for your shape, and produce an array of tangents that specify that mapping and rotation.

The `PolarToPoint` and `MapPoints` functions are described the "QuickDraw GX Mathematics" chapter in *Inside Macintosh: QuickDraw GX Environment and Utilities.*

Glyph Shapes Reference

Functions

This section describes the functions you use for creating, changing, or setting parts of QuickDraw GX glyph shapes.

Creating and Drawing Glyph Shapes

The GXNewGlyphs function creates a new glyph shape. You can also use the GXNewShape function to create a glyph shape that has the default style, ink, and transform objects. GXNewShape is described in *Inside Macintosh: QuickDraw GX Objects*.

The GXDrawGlyphs function draws a string of glyphs in the view port without actually creating a glyph shape. To draw an existing glyph shape, you use the GXDrawShape, which is described in *Inside Macintosh: QuickDraw GX Objects*.

GXNewGlyphs

You can use the GXNewGlyphs function to create a new glyph shape.

```
gxShape GXNewGlyphs(long charCount, const unsigned char text[],
                    const gxPoint positions[],
                    const long advance[],
                    const gxPoint tangents[],
                    const short styleRuns[],
                    const gxStyle glyphStyles[]);
```

`charCount` The number of character codes in the `text` parameter. The value of `charCount` must be greater than or equal to 0; if it is not 0, the `text` parameter must contain an array of character codes. For non-Roman scripts, the byte length may be up to twice the number of character codes.

`text` An array of character codes or glyph codes. Whether each character code is 8-bit or 16-bit depends on both the platform in the glyph shape's style object and on the actual byte values. If the value of the glyph attribute `gxIgnorePlatformShape` is clear, then it returns an array of character codes. If it is set, it is returns an array of 16-bit glyph codes.

`positions` An array of points, one for each character code in the `text` parameter. Depending on the corresponding values in the advance bits array, these points are either the relative or absolute positions of the glyphs in geometry space. If you pass `nil` for this parameter, GXNewGlyphs places the first glyph at (0.0,0.0).

`advance` An array of bits, one bit for each character code in the `text` parameter. The high bit in the long integer corresponds to the first character code in the shape. If the bit is set, then the corresponding position for that glyph in the `positions` parameter is an absolute position in geometry space. If the bit is not set, the position for that glyph is relative to the end of the advance width of the previous glyph. If you pass `nil` for both this parameter and the `positions` parameter, GXNewGlyphs positions each glyph relative according to the advance width of the preceding glyph. If you pass `nil` for this parameter but not for the `positions` parameter, GXNewGlyphs gives each glyph an absolute position from the positions array.

`tangents` An array of points that serve as tangents. This array determines the scaling or rotation of each glyph. If you pass `nil` for this parameter, `GXNewGlyphs` does not apply optional scaling or rotation to each character.

`styleRuns` An array of values that specify how many consecutive glyphs use the same style; the styles are stored in the `glyphStyle` list. The number of entries in the `styleRuns` array must match the number of entries in the `glyphStyles` parameter; the sum of the entries in this array should equal the value in the `charCount` parameter, so that every glyph is part of a style run. The value of each entry must be greater than or equal to 1. If this parameter is `nil`, `GXNewGlyphs` uses the default style object for all glyphs in the shape.

`glyphStyles`
An array of references to style objects. This array should be present only if the style-runs array is present. If you pass `nil` for this parameter, `GXNewGlyphs` assigns the style object associated with the glyph shape object to all glyphs in the shape. If an entry in the `glyphStyles` array is `nil`, that entry refers to the style object associated with the shape.

function result A reference to the new glyph shape.

DESCRIPTION

The `GXNewGlyphs` function creates a copy of the default glyph shape, sets the owner count of the copy to 1, initializes its geometry with the values in the function's parameters, and returns a reference to the glyph shape as the function result.

The new glyph shape returned by this function contains references to the same style, ink, and transform objects as the default glyph shape.

Most of parameters to `GXNewGlyphs`—`positions`, `advance`, `tangents`, `styleRuns`, and `glyphStyles`—are optional; if you set them to `nil`, QuickDraw GX sets their values to those of the default glyph shape.

The `charCount` parameter indicates the number of character codes present, which may not necessarily equal the number of bytes in the length of the `text` parameter. The interpretation of characters depends on the glyph shape's style attribute.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`	
`parameter_is_nil`	(debugging version)
`missing_parameter`	(debugging version)
`inconsistent_parameters`	(debugging version)
`count_is_less_than_zero`	(debugging version)
`parameter_out_of_range`	(debugging version)

SEE ALSO

You can also use the `GXNewShape` function, described in the chapter "Shape Objects" in *Inside Macintosh: QuickDraw GX Objects*, to create a new glyph shape. The shape type of the glyph shape is `gxGlyphType`. The default glyph shape is described on page 4-10.

To create a new text shape, use the `GXNewText` function, described in the chapter "Text Shapes" in this book.

To create a new layout shape, see the description of the `GXNewLayout` function in the chapter "Layout Shapes" in this book.

For information about the positions and advance bits arrays, see "The Positions and Advance Bits Arrays" on page 4-5. For information about the tangents array, see "The Tangents Array" on page 4-6.

GXDrawGlyphs

You can use the `GXDrawGlyphs` function to draw a string of glyphs without first creating a glyph shape.

```
void GXDrawGlyphs(long charCount, const unsigned char text[],
                  const gxPoint positions[], const long advance[],
                  const gxPoint tangents[], const short styleRuns[],
                  const gxStyle glyphStyles[]);
```

`charCount` The number of character codes in the `text` parameter. For non-Roman scripts, the byte length may be up to twice the number of character codes. This value must be greater than 0, and the `text` parameter must point at an array of character codes. (If the value of this parameter is 0, the `GXDrawGlyphs` function does nothing.)

`text` An array of character codes.

`positions` An array of points, one for each character code in the `text` parameter. Depending on the corresponding values in the advance bits array, these points are either the relative or absolute positions of the glyphs in geometry space. If you pass `nil` for this parameter, `GXDrawGlyphs` places the first glyph at (0.0,0.0).

`advance` An array of bits, one bit for each character code in the `text` parameter. The high bit in the long integer corresponds to the first character code in the shape. If the bit is set, then the corresponding position for that glyph in the `positions` parameter is an absolute position in geometry space. If the bit is not set, the position for that glyph is relative to the end of the advance width of the previous glyph. If you pass `nil` for both this parameter and the `positions` parameter, `GXDrawGlyphs` positions each glyph at the end of the advance width of the preceding glyph. If you pass `nil` for this parameter but not for the `positions` parameter, `GXDrawGlyphs` gives each glyph an absolute position from the positions array.

`tangents` An array of tangents that determines the scaling or rotation of each glyph. If you pass `nil`, `GXDrawGlyphs` does not apply optional scaling or rotation to each character.

`styleRuns` An array of values that specify how many consecutive glyphs use the same style; the styles are stored in the style list. The number of entries in the style-runs array must match the number of entries in the `glyphStyles` parameter; the sum of the entries in the style-runs array should equal the value in the `charCount` parameter, so that every glyph is part of a style run. The value of each entry must be greater than or equal to 1. If you pass `nil` for this parameter, `GXDrawGlyphs` uses the default style object when drawing the glyphs.

`glyphStyles`
An array of references to style objects. This array should be present only if the style-runs array is present. If you pass `nil`, `GXDrawGlyphs` assigns the glyphs the default style.

DESCRIPTION

The `GXDrawGlyphs` function draws the text using the default transform and ink objects. If the `glyphStyles` parameter contains no values, `GXDrawGlyphs` also uses the default style object. The function does not create a new glyph shape that you can subsequently use.

The parameter `charCount` indicates the number of characters present, which may not necessarily equal the number of bytes in the length of the `text` parameter. The interpretation of characters depends on the glyph shape's style attribute.

ERRORS, WARNINGS, AND NOTICES

Errors

`parameter_is_nil`	(debugging version)
`out_of_memory`	

SEE ALSO

To draw an existing glyph shape, use the `GXDrawShape` function, described in the "Shape Objects" chapter in *Inside Macintosh: QuickDraw GX Objects*.

To draw simple text that shares the same font and other font characteristics, see the description of the `GXDrawText` function in the chapter "Text Shapes" in this book.

To draw a layout shape, see the description of the `GXDrawLayout` function in the chapter "Layout Shapes" in this book.

Getting and Setting the Properties of Glyph Shapes

You can get the settings of the various arrays of glyph shapes—the positions, advance bits, tangents, style-runs arrays, and the styles—with the `GXGetGlyphs` function. You can change any type of shape into a glyph shape with the `GXSetGlyphs` function.

The GXGetGlyphParts function allows you to examine parts of a glyph shape, such as some of the character codes stored in the shape or a portion of the positions array. The GXSetGlyphParts function allows you to change parts of a glyph shape.

The GXGetGlyphPositions function allows you to examine the positions and advance bits arrays of a glyph shape. The GXSetGlyphPositions function lets you change the values of the positions and advance bits arrays.

The GXGetGlyphTangents function allows you to examine the tangents array of a glyph shape. The GXSetGlyphTangents function lets you change the values of the tangents array.

GXGetGlyphs

You can use the GXGetGlyphs function to retrieve the information in the geometry of a glyph shape.

```
long GXGetGlyphs(gxShape source, long *charCount,
                 unsigned char text[], gxPoint positions[],
                 long advance[], gxPoint tangents[],
                 long *runCount, short styleRuns[],
                 gxStyle glyphStyles[]);
```

`source` A reference to the glyph shape.

`charCount` A pointer to a `long` value. On return, the value is the number of character codes in the `text` parameter. For non-Roman scripts, the byte length may be up to twice the number of character codes.

`text` An array of character codes or glyph codes. On return, the array contains the character codes of the source glyph shape.

`positions` An array of points. On return, the array contains the positions array of the source glyph shape.

`advance` An array of bits. On return, the array contains the advance bits array of the source glyph shape.

`tangents` An array of points that serve as tangents. On return, the array contains the tangents array of the source glyph shape.

`runCount` A pointer to a `long` value. On return, the value is the number of style runs.

`styleRuns` An array of `short` values. On return, each element specifies how many consecutive glyphs use the same style (the style runs array).

`glyphStyles` An array of references to style objects. On return, the array contains the style list for the source glyph shape.

function result The number of bytes needed to store the character codes of the glyph shape.

DESCRIPTION

The `GXGetGlyphs` function gets information from a glyph shape: the character or glyph codes, positions, advance bits, tangents, style runs, and style lists. The parameters, except for the shape reference, are pointers to storage for the other information, such as the number of characters (which may not be equal to the number of bytes, depending on the platform) or the positions array of the shape.

You must allocate the required space for the `charCount`, `text`, `positions`, `advance`, `tangents`, `styleRuns`, and `glyphStyles` parameters.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`
`illegal_type_for_shape` (debugging version)

Notices (debugging version)

`glyph_tangents_have_no_effect`
`glyph_positions_determined_by_advance`

SEE ALSO

To retrieve the description of a text shape, use the `GXGetText` function, described in the chapter “Text Shapes” in this book.

GXSetGlyphs

You can use the `GXSetGlyphs` function to change the information in a glyph shape.

```
void GXSetGlyphs(gxShape target, long charCount,
                 const unsigned char text[],
                 const gxPoint positions[], const long advance[],
                 const gxPoint tangents[], const short styleRuns[],
                 const gxStyle glyphStyles[]);
```

`target` A reference to the shape whose arrays you want to change.

`charCount` The number of character codes in the `text` parameter. For non-Roman scripts, the byte length may be up to twice the number of character codes. This value must be greater than or equal to 0; if it is not 0, the `text` parameter must point to an array of character codes.

`text` An array of character codes or glyph codes. Whether each character code is 8-bit or 16-bit depends on both the platform in the glyph shape’s style object and on the actual byte values. If the value of the shape attribute `gxIgnorePlatformShape` is clear, then it returns an array of character codes. If it is set, it is returns an array of 16-bit glyph codes.

`positions` An array of points, one for each character code in the `text` parameter. Depending on the corresponding values in the advance bits array, these points are either the relative or absolute positions of the glyphs in geometry space. If you pass `nil` for this parameter, `GXSetGlyphs` places the first glyph at (0.0,0.0).

`advance` An array of bits, one bit for each character code in the `text` parameter. The high bit in the long integer corresponds to the first character code in the shape. If the bit is set, then the corresponding position for that glyph in the positions array is an absolute position in geometry space. If the bit is not set, the position for that glyph is relative to the end of the advance width of the previous glyph. If you pass `nil` for both this parameter and the `positions` parameter, `GXSetGlyphs` positions each glyph relative to the advance width of the preceding glyph. If you pass `nil` for this parameter but not for the `positions` parameter, `GXSetGlyphs` gives each glyph an absolute position from the positions array.

`tangents` An array of points that serve as tangents. This array determines the scaling or rotation of each glyph. If you pass `nil`, `GXSetGlyphs` does not apply optional scaling or rotation to each character.

`styleRuns` An array of values that specify how many consecutive glyphs use the same style; the styles are stored in the style list. The sum of the entries in the style runs array should equal the value in the `charCount` parameter. The value of each entry must be greater than or equal to 1. If you pass `nil` for this parameter, `GXSetGlyphs` uses the default style object for all glyphs in the shape.

`glyphStyles`
An array of references to style objects. This array should be present only if the value of the `styleRuns` parameter is not `nil`. If you pass `nil` for `glyphStyles`, `GXSetGlyphs` assigns the style object associated with the shape object to all glyphs in the shape. If an entry in the style list is `nil`, that entry refers to the style object associated with the shape.

DESCRIPTION

The `GXSetGlyphs` function changes the information in a glyph shape. If you use `GXSetGlyphs` to change one of the elements of one of the arrays in a glyph shape—for example, a point in the positions array—you must send the entire array for the shape.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`	
`shape_is_nil`	
`parameter_is_nil`	(debugging version)
`missing_parameter`	(debugging version)
`inconsistent_parameters`	(debugging version)
`cannot_add_unspecified_new_glyphs`	(debugging version)
`style_run_array_does_not_match_number_of_characters`	(debugging version)

Warnings

`shape_access_not_allowed`	(debugging version)
`first_glyph_advance_must_be_absolute`	(debugging version)

SEE ALSO

To change a portion of an array in a glyph shape, use the `GXSetGlyphParts` function, described on page 4-30.

GXGetGlyphParts

You can use the `GXGetGlyphParts` function to retrieve parts of a glyph shape.

```
long GXGetGlyphParts(gxShape source, long index, long charCount,
                     long *byteLength, unsigned char text[],
                     gxPoint positions[], long advanceBits[],
                     gxPoint tangents[], long *runCount,
                     short styleRuns[], gxStyle styles[]);
```

`source` A reference to the shape whose information you want to retrieve.

`index` The number of the first character you want to retrieve.

`charCount` The number of character codes you want to retrieve. The number of actual glyphs returned may be different from this value. (The function itself returns the number of glyphs retrieved.)

`byteLength` A pointer to a `long` value. On return, the value is the number of bytes used by the character codes.

`text` An array of character codes. On return, the array contains the character codes from the source shape.

`positions` An array of `gxPoint` values. On return, the array contains the positions of each of the glyphs.

`advanceBits` An array of `long` values. On return, the array contains the advance bits for the glyph shape.

`tangents` An array of `gxPoint` values. On return, the array contains the tangents for glyphs you specify.

`runCount` A pointer to a `long` value. On return, the value is the number of style runs.

`styleRuns` An array of `short` values. On return, the array contains the number of glyphs associated with each style run.

`styles` An array of style-object references. On return, the array contains the styles of the glyphs you specify.

function result The number of glyphs retrieved.

DESCRIPTION

The `GXGetGlyphParts` function retrieves the specified glyphs from the source shape, which must be of type `glyphsType`, and places the character codes into the `text` parameter, the positions into the `positions` parameter, the advance bits into the `advanceBits` parameter, and so on. It also returns the length in bytes of these characters in the `byteLength` parameter and the number of style runs in the `runCount` parameter. You must allocate the memory to store the information the function returns.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`	
`shape_is_nil`	
`index_is_less_than_one`	(debugging version)
`count_is_less_than_one`	(debugging version)
`illegal_type_for_shape`	(debugging version)

Warnings

`index_out_of_range`
`count_out_of_range`

SEE ALSO

To retrieve part of a text shape, use the `GXGetTextParts` function, described in the chapter “Text Shapes” in this book.

GXSetGlyphParts

You can use the `GXSetGlyphParts` function to replace specific parts of a glyph shape with new information.

```
void GXSetGlyphParts(gxShape source, long index,
                     long oldCharCount, long newCharCount,
                     const unsigned char text[],
                     const gxPoint positions[],
                     const long advanceBits[],
                     const gxPoint tangents[],
                     const short styleRuns[],
                     const gxStyle styles[]);
```

`source` A reference to the shape you want to change.

`index` The index number of the first character code that you want to replace in the source shape. This value must be greater than or equal to 1.

`oldCharCount`
: The number of character codes that you want to replace in the source shape. This value may be greater than or equal to 0, or it may be `gxSelectToEnd`, which selects all characters after and including the character code specified by the value of the `index` parameter.

`newCharCount`
: The number of character codes you want to add to the source shape.

`text` An array of the new characters you want to add.

`positions` The new values for the positions array.

`advanceBits`
: The new values for the advance bits array.

`tangents` The new values for the tangents array.

`styleRuns` The new style-runs array.

`styles` The new style list.

DESCRIPTION

The `GXSetGlyphParts` function replaces the specified glyphs in the source shape with the new glyphs specified by the `text`, `positions`, `advanceBits`, `tangents`, `styleRuns`, and `styles` parameters.

If you send new text to the shape, the values in the `positions`, `advanceBits`, `tangents`, `styleRuns`, and `styles` parameters apply only to new text, not to the shape as a whole.

You cannot change the style list and style-runs array independently of each other. If you change one, you must resend the values for the other.

Using the `GXSetGlyphParts` function, you can change the positions and advance bits arrays independently, and you can send part of or all of a positions or advance bits array.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`	
`shape_is_nil`	
`inconsistent_parameters`	(debugging version)
`index_is_less_than_zero`	(debugging version)
`count_is_less_than_zero`	(debugging version)
`illegal_type_for_shape`	(debugging version)
`cannot_add_unspecified_new_glyphs`	(debugging version)
`style_run_array_does_not_match_number_of_characters`	(debugging version)

Warnings

`index_out_of_range`	
`count_out_of_range`	
`shape_access_not_allowed`	(debugging version)
`first_glyph_advance_must_be_absolute`	(debugging version)

SEE ALSO

For more information on how to use the `GXSetGlyphParts` function, see "Changing Parts of a Glyph Shape" beginning on page 4-13.

To change parts of a text shape, use the `GXSetTextParts` function, described in the chapter "Text Shapes" in this book.

GXGetGlyphPositions

You can use the `GXGetGlyphPositions` function to retrieve a number of entries in the positions and advance bits arrays of a glyph shape.

```
long GXGetGlyphPositions(gxShape source, long index, long count,
                         long advance[], gxPoint positions[]);
```

`source` A reference to the glyph shape whose positions and advance bits arrays you want to retrieve.

`index` The first entry in the positions and advance bits arrays you want.

`count` The number of entries in the positions and advance bits arrays to return, starting at the value of the `index` parameter. This value may be greater than or equal to 0. If you pass 0, the function result is a Boolean value indicating whether the glyph shape has any entries set in the advance bits array. A function result of `false` indicates that all the advance bits array values are the default values.

`advance` An array of `long` values. On return, the array contains the requested portion of the advance bits.

`positions` An array of points. On return, the array contains the requested portion of the positions.

function result The number of retrieved entries, if the value of `count` is greater than 0; otherwise, a Boolean value indicating whether the glyph shape has any entries set in the advance bits array.

DESCRIPTION

The `GXGetGlyphPositions` function retrieves the specified number of entries from the positions and advance bits arrays of the source glyph shape.

If the glyph shape's advance bits array is not set, `GXGetGlyphPositions` fills the advance bits array with zeros. If the glyph shape's positions array is empty, the function returns the positions array filled with the coordinate point (0.0,0.0), unless it is also returning the first point in the shape; in this case, the function returns the initial position of the glyph shape and fills the rest of the array with (0.0,0.0).

If you do not want either the advance bits or positions array, send `nil` for that parameter.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`	
`index_is_less_than_one`	(debugging version)
`count_is_less_than_zero`	(debugging version)
`illegal_type_for_shape`	(debugging version)
`out_of_memory`	

Warnings

`index_out_of_range`
`count_out_of_range`

GXSetGlyphPositions

You can use the `GXSetGlyphPositions` function to replace the selected positions and advance bits arrays of a glyph shape with new arrays.

```
void GXSetGlyphPositions(gxShape target, long index, long count,
                         const long advance[],
                         const gxPoint positions[]);
```

`target` A reference to the shape whose advance bits and positions arrays you want to replace.

`index` The first entry in the positions and advance bits arrays you want to replace.

`count` The number of entries to replace. This value may be greater than or equal to 1. It may also be equal to `gxSelectToEnd`, which indicates that `GXSetGlyphPositions` should select all positions and advance bits beginning at the entry indicated by `index`.

`advance` The advance bits array you want to put into the target shape. If you pass `gxSetToNil`, `GXSetGlyphPositions` sets all the bits, except the first one, in this array to 0.

`positions` The positions array you want to put into the target shape. If you pass `gxSetToNil`, `GXSetGlyphPositions` sets all the positions, except the first one, in this array to (0.0,0.0).

DESCRIPTION

The `GXSetGlyphPositions` function replaces the selected sections of the advance bits and positions arrays of the target shape with the arrays in the `advance` and `positions` parameters.

ERRORS, WARNINGS, AND NOTICE

Errors

`out_of_memory`	
`shape_is_nil`	
`parameter_is_nil`	(debugging version)
`index_is_less_than_zero`	(debugging version)
`count_is_less_than_zero`	(debugging version)
`illegal_type_for_shape`	(debugging version)

Warnings

`index_out_of_range`	
`count_out_of_range`	
`shape_access_not_allowed`	
`first_glyph_advance_must_be_absolute`	(debugging version)

Notices (debugging version)

`glyph_positions_are_already_set`

GXGetGlyphTangents

You can use the `GXGetGlyphTangents` function to get some or all of the information about tangents in a glyph shape.

```
long GXGetGlyphTangents(gxShape source, long index, long count,
                        gxPoint tangents[]);
```

`source` A reference to the shape from which you want the tangent information.

`index` The index number of the first tangent to return. This value must be greater than or equal to 1.

`count` The number of tangents to return. This value may be greater than or equal to 0. If you pass 0, the function result changes to a Boolean value indicating whether the glyph shape has any explicit tangents at all.

`tangents` An array of `gxPoint` tangents. On return, the array of tangents.

function result The number of retrieved tangents, or, if the value of `count` is 0, a Boolean value indicating whether the glyph shape has any explicit tangents. A function result of `false` indicates that all the tangents are the default settings.

DESCRIPTION

The `GXGetGlyphTangents` function retrieves the specified number of tangent points from the glyph shape. If the glyph shape does not have tangents, then `GXGetGlyphTangents` returns an array filled with the value (1.0,0.0) in the `tangents` parameter.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`	
`shape_is_nil`	
`index_is_less_than_one`	(debugging version)
`count_is_less_than_one`	(debugging version)
`illegal_type_for_shape`	(debugging version)
`parameter_out_of_range`	(debugging version)

Warnings

`index_out_of_range`
`count_out_of_range`

GXSetGlyphTangents

You can use the `GXSetGlyphTangents` function to replace selected tangents in a glyph shape with new ones.

```
void GXSetGlyphTangents(gxShape target, long index, long count,
                        const gxPoint tangents[]);
```

`target`	A reference to the glyph shape.
`index`	The first tangent you want to replace.
`count`	The number of tangents to replace. This value may be greater than or equal to 1. It may also be equal to `gxSelectToEnd`, which indicates that the function should select all tangents, from the one indicated by `index` to the end of the tangents array.
`tangents`	The array of tangents you want to put into the glyph shape. You can pass `gxSetToNil` for an entry in this array; this value sets the tangents to the default value of (1.0,0.0).

DESCRIPTION

The `GXSetGlyphTangents` function replaces the selected tangents with the new ones you pass in the `tangents` parameter.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`	
`shape_is_nil`	
`parameter_is_nil`	(debugging version)
`index_is_less_than_one`	(debugging version)
`count_is_less_than_zero`	(debugging version)
`illegal_type_for_shape`	(debugging version)
`parameter_out_of_range`	(debugging version)

Warnings
`index_out_of_range`
`count_out_of_range`
`shape_access_not_allowed`

Notices (debugging version)
`glyph_tangents_already_set`

Summary of Glyph Shapes

Functions

Creating and Drawing Glyph Shapes

```
gxShape GXNewGlyphs          (long charCount, const unsigned char text[],
                              const gxPoint positions[],
                              const long advance[],
                              const gxPoint tangents[],
                              const short styleRuns[],
                              const gxStyle glyphStyles[]);
void GXDrawGlyphs            (long charCount, const unsigned char text[],
                              const gxPoint positions[],
                              const long advance[], const gxPoint tangents[],
                              const short styleRuns[],
                              const gxStyle glyphStyles[]);
```

Getting and Setting the Properties of Glyph Shapes

```
long GXGetGlyphs             (gxShape source, long *charCount,
                              unsigned char text[], gxPoint positions[],
                              long advance[], gxPoint tangents[],
                              long *runCount, short styleRuns[],
                              gxStyle glyphStyles[]);
void GXSetGlyphs             (gxShape target, long charCount,
                              const unsigned char text[],
                              const gxPoint positions[],
                              const long advance[],
                              const gxPoint tangents[],
                              const short styleRuns[],
                              const gxStyle glyphStyles[]);
long GXGetGlyphParts         (gxShape source, long index, long charCount,
                              long *byteLength, unsigned char text[],
                              gxPoint positions[], long advanceBits[],
                              gxPoint tangents[], long *runCount,
                              short styleRuns[], gxStyle styles[]);
void GXSetGlyphParts         (gxShape source, long index, long oldCharCount,
                              long newCharCount, const unsigned char text[],
                              const gxPoint positions[],
                              const long advanceBits[],
                              const gxPoint tangents[],
                              const short styleRuns[],
                              const gxStyle styles[]);
```

```
long GXGetGlyphPositions    (gxShape source, long index, long count,
                             long advance[], gxPoint positions[]);
void GXSetGlyphPositions    (gxShape target, long index, long count,
                             const long advance[],
                             const gxPoint positions[]);
long GXGetGlyphTangents     (gxShape source, long index, long count,
                             gxPoint tangents[]);
void GXSetGlyphTangents     (gxShape target, long index, long count,
                             const gxPoint tangents[]);
```

CHAPTER 5

Layout Shapes

Contents

This chapter describes the basic features of the layout shape object and the functions you can use to manipulate them. Read this chapter if you create or use layout shapes in your application.

Before reading this chapter, you should be familiar with the information in the chapters "Introduction to QuickDraw GX Typography" and "Typographic Shapes" in this book. You may also need to refer to the book *Inside Macintosh: QuickDraw GX Objects* before reading this chapter.

This chapter describes the basic elements of a layout shape. For information on creating carets, highlighting, and hit-testing a layout shape, read the chapter "Layout Carets, Highlighting, and Hit-Testing for Layout Shapes." For information about baselines, line measurement, line breaking, and text direction, read the chapter "Layout Line Control." For information on style-object properties used by layout shapes, see the chapter "Layout Styles."

This chapter introduces the concept of a layout shape, describes how it relates to the other typographic shapes, and describes the layout shape's properties. It then shows how to use QuickDraw GX functions to

- create and draw layout shapes
- manipulate the contents of a layout shape
- retrieve glyph information from a layout shape

About Layout Shapes

A **layout shape,** like a text shape or glyph shape, produces a line of text that QuickDraw GX can draw on the screen. Unlike the text and glyph shapes, however, the layout shape deals with text primarily in *typographic* terms ("kern by this amount" or "change the orientation of this text in a vertical line") rather than *graphic* terms ("place this glyph at the following (x,y) location" or "set this glyph's rotation to 12 degrees").

Some of the things your application can do with layout shapes include

- creating contextual forms and ligatures automatically
- manual and automatic kerning, tracking, and letterspacing
- justifying text in sophisticated ways, including the use of language-specific justification, such as Arabic kashidas
- automatically rearranging text for languages such as Arabic, Hebrew, Hindi, and other non-Roman script systems that require rearrangement
- highlighting some or all of the text in the layout, including text in two different scripts, such as Roman and Arabic
- hit-testing within the text
- determining the caret or carets for some location within the text
- supporting your application's line-breaking decisions with fast measurement functions

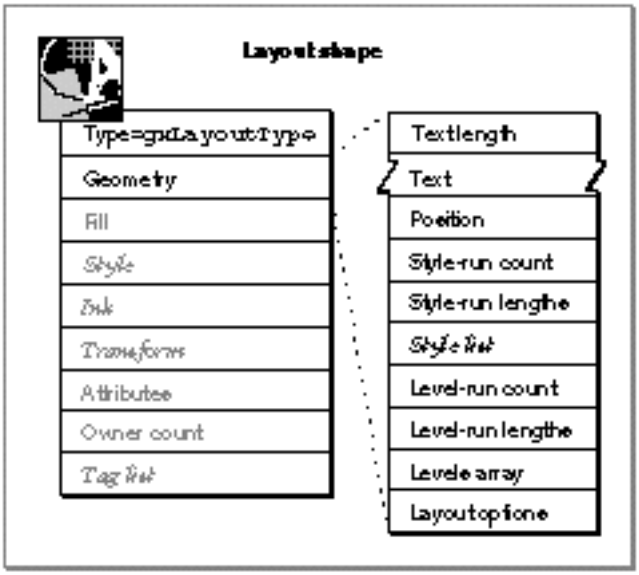
A layout shape generally describes a single line of text. It is not suited for large blocks of text, such as a paragraph. However, you can use QuickDraw GX functions to break up a paragraph into several layouts that your application can work with.

Properties of the Layout Shape

The geometry of a layout shape, shown in Figure 5-1, has three main components: text, style-run information, and direction-level information. Note that, because a layout shape is an object and not a data structure, the order of the properties as shown in Figure 5-1 is completely arbitrary.

Figure 5-1 shows the ten accessible properties of the layout shape object.

Figure 5-1 Geometry of a layout shape



The geometry of a layout shape contains these elements:

- **Text length.** The count of the total number of bytes contained in the text. Note that with layout shapes, you pass in byte counts, unlike with text and glyph shapes.
- **Text.** The text of a layout shape is stored as a single run of character codes, although you can supply pointers to several separate runs when you create or modify a layout.
- **Position**. The starting position of the layout shape in geometry space. This position always marks the left (if horizontal) or top (if vertical) edge of the shape. If the shape is left-aligned, this position corresponds to the intersection of the baseline with the leftmost glyph of the shape.
- **Style-run count.** The number of style runs in the layout shape—that is, text sequences that each share the same font, size, style, and script system.
- **Style-run lengths.** An array that specifies the length, in bytes, of each style run in the layout shape.

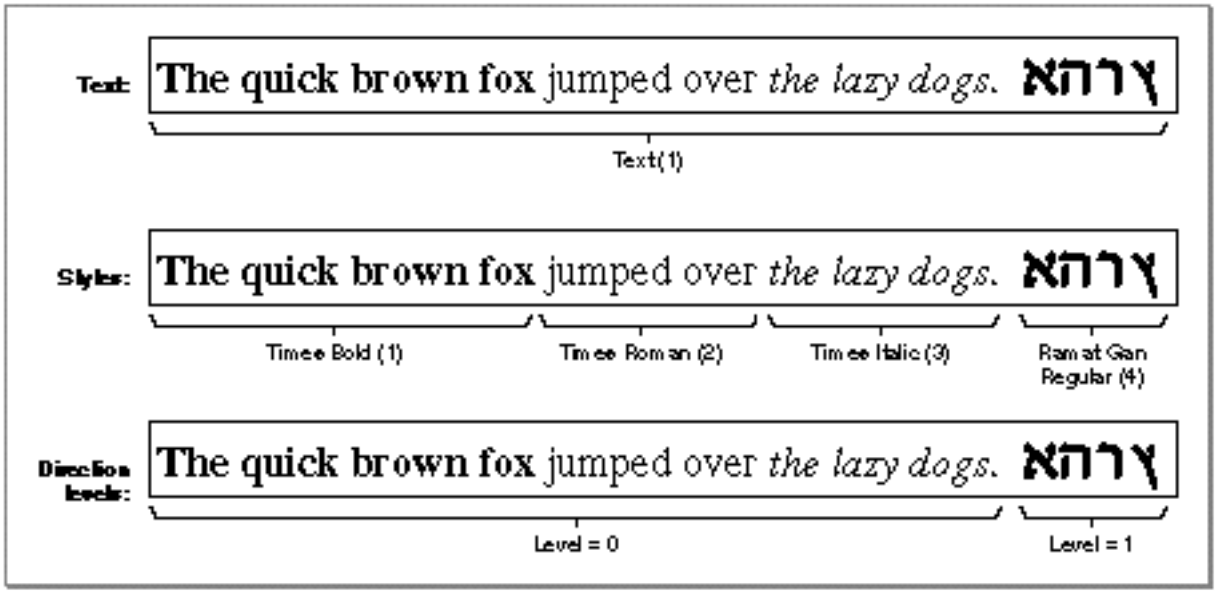
- **Style list.** An array of references to the style objects for each of the style runs. If the layout shape has only one style run or is `nil`, its style list may be empty, and the one style object may be referenced instead in the style property of the shape object itself.
- **Level-run count.** The number of direction-level runs in a layout shape. Each **direction-level run** defines a direction—left to right or right to left—for reading the text of that run. Because the runs can be nested (a left-to-right phrase may be embedded within a right-to-left phrase), each run has a *level* that describes its depth of embedding. Left-to-right phrases are given even numbers; right-to-left phrases are given odd numbers.
- **Level-run lengths.** An array that specifies the length, in bytes, of each level run in the layout shape.
- **Levels array.** An array that specifies the length in bytes of each level run in the layout shape.
- **Layout options.** A set of values and flags that are general controls for the line described by the layout shape: the width of the text area from the left margin to the right margin, the alignment of the text, the justification of the text, and the locations of the various baselines for the text.

Some functions, such as `GXSetLayoutParts` and `GXSetLayout`, manipulate the geometry, and other functions, such as `GXHitTestLayout` and `GXGetLayoutHighlight`, allow you to interact with the layout shape.

Runs in a Layout Shape

Most of the information in a layout shape is in the form of three kinds of runs. In Figure 5-2, a single layout shape has one text run, four style runs, and two direction-level runs.

Figure 5-2 An example of a layout with its text, style, and direction-level runs marked



Text Runs

A **text run** is an ordered array of character codes or glyph codes. These codes may be in any character encoding and therefore may be 1-byte codes, 2-byte codes, or a mixture of 1- and 2-byte codes. When you create a new layout shape, you specify the number of text runs, the byte length of each run, and the text in the runs that are to make up the **source text** of the layout shape. For example, your source text would include the two separate characters “f” and “i” and not the “fi” ligature, because the layout shape allows you to create ligatures. (The purpose of the layout shape is to manipulate the appearance of the text.)

The text in a layout shape is stored as a single text run, but you can maintain pointers to separate text runs in your own data structures. You pass those pointers to functions, such as `GXSetLayout` or `GXSetLayoutParts`, that modify the text of a layout shape.

QuickDraw GX assumes that the character codes of layout shape’s source text are always stored in **input order**—the order in which characters are typically entered when the user creates a document. (Note that sometimes the term *phonetic order* is used as a synonym for *input order*.) This may be different from the order in which QuickDraw GX displays them, which is the **display order.**

Note

If the `gxIgnorePlatformShape` attribute is set for the layout shape, then the text consists only of glyph codes. Otherwise, the platform controls of each run determine whether that run consists of character codes or glyph codes. For more information on the `gxIgnorePlatformShape` attribute, see the chapter “Typographic Shapes” in this book. u

The input order and display order may differ for certain text directions—the directions in which text of a particular language is written and read. Written English has a left-to-right direction; written Arabic has a (predominantly) right-to-left direction, and Japanese has a top-to-bottom direction, as shown in Figure 5-3.

Figure 5-3 English, Arabic, and Japanese text directions

If the text direction is right to left, QuickDraw GX may need to rearrange text from its input order to its display order. Layout shapes provide the information QuickDraw GX needs to rearrange text where necessary. The important point to remember is that you should store the source text of your layout shapes in input order, not display order.

In simple cases of left-to-right or right-to-left text, QuickDraw GX handles the rearrangement automatically. However, in certain situations you may need to specify direction, which you can do using direction-level runs. See “Direction-Level Runs” on page 5-9.

Style Runs

A single layout shape can have multiple **style runs.** When you create style runs for a layout shape, you specify the number of style runs, the number of bytes of text in each style run, and the style objects themselves. The style object references are contained in an array (the style list) within the layout shape geometry; these style objects are like the style object referenced in the style property of the layout shape object, except that there can be more than one of them per shape. If you don’t add any style objects to the layout shape geometry, QuickDraw GX uses the default style object for the entire shape. If you do add styles to the shape, QuickDraw GX ignores the default style object, except that if one of the styles in the style list is `nil`, QuickDraw GX uses the style object attached to the shape for that particular style run.

Note

If the geometry of a layout shape contains the style object or objects used with the layout shape, the function `GXSetShapeStyle`, which affects only the style object referenced in the style property of the shape, has no effect on the look of the layout shape. To change the styles of the shape, you must call `GXSetLayout`, `GXSetLayoutParts`, or `GXSetLayoutShapeParts`, which are described in this chapter. u

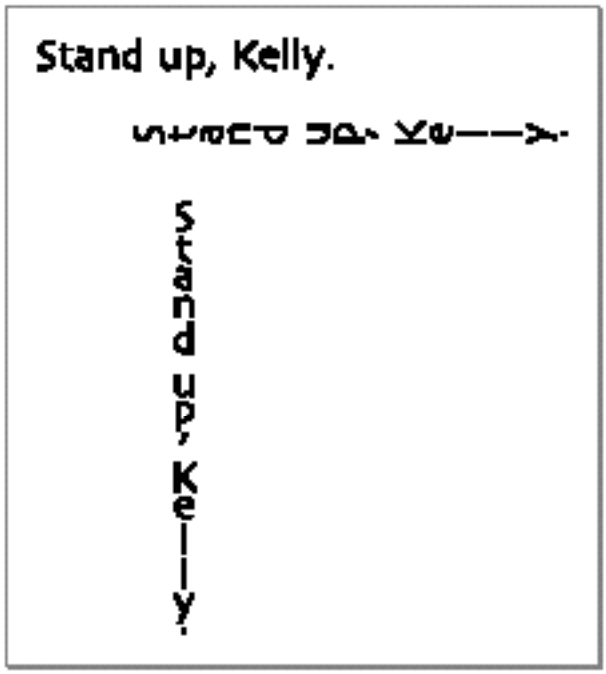
Because each style object in the style list contains the font, the text size, and the character encoding that you want to apply to one run of text in the shape, you can create one layout with several different fonts and scripts, such as Roman and Arabic. (For more information about the basic contents of the typographic elements of the style object, see the chapter “Typographic Styles” in this book. For information about the contents of a font, such as its script and language, see the chapter “Font Objects” in this book.)

Style objects, when used with layout shapes, also contain the run controls, font features, glyph justification overrides array, priority justification override structure, kerning adjustments array, and glyph substitutions array of a layout shape. These are described in the chapters “Layout Line Control” and “Layout Styles” in this book.

Vertical Text in a Layout Shape

Fundamentally, there is no vertical text direction for layout shapes. Because transform objects allow you to rotate any shape by any amount, to draw a layout shape as a vertical line of text you must rotate it 90 degrees before drawing it, as in Figure 5-4.

Figure 5-4 A line of text rotated into a vertical position



For measurement and analysis, a vertical line of text is considered to be left to right and horizontal. If a style run is to be vertical, that is, if its glyphs are to appear in their proper orientation when the line is rotated, you set the `gxVerticalText` text attribute. That setting rotates the glyphs 90 degrees counterclockwise before they are positioned on the baseline. You then perform operations (such as measuring the line) while the line is still considered horizontal, rotate the shape 90 degrees clockwise, and position it correctly before drawing it.

For more information and examples of drawing vertical lines, see the chapter “Layout Line Control” in this book.

Font Features

Font features are glyph-substitution capabilities that are built into QuickDraw GX fonts specifically for the layout shape. For example, a font can contain ligatures that your application can use whenever certain glyphs (such as “f” and “i”) appear in sequence on a line of text. Other examples of font features are cursive connections, special typestyles of number sets, and automatic formation of fractions.

The use of font features with the layout shape is an advanced topic discussed in detail in the chapter “Layout Styles” in this book.

Direction-Level Runs

If a text run (the text in a shape) has only one direction, whether left to right or right to left, displaying its characters in the proper order is simple. (The inherent direction of glyphs is determined by linguistic rules and stored by the font designer in the glyph properties table of the font.) However, if a text run contains a mixture of left-to-right and right-to-left text, the display order of its characters can be more complex.

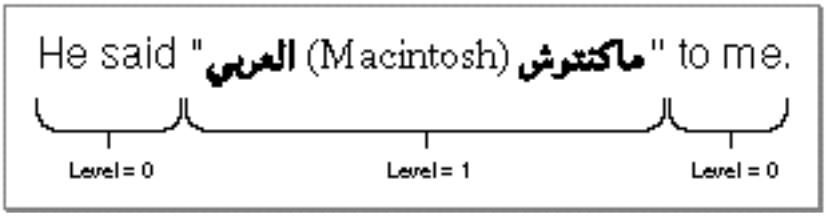
To help QuickDraw GX determine the display order of the characters in a layout shape containing text running in multiple directions, you can create direction-level runs by supplying a **direction-level** code to each block of text that runs in a different direction. The direction-level code determines the dominant direction of the target text run; the code affects only multidirectional text. A text run with a dominant direction can have embedded direction-level runs that must be displayed in other directions.

Furthermore, when you create a layout shape that contains multidirectional text, you can in most cases simply assign a single direction-level code to the entire shape, which gives it an overall dominant direction of left to right or right to left. When you do, QuickDraw GX automatically draws the embedded text in the proper direction. It is only when embedded text itself contains embedded text of another direction that the assignment of direction-level codes to portions of the text run becomes important.

Direction-level codes do not affect the order of the character codes as stored in the source text of a layout shape. Furthermore, direction-level codes affect only the ordering of blocks of text of a given direction; they do not affect the display order of the individual characters of a given direction. That is, if a layout shape contains a text run that is all right to left and you assign the shape a direction-level code that specifies a left-to-right dominant direction, QuickDraw GX does not rearrange the characters so they read left to right—it displays them correctly as right to left.

Figure 5-5 shows an example of right-to-left text embedded within left-to-right text; for an explanation of the numbering scheme used in this example, see the chapter “Layout Line Control” in this book.

Figure 5-5 A line of right-to-left of text with multiple direction levels



See the chapter “Layout Line Control” in this book for more information and for examples of how to create layout shapes with several level runs and multiple embedded text directions.

Layout Options

Layout options are values that apply to the entire layout shape and are stored in the layout options structure. The layout options determine the following basic characteristics of a layout shape:

- The width of the layout shape.
- The alignment of the layout shape. The default is zero or left-flush.
- The justification of the layout shape—that is, how the white space on the line of text is distributed between the words and glyphs. A layout shape can have no justification, full justification (all the extra white space on the line is distributed), or some fractional value in between. The default value is 0, or no justification.
- A pointer to a structure containing the distances between the y-positions of the baselines to use for this layout shape. (See the chapter "Layout Line Control" for more information about baselines.) The default value is `nil`.
- Layout option flags, which allow you to set some basic attributes of the layout shape. The default value is 0, or no flags set.

The layout options structure is described on page 5-29; values for the layout options flags are described on page 5-30.

Width

The `width` field of the layout options structure specifies the width of the line, from left margin to right margin. This value is a fixed number in typographic points (72 per inch), not QuickDraw GX coordinates.The default value is 0. Table 5-1 shows the various interactions of the `width`, `just`, and `flush` fields.

If you do not want to justify the text in the layout shape, the actual width of the line may differ from the value specified in this field. See "Setting the Width of a Layout Shape" on page 5-24.

Also, if the line contains glyphs with large negative side bearings, hanging punctuation, or optically aligned edges, the final width of the displayed layout shape may be different from the value you specify in the `width` field. (Hanging punctuation and optical alignment are described in the chapter "Layout Styles" in this book.)

Table 5-1 Interactions between the `width`, `just`, and `flush` fields

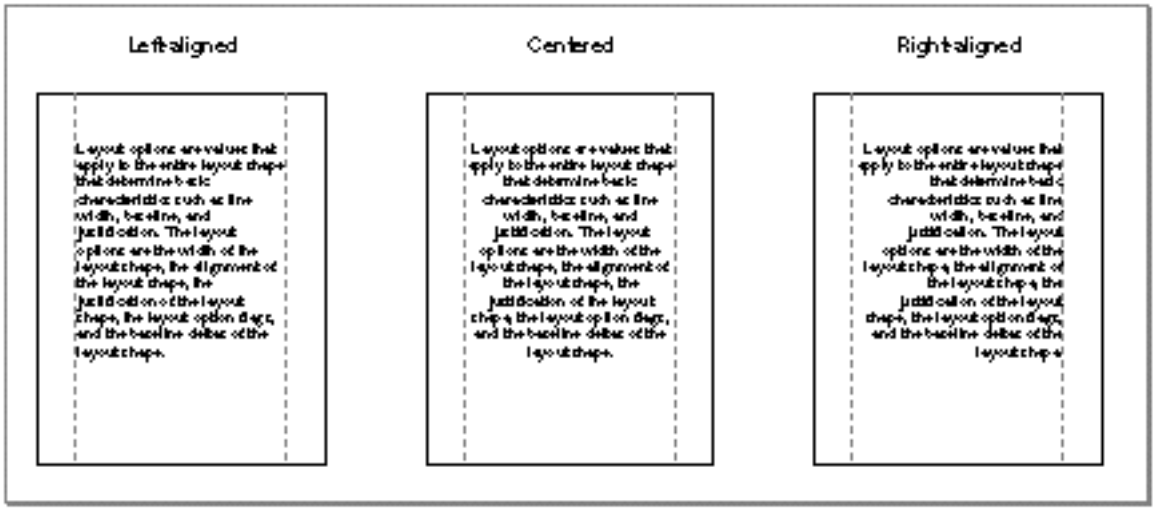
Width	Just	Flush	Effect
0	0	0	Unjustified layout is flush left.
0	0	>0	Unjustified layout moves around the origin proportionally. For example, if the `flush` field equals 0.5, layout is centered on the origin, or, if the `flush` field equals 1.0, the right edge of layout is aligned to the origin.
0	>0	0	Unjustified layout compresses, flush left.
0	>0	>0	Unjustified layout compresses at the point specified by the `flush` field.
>0	0	0	Unjustified layout is flush left, unless the unjustified width is greater than the specified width. In this case, the layout is compressed into the specified width.
>0	0	>0	Unjustified layout is at the point specified by the `flush` field within the specified width (rather than around the origin, as happens when the width is 0). If the unjustified width is greater than the specified width, layout is compressed into the specified width.
>0	>0	0	Justified layout is in the specified width, flush left, unless the `just` field is equal to 1.0. In this case, both edges are flush.
>0	>0	>0	Justified layout is in the specified width at the point specified by the `flush` field within the specified width, unless the `just` field is equal to 1.0. In this case, both edges are flush.

Alignment

Alignment, or flushness, is the placement of lines of text with respect to the left and right, or top and bottom margins (edges of the text area). Text can be left-aligned, right-aligned, centered, or positioned elsewhere between the margins. (Text that is both left-aligned and right-aligned is said to be *fully justified*; see “Justification” beginning on page 5-13.)

In Figure 5-6, three types of alignment are shown: left, right, and centered. Note how the words of the text are spaced normally. Unlike justification, alignment does not affect the spacing between words or individual glyphs.

Figure 5-6 Types of alignment



The default value for the alignment of a layout shape is 0, or alignment at the left margin. (A value of 0 also indicates the left-alignment for right-to-left text, such as Hebrew.) The `flush` field of the layout options structure, not the alignment values in the style objects associated with the shape, determines alignment. The `flush` field specifies whether the text is left-aligned, right-aligned, or centered in relation to the text margins, as shown in Figure 5-7.

Figure 5-7 Use of the `flush` field

If the value of the `flush` field is 0.0, the text appears aligned at the left or top edge. If the value is 0.5, the text is centered. If the value is 1.0, the text appears aligned at the right or bottom edge. Other values can be between these main values: for example, a value of 0.25 aligns the edge of the text halfway between where it would appear if the values were 0.0 and 0.5.

Justification

Justification is a type of alignment that involves expanding or compressing a line to occupy a given line width. The line width is specified by the value of the `width` field; QuickDraw GX uses the value in the `just` field to distribute the glyphs on the line.

QuickDraw GX justifies Roman text primarily by adjusting the white space between words and glyphs, as shown in Figure 5-8. Note that white space is added not only between words but also between glyphs. Also notice that ligatures such as “fi” and “ffi” can be broken during justification.

Figure 5-8 Alignment and justification in English

When Arabic text is justified, QuickDraw GX distributes the available white space on the line by automatically lengthening or shortening the **kashidas,** which are the extender bars stretching between some of the glyphs of a word, as shown in Figure 5-9.

Figure 5-9 Alignment and justification in Arabic

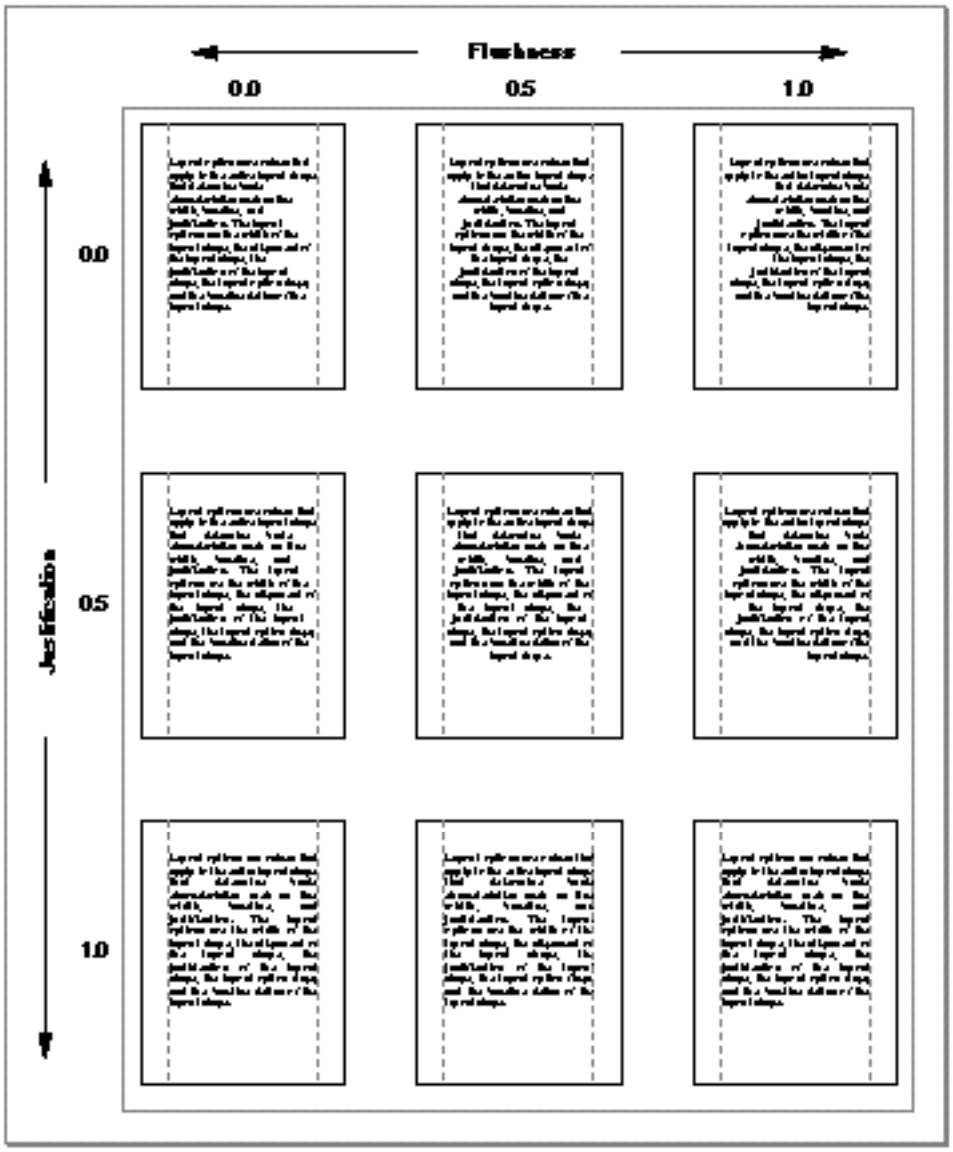
The `just` field of the layout options structure can have valid fractional values from 0 through 1. A value of 0.0 means no justification; a value of 1.0 means full justification (to the value of the `width` field). QuickDraw GX interprets intermediate values, such as 0.5, to mean partial justification (also called *ragged justification*), as shown in Figure 5-10.

Figure 5-10 Use of the `just` field

Figure 5-11 shows how the `just` field and `flush` (alignment) field of the layout options structure interact with values ranging from 0.0 to 1.0. Note that when the user chooses full justification—that is, when the `just` field equals 1.0—the value in the `flush` field has no effect.

For more information on how to control justification and alignment, see the chapter “Layout Line Control” in this book.

Figure 5-11 How different values for justification and alignment affect text in a layout shape

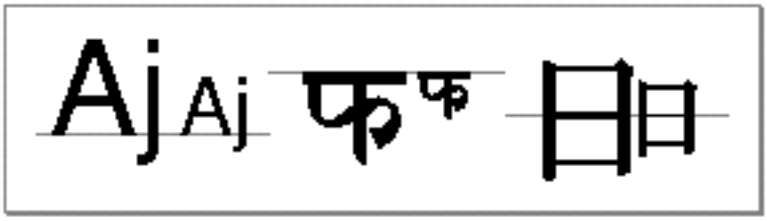


Baselines

In general, a run of text has a default **baseline,** the line to which all glyphs are visually aligned when the text is laid out. For example, in a run of Roman text, the default baseline is the Roman baseline, upon which glyphs sit (except for descenders, which extend below the baseline). In some other writing systems, glyphs hang from the baseline. When text in a line comprises runs using multiple baselines, the layout shape uses information in the baseline record of the layout options structure to determine how to align the runs with each other.

The baseline structure of the layout options structure contains an array of distances, in points, from a delta of 0 from the y-coordinate of the layout's origin to the other baseline types the layout shape contains. Positive values indicate baselines above the default baseline, and negative values indicate baselines below it. QuickDraw GX can use these values to position text in relation to the default baseline. Figure 5-12 shows an example of text with multiple baselines aligned according to information in the baseline structure.

Figure 5-12 Text with multiple baselines aligned to the default baseline



Baseline types and other uses for baselines are described in the chapter "Layout Line Control" in this book.

Flags

The flags of the layout options structure allow you to set certain characteristics of the layout shape as a whole.

The layout options flags can have the following settings:

- `gxNoLayoutOptions`. In this case, no layout options flags are set.
- `gxLineIsDisplayOnly`. This setting indicates that QuickDraw GX creates the shape without the internal information needed for editing the layout shape. This shape conserves memory and is for display purposes only. Performing any editing on this shape will clear this flag.

The Default Layout Shape

The default layout shape has no text and no layout options associated with it. The `gxMapTransformShape` attribute is set by default. Like text and glyph shapes, the default layout shape is of type `gxWindingFill`.

The default layout shape is associated with a style object. (See the chapter “Style Objects” in *Inside Macintosh: QuickDraw GX Objects*.) The default settings for all typographic shapes are described in the “Typographic Shapes” chapter in this book.

Using Layout Shapes

This section describes how to perform basic operations with the simple layout shape described in this chapter. These operations include

- creating and drawing a layout shape
- changing its parts: that is, changing some or all of the text, styles, or direction levels in the shape or inserting the geometry of another typographic shape
- setting the layout options of a layout shape, including its width, its alignment, and its justification
- getting glyph information from a layout shape
- converting a layout shape to a glyph shape to perform certain graphic operations on the shape

For information about more complex actions you can take with a layout shape, see the chapters “Layout Styles,” “Layout Carets, Highlighting, and Hit-Testing,” and “Layout Line Control” in this book.

Creating and Drawing a Layout Shape

You can use the `GXNewLayout` function (described on page 5-31) to create a new layout shape based on the text and style information you specify. All information about style runs is stored in the geometry of the shape, not in the style object associated with the layout shape object.

Alternatively, you can use the `GXNewShape` function followed by the `GXSetLayout` function to create and initialize the values of a layout shape. The `GXNewShape` function is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects.* Unlike `GXNewLayout`, `GXSetLayout` (page 5-36) uses an existing shape rather than creating a new shape, but both functions allow you to set the values of text runs, style runs, and direction-level runs.

If you have a pointer to some text and one or more style objects to apply to that text, you can create the layout shape and then draw it. Listing 5-1 illustrates this operation. The `GXNewLayout` function takes a string, a style, and a position, and puts a reference to the new layout shape into the variable `myLayout`.

Listing 5-1 Creating and drawing a layout shape

```
gxShape myLayout;
char   *myText = "Hi there!";
short  textLen = strlen(myText);
gxStyle myStyle = GXNewStyle();
gxPoint position = {ff(100), ff(100)};

myLayout = GXNewLayout(1, &textLen, &myText,
                       1, &textLen, &myStyle,
                       0, nil, nil,
                       nil, &position);
GXDrawShape(myLayout);
```

The `GXDrawLayout` function (page 5-33) speeds the process of creating, drawing, and then disposing of a layout shape. If you wanted to use `GXDrawLayout` instead of `GXNewLayout` and `GXDrawShape` in Listing 5-1, you could replace the last five lines of the listing with this code:

```
GXDrawLayout(1, &textLen, &myText,
            1, &textLen, &myStyle,
            0, nil, nil,
            nil, &position);
```

However, `GXDrawLayout` does not create a new layout shape that you can subsequently use. If you want to draw an existing layout shape or anticipate that you may want to draw a layout shape repeatedly, use the `GXDrawShape` function as shown in Listing 5-1 and as described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

Creating a Layout Shape With Multiple Style Runs

If a layout shape does not contain style list in its geometry, QuickDraw GX uses the style object referenced in the style property of the shape object. If the layout shape does have a style list, however, QuickDraw GX stores in it the information about the styles used in the layout shape, including information about the fonts, text size, typestyles, justification, and text attributes.

Listing 5-2 creates a layout shape that contains three style runs. The code creates two styles: the Palatino® Regular font at point size 20, and the Hoefler Text Italic font at point size 20. It assigns the first and third styles runs to Palatino, and the second style run to

Hoefler Text Italic. Listing 5-2 sets the style run lengths to the appropriate byte counts, and then it creates and draws the shape. The routine makes use of a library function, `NewLayoutStyle`, to create and initialize a style object.

Listing 5-2 Creating a line containing multiple style runs

```
static gxLayoutOptions myLayoutOptions;
static gxPoint  myPosition = {ff(20), ff(100)};
static short  len;

static short myStyleRunCount = 3;
gxStyle  myStyles[3];
static short myStyleRunLengths[] = {7,10,12};

static const char *myString = "Jeff's excellent layout shape";

/* Initialize as Pascal strings.
 */
static char palatinoName[] = "\pPalatino"
static char hoeflerName[] = "\pHoefler Text Italic"

len = strlen(myString);
InitializeLayoutOptions(&myLayoutOptions);

/* Set up styles: call the library function NewLayoutStyle to
 create the styles.
 */
myStyles[0] = NewLayoutStyle(palatinoName,ff(20),0, nil, nil, 0,
                             nil);
myStyles[1] = NewLayoutStyle(hoeflerName,ff(20),0, nil, nil, 0,
                             nil);
myStyles[2] = myStyles[0];

/* Create layout. */
gShape = GXNewLayout(1, &len, (void *)&myString,
                  myStyleRunCount, myStyleRunLengths, myStyles,
                  0, nil, nil, &myLayoutOptions,
                  &myPosition);
GXDrawShape(gShape);
```

Listing 5-2 creates the output shown in Figure 5-13.

Figure 5-13 A layout shape with multiple style runs

Positioning a Layout Shape

To position a layout shape, you can use either the `GXNewLayout` or `GXSetLayout` function. The value you supply in the `position` parameter of these functions generally determines the position, in the local coordinates of the view port, of the baseline of the intersection on the margin and the layout shape.

Alternatively, you can call the `GXMoveShapeTo` function to draw the shape at a specified point in the view port, as in this example:

```
GXMoveShapeTo(myLayoutShape, ff(50), ff(125));
```

However, the `GXMoveShapeTo` function does not change the value of the position stored in the layout shape. If you want to change the position stored in the shape, you must call the `GXSetLayout` function (page 5-36).

For a description of the `GXMoveShapeTo` function, see the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

Changing Parts of an Existing Layout Shape

Three similarly named functions allow you to change values in the geometry of a layout shape: `GXSetLayout`, `GXSetLayoutParts`, and `GXSetLayoutShapeParts`. These functions have companion functions that are also similarly named and allow you to retrieve values: `GXGetLayout`, `GXGetLayoutParts`, and `GXGetLayoutShapeParts`. Each set of functions serves a different purpose:

- If you want to retrieve or change information for the whole layout (text, style, levels) in a layout shape, use the `GXGetLayout` and `GXSetLayout` functions.
- If you want to retrieve or change parts of runs of information in a layout shape, use the `GXGetLayoutParts` and `GXSetLayoutParts` functions.
- If you want to create a second layout shape using some or all of a first layout shape, use the `GXGetLayoutShapeParts` function. This function allows you to extract a section from a layout shape—including the text, styles, and levels—and create a new layout shape out of it.
- If you want to change the geometry of an existing layout shape using the geometry of another typographic shape (whether text, glyph, or layout), use the `GXSetLayoutShapeParts` function. This function allows you to insert the text and styles of another typographic shape into an existing layout shape. Any styles attached

to the text from the inserted shape are also inserted into the layout shape, but other arrays of information, such as the positions and advance bits arrays of the glyph shape, are not.

Changing Text in a Layout Shape

You can change text in a layout shape using the `GXSetLayout` or `GXSetLayoutParts` function. Use the former if you want to replace all of the text in the shape.

However, for most editing operations you may want to perform on the shape—whether adding text, deleting text, or replacing text—you should use the faster `GXSetLayoutParts` function. Table 5-2 lists some of the parameter settings for changing text in a layout shape using this function.

Table 5-2 Changing text in a layout shape using the `GXSetLayoutParts` function

Action	Starting offset in the text to be edited	Ending offset in the text to be edited
Inserting new text	The offset (in source text) at which the new text should start, or `gxSelectToEnd`, if you want to insert at the end	The same as `oldStartOffset`
Replacing and deleting text	The offset of the first byte you want to replace	The offset of the last byte you want to replace
Replacing all text in the shape	0	The value `gxSelectToEnd`

In Listing 5-3, the original layout shape contains the string “ABC”. The `GXSetLayoutParts` function inserts the new text, “ DEF”, at the end of the original string. The layout shape then reads “ABC DEF”.

Listing 5-3 Adding text to a layout shape using the `GXSetLayoutParts` function

```
gxShape myLayout;
char *originalText = "ABC";
char *newText = " DEF";
short originalLen, newLen;
gxPoint position = {ff(100), ff(100)};

originalLen = strlen(originalText);
myLayout = GXNewLayout(1, &originalLen, (void *)&originalText,
                       0, nil, nil,
                       0, nil, nil,
                       nil, &position);
```

```
newLen = strlen(newText);
GXSetLayoutParts(myLayout, 3, gxSelectToEnd,
                1, &newLen, (void *)&newText,
                0, nil, nil,
                0, nil, nil);
```

If you use the `GXSetLayout` function to perform the same action, you must pass the entire string of text (“ABC DEF”), and not simply the string “ DEF”. (You also must pass the new text length and resend the text run count, as well as the style run lengths and direction-level run lengths.)

The values in Table 5-2 apply if you are changing the style runs or direction-level runs in the layout shape. However, you can’t insert style runs or direction-level runs without changing the values of the runs already stored in the shape.

Inserting a Typographic Shape Into a Layout Shape

To insert the geometry of another typographic shape—whether a text, glyph, or layout shape—into an existing layout shape, you can use the `GXSetLayoutShapeParts` function (page 5-44). The function inserts both the text of the typographic shape and the associated style or styles into the layout shape.

Listing 5-4 creates a layout shape that reads “, , and layout”. It then creates a glyph shape that reads “glyph” and inserts it into the original layout shape, changing the shape to read “, glyph, and layout”. It then creates a text shape that reads “text” and inserts that into the layout shape, leaving the shape with the text “text, glyph, and layout”. In this example, the glyph and text shapes use the default style object, as the layout shape does. However, if the inserted shapes had had different styles, those styles would have been inserted into the layout shape as well, creating a layout shape with several style runs.

Listing 5-4 Inserting a text shape and a glyph shape into a layout shape

```
char   *layoutText = ", , and layout"
short  layoutLen;
gxPoint position = {ff(100), ff(100)};
gxShape myLayout, myGlyph, myText;

layoutLen = strlen(layoutText);
myLayout = GXNewLayout(1, &layoutLen, (void *)&layoutText,
                      0, nil, nil,
                      0, nil, nil,
                      nil, &position);

myGlyph = GXNewGlyphs(5, (unsigned char *)"glyph",
                     nil, nil, nil, nil, nil);
GXSetLayoutShapeParts(myLayout, 2, 2, myGlyph);
```

```
myText = GXNewText(4, (unsigned char *)"text", nil);
GXSetLayoutShapeParts(myLayout, 0, 0, myText);

GXDrawShape(myLayout);

GXDisposeShape(myLayout);
GXDisposeShape(myGlyph);
GXDisposeShape(myText);
```

Note that if you insert the geometry of a glyph shape into a layout shape, you lose the advance bits, positions, and tangents arrays of that glyph shape.

Extracting a Layout Shape From Part of an Existing Layout Shape

To copy part or all of an existing layout shape to another layout shape, you can use the `GXGetLayoutShapeParts` function (page 5-42). You can copy to an existing layout shape (in which case the function replaces the entire geometry of the shape) or a new layout shape, which `GXGetLayoutShapeParts` creates.

Listing 5-5 creates a layout shape that reads “blue & red balloon”. Using the `GXGetLayoutShapeParts` function, it then extracts the text “red ball” from the layout shape and creates a new layout shape that contains that text. If the text had styles attached to it, the function would include references to these styles in the new layout shape. (In this example, however, the default layout shape’s style object is used.)

Listing 5-5 Creating a new layout shape from a previously existing one

```
char *layoutText = "blue & red balloon";
short layoutLen;
gxPoint position = {ff(100), ff(100)};
gxShape myLayout, newLayout;

layoutLen = strlen(layoutText);
myLayout = GXNewLayout(1, &layoutLen, (void *)&layoutText,
                       0, nil, nil, 0, nil, nil,
                       nil, &position);

newLayout = GXGetLayoutShapeParts(myLayout, 7, 15, nil);

GXDrawShape(newLayout);

GXDisposeShape(myLayout);
GXDisposeShape(newLayout);
```

Setting Layout Options

The layout options structure is described in detail on page 5-29. Values for layout options flags are described on page 5-30.

Setting the Width of a Layout Shape

You can set the width of a layout shape by setting the `width` field of the layout options structure. This field specifies the desired width, in typographic points, of the line when you draw justified or right-aligned text. However, the width of the line and the width of the (unjustified) layout shape itself may be different, and if the line is *not* justified or right-aligned, the `width` field is ignored, unless the unjustified width of the layout would exceed the specified width, in which case the layout is squeezed to fit (see Table 5-1).

Also, if the line contains glyphs with large negative side bearings, hanging punctuation, or optically aligned edges, the final width of the displayed layout shape may be different from the value specified here.

For more information and examples on line measurement and the use of the `width` field for justification, see the chapter “Layout Line Control” in this book.

Setting the Alignment of a Layout Shape

You can set the alignment of a layout shape by changing the value of the `flush` field in the layout options structure. A value of 0 specifies left alignment, a value of 1 specifies right alignment, and fractional values between 0 and 1 position the text proportional distances from the left and right margins.

Listing 5-6 creates a layout shape containing the text “A line of text”. It then aligns the string at five different positions by changing the value of the `flush` field. This code uses library functions `InitializeRunControls` and `InitializeLayoutOptions` to initialize the data structures, and `NewLayoutStyle` to create a style object.

Listing 5-6 Altering the alignment of a layout shape

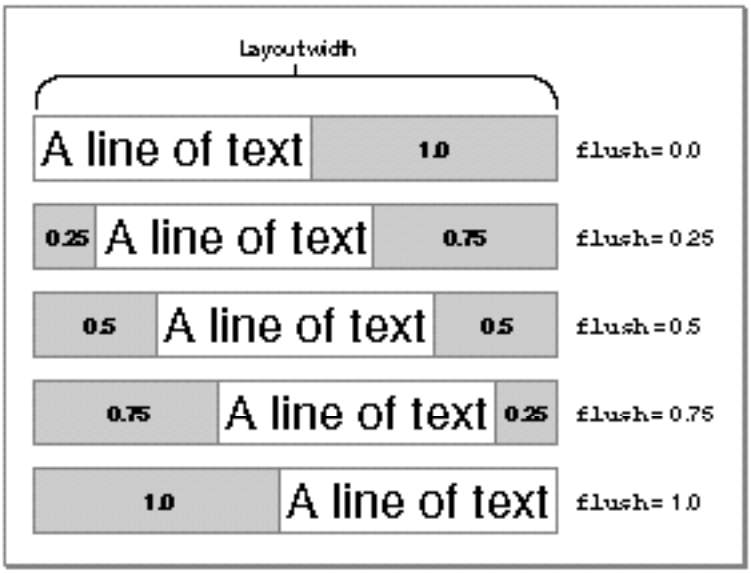
```
char  *myString = "A line of text";
gxLayoutOptions layoutOptions;
gxLine myLine;
static gxPoint myPoint = {ff(30), ff(50)};
gxShape layout;
short len;
gxStyle myStyle;

len = strLen(myString);

InitializeLayoutOptions(&layoutOptions);
layoutOptions.width = ff(500);
```

```
/* The initial alignment of the layout shape is set to 0.0. */
layoutOptions.flush = 0;

myLine.first.x = myLine.last.x = myPoint.x;
myLine.first.y = 0;
myLine.last.y = ff(1000);
GXDrawLine(&myLine);

myLine.first.x = myLine.last.x = myPoint.x + layoutOptions.width;
GXDrawLine(&myLine);

myStyle = NewLayoutStyle((char *) "\pTimes Roman", ff(50), 0,
                         nil, nil, 0, nil);

layout = GXNewLayout(
   1, &len, (void *) &myString,
   1, &len, &myStyle,
   0, nil, nil,
   nil, &myPoint);
GXDrawShape(layout);

/* The alignment of the layout shape is set to 0.25. */
layoutOptions.flush = fract1 / 4;
GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
&layoutOptions, nil);
GXMoveShape(layout, 0, ff(75));
GXDrawShape(layout);

/* The alignment of the layout shape is set to 0.5. */
layoutOptions.flush = fract1 / 2;
GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
&layoutOptions, nil);
GXMoveShape(layout, 0, ff(75));
GXDrawShape(layout);

/* The alignment of the layout shape is set to 0.75. */
layoutOptions.flush = 3 * (fract1 / 4);
GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
&layoutOptions, nil);
GXMoveShape(layout, 0, ff(75));
GXDrawShape(layout);
```

```
/* Here the alignment of the layout shape is set to 1.0. */
layoutOptions.flush = fract1;
GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
&layoutOptions, nil);
GXMoveShape(layout, 0, ff(75));
GXDrawShape(layout);

GXDisposeShape(layout);
GXDisposeStyle(myStyle);
```

Figure 5-14 shows the output of Listing 5-6. The numbers in the gray boxes specify what fraction of the gap appears on either side of the text.

Figure 5-14 Changing the alignment of a layout shape



See “Alignment” on page 5-11 for more information.

Justifying Text in a Layout Shape

You can control the justification of a layout shape by setting the value of the `just` field in the layout options structure. A value of 0 specifies no justification, a value of 1 specifies full justification, and fractional values between 0 and 1 distribute the extra space proportionally. The `width` field in the layout options structure defines the text width for justification.

Justification is easy to specify, but its internal workings are complex, powerful, and controlled by settings in the style object for each style run. For more information on and examples of the QuickDraw GX justification process, see the chapter “Layout Line Control” in this book.

Getting Glyph Information From a Layout Shape

In a layout shape, there can be a difference between the number of character codes in the source text and the number and order of glyphs displayed, because the layout shape can automatically form ligatures or other contextual forms. (The other typographic shapes, text and glyph, have a one-to-one correspondence between the number of character codes and the number of glyphs displayed.)

You can get information about the character codes stored in the shape using the `GXGetLayout` or `GXGetLayoutParts` function, as described in "Changing Parts of an Existing Layout Shape" on page 5-20. However, you can also get information about the glyphs in a layout shape using the `GXGetLayoutGlyphs` function, described on page 5-45.

The difference between the information returned by the `GXGetLayoutGlyphs` function and by the `GXGetLayout` function can be illustrated as follows. Take, for example, the layout shape "office", which has a source text of six character codes but is displayed using only four glyphs because of the "ffi" ligature. If you look at the contents of the `text` parameter returned by the `GXGetLayout` function, you will see six character codes. The `GXGetLayoutGlyphs` function returns only four glyph codes—and all four may be different from the glyph codes that individually correspond to the original character codes, depending on whether alternate forms are used. Likewise, the style runs are different: there is a different count for character codes than for glyph codes.

The `GXGetLayoutGlyphs` function treats the layout shape as though it were a glyph shape—that is, it returns information about the advance bits, positions, and tangents arrays, although the layout shape does not explicitly contain any of these arrays.

The tangents array gives every entry a default value of (1.0,0.0). However, if you set the `gxVerticalText` text attribute in the layout shape, the array contains individual tangents for each glyphs.

Converting a Layout Shape Into a Glyph Shape

You can convert a layout shape into a text shape, a glyph shape, or any other type of shape.

As with all typographic shapes, you cannot convert any type of geometric shape (point, line, rectangle, and so on) into a layout shape.You can include, in the layout shape's style list, a text face that is patterned. Text faces are described in the chapter "Typographic Styles" in this book.

You can use the `GXPrimitiveShape` function, described in *Inside Macintosh: QuickDraw GX Graphics*, to convert a layout shape into a glyph shape. This conversion lets you take advantage of the special capabilities of a glyph shape. For instance, you may want to alter the tangents or positions of the individual glyphs in the shape.

Keep in mind that, after conversion, the resulting glyph shape looks exactly the same as the layout shape, but its internal data may be very different. For example, the order of character codes in the source text is the same as the display order of the glyphs in the glyph shape, which may not have been the case with the original layout shape.

In addition, you cannot restore the original layout shape by converting the glyph shape back into a layout shape. However, you can use the glyph shape for clipping, dashing, and patterns.

Also keep in mind that, after conversion, the resulting glyph shape may not retain alignment settings, justification settings, and other information originally in the layout shape. This information is lost, for instance, if you alter the positions of glyphs in the displayed shape by calling `GXSetGlyphTangents` and other functions described in the chapter "Glyph Shapes" in this book.

Note

You can also use the `GXSetShapeType` function to convert a layout shape into a glyph shape, although the resulting shape may not look the same. The `GXSetShapeType` function does not preserve all layout functionality in the destination shape, whereas the `GXPrimitiveShape` function does. u

Layout Shapes Reference

This section describes the constants and data types you use with layout shapes, as well as the basic functions you need to create a layout shape, change the information stored in a layout shape, and retrieve information from a layout shape.

Functions that relate to other aspects of layout shapes are described in the following chapters in this book:

- The chapter "Layout Styles" describes functions you can use to override the kerning and justification behavior and manipulate the special typographic features of a layout shape.
- The chapter "Layout Line Control" describes functions you can use to measure and break lines in a layout shape.
- The chapter "Layout Carets, Highlighting, and Hit-Testing" describes functions you can use to create and manipulate carets and highlighted sections of a layout shape.

Constants and Data Types

This section describes the constants and data types that you use when creating layout shapes.

Layout Options Structure

The layout options structure contains information that is relevant to the entire line of text rather than to any individual text run, style run, or direction-level run in the shape.

```
typedef struct {
   Fixed                  width;
   fract                  flush;
   fract                  just;
   gxLayoutOptionsFlags   flags;
   gxLineBaselineRecord   *baselineRec;
} gxLayoutOptions;
```

Field descriptions

`width` The desired width of the line, measured in points (72 per inch) in geometry space. If you want the layout shape to be fully left-aligned and not justified, you do not need this field and you can set it to 0. See "Width" on page 5-10.

`flush` The alignment of the text, based on the layout shape's position and the line width as specified in the `width` field. Alignment is a continuously varying, fractional value. A value of 0 means the text is to be left-aligned; its left or top edge coincides with the shape's position. A value of 1 means the text is to be right-aligned; its right or bottom edge coincides with the shape position plus the text width. Intermediate values place the text between the left and right edges. (A value of 0.5 centers the text.) The following constants are available for specifying typical values in the `flush` field:

```
#define gxFlushLeft        0
#define gxFlushCenter      (fract1/2)
#define gxFlushRight       fract1
```

`just` The degree of justification in the line, defined as a continuously varying fractional value between 0 and 1. A value of 0 means no justification. A value of 1 means full justification between the edges defined by the layout position and the sum of the layout position and the `width` field. Intermediate values cause a fractional amount of the extra white space to be taken up by justification. The following constants are available for specifying typical values in the `just` field:

```
#define gxNoJustification     0
#define gxFullJustification   fract1
```

`flags` Flag values that describe certain aspects of the entire layout shape. Values for the `flags` field are described in the next section, "Layout Options Flags."

`baselineRec` An array of distances, in points (72 per inch), from the primary baseline for this layout shape to other baseline types. If you fill in this structure manually, you need to fill in only those values that correspond to the set of baselines present on the line. If you specify `nil` for this value, the line uses the Roman baseline, and all text is aligned to it. For more information on baselines and baseline alignment, see the chapter "Layout Line Control" in this book.

The layout options structure is used by the functions `GXNewLayout` (page 5-31), `GXGetLayout` (page 5-34), `GXSetLayout` (page 5-36), and `GXDrawLayout` (page 5-33).

Layout Options Flags

The layout options flags allow you to set certain characteristics of the layout shape as a whole. You set these flags through the `flags` field of the layout options structure.

```
#define gxNoLayoutOptions        0
#define gxLineIsDisplayOnly      0x00000001
```

Flag descriptions

`gxNoLayoutOptions`
Indicates that no layout option flags are used in this layout shape.

`gxLineIsDisplayOnly`
Indicates that the layout shape will be displayed but not edited in any way. If this bit is set, the shape is not edited, and QuickDraw GX does not recalculate any changes to the shape, such as the addition of ligatures or kashidas. This allows QuickDraw GX to display the layout shape faster and make the shape smaller.

Functions

This section describes functions for creating and drawing layout shapes, getting information from layout shapes, and editing layout shapes.

Some functions in this section use a byte offset parameter type. The `gxByteOffset` data type defines a byte offset into the text stored in the layout shape.

```
typedef long gxByteOffset;
```

The number of bytes is not necessarily equal to the number of character codes or glyph codes.

Creating and Drawing Layout Shapes

When you create a layout shape, you can supply it with different numbers of text runs, style runs, and direction-level runs. The `GXNewLayout` function creates a new layout shape for you to use: the `GXDrawLayout` function creates, draws, and disposes of a layout shape with a single call.

Note that you can also create and draw a layout shape by using the GXNewShape and GXDrawShape functions, described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects.*

GXNewLayout

You can use the GXNewLayout function to create a new layout shape.

```
gxShape GXNewLayout(long textRunCount,
                    const short textRunLengths[],
                    const void *text[],
                    long styleRunCount,
                    const short styleRunLengths[],
                    const gxStyle styles[],
                    long levelRunCount,
                    const short levelRunLengths[],
                    const short levels[],
                    const gxLayoutOptions *layoutOptions,
                    const gxPoint *position);
```

`textRunCount`
: The number of text runs supplied (the number of entries in the `text` parameter).

`textRunLengths`
: An array containing the byte length of each text run (the length of each entry in the `text` parameter).

`text`
: An array of pointers to runs of text. The text from these runs is concatenated, in order, to make up the source text of the layout shape.

`styleRunCount`
: The number of style runs in the layout shape. (The number of entries in the `styles` parameter.)

`styleRunLengths`
: An array containing the byte length of each style run.

`styles`
: The style list: an array of references to the style objects associated with the layout shape, one for each style run. If you pass `nil` for this parameter, QuickDraw GX assigns the default layout style object to the layout shape and leaves the style list empty. If you pass a non-`nil` value for this parameter, then any `nil` entries in the array also refer to the shape’s style.

`levelRunCount`
: The number of direction-level runs in this layout shape (the number of entries in the `levels` parameter).

`levelRunLengths`
: An array containing the byte length of each direction-level run.

`levels` The levels array: an array of nested direction levels that control the dominant text direction within the layout shape. If pass `nil` for this parameter, QuickDraw GX assumes that the layout shape has an overall dominant direction of left to right.

`layoutOptions` A pointer to a layout options structure. If you specify `nil` for this parameter, the default values are: left-aligned, unjustified, horizontal text on a Roman baseline.

`position` The position of the baseline in the geometry coordinates. If you specify `nil` for this parameter, `GXNewLayout` sets the position to (0.0,0.0).

function result A reference to the newly created layout shape.

DESCRIPTION

The `GXNewLayout` function creates a layout shape, sets its owner count to 1, initializes its geometry with the values in the function's parameters, and returns a reference to it as the function result.

Although this function creates a new layout shape, it does not create new style, ink, or transform objects. The new layout shape returned by `GXNewLayout` contains references to the default style, ink, and transform objects.

Most of the parameters to `GXNewLayout` are optional; if you set them to `nil`, QuickDraw GX sets the layout shape's equivalent properties to the default values, which are no text, no styles, and no levels.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
count_is_less_than_zero
parameter_out_of_range
inconsistent_parameters
```

SEE ALSO

To create a new layout shape without specifying an initial geometry, see the description of the `GXNewShape` function in the chapter "Shape Objects" in *Inside Macintosh: QuickDraw GX Objects*.

GXDrawLayout

You can use the `GXDrawLayout` function to create, draw, and dispose of a layout shape with one call.

```
void GXDrawLayout(long textRunCount, const short textRunLengths[],
                  const void *text[], long styleRunCount,
                  const short styleRunLengths[],
                  const gxStyle styles[], long levelRunCount,
                  const short levelRunLengths[],
                  const short levels[],
                  const gxLayoutOptions *layoutOptions,
                  const gxPoint *position);
```

`textRunCount`
: The number of text runs supplied (the number of entries in the `text` parameter).

`textRunLengths`
: An array containing the byte length of each text run (the length of each entry in the `text` parameter).

`text`
: An array of pointers to runs of text. The text from these runs is concatenated, in order, to make up the source text of the (temporary) layout shape.

`styleRunCount`
: The number of style runs in the layout shape. (The number of entries in the `styles` parameter.)

`styleRunLengths`
: An array containing the byte length of each style run.

`styles`
: The style list: an array of references to the style objects associated with the (temporary) layout shape, one for each style run. If you pass `nil` for this parameter, QuickDraw GX assigns the default layout style object to the layout shape and leaves the style list empty. If you pass a non-`nil` value for this parameter, then any `nil` entries in the array also refer to the shape's style.

`levelRunCount`
: The number of direction-level runs in this layout shape (the number of entries in the `levels` parameter).

`levelRunLengths`
: An array containing the byte length of each direction-level run.

`levels`
: The levels array: an array of nested direction levels that control the dominant text direction within the layout shape. If you specify `nil` for this parameter, QuickDraw GX assumes that the layout shape has an overall dominant direction of left to right.

`layoutOptions`
: A pointer to a layout options structure. If you specify `nil` for this parameter, the default values are: left-aligned, unjustified, left to right horizontal text on a Roman baseline.

`position` The position of the baseline in the geometry coordinates. If you specify `nil` for this parameter, `GXNewLayout` sets the position to (0.0,0.0).

DESCRIPTION

The `GXDrawLayout` function creates, draws, and then disposes of a layout shape. You may want to use this function if you do not need to store a layout shape or draw the shape more than once.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
count_is_less_than_zero
parameter_out_of_range
inconsistent_parameters
```

SEE ALSO

The `GXNewLayout` function, described on page 5-31, creates a new layout shape that you can store and draw more than once.

You can use the `GXDrawShape` function, described in *Inside Macintosh: QuickDraw GX Objects*, to draw an existing layout shape.

Getting and Setting the Geometry of a Layout Shape

When you retrieve information about the text, style, and direction-level runs from a layout shape, you can get an entire array from the geometry—for instance, the complete style list—using the `GXGetLayout` function. You can change an entire array in the geometry—for instance, all of the text runs in the shape—using the `GXSetLayout` function.

GXGetLayout

You can use the `GXGetLayout` function to get all the information from the geometry of a layout shape.

```
long GXGetLayout(gxShape layout, void *text, long *styleRunCount,
                 short styleRunLengths[], gxStyle styles[],
                 long *levelRunCount, short levelRunLengths[],
                 short levels[], gxLayoutOptions *layoutOptions,
                 gxPoint *position);
```

`layout` A reference to the layout shape whose information you need.

`text` A pointer to a space for a text string. On return, the string contains all of the text from the layout shape (as a single text run).

`styleRunCount`
A pointer to a `long` value. On return, the value is the number of style runs in the shape (the number of entries in the style list).

`styleRunLengths`
An array of `short` values. On return, the array contains the byte length of each style run in the layout shape.

`styles` An array of style-object references. On return, the array contains the style list for the layout shape.

`levelRunCount`
A pointer to a `long` value. On return, the value is the number of direction-level runs in this layout shape (the number of entries in the `levels` parameter).

`levelRunLengths`
An array of `short` values. On return, the array contains the byte length of each direction-level run.

`levels` An array of `short` values. On return, the array is the levels array for the layout shape: an array of nested direction levels that control the dominant text direction within the layout shape.

`layoutOptions`
A pointer to the `gxLayoutOptions` structure. On return, the structure contains the layout options for the layout shape.

`position` A pointer to a `gxPoint` value. On return, this parameter contains the position of the layout shape in geometry coordinates.

function result The number of bytes of text returned in the `text` parameter.

DESCRIPTION

The `GXGetLayout` function returns all the information from the geometry of the specified layout shape. If you specify `nil` for any parameter, `GXGetLayout` does not return that information.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
illegal_type_for_shape
index_is_less_than_zero
parameter_out_of_range
inconsistent_parameters
```

SEE ALSO

To change the geometry of an existing layout shape by replacing an entire array in the geometry, use the `GXSetLayout` function, described next.

To get a portion of one or more arrays from the geometry of an existing layout shape, use the `GXGetLayoutParts` function, described on page 5-38. To change the geometry of an existing layout shape by replacing a portion of one or more of its arrays, use the `GXSetLayoutParts` function, described on page 5-40.

To get a portion of the geometry of an existing layout shape and put that portion into another layout shape, use the `GXGetLayoutShapeParts` function, described on page 5-42. To change the geometry of an existing layout shape by inserting the geometry of another typographic shape, use the `GXSetLayoutShapeParts` function, described on page 5-44.

GXSetLayout

You can use the `GXSetLayout` function to assign a new text array, style list, direction-levels array, or other property in the geometry of a layout shape.

```
void GXSetLayout(gxShape layout, long textRunCount,
                 const short textRunLengths[], const void *text[],
                 long styleRunCount,
                 const short styleRunLengths[],
                 const gxStyle styles[], long levelRunCount,
                 const short levelRunLengths[],
                 const short levels[],
                 const gxLayoutOptions *layoutOptions,
                 const gxPoint *position);
```

`layout` A reference to the layout shape whose properties you want to set.

`textRunCount`
The number of text runs supplied (the number of entries in the `text` parameter).

`textRunLengths`
An array containing the byte length of each text run (the length of each entry in the `text` parameter).

`text` An array of pointers to runs of text. The text from these runs is collated, in order, to make up the new source text of the layout shape.

`styleRunCount`
The number of style runs to put in the layout shape. (The number of entries in the `styles` parameter.)

`styleRunLengths`
An array containing the byte length of each style run.

`styles` The new style list for the layout shape.

`levelRunCount`
: The number of direction-level runs to put in this layout shape (the number of entries in the `levels` parameter).

`levelRunLengths`
: An array containing the byte length of each direction-level run.

`levels`
: The new levels array for the layout shape.

`layoutOptions`
: A pointer to the layout options structure to use for this layout shape. If you specify `nil` for this parameter, the layout shape's current layout options are not changed.

`position`
: The position of the baseline in the geometry coordinates. If you specify `nil` for this parameter, the layout shape's current position is not changed.

DESCRIPTION

The `GXSetLayout` function sets one or more entire field properties of a layout shape. You can change the values in any of the text, styles, or direction-levels arrays, or in the layout options or position. If you change one value having to do with the text, styles, or direction-levels components, you must send values for all the parameters having to do with that part—run count, run lengths, or the main component itself—even if those values are not changing.

If you want to add new values to existing values, you must send both the old and new values. For example, if you want to change the text "ABC" to "ABC DEF", you must send the entire string, not simply the string "DEF". (You must also send the new text length and resend the text run count. If you don't, the function returns the error `inconsistent_parameters`. In addition, you must update the style list and change the level run length.)

If you don't want to change one property of a layout (the text, styles, or direction levels), you can specify in 0, `nil`, and `nil` for the corresponding count, run lengths, and array parameters.

If you don't want to change the layout options or position, specify `nil` for the corresponding parameters.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
length_is_less_than_zero
index_is_less_than_zero
inconsistent_parameters
count_is_out_of_range
count_is_less_than_zero
```

Warnings

```
shape_access_not_allowed
shape_contains_invalid_data
```

SEE ALSO

For an example of how to use the `GXSetLayout` function, see "Changing Parts of an Existing Layout Shape" beginning on page 5-20.

To get one or more complete arrays from the geometry of an existing layout shape, use the `GXGetLayout` function, described on page 5-34.

To get a portion of one or more arrays from the geometry of an existing layout shape, use the `GXGetLayoutParts` function, described next. To change the geometry of an existing layout shape by replacing a portion of one or more of its arrays, use the `GXSetLayoutParts` function, described on page 5-40.

To get a portion of the geometry of an existing layout shape and put that portion into another layout shape, use the `GXGetLayoutShapeParts` function, described on page 5-42. To change the geometry of an existing layout shape by inserting the geometry of another typographic shape, use the `GXSetLayoutShapeParts` function, described on page 5-44.

Getting and Setting Portions of a Layout Shape's Geometry

If you want to retrieve information about only a part of the shape—for example, the part of the style list associated with the characters 9 to 15 of the shape in the source text—you can use the `GXGetLayoutParts` function. If you want to change only a portion of the geometry—for example, if you want to insert four character codes in the middle of the existing shape's source text—you can use the `GXSetLayoutParts` function.

GXGetLayoutParts

You can use the `GXGetLayoutParts` function to get a portion of one of the various arrays of a layout shape, such as the style runs or layout options.

```
long GXGetLayoutParts(gxShape layout, gxByteOffset startOffset,
                      gxByteOffset endOffset, void *text,
                      short *styleRunCount,
                      short styleRunLengths[], gxStyle styles[],
                      short *levelRunCount,
                      short levelRunLengths[], short levels[]);
```

`layout` A reference to the layout shape whose information you need.

`startOffset`
The edge offset in source text preceding the first character you want to retrieve from the layout shape.

`endOffset` The edge offset in source text following the final character you want to retrieve from the layout shape.

`text` A pointer to a text string. On return, the specified portion of the text of the layout shape.

`styleRunCount`
A pointer to a `short` value. On return, the number of entries in the style list.

`styleRunLengths`
An array of `short` values. On return, the array contains the number of bytes of text associated with each entry in the `styles` parameter.

`styles` An array of style-object references. On return, the array is a style list associated with the `text` parameter.

`levelRunCount`
A pointer to a `short` value. On return, the number of entries in the `levels` parameter.

`levelRunLengths`
An array of `short` values. On return, the array contains the number of bytes per level associated with each entry in the `levels` parameter.

`levels` An array of `short` values. On return, the array of direction-level codes associated with the text.

function result The byte count of the part of the text specified, which may not be equal to the character count or the glyph count, depending on the character codes or glyph codes used and the platform.

DESCRIPTION

The `GXGetLayoutParts` function queries a layout shape and retrieves a portion of the information from the shape, such as the style runs or layout options. For example, if a layout shape has three styles attached to it, you can retrieve one of those styles, rather than all three.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
illegal_type_for_shape
index_is_less_than_zero
parameter_out_of_range
inconsistent_parameters
```

SEE ALSO

To change the geometry of an existing layout shape by replacing a portion of one or more of its arrays, use the `GXSetLayoutParts` function, described next.

To get one or more complete arrays from the geometry of an existing layout shape, use the `GXGetLayout` function, described on page 5-34. To change an existing layout shape by replacing an entire array in its geometry, use the `GXSetLayout` function, described on page 5-36.

To get a portion of the geometry of an existing layout shape and put that portion into another layout shape, use the `GXGetLayoutShapeParts` function, described on page 5-42. To change the geometry of an existing layout shape by inserting the geometry of another typographic shape, use the `GXSetLayoutShapeParts` function, described on page 5-44.

GXSetLayoutParts

You can use the `GXSetLayoutParts` function to change a portion of the text, styles, or direction-levels arrays in the geometry of a layout shape.

```
void GXSetLayoutParts(gxShape layout, gxByteOffset oldStartOffset,
                      gxByteOffset oldEndOffset,
                      long newTextRunCount,
                      const short newTextRunLengths[],
                      const void *newText[],
                      long newStyleRunCount,
                      const short newStyleRunLengths[],
                      const gxStyle newStyles[],
                      long newLevelRunCount,
                      const short newLevelRunLengths[],
                      const short newLevels[]);
```

`layout` A reference to the layout shape you want to modify.

`oldStartOffset`
The edge offset in the source text preceding the first character to replace.

`oldEndOffset`
The edge offset in the source text following the last character to replace.

`newTextRunCount`
The number of text runs supplied in the `newText` parameter.

`newTextRunLengths`
An array containing the byte length of each text run in the `newText` parameter.

`newText` An array of pointers to runs of text. If you pass `nil` for this parameter, QuickDraw GX uses the default text object.

`newStyleRunCount`
The number of styles in the style list.

`newStyleRunLengths`
An array containing the byte length of each style run.

`newStyles` An array of references to style objects, one for each style run. If you pass `nil` for this parameter, QuickDraw GX uses the default style object. If you pass a non-`nil` value for this parameter, then any `nil` entries in the array also refer to the shape's style.

`newLevelRunCount`
: The number of direction-level runs.

`newLevelRunLengths`
: An array containing the byte lengths of each direction-level run.

`newLevels`
: An array of nested direction levels that control text direction within the layout shape.

DESCRIPTION

The `GXSetLayoutParts` function sets or changes parts of the geometry of a layout shape. You can make changes to an array in the layout shape without sending the old values in addition to the new values. For example, if you want to change the text “ABC” to “ABC DEF”, send only the string “DEF”. You must also set the `oldStartOffset` and `oldEndOffset` parameters to the character offset of the last character offset, in the source text, of the original string; in the previous example, both parameters should be set to 3.

Any new values you add must be consistent with values already in the shape, unless you are discarding the old values. For example, if you change the text in the layout shape, the new text must be consistent with the values of the `newTextRunCount` and `newTextRunLengths` parameters, or you must enter new values for these parameters.

If you don’t want to change one component of a layout (text, styles, or levels), you can specify 0, `nil`, and `nil` for the corresponding count, run lengths, and array parameters.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
length_is_less_than_zero
index_is_less_than_zero
inconsistent_parameters
count_is_out_of_range
count_is_less_than_zero
```

Warnings

```
shape_access_not_allowed
shape_contains_invalid_data
```

SEE ALSO

For an example of how to use the `GXSetLayoutParts` function, see “Changing Parts of an Existing Layout Shape” beginning on page 5-20.

To get a portion of one or more arrays from the geometry of an existing layout shape, use the `GXGetLayoutParts` function, described on page 5-38.

To get one or more complete arrays from the geometry of an existing layout shape, use the `GXGetLayout` function, described on page 5-34. To change the geometry of an existing layout shape by replacing an entire array in the geometry, use the `GXSetLayout` function, described on page 5-36.

To get a portion of the geometry of an existing layout shape and put that portion into another layout shape, use the `GXGetLayoutShapeParts` function, described on page 5-42. To change the geometry of an existing layout shape by inserting the geometry of another typographic shape, use the `GXSetLayoutShapeParts` function, described on page 5-44.

Extracting or Inserting Parts of a Layout Shape

If you want to get part of the geometry of a layout shape and put it into another layout shape, use the `GXGetLayoutShapeParts` function. If you want to insert the geometry of another typographic shape, whether a text shape, glyph shape, or layout shape into a layout shape, use the `GXSetLayoutShapeParts` function.

GXGetLayoutShapeParts

You can use the `GXGetLayoutShapeParts` function to extract a copy of a specified range of the geometry of a layout shape and encapsulate it in another layout shape.

```
gxShape GXGetLayoutShapeParts(gxShape layout,
                              gxByteOffset startOffset,
                              gxByteOffset endOffset,
                              gxShape dest);
```

`layout` A reference to the layout shape containing the geometry you want to use.

`startOffset`
The edge offset in the source text preceding the starting character to retrieve from the layout shape.

`endOffset` The edge offset in the source text following the ending character to retrieve from the layout shape.

`dest` A reference to the layout shape that will receive the extracted geometry. If you set this parameter to `nil`, a new layout shape is returned as the function result. Even if it's not set to `nil`, the function result is still a copy of the old shape's reference.

function result A reference to the layout shape referenced by the `dest` parameter or to a new layout shape if the value of the `dest` parameter is `nil`.

DESCRIPTION

The `GXGetLayoutShapeParts` function extracts a copy of a portion of the layout shape in the `layout` parameter, including the text, styles, and levels arrays, and copies the geometry into an existing layout shape, specified by the `dest` parameter. The function returns a reference to the layout shape specified by the `dest` parameter

or a new layout shape, if `dest` is set to `nil`. The extracted data is bounded by the `startOffset` and `endOffset` values, which refer to byte offsets, in the source text, of the original layout shape.

If you want to put the copy in an existing layout shape, put a reference to that layout shape in the `dest` parameter; otherwise, leave that parameter set to `nil`, and the function returns a reference to a new layout shape in the function result.

SPECIAL CONSIDERATIONS

The `GXGetLayoutShapeParts` function is analogous to the `GXGetShapeParts` function, described in *Inside Macintosh: QuickDraw GX Objects.* However, the `GXGetLayoutShapeParts` function uses zero-based indexing and two offsets to mark a section within a layout shape; the `GXGetShapeParts` function uses 1-based indexing, a single offset, and a count to mark a section within a shape.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
illegal_type_for_shape
index_is_less_than_zero
parameter_out_of_range
inconsistent_parameters
```

SEE ALSO

For an example of the use of `GXGetLayoutShapeParts`, see “Extracting a Layout Shape From Part of an Existing Layout Shape” on page 5-23.

To change the geometry of an existing layout shape by inserting the geometry of another typographic shape, use the `GXSetLayoutShapeParts` function, described next.

To get one or more entire arrays from the geometry of an existing layout shape, use the `GXGetLayout` function, described on page 5-34. To change an existing layout shape by replacing an entire array in its geometry, use the `GXSetLayout` function, described on page 5-36.

To get a portion of one or more arrays from the geometry of an existing layout shape, use the `GXGetLayoutParts` function, described on page 5-38. To change the geometry of an existing layout shape by replacing a portion of one or more of its arrays, use the `GXSetLayoutParts` function, described on page 5-40.

GXSetLayoutShapeParts

You can use the `GXSetLayoutShapeParts` function to replace the geometry in a layout shape with another typographic shape's geometry.

```
void GXSetLayoutShapeParts(gxShape layout,
                           gxByteOffset startOffset,
                           gxByteOffset endOffset,
                           gxShape insert);
```

`layout` A reference to the layout shape whose geometry you want to edit.

`startOffset`
The edge offset in the source text preceding the starting character to retrieve from the layout shape.

`endOffset` The edge offset in the source text following the ending character to retrieve from the layout shape.

`insert` The typographic shape whose geometry you want to insert. This shape may be a text, glyph, or layout shape. Note: the whole shape's geometry is inserted; the start and end offsets refer to the edited shape only.

DESCRIPTION

The `GXSetLayoutShapeParts` function inserts the entire text of the shape specified by the `insert` parameter into the layout shape specified by the `layout` parameter. The inserted text may have its own styles; however, you cannot add positions and tangents arrays of a glyph shape to a layout shape.

The `startOffset` and `endOffset` parameters are the starting offset and ending offset in the layout shape where you want to insert the new geometry. As long as the values of `startOffset` and `endOffset` are equal, `GXSetLayoutShapeParts` performs the insertion. If they are not equal, it replaces the old text between `startOffset` and `endOffset`.

If the value of `insert` is not a text, glyph, or layout shape, the function returns the warning `illegal_type_for_shape`.

SPECIAL CONSIDERATIONS

The `GXSetLayoutShapeParts` function is analogous to the `GXSetShapeParts` function, described in *Inside Macintosh: QuickDraw GX Objects*. However, the `GXSetLayoutShapeParts` function uses 0-based offsets to mark a section within a layout shape; `GXSetShapeParts` uses 1-based indexing, an offset, and a count to mark a section within a shape.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
inconsistent_parameters
count_is_less_than_zero
count_is_out_of_range
index_is_less_than_zero
length_is_less_than_zero
```

Warnings

```
shape_access_not_allowed
shape_contains_invalid_data
illegal_type_for_shape
```

SEE ALSO

For an example of how to use the `GXSetLayoutShapeParts` function, see "Changing Parts of an Existing Layout Shape" beginning on page 5-20.

To get a portion of the geometry of an existing layout shape and put that portion into another layout shape, use the `GXGetLayoutShapeParts` function, described on page 5-42.

To get one or more entire arrays from the geometry of an existing layout shape, use the `GXGetLayout` function, described on page 5-34. To change the geometry of an existing layout shape by replacing an entire array in the geometry, use the `GXSetLayout` function, described on page 5-36.

To get a portion of one or more arrays from the geometry of an existing layout shape, use the `GXGetLayoutParts` function, described on page 5-38. To change the geometry of an existing layout shape by replacing a portion of one or more of its arrays, use the `GXSetLayoutParts` function, described on page 5-40.

Obtaining Glyph Information From a Layout Shape

You can obtain information about the glyphs in a layout shape by using the `GXGetLayoutGlyphs` function.

GXGetLayoutGlyphs

You can use the `GXGetLayoutGlyphs` function to get information about each of the glyphs in a layout shape.

```
long GXGetLayoutGlyphs(gxShape layout, gxGlyphcode glyphs[],
                       gxPoint positions[], long advance[],
                       gxPoint tangents[], long *runCount,
                       short styleRuns[], gxStyle glyphStyles[]);
```

`layout`	A reference to the layout shape whose glyph information you need.
`glyphs`	A pointer to an array of glyph codes. On return, the array contains the glyph codes for all the glyphs in the layout shape.
`positions`	An array of `gxPoint` values. On return, the array contains the positions of each of the glyphs in the layout shape.
`advance`	An array of `long` values. On return, the array contains the advance bits for the glyphs in the layout shape with the first bit on, the others off.
`tangents`	An array of `gxPoint` values. On return, the array contains the tangents for the glyphs in the layout shape. If the layout contains runs with the `gxVerticalText` flag set, the array contains the tangents for the individual glyphs; otherwise it is filled with [1.0,0.0].
`runCount`	A pointer to a `long` value. On return, the value is the number of style runs in the glyph shape that is equivalent to this layout shape.
`styleRuns`	An array of `short` values. On return, the array contains the number of glyphs in each style run.
`glyphStyles`	An array of style-object references. On return, the array contains the style list for the glyph shape that is equivalent to this layout shape.

function result The number of glyphs in the shape.

DESCRIPTION

The `GXGetLayoutGlyphs` function provides access to a layout shape's glyph information without requiring you to first convert the layout shape to a glyph shape. It returns the glyph code and positioning information for each glyph, in a form analogous to the information returned for glyph shapes by the `GXGetGlyphs` function.

This function treats the layout shape as if it were a glyph shape. The `advance` and the `positions` parameters return the advance bits and positions arrays, respectively, which contain information about whether the positions stored in the positions array (one position per glyph in the shape) are absolute or relative. For layout shapes, the first bit of the advance bits array is always absolute, and the remaining bits are relative. The `tangents` parameter contains as many tangents as there are glyphs in the shape and determines the direction and scaling of the individual glyph. For more information about advance bits, positions, and tangents, see the chapter "Glyph Shapes" in this book.

Besides glyph identity and positioning, this function returns style-run information indexed by glyph (rather than by character code, as is true for other functions). See "Getting Glyph Information From a Layout Shape" on page 5-27 for a demonstration of the difference between the `GXGetLayoutGlyphs` function and the `GXGetLayout` function, which returns information based upon the character codes stored in the shape.

If you pass `nil` for a parameter, `GXGetLayoutGlyphs` does not return values in that parameter.

ERRORS, WARNINGS, AND NOTICES

Errors
`shape_is_nil`

Notices (debugging version)
`glyph_tangents_have_no_effect`

SEE ALSO

This function is similar in form and purpose to the `GXGetGlyphs` function, described in the chapter “Glyph Shapes” in this book. The advance bits and tangent arrays are properties of the glyph shape, also described in the chapter “Glyph Shapes” in this book.

Summary of Layout Shapes

Constants and Data Types

```
typedef long gxByteOffset;
```

Layout Options Structure

```
typedef struct {
  Fixed               width;
  fract               flush;
  fract               just;
  gxLayoutOptionsFlags  flags;
  gxLineBaselineRecord  *baselineRec;
  } gxLayoutOptions;

#define gxNoLayoutOptions 0
#define gxLineIsDisplayOnly   0x00000100
#define gxMaxRunLevel 15
#define gxFlushLeft 0
#define gxFlushCenter (frac1/2)
#define gxFlushRight frac1
#define gxNoJustification 0
#define gxFullJustification frac1
typedef unsigned long gxLayoutOptionsFlags;
```

Layout Shape Functions

Creating and Drawing Layout Shapes

```
gxShape GXNewLayout         (long textRunCount,
                             const short textRunLengths[],
                             const void *text[], long styleRunCount,
                             const short styleRunLengths[],
                             const gxStyle styles[], long levelRunCount,
                             const short levelRunLengths[],
                             const short levels[],
                             const gxLayoutOptions *layoutOptions,
                             const gxPoint *position);
```

```
void GXDrawLayout           (long textRunCount,
                             const short textRunLengths[],
                             const void *text[], long styleRunCount,
                             const short styleRunLengths[],
                             const gxStyle styles[], long levelRunCount,
                             const short levelRunLengths[],
                             const short levels[],
                             const gxLayoutOptions *layoutOptions,
                             const gxPoint *position);
```

Getting and Setting the Geometry of a Layout Shape

```
long GXGetLayout            (gxShape layout, void *text,
                             long *styleRunCount, short styleRunLengths[],
                             gxStyle styles[], long *levelRunCount,
                             short levelRunLengths[], short levels[],
                             gxLayoutOptions *layoutOptions,
                             gxPoint *position);
void GXSetLayout            (gxShape layout, long textRunCount,
                             const short textRunLengths[],
                             const void *text[], long styleRunCount,
                             const short styleRunLengths[],
                             const gxStyle styles[], long levelRunCount,
                             const short levelRunLengths[],
                             const short levels[],
                             const gxLayoutOptions *layoutOptions,
                             const gxPoint *position);
```

Getting and Setting Portions of a Layout Shape's Geometry

```
long GXGetLayoutParts       (gxShape layout, gxByteOffset startOffset,
                             gxByteOffset endOffset, void *text,
                             short *styleRunCount, short styleRunLengths[],
                             gxStyle styles[], short *levelRunCount,
                             short levelRunLengths[], short levels[]);
void GXSetLayoutParts       (gxShape layout, gxByteOffset oldStartOffset,
                             gxByteOffset oldEndOffset,
                             long newTextRunCount,
                             const short newTextRunLengths[],
                             const void *newText[], long newStyleRunCount,
                             const short newStyleRunLengths[],
                             const gxStyle newStyles[],
                             long newLevelRunCount,
                             const short newLevelRunLengths[],
                             const short newLevels[]);
```

Extracting or Inserting Parts of a Layout Shape

```
gxShape GXGetLayoutShapeParts
                            (gxShape layout, gxByteOffset startOffset,
                             gxByteOffset endOffset, gxShape dest);
void GXSetLayoutShapeParts  (gxShape layout, gxByteOffset startOffset,
                             gxByteOffset endOffset, gxShape insert);
```

Obtaining Glyph Information From a Layout Shape

```
long GXGetLayoutGlyphs      (gxShape layout, gxGlyphcode *glyphs,
                             gxPoint positions[], long advance[],
                             gxPoint tangents[], long *runCount,
                             short styleRuns[], gxStyle glyphStyles[]);
```

CHAPTER 6

Typographic Styles

Contents

This chapter describes how typographic shapes use the style object. Read this chapter if you want to create or use any styles with QuickDraw GX typographic shapes.

Before reading this chapter, you should be familiar with the information in the chapter "Introduction to QuickDraw GX Typography" in this book. You should also be familiar with typographic shapes, as discussed in the chapter "Typographic Shapes" in this book.

For more information on style objects, see the chapter "Style Objects" in *Inside Macintosh: QuickDraw GX Objects*.

This chapter does *not* cover style object properties used exclusively by layout shapes, such as run controls and the kerning adjustments array, which are also part of the style object. For more information about style properties used by layout shapes, see the chapter "Layout Styles" in this book.

This chapter introduces the properties of the style object and how they are associated with QuickDraw GX typographic shapes. It then shows how to use QuickDraw GX functions to

- get and set the style object's properties, such as its font, text face, text size, and text attributes
- create a text face in order to create stylistic variations of a font
- orient text vertically instead of horizontally

About Typographic Styles

Typographic styles exist to provide information about typographic shapes. Each QuickDraw GX typographic style associated with a particular typographic shape defines much of that shape's appearance, such as what font the shape uses and where it is placed. Typographic styles also describe what glyphs are represented by a font's character encoding and its text size, as well as the stylistic variations of a font and its text faces.

Typographic styles are device-independent. The styles of a typographic shape are not affected by the properties of the display device for which the shape is drawn.

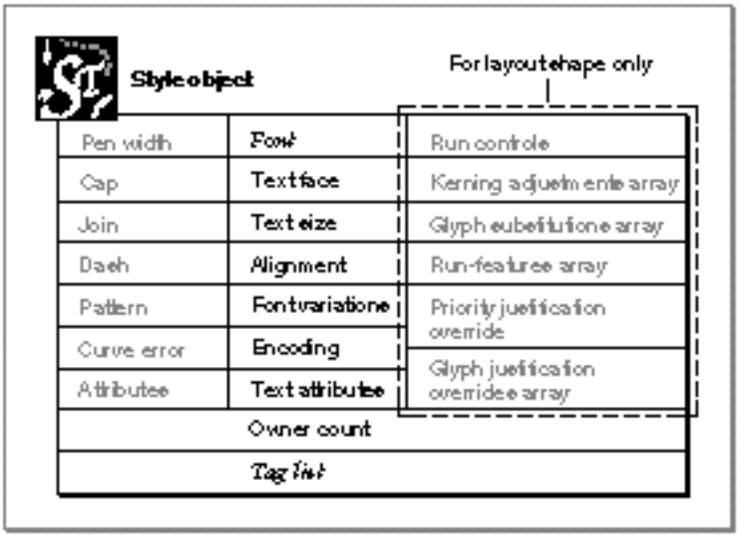
Although a typographic style is the same as a geometric style, typographic shapes—text, glyph, and layout shapes—use different properties from the style object than those used by geometric shapes, such as caps, joins, and dashes.

Style Properties Associated With Typographic Shapes

The interface to style objects is entirely procedural. You manipulate the information in a style object by modifying its properties using QuickDraw GX functions.

Style objects have 22 accessible properties, as shown in Figure 6-1. Note that, because a style is an object and not a data structure, the order of the properties as shown in Figure 6-1 is completely arbitrary.

Figure 6-1 The style object used by all typographic shapes



Thirteen of the style object's properties pertain only to typographic styles—that is, styles associated with typographic shapes. Seven apply to all typographic shapes:

- **Font.** A reference to the font to use in drawing the text of this shape.
- **Text face.** The text face—the constructed stylistic variation from plain text—to apply in drawing the text of this shape.
- **Text size.** The size, in typographic points (72 per inch), at which to draw the text of this shape.
- **Alignment.** The alignment value to use when drawing the text of this shape. Text may be left-aligned, right-aligned, anywhere between the two alignments (such as centered), or fully justified.
- **Font variations.** The list of font variations—stylistic variations built into the font—available for drawing the text of this shape.
- **Encoding.** The type of character encoding used to represent the text of this shape, as well as its script and language.
- **Text attributes.** The set of flags that allow you to specify how QuickDraw GX alters glyph outlines or chooses the proper metrics for horizontal or vertical text.

Most of the properties of the style object associated with geometric shapes—pen size, cap, join, dash, curve error, and style attributes—can be set in a style object used by a typographic shape, but they do *not* affect the appearance of the shape. (Patterns are an exception; they do affect the appearance of a typographic shape. See Figure 6-20

on page 6-33 for example.) However, you can take advantage of the geometric style properties by using the text face property of the style object. In this way, you can apply cap, join, dash, and pattern properties to your typographic shape. See the section "Applying Patterns and Dashes to Text Faces" on page 6-32.

Note

Both glyph shapes and layout shapes may have arrays of styles in their shape geometry and therefore do not necessarily use the style object associated with the shape object. However, any operation you can perform on the shape's style object can also be performed on the styles stored in the shape's geometry. (Note that any `nil` member of an array causes the shapes's style object to be used.) u

QuickDraw GX provides functions for manipulating each of these style object properties. The properties and functions pertaining to typographic styles are described in the following sections.

Font

The font property of style objects specifies the font or font family of a style object. QuickDraw GX provides your application with functions for retrieving and specifying font information for a style object, as well as retrieving and specifying information for a style object associated with a particular shape. The functions for getting and setting the font of a style object are described in the "Typographic Styles Reference" section of this chapter.

For more information about fonts—specifically, encodings and font variations—see the chapter "Font Objects" in this book.

Text Face

If your application needs to apply a typestyle to text, you have three options. (A **typestyle** is a specific variation in the appearance of a glyph that can be applied consistently to all the glyphs in a font family.) You can

- use a font specifically designed for that typestyle, as described in the chapter "Font Objects"
- use a font that has a font variation for that typestyle, as described in the chapter "Font Objects"
- apply a text face created by your application to that text

The first two methods involve using information already present in the font. If there isn't a separate font in the appropriate typestyle, or the font doesn't contain the font variation that the user needs, your application can use text faces to create the desired typestyle. A **text face** is a typestyle created by an algorithmic method, which allows you to control the appearance and placement of glyphs in a font.

A text face consists of a mapping, which affects the advance widths and interglyph spacing—but not the shape of the glyphs—and a number of optional face layers, which describe the appearance of the glyphs.

A **face layer** specifies the manner in which the glyphs are drawn. When all of the face layers are combined, they form the visual composite. This creates the particular look of the text face—that is, bold, oblique, underline, condensed, extended, or a more unusual pattern, dash, join, or other style. If a text face has multiple face layers, the layers combine to form the final text face, as shown in Figure 6-2.

A face layer contains the following elements:

- a shape fill
- an optional style object
- an optional transform object
- bold values for the x and y directions
- layer flags

The Shape Fill

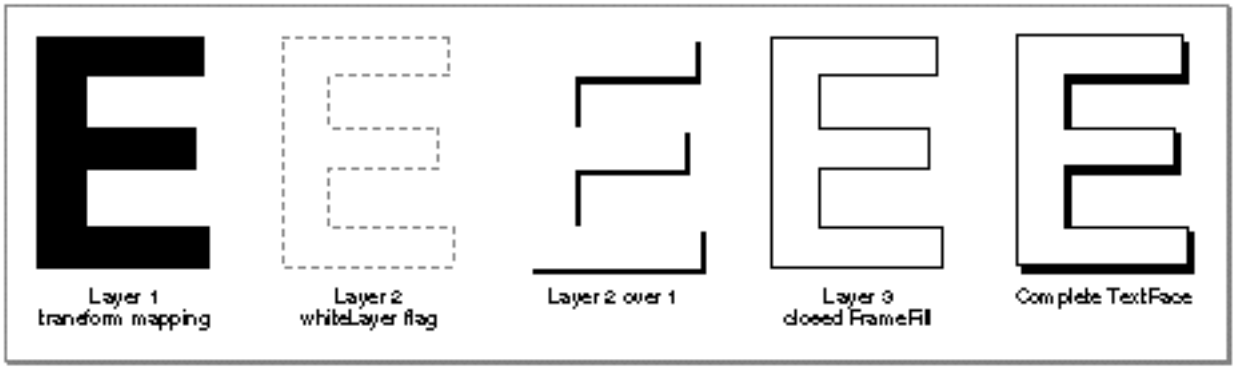
The shape fill determines the kind of geometry—that is, fill or frame—used by the layer. If the fill is an open frame or closed frame fill, then the glyph is converted to a path shape.

For more information about shape fills, see the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects.*

The Style

The style in a face layer allows the text face to take advantage of the geometric, as opposed to typographic, properties of the style object. The only restriction is that the text face’s style cannot have its own text face.

Figure 6-2 Face layers combined to form the visual composite of a Roman “E”



The style is the overriding style for the text face. The values in this style are interpreted in a unique way because of the text face. The style's pen size, pattern, dash, and join are scaled by a factor corresponding to the point size of the text.

The fields of the outline style are multiplied by the text size of the text being drawn. Thus, a pattern in an outline style is scaled relative to the text being patterned, so that, for example, a 30.0-point "A" will have a pattern shape twice as large as a 15.0-point "A".

The bulleted items listed are the fields of the outline style that are scaled. Wherever the text "scaled by textSize" appears, the text size refers to the size of the text being drawn. This text size comes from the text size field of the text's style. The scaled fields include:

- Pen width: scaled by the text size
- Dash (if present): advance scaled by the text size; dash shape scaled in X only by the text size
- Pattern (if present): u and v vectors scaled in X and Y by the text size; pattern shape scaled in X and Y by the text size
- Start and End Caps (if present): scaled in X and Y by the text size.
- Join (if present and join type is `gxShapeJoin`): join shape is scaled in X and Y by the text size.
- Text Size: scaled by the text size

If the outline style's text size is 1.1, for example, the text will end up being drawn with a point size 10 percent larger than if there were no text face. A 12.0 point text will be drawn at 13.2 points.

You can use the style in the text face to pattern a typographic shape. See "Applying Patterns and Dashes to Text Faces" on page 6-32.

The Transform

The transform in the face layer allows you to produce oblique, condensed, and extended typestyles, as well as unusual special effects. You can change, skew, and scale the text face's transform as you would any other transform.

For more information about the transform object, its properties, and the `GXNewTransform`, `GXSkewTransform`, `GXScaleTransform`, and `GXSetTransformMapping` functions, see *Inside Macintosh: QuickDraw GX Objects.*

The Bold Outset

The x and y values in the `boldOutset` field of the face layer scale the thicknesses of the glyphs in the displayed shape. The values for bold are treated as though they are for a 1-point font. For example, to get 3 points worth of bold at 48-point text, you set the bold outset to `ff(3)/ff(48)`.

Layer Flags

The **layer flags** describe the characteristics of one layer of a text face. They are used primarily to determine the underlining capabilities of the text face. By setting the layer flags, you can affect where the underlining goes. Figure 6-3 indicates underlining a glyph with tangent values.

Figure 6-3 An underlined glyph with tangent values

QuickDraw GX allows your application to

- underline all of the glyphs in that text face
- underline only the glyphs that make up words (skipping whitespace glyphs)
- connect text of different text faces

Figure 6-4 shows underlining with intervals, and with a style change from italic to Roman.

Figure 6-4 Underlining with interval and with style changes

A layer flag can also indicate whether the layer adds to or subtracts from the previous layers.

Figure 6-5 shows underlining vertical text, with centering.

Figure 6-5 Underlining vertical text through its center

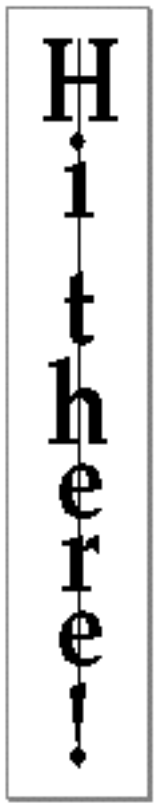


Table 6-1 describes values for layer flags. When you pass a value for one of the flags, QuickDraw GX performs the described action.

Table 6-1 Layer flag values and descriptions

Flag	Value	Description
`gxUnderlineAdvanceLayer`	1	Draws an underline from the beginning of the text that shares one text face to the end, including white spaces.
`gxSkipWhiteSpaceLayer`	2	Does not draw a line under glyphs that have no contours (such as the space character) if the `gxUnderlineAdvanceLayer` bit is also set.
`gxUnderlineIntervalLayer`	4	Extends the underline through the gaps between text. If you set this bit, you must also set the `gxStringLayer` bit.
`gxUnderlineContinuationLayer`	8	Draws an underline across text of different style runs. If you set this bit, you must also set the `gxStringLayer` bit. Also, you must set the `gxUnderlineAdvanceLayer` flag; if you do not, you get an error.
`gxWhiteLayer`	16	Subtracts portions of previously drawn layers. You can set this bit only for the second or greater layer.
`gxClipLayer`	32	Clips the underline with the outline as a glyph.
`gxStringLayer`	64	Connects the text of different text faces and style runs.

Text Size

The text size property of style objects specifies the size, in fractional typographic points, of the text of a style object.

QuickDraw GX provides functions to retrieve and specify the text size from a style object as well as functions to retrieve and specify the text size for the style object associated with a particular shape. The functions for getting and setting the text size of a style object are described later in the “Typographic Styles Reference” section of this chapter.

Alignment

Alignment is the process of placing text in relation to one or both margins, which are the left and right sides (or top and bottom sides) of the text area. Text can be aligned left, right, center, or at full justification (full justification allows you to “stretch” or “shrink” a line of text to fit within a given width). Alignment should be used only to compensate for the differences between ideal measurements of characters and device-specific measurements of characters.

The alignment is set by getting and setting the alignment values of the style object. Table 6-2 describes some alignment values of the style object.

Table 6-2 Alignment values and descriptions

Alignment	Value	Description
Left	0.0	Text is drawn to the right of the left margins (or below the top margin, for vertical text).
Right	1.0	Text is drawn to the left of the right margin (or above the bottom margin, for vertical text).
Center	0.5	Text is drawn between the left and right margins with an equal amount of space on either side.
Full Justification	–1.0	Text is evenly distributed between the margins (left and right for horizontal text or top and bottom for vertical text) because the white space on the line is distributed between the words and, to a lesser extent, between the glyphs in the words.

Note

The alignment value in the style object is used only by text and glyph shapes. The layout shape has its own method of justifying and aligning text, as described in the chapter “Layout Line Control.” u

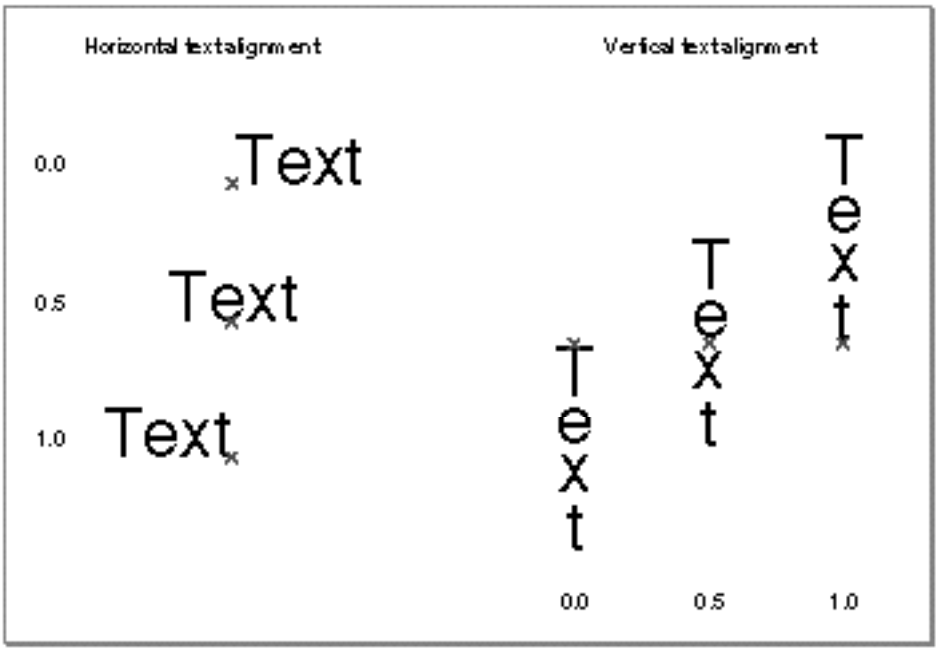
As shown in Table 6-2, several values are predefined for different types of alignment. However, any fractional value between 0.0 and 1.0 is legal for alignment and represents a gradation of one of the types of alignment. (Note that values between 0.0 and –1.0 are illegal.) For example, a value of 0.25 indicates that QuickDraw GX should draw the shape one quarter of the way from the left text position, or one quarter of the way from the left and right edges.

Figure 6-6 compares alignment values for horizontal and vertical text.

IMPORTANT

If you use text and glyph shapes with non-Roman scripts, you are not assured of getting the proper results when you use these alignment values. For example, if you use a script such as Arabic that relies on kashidas (extending bars), then full justification may break that alignment apart. To guarantee correct results in non-Roman cases, you should use layout shapes. s

Figure 6-6 Comparing alignment values for horizontal and vertical text



Full Justification

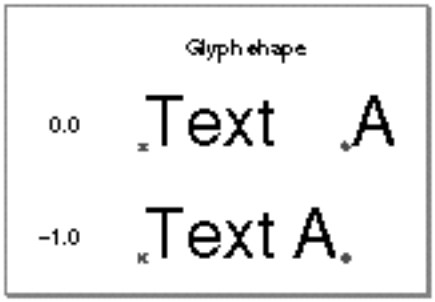
In full justification, the glyphs are distributed evenly between the margins, the left and right margins for horizontal text or the top and bottom margins for vertical text. Full justification does not affect text or layout shapes; it affects only glyph shapes that have two absolute positions. The first absolute position is first entry in the glyph shape's position array; the second absolute position is the final entry in the glyph shape's position array.

For each pair of absolute positions in the glyph shape, the first position specifies the left alignment for the glyph that corresponds to that position; the second of the pair specifies the right alignment for the glyph that corresponds to that position. QuickDraw GX justifies the remaining glyphs between the pair of glyphs corresponding to the absolute positions. The glyphs are justified using the glyph advance and the relative position.

QuickDraw GX justifies the whitespace glyphs (whitespace glyphs are those with no contours, such as spaces between characters) after justifying the other glyphs. If there is a difference between the advance after the next-to-the-last glyph and the initial position of the last glyph, the difference is divided by the number of whitespace glyphs. If there are no whitespace glyphs, or if changing their width is not enough to compensate for the mismatch in widths, the positions of all other characters are also adjusted.

Figure 6-7 shows the absolute positions for two different alignment values.

Figure 6-7 Comparing alignment values for full justification



Font Variations

The font variations property of a style object specifies the font variations that will be applied to the style's font.

QuickDraw GX provides your application with functions to retrieve and specify the font variations from a style object, as well as to retrieve and specify the font variations associated with a particular shape. See "Getting and Setting the Font Variations of a Style Object" on page 6-51 and the chapter "Font Objects" in this book.

The QuickDraw GX Font Variation Suite

A typographic style may specify the values for any number of variation axes. (A variation axis has a name that identifies the typestyle which the axis represents, a set of maximum and minimum values for the axis, and the default value of the axis.)

A style's font might not support, however, all of the specified axes, or support axes not explicitly mentioned in the style. In addition, the style may specify a value for an axis that is beyond the supported range for that axis by the style's font. QuickDraw GX manages all of these cases by converting the style's list of variations into a canonical form before using them. This canonical form is called a **font variation suite.**

The canonical form is a complete listing of every axis supported in the font (see the `GXCountFontVariations` function described in the "Font Objects" chapter in this book), in the order specified by the font (see the `GXGetFontVariation` function). Each axis is given a value. If the style specifies a value for an axis, the value is pinned to lie within (inclusively) the font's minimum and maximum value for that axis. Axes not specified in the style are set to their default values.

The functions GXGetStyleFontVariationSuite and GXGetShapeFontVariationSuite return a gxFontVariation array in this canonical form. For more information, see “Retrieving the Elements in a Font Variation Suite” on page 6-55.

The functions return the number of elements in the variation suite for the font specified by the style or shape. This number is the same as the number of font variation axes returned by GXCountFontVariations, described in the chapter “Font Objects” in this book. If the variations parameter is not nil, the variations specified in the style are converted into their canonical form, and then copied into variations.

Font Metrics

The font metrics property allows you to retrieve font metrics for specified style objects or style objects associated with a shape object.

You can retrieve the font metrics, including line spacing and caret angle, for a style object. You can also retrieve the font metrics for the style object associated with a shape object. You can retrieve the font metrics for the style object associated with a shape, taking into account the shape’s transform or the mappings on the specified view port and view device. See “Retrieving Font Metrics” on page 6-57.

Encoding

The encoding property of a style object represents a combination of its platform, script, and language. Platforms, scripts, and languages are described in the chapter “Font Objects” in this book.

QuickDraw GX provides your application with functions to retrieve and specify the encoding information of a style object, as well as to retrieve and specify the encoding information for the style object associated with a particular shape. The functions are described in “Getting and Setting the Encoding of a Style Object” on page 6-61.

For more information on encoding, see the chapters “Typographic Shapes” and “Font Objects” in this book.

Text Attributes

Each style object has a set of text attributes that modify the behavior of the style object associated with typographic shapes. **Text attributes** are a collection of flags which individually affect different properties of typographic styles.

Table 6-3 describes text attributes and their values.

As shown in Table 6-3, the text attribute flags allow you to specify how QuickDraw GX alters glyph outlines or chooses the proper types of metrics for horizontal or vertical text. For example, the gxAutoAdvanceText text attribute determines the difference between the advance width of the glyph from the font and the advance width of the same glyph with the text face applied. It then adds this difference to the original advance width.

Table 6-3 Text attributes and their values

Attribute	Value	Description
gxAutoAdvanceText	0x0001	QuickDraw GX automatically changes the advance width of a glyph to match the change of glyph width due to a text face.
gxNoContourGridText	0x0002	QuickDraw GX doesn't use hinted outlines before displaying the text—that is, it instructs the scaler not to use hints when imaging. If you set this bit, you must also set the gxNoMetricsGridText bit.
gxNoMetricsGridText	0x0004	Uses the ideal metrics to space the glyphs in a typographic shape—that is, instructs the scaler not to use hints when measuring the character.
gxAnchorPointsText	0x0008	Includes data for all the outline's points. QuickDraw GX's default action is to return a basic outline, removing points that do not affect the shape, such as single-point contours. If set, and a typographic shape is converted to a path, all points are returned.
gxVerticalText	0x0010	Uses the vertical advance and side bearing metrics. The default values are the horizontal advance and side bearing metrics.
gxNoOpticalScaleText	0x0020	QuickDraw GX automatically specifies an optical scale variation value equal to the style's text size, if the style's font supports optical scaling variations and the style does not already specify a value for this axis. If set, this bit prevents QuickDraw GX from specifying an optical scale variation value.

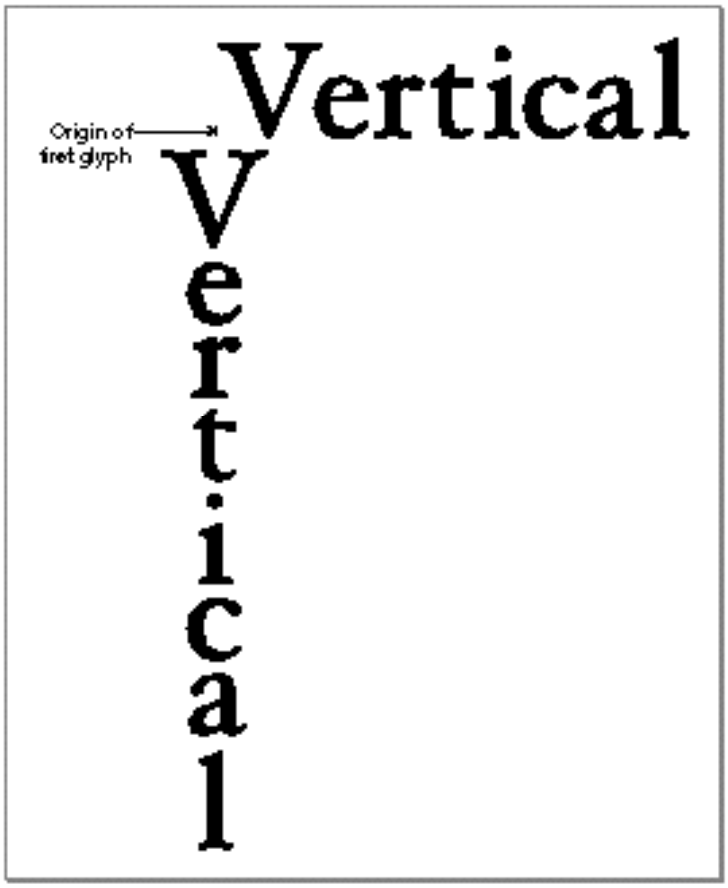
Given a specific point size and resolution, hinted outlines produce better-looking text for a specific point size and resolution by moving points or otherwise altering the outline. A hinted outline for a particular point size that is converted to a path or polygon shape may not respond to scaling as well as graphic objects or unhinted outlines do on a high-resolution display device. The gxNoContourGridText text attribute does not use hinted outlines before displaying the text.

The gxNoMetricsGridText text attribute uses the ideal metrics to space the glyphs in a typographic shape. Ideal metrics always give the same interglyph spacing on a high-resolution device, but you may get poorer spacing if you use ideal metrics on a low-resolution device such as the screen. Also, ideal metrics scale linearly with the shape's transform and device. (Your application may want to use ideal metrics for such features as linebreaking.) The alternative, device-specific metrics, change non-linearly with scaling factors in the transform and device.

When you convert a glyph to a path, the `gxAnchorPointsText` text attribute includes data for all the points on a glyph's outline if this bit is set. QuickDraw GX's default action is to return a basic outline, removing points that do not affect the shape, such as single-point contours.

The `gxVerticalText` text attribute uses the vertical advance and side bearing metrics. You should set this bit if you want QuickDraw GX to draw text vertically. The default values are the horizontal advance and side bearing metrics. Figure 6-8 shows an example of orienting text both vertically and horizontally.

Figure 6-8 Orienting text vertically and horizontally



Typographic Properties of the Default Style Object

When QuickDraw GX first creates a style object, that object has default characteristics defined by QuickDraw GX. The typographic properties of the default style object have the following values:

- A `nil` font reference, meaning that the default font is used for text. The default font is described in the chapter "Font Objects" in this book.
- A text size of 12.0

- A text attributes value of `gxNoAttributes` (0)
- A scale value (within the dash property) of 1.0
- A joins miter value of 1.0
- A value of 0 or `nil` for all other typographic properties

Using Typographic Styles

This section describes the basic method of manipulating the style object for typographic shapes. Unless otherwise noted, these methods apply to all three typographic shapes. For detailed information on using a specific typographic shape—text, glyph, or layout—see the chapter describing that particular shape.

This section describes how you can

- create text faces in your application
- set text attributes and orient the vertical text attribute
- apply patterns and dashes to text faces
- create unusual effects with text faces

For more information on setting font variations and encoding, see the chapter “Font Objects” in this book.

Creating Text Faces

You can include text faces in your application that create the basic typestyles: bold, italic, condensed, extended, outlined, and underlined. You may also want to include unusual text faces for users.

Note that your application must allocate enough memory to store the text face and all of the face layers of that text face. For example, to create a text face with three layers, you can use the following code:

```
gxTextFace*face;
long     numOfLayers = 3;

face = (gxTextFace *)NewPtr(sizeof(gxTextFace) +
       (numOfLayers - gxAnyNumber)* sizeof(gxFaceLayer));
face->faceLayers = numOfLayers;
```

You can then set the values of the advance mapping and the individual face layers.

Setting the Advance Mapping

The advance mapping in the text face affects the positions of the glyphs using the text face. It does not affect the size of the glyphs themselves or change the positions of the shape as stored in the shape's geometry.

Listing 6-1 is an example of a routine that shows advance mapping. Note that the translation component of the advance mapping is multiplied by the text size before being applied.

Listing 6-1 Advance mapping

```
static void ShowAdvanceMapping(void)
{
   gxTextFace mappingOnlyFace;
   ResetMapping(&mappingOnlyFace.advanceMapping);
   mappingOnlyFace.advanceMapping.map[2][1] = -fixed1/3;
   mappingOnlyFace.faceLayers = 0;

   GXSetShapeFace(GXGetDefaultShape (gxTextType),
            &mappingOnlyFace);
   GXDrawText(6, (unsigned char*) “Mapped”, nil);
}
```

Figure 6-9 shows the results of executing the code in Listing 6-1.

Figure 6-9 A shape with the advance mapping applied

Note
If you scale the advance mapping, you're scaling on a glyph-by-glyph basis. u

Setting a Face Layer

To modify the appearance of the text face, you can use the face layer in a text face. The various layers of the text face are composited on top of one another; the first face layer of the text face is the bottom most layer. Subsequent layers are added to or subtracted from previous layers.

The Transform

To produce italic, condensed, and extended typestyles, as well as unusual effects, you can use the transform in the face layer. You can change, skew, and scale the text face's transform as you would any other transform.

To create an italic text face, for example, you can use the following code in Listing 6-2.

Listing 6-2 Creating an italic text face

```
void ApplyItalicTextFace(gxShape target, Fixed italicValue)
{
   gxTextFace italicFace;

   ResetMapping(&italicFace.advanceMapping);
   italicFace.faceLayers = 1;
   italicFace.faceLayer[0].outlineFill = gxWindingFill;
   italicFace.faceLayer[0].flags = 0;
   italicFace.faceLayer[0].outlineStyle = nil;
   italicFace.faceLayer[0].outlineTransform = GXNewTransform();
   GXSkewTransform(italicFace.faceLayer[0].outlineTransform,
italicValue, 0, 0, 0);
   italicFace.faceLayer[0].boldOutset.x = 0;
   italicFace.faceLayer[0].boldOutset.y = 0;

   GXDisposeTransform(italicFace.facelayer[0].outlineTransform)
}

void DisplayItalicText(void)
{
   gxShape text = GXNewText(6, (unsigned char *)"Italic", nil);

   GXMoveShape(text, 0, ff(20));
   ApplyItalicTextFace(text, -fixed1/6);
   GXDrawShape(text);

   GXMoveShape(text, 0, ff(20));
   ApplyItalicTextFace(text, -fixed1/4);
   GXDrawShape(text);
```

```
   GXMoveShape(text, 0, ff(20));
   ApplyItalicTextFace(text, -fixed1/3);
   GXDrawShape(text);

   GXMoveShape(text, 0, ff(20));
   ApplyItalicTextFace(text, -fixed1/2);
   GXDrawShape(text);

   GXDisposeShape(text);
}
```

Figure 6-10 shows the results of executing the code in Listing 6-2.

Figure 6-10 An italic text face

To create a condensed or extended text face, scale the transform of the text face. The transforms of condensed text faces are scaled to less than 1; those of extended text faces are scaled to more than 1.

```
shrink = ff(60)/100;
face->faceLayer[0].outlineTransform = GXNewTransform();
GXScaleTransform(face->faceLayer[0].outlineTransform, shrink,
                 fixed1, 0, 0);

ScaleMapping(&face->advance mapping, shrink, fix1, 0, 0)
```

The preceding code produces the text face in Figure 6-11.

Figure 6-11 A condensed text face

You can combine these steps to produce a condensed-italic text face. You can also use the transform of a face layer in a text face with more than one layer to produce a drop shadow or other unusual text face.

Listing 6-3 creates a text face with two layers. The first layer is the drop shadow, which is drawn with a closed frame fill; the second layer does not alter the appearance of the text at all. Together, they produce the text face in Figure 6-12.

Listing 6-3 Creating a drop-shadow text face

```
ResetMapping (& layer2map);

layer1map.map[2][0] = fl(0.2);
layer1map.map[2][1] = fl(0.2);

face->faceLayer[0].flags = gxNoAttributes;
face->faceLayer[0].outlineFill = gxClosedFrameFill;
face->faceLayer[0].outlineStyle = nil;
GXSetTransformMapping(face->faceLayer[0].outlineTransform =
                    GXNewTransform(), &layer1map);
face->faceLayer[0].boldOutset.x = 0;
face->faceLayer[0].boldOutset.y = 0;

mapping  layer1map;

face->faceLayer[1].flags = gxNoAttributes;
face->faceLayer[1].outlineFill = gxWindingFill;
face->faceLayer[1].outlineStyle = nil;
face->faceLayer[1].outlineTransform = nil;
face->faceLayer[1].boldOutset.x = 0;
face->faceLayer[1].boldOutset.y = 0;
```

Figure 6-12 shows the results of executing the code in Listing 6-3.

Figure 6-12 A drop-shadow text face

For more information about the transform object, its properties, and the `GXNewTransform`, `GXSkewTransform`, `GXScaleTransform`, and `GXSetTransformMapping` functions, see *Inside Macintosh: QuickDraw GX Objects.*

The Bold Outset

The x and y values in the `boldOutset` field of the face layer scale the thicknesses of the glyphs in the displayed shape. The values for bold are treated as though they are for a 1-point font. For example, to get 3 points worth of bold at 48-point text, you set the bold outset to `ff(3)/48`.

You can produce a simple bold text face by setting the x value of the `boldOutset` field to any positive value other than 0 and setting the y value to 0. In this way, you can create several kinds of boldface text: bold, demibold, black, and so on. See Figure 6-13.

In general, bold fonts tend to get heavy in just the horizontal direction. You can use the y bolding as well.

IMPORTANT

You can decrease the amount of bold by using negative values in the bold outset field. s

Figure 6-13 Different values of boldface

Setting the Layer Flags

The layer flags primarily allow you to determine the underlining characteristics of the text face. You can also use the layer flags to create a “white layer.”

Listing 6-4 is a sample routine that shows how to create a simple underline text face.

Listing 6-4 Creating a simple underline text face

```
#include “graphics routines.h”
#include “math routines.h”

static ShowSimpleUnderline(void)
{
   gxPoint position = {0, ff(400)};
   gxShape text = GXNewText(9, (unsigned char *)"Underline",
            &position);

   gxTextFace underlineFace = {1, {{{fixed1, 0, 0},
            {0, fixed1, 0}, {0, 0, fract1}}},
            gxUnderlineAdvanceLayer, gxWindingFill, nil, nil,
            {0, 0}};

   GXSetShapeFace(text, &underlineFace);
   GXDrawShape(text);
   GXDisposeShape(text);
}
```

Figure 6-14 shows the results of executing the code in Listing 6-4.

Figure 6-14 A simple underline text face

Listing 6-5 is a sample function that produces a thicker underline text face.

Listing 6-5 Creating a thicker underline

```
void ShowBetterUnderline(void)
{
   gxShape text = GXNewText(9, (unsigned char *)"Underline", nil);
   gxTextFace *underlineFace;
   gxStyle thickPenStyle;
   gxTransform moveDownTransform;


   underlineFace = (gxTextFace *)NewPtr(sizeof(gxTextFace) +
                   sizeof(gxFaceLayer));

   ResetMapping(&underlineFace->advanceMapping);
   underlineFace->faceLayers = 2;
   underlineFace->faceLayer[0].flags = gxNoAttributes;
   underlineFace->faceLayer[0].outlineFill = gxWindingFill;
   underlineFace->faceLayer[0].outlineStyle = nil;
   underlineFace->faceLayer[0].outlineTransform = nil;
   underlineFace->faceLayer[0].boldOutset.x = 0;
   underlineFace->faceLayer[0].boldOutset.y = 0;

   /* Create a style to thicken the underline. The pen size will
   be scaled by the size of the text drawn,so we make the pen size
   1/12, so that 12 point text gets a 1 pixel underline and larger
   text will get a proportionally thicker underline.
   */
   thickPenStyle = GXNewStyle();
   GXSetStylePen(thickPenStyle, fixed1/12);

   /* Create a transform to position the underline. We want it to
    be 2 pixels below the baseline, so we translate it in the
    positive y direction by 2/12.
   */
   moveDownTransform = GXNewTransform();
   GXMoveTransform(moveDownTransform, 0, fixed1/6);

   underlineFace->faceLayer[1].flags = gxUnderlineAdvanceLayer;
   underlineFace->faceLayer[1].outlineFill = gxNoFill;
   underlineFace->faceLayer[1].outlineStyle = thickPenStyle;
   underlineFace->faceLayer[1].outlineTransform =
                   moveDownTransform;
   underlineFace->faceLayer[1].boldOutset.x = 0;
   underlineFace->faceLayer[1].boldOutset.y = 0;
```

```
    GXSetShapeFace(text, underlineFace);
    GXMoveShape(text, 0, ff(400));
    GXDrawShape(text);

    DisposePtr((void *)underlineFace);
    GXDisposeShape(text);
}
```

Figure 6-15 shows the results of executing the code in Listing 6-5.

Figure 6-15 A thicker underline

Setting Text Attributes

By setting the text attributes of a style, you affect how QuickDraw GX treats individual glyphs in a shape.

You should always get the current settings of the text attributes before setting any of them. The `GXSetStyleTextAttributes` function replaces all of the attributes currently associated with the style; if you want any attributes to remain the same, you must be include them in the call.

For example, to set the `gxVerticalText` text attribute of a style, you can use the following code:

```
GXSetStyleTextAttributes(myStyle,
            GXGetStyleTextAttributes(myStyle) | gxVerticalText);
```

If you want to clear only that text attribute from a style, you can use the code:

```
GXSetStyleTextAttributes(myStyle,
           GXGetStyleTextAttributes(myStyle) & ~gxVerticalText);
```

Setting the Automatic Text Advance Attribute

By setting the `gxAutoAdvanceText` text attribute you tell QuickDraw GX to take into account changes to the widths of the glyphs in the shape. For example, by applying a bold text face, you increase the widths of the glyphs. If you set the `gxAutoAdvanceText` attribute, QuickDraw GX increases the advance widths of the glyphs by a percentage corresponding to the increase in the filled widths of the bold glyphs.

Listing 6-6 is a pair of sample routines that show how the automatic text advance attribute increases the advance width of glyphs in the case of applying a bold text face.

Listing 6-6 Using the automatic text advance attribute

```
void MakeBoldTextFace(gxTextFace* face)
{

   face->faceLayers = 1;
   ResetMapping(&face->advanceMapping);
   face->faceLayer[0].outlineFill = gxWindingFill;
   face.faceLayer[0].flags = 0;
   face.faceLayer[0].outlineStyle = nil;
   face.faceLayer[0].boldOutset.x = fixed1/12;
   face.faceLayer[0].boldOutset.y = fixed1/36;
}

void MakeBoldText(void)
{
   gxShape text;
   gxPoint loc = { ff(50), ff(250);
   gxTextFace myFace;

   text = MakeTextShape(“Bolded”, “Hoefler Text”, ff(200),
            &loc);
   GXDrawShape(text);

   GXMoveShape(text, 0, ff(200));
   MakeBoldTextFace(&myFace);
   GXSetShapeFace(text, &myFace);
   GXDrawShape(text);

   GXMoveShape(text, 0, ff(200));
   GXSetShapeTextAttributes(text,
            GXGetShapeStyleAttributes(text) | gxAutoAdvanceText);
   GXDrawShape(text);

   GXDisposeShape(text);
}
```

Figure 6-16 shows the results of executing the code in Listing 6-6. The same typographic shape is drawn three times. The first time, the shape is drawn with no text face applied. The second time, the shape is drawn with a bold text face, but the `gxAutoAdvanceText`

text attribute not set. The third time, the shape is drawn with the text bolded and with the `gxAutoAdvanceText` text attribute set. Note that in the third case the glyph spacing has expanded to account for the bold text face.

Figure 6-16 Drawing text using the `gxAutoAdvanceText` text attribute

Setting the No Contour Grid Attribute

If you set the `gxNoContourGridText` text attribute, QuickDraw GX instructs the scaler *not* to use hints when imaging. If you set this bit, you must also set the `gxNoMetricsGridText` bit.

Listing 6-7 is an example of a routine that shows results of turning off the `gxNoContourGridText` attribute, then turning the shapes into path shapes.

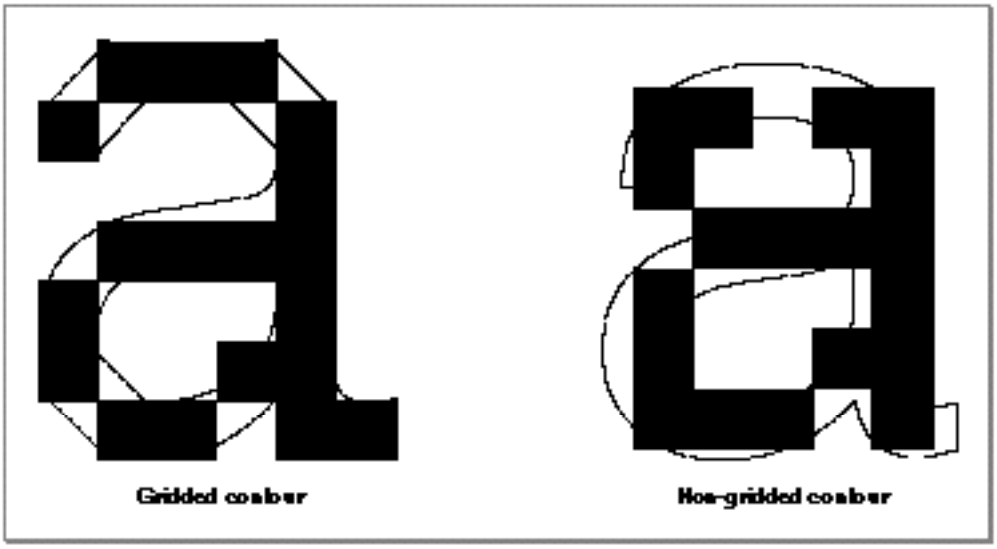
Listing 6-7 Using the no contour grid text attribute

```
void CompareNoContourGrid(void)
{
   gxShape contourGriddedA = GXNewText(1, (unsigned char *)"a",
nil);
   gxShape noContourGriddedA;

   GXSetShapeTextSize(contourGriddedA, ff(9));/* A small point
size increases the effect of gridding */
   GXSetShapePen(contourGriddedA, ff(4));

/* Make a copy of the original "a" and turn off contour gridding
*/
   noContourGriddedA = GXCopyToShape(nil, contourGriddedA);
   GXSetShapeTextAttributes(noContourGriddedA,
      GXGetShapeTextAttributes(noContourGriddedA) |
gxNoContourGridText|gxNoMetricsGridText);
```

```
/* Turn both shapes into paths, and magnify them 75x to show the
differences */
   GXSetShapeType(contourGriddedA, gxPathType);
   GXSetShapeFill(contourGriddedA, gxClosedFrameFill);
   GXSetShapeType(noContourGriddedA, gxPathType);
   GXSetShapeFill(noContourGriddedA, gxClosedFrameFill);
   GXScaleShape(contourGriddedA, ff(75), ff(75), 0, 0);
   GXScaleShape(noContourGriddedA, ff(75), ff(75), 0, 0);
   GXMoveShape(contourGriddedA, ff(20), ff(400));
   GXMoveShape(noContourGriddedA, ff(400), ff(400));

   GXDrawShape(contourGriddedA);
   GXDrawShape(noContourGriddedA);

   GXDisposeShape(contourGriddedA);
   GXDisposeShape(noContourGriddedA);
}
```

Figure 6-17 shows the results of executing the code in Listing 6-7. The “a” to the left shows an example with the `gxNoContourGridText` attribute clear, the “a” to right with it set. Note that at small point sizes, more realistic characters can be produced with contour gridding on (with `gxNoContourGridText` clear).

Figure 6-17 Turning the no contour grid attribute off and on



Setting the Vertical Text Attribute

If you set the `gxVerticalText` text attribute in the style, QuickDraw GX returns the font's vertical metrics for the glyphs in the shape when you call a function such as `GXGetGlyphMetrics`. (See the chapter "Typographic Shapes" in this book for a description of this function).

Listing 6-8 is a sample routine that shows how to set the vertical text attribute.

Listing 6-8 Setting the vertical text attribute

```
static void ShowVerticalText(void)
{
   gxShape text = GXNewText(8, (unsigned char *)"Vertical", nil);

   GXMoveShape(text, ff(100), ff(100));
   GXDrawShape(text);

   GXSetShapeTextAttributes(text,
               GXGetShapeTextAttributes(text) | gxVerticalText);
   GXDrawShape(text);

   GXDisposeShape(text);
}
```

Figure 6-8 on page 6-16 shows the results of the executing code in Listing 6-8.

Depending on the type of shape you're using, setting the `gxVerticalText` text attribute has two very different results. If the shape is a text or glyph shape, QuickDraw GX automatically draws the glyphs that use that style in a vertical line.

Listing 6-9 creates a glyph shape that uses two styles: the first part of the shape uses a Times® Roman style that does not have the vertical text attribute set, and second half of the shape uses a Helvetica style that does.

Listing 6-9 The effects of the vertical text attribute on a glyph shape

```
gxShape        gShape;
gxStyle        helveticaStyle, timesStyle;
gxFont         helveticaFont, timesFont;
stat const unsigned charvertText[] = "one way another";
static const shortmyStyleRuns[] = {8,7};
static gxStyle myStyles[2];
```

```
GXFindFonts (gxFullFontName, gxMacintoshPlatform, gxRomanScript,
   gxEnglishLanguage, 9, "Helvetica", 1, 1, &helveticaFont);
helveticaStyle = GXNewStyle ();
GXSetStyleTextSize(helveticaStyle, ff(40));
GXSetStyleFont(helveticaStyle, helveticaFont);
GXSetStyleTextAttributes (helveticaStyle,
   GXGetStyleTextAttributes (helveticaStyle) | gxVerticalText);

GXFindFonts (gxFullFontName,gxMacintoshPlatform, gxRomanScript,
   gxEnglishLanguage, 9, 11, "Times Roman",1, 1, &timesFont);
timesStyle = GXNewStyle ();
GXSetStyleTextSize(timesStyle, ff(40));
GXSetStyleFont(timesStyle, timesFont);

GXSetStyleTextAttributes (timesStyle, gxVerticalText);
myStyles[0] = timesStyle;
myStyles[1] = helveticaStyle;
totalLength = sizeof(vertText);

gShape = GXNewGlyphs(totalLength, vertText,
                     nil, nil, nil, myStyleRuns, myStyles);

GXMoveShapeTo(gShape, ff(100), ff(100));

GXDrawShape(gShape);

GXDisposeShape(gShape);
GXDisposeStyle(myStyles[0]);

GXDisposeStyle(myStyles[1]);
```

Figure 6-18 shows the results of the code in Listing 6-9. The first half of the shape is drawn in a horizontal line; the second half in a vertical line. Notice that the individual glyphs in the second half of the shape retain their original horizontal orientation but are stacked on top of one another. The glyphs that use the style with the vertical text attribute set are aligned through their centers, and the vertical glyph origin, which is at the top of the glyph “a”) is placed exactly where the advance width of the previous glyph—the space glyph—ends.

Keep in mind, however, that a layout shape describes a single *line* of text. Therefore, when you set the vertical text attribute on a style in a layout shape, QuickDraw GX does not give those glyphs a vertical orientation. It *rotates* them in line and makes them line-up vertically.

Figure 6-18 Using the `gxVerticalText` attribute with a text or glyph shape

For example, you can use the glyph shape in Listing 6-9 to describe the same appearance, using the following code:

```
gxPoint  textPositions = {ff(100), ff(100)};

gShape = GXNewLayout(1, &totalLength, (void *)&vertText,
                     2, myStyleRuns, myStyles,
                     0, nil, nil,
                     nil, &textPositions);
```

Figure 6-19 shows the resulting shape of executing the code. Notice how the glyphs using the Helvetica style, which has the vertical text attribute set, are center aligned just as they are in Figure 6-18, and the vertical glyph origin of the glyph “a” begins where the advance width of the previous space glyph ends.

In the layout shape, however, the entire shape must have one orientation in order to describe a single line. Therefore, in order to orient the second half of the shape correctly, the entire shape needs to be rotated.

Figure 6-19 Using the `gxVerticalText` attribute with a layout shape

See the chapter “Layout Line Control” in this book for more information on creating vertical text in layout shapes.

Applying Patterns and Dashes to Text Faces

You can fill a typographic shape with a pattern, or outline the glyphs in the shape with dashes, just as you can with any other shape. Listing 6-10 is a set of sample routines that creates a repeating sunburst pattern and applies it to a typographic shape through the shape's style object.

Listing 6-10 Filling a typographic shape with a pattern

```
void MakePatternText(void)
{
   gxShape text;
   gxPatternRecord myPat;
   gxPoint loc = { ff(50), ff(200) };

   text = MakeTextShape("Patterned words!", "Hoefler Text",
            ff(144), &loc);
   MakeBurstPattern(&myPat, ff(5), ff(64));
   GXSetShapePattern(text, &myPat);
   GXDisposeShape(myPat.pattern);
   GXDrawShape(text);
   GXDisposeShape(text);
}

/* A utility function that creates a text shape.*/

gxShape MakeTextShape(const char text[], const char font[],
            Fixed textSize, const gxPoint* loc)
{
   gxFont fontID;
   gxShape shape = GXNewText(strlen(text), (unsigned char*)text,
            loc);

   GXFindFonts(nil, gxFullFontName, gxMacintoshPlatform,
            gxRomanScript, gxEnglishLanguage,
            strlen(font), (unsigned char*)font, 1, 1, &fontID);
   GXSetShapeFont(shape, fontID);
   GXSetShapeTextSize(shape, textSize);
   return shape;
}

/* A utility function that creates the sunburst pattern.*/
```

```
void MakeBurstPattern(gxPatternRecord* pat,
            Fixed rotateAmount, Fixed scaleAmount)
{
   gxLine lineData = { { -fixed1, 0 }, { fixed1, 0 } };
   gxShape line = GXNewLine(&lineData);
   gxShape rotated = GXNewShape(gxPolygonType);
   Fixed angle;

   GXSetShapeFill(rotated, gxOpenFrameFill);
   for (angle = 0; angle < ff(180); angle += rotateAmount)
   {  GXSetShapeParts(rotated, 0, 0, line, gxBreakLeftEdit);
      GXRotateShape(line, rotateAmount, 0, 0);
   }
   GXDisposeShape(line);

   GXScaleShape(rotated, scaleAmount, scaleAmount, 0, 0);

   pat->attributes = 0;
   pat->pattern = rotated;
   pat->u.x = pat->v.x = MultiplyDivide(scaleAmount, 3, 2);
   pat->u.y = MultiplyDivide(scaleAmount, FixedSquareRoot(ff(3)),
            ff(2));
   pat->v.y = -pat->u.y;
}
```

Figure 6-20 shows the results of executing the code in Listing 6-10. Note that the sunburst pattern is applied to the shape as a whole, repeating at regular intervals across the text.

Figure 6-20 A typographic shape with a pattern

Creating Unusual Effects With Text Faces

Listing 6-11 is a sample routine that creates a similar sunburst effect, but with a pattern in a text face rather than as a style pattern. This listing uses the same utility functions as does Listing 6-10.

Listing 6-11 Creating an unusual effect

```
void MakePatternTextFace(void)
{
   gxShape text;
   gxPoint loc = { ff(50), ff(200) };
   gxPatternRecord myPat;
   gxTextFace myFace;

   text = MakeTextShape("Patterned words!", "Hoefler Text",
            ff(144), &loc);
   MakeBurstPattern(&myPat, ff(5), fixed1/2);
   MakePatternFace(&myFace, &myPat);
   GXDisposeShape(myPat.pattern);
   GXSetShapeFace(text, &myFace);
   GXDisposeStyle(myFace.faceLayer[0].outlineStyle);
   GXDrawShape(text);
   GXDisposeShape(text);
}

/* A utility function that creates a patterned text shape.*/

void MakePatternFace(gxTextFace* face,
            const gxPatternRecord* pat)
{
   face->faceLayers = 1;
   ResetMapping(&face->advanceMapping);
   face->faceLayer[0].outlineFill = gxWindingFill;
   face->faceLayer[0].flags = 0;
   face->faceLayer[0].outlineStyle = GXNewStyle();
   face->faceLayer[0].outlineTransform = nil;
   face->faceLayer[0].boldOutset.x = 0;
   face->faceLayer[0].boldOutset.y = 0;
   GXSetStyleTextSize(face->faceLayer[0].outlineStyle, fixed1);
   GXSetStylePattern(face->faceLayer[0].outlineStyle, pat);
}
```

Figure 6-21 shows the results of executing the code in Listing 6-11. Note that the sunburst pattern is applied individually to each character; for example, unlike in Figure 6-20, repeated letters are identically patterned wherever they appear.

Figure 6-21 An unusual effect with text faces

Typographic Styles Reference

This section describes the constants and data types, data structures, and functions that are specific to the text shape, glyph shape, and layout shape.

The “Constants and Data Types” section lists the enumerated types that provide information about the shapes and the data structures that contain information about the typographic shapes.

The “Functions” section, beginning on page 6-39, lists the QuickDraw GX functions you use to manipulate the values in the style objects associated with typographic shapes.

The layout shape has its own style and shape routines, in addition to the ones described in this chapter. These routines are described in the chapter “Layout Shapes” in this book.

Constants and Data Types

This section describes the data types that you use to provide information about and retrieve information from the typographic shapes and their associated styles. The data types discussed in this section include

- the `gxTextFace` structure, which you use to define an algorithmically added font style, such as bold or italic, or a more complex font style, such as a special kind of pattern or fill you add to the glyphs of a typographic shape
- the `gxFaceLayer` structure, which you use to describe one of the layers that make up a text face
- the `gxLayerFlags` enumeration, which specifies the characteristics of a face layer in a text face
- the values for alignment, which allow you to pick a predefined alignment setting or create your own
- the `gxTextAttributes` enumeration, which allows you to control how QuickDraw GX alters glyph outlines or sets text to be horizontal or vertical

Text Face

A **text face** is an algorithmically applied typestyle that you define. A text face has both a mapping and face layers. The mapping affects the advance widths and interglyph spacing, but not the shape of the glyphs. The face layers, which are optional, specify the manner in which the glyphs are drawn. All of the face layers are combined to form the visual composite. QuickDraw GX then draws the filled parts of the glyph as it appears with the text face.

The `gxTextFace` structure specifies how to distort the outline, which can be used, among other things, to mimic popular variants of a plain text face.

```
typedef struct {
   long        faceLayers;
   gxMapping   advanceMapping;
   gxFaceLayer gxFaceLayer[gxAnyNumber];
} gxTextFace;
```

Field descriptions

`faceLayers`	The number of face layers.
`advanceMapping`	The mapping applied to the advance width for each glyph when the text face is applied. See “Setting the Advance Mapping” on page 6-18.
`gxFaceLayer`	The face layers attached to this text face. There may be 0 or more face layers, and the first face layer is drawn first. The `gxFaceLayer` data structure is described in “Face Layers” on page 6-36.

For more information on how to create a text face, see “Creating Text Faces” on page 6-17.

Face Layers

A **face layer** is a description of a part of a text face that you define using the `gxTextFace` structure. The `gxFaceLayer` structure contains the shape fill, the layer flags, a style, a transform, and a degree of boldness that QuickDraw GX should apply. A text face may contain 0 or more face layers.

For more information about how to create a face layer, see “Setting a Face Layer” on page 6-19.

```
typedef struct {
   gxShapeFill outlineFill;
   gxLayerFlag flags;
   gxStyle     outlineStyle;
   gxTransform outlineTransform;
   gxPoint     boldOutset;
} gxFaceLayer;
```

Field descriptions

`outlineFill` The shape fill for each layer. The possible values for this field are described in *Inside Macintosh: QuickDraw GX Graphics*. The most useful values with typographic shapes are `gxWindingFill` and `gxClosedFrameFill`. The `gxEvenOddFill` fill may give you unpredictable results.

`flags` The flags that describe the drawing and composition of the layers of the text. Possible values for this field are discussed in "Functions" on page 6-39.

`outlineStyle` The optional overriding style for the face. The style's pen size is scaled by a factor corresponding to the point size of the text. This style cannot have its own text face; if so QuickDraw GX posts an error.

`outlineTransform` The matrix and clip used to distort the glyph. This field governs the algorithmic application of the italic, condensed, and extended typestyles as well as any designs and patterns in the text face. All distortions are scaled factors corresponding to text point size and by any mappings affecting the text. This clip has a more global effect than the clip specified in the layer flags.

`boldOutset` The degree of boldness QuickDraw GX should apply in the x and y directions for 1-point text.

Layer Flags

The layer flags describe the characteristics of one layer of a text face. For more information about the meanings of the flags, see "Setting the Layer Flags" on page 6-23.

```
enum gxLayerFlags{
   gxUnderlineAdvanceLayer    = 0x0001,
   gxSkipWhiteSpaceLayer      = 0x0002,
   gxUnderlineIntervalLayer   = 0x0004,
   gxUnderlineContinuationLayer= 0x0008,
   gxWhiteLayer               = 0x0010,
   gxClipLayer                = 0x0020,
   gxStringLayer              = 0x0040
};

   typedef long gxLayerFlag;
```

Constant descriptions

`gxUnderlineAdvanceLayer` Draws an underline from the beginning of the text that shares one text face to the end, including white spaces. This bit must be set if the `gxSkipWhiteSpaceLayer`, `gxUnderlineIntervalLayer`, or `gxUnderlineContinuationLayer` bit is set.

`gxSkipWhiteSpaceLayer`
: Does not draw an underline with glyphs that have no contours (such as the space character) if the `gxUnderlineAdvanceLayer` bit is also set.

`gxUnderlineIntervalLayer`
: Draws an underline through the gaps between text of different text faces. If you set this bit, you must also set the `gxStringLayer` bit.

`gxUnderlineContinuationLayer`
: Draws an underline across text of different style runs. If you set this bit, you must also set the `gxStringLayer` bit. Also, the previous style in the shape must have `gxUnderlineAdvanceLayer` set; if it doesn't, you will get an error.

`gxWhiteLayer` Erases portions of previously drawn layers. You can only set this bit for the second or greater layer.

`gxClipLayer` Clips the layer to the original outline of the glyph. You should set this bit if you want to clip a pattern or fill to the text.

`gxStringLayer` Connects the text of different text faces and style runs.

Alignment Values

Several values are predefined for various types of alignment.

```
#define gxLeftJustify   0
#define gxCenterJustify (fract1/2)
#define gxRightJustify  fract1
#define gxFillJustify   -1
```

Constant descriptions

`gxLeftJustify` Draws left-aligned text.

`gxCenterJustify` Draws centered text.

`gxRightJustify` Draws right-aligned text.

`gxFillJustify` Draws fully justified text.

For more information see "Alignment" on page 6-11.

Text Attributes

Each style object has a set of **text attributes,** which consist of a group of flags that modify the behavior of the style object associated with typographic shapes. These flags allow you to specify how QuickDraw GX alters glyph outlines or chooses the proper types of metrics for horizontal or vertical text. These flags are defined in the `gxTextAttributes` enumeration.

```
enum gxTextAttributes{
   gxAutoAdvanceText    = 0x0001,
   gxNoContourGridText  = 0x0002,
   gxNoMetricsGridText  = 0x0004,
```

```
   gxAnchorPointsText   = 0x0008,
   gxVerticalText       = 0x0010,
   gxNoOpticalScaleText = 0x0020
};
```

Constant descriptions

`gxAutoAdvanceText`
: Tells QuickDraw GX to take into account changes to the widths of glyphs in the shape.

`gxNoContourGridText`
: Prevents QuickDraw GX from using hinted outlines before displaying the text.

`gxNoMetricsGridText`
: Tells QuickDraw GX to use the ideal metrics to space the glyphs in a typographic shape.

`gxAnchorPointsText`
: Tells QuickDraw GX to include data for all the outline's points. QuickDraw GX's default action is to return a basic outline, removing points that do not affect the shape, such as single-point contours.

`gxVerticalText`
: Tells QuickDraw GX to return the vertical advance and side bearing metrics. The default values are the horizontal advance and side bearing metrics.

`gxNoOpticalScaleText`
: Disables QuickDraw GX's attempt to automatically set the optical scale variation value.

Text attributes are described in "Text Attributes" on page 6-14.

Functions

This section describes the routines that access, retrieve, change, or delete information about the typographic shapes.

You can retrieve and set basic shape and style information, such as the text attributes, the text face, the font, the text size, the justification amount, the platform, or the font variation descriptions.

Getting and Setting the Font of a Style Object

The font property of style objects specifies the font (or font family) of a style object. You use the `gxFont` data structure, which is described in the chapter "Fonts Objects" of this book, when retrieving or setting the font used by a style object.

You can use the `GXGetStyleFont` function to retrieve the font information from a style object and the `GXSetStyleFont` function to specify the font information for a style object.

The `GXGetShapeFont` and `GXSetShapeFont` functions provide a way to retrieve and specify the font information for the style object associated with a particular shape.

GXGetStyleFont

You can use the `GXGetStyleFont` function to determine the font currently set by a style object.

```
gxFont GXGetStyleFont(gxStyle source);
```

`source` A reference to the style object whose font you want to determine

function result The font associated with the style object.

DESCRIPTION

The `GXGetStyleFont` function returns the font associated with the style object. To get the name of the font, you can use the `GXFindFontName` function.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
shape_is_nil
```

SEE ALSO

The `GXGetStyleFace` function (page 6-43) determines the text face currently set by a style object.

Fonts are discussed in the chapter “Font Objects” in this book.

GXSetStyleFont

You can use the `GXSetStyleFont` function to set or change the font used by a style object.

```
void GXSetStyleFont(gxStyle target, gxFont aFont);
```

`target` A reference to the style object whose font you want to set or change.

`aFont` The new font.

DESCRIPTION

The `GXSetStyleFont` function sets the font used by the style object specified by the `target` parameter. If you want the default font, pass `nil` in the `aFont` parameter.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`style_is_nil`
`illegal_font_parameter`

Notices (debugging version)
`font_already_set`

SEE ALSO

Fonts are discussed in the chapter "Font Objects" in this book.

GXGetShapeFont

You can use the `GXGetShapeFont` function to determine the font set for the style object of a particular QuickDraw GX typographic shape.

```
gxFont GXGetShapeFont(gxShape source);
```

`source` A reference to the shape whose font you want to determine.

function result The font of the style object of a shape.

DESCRIPTION

The `GXGetShapeFont` function returns as the function result the font of the style object of a shape.

The function returns information about the font setting of the style object associated with the shape object. However, the glyph and layout shapes may have arrays of styles in their geometries and therefore do not necessarily use the style object attached to the shape object. In this case, you should use the `GXGetStyleFont` (page 6-40) and `GXSetStyleFont` (page 6-40) functions to determine the fonts of the styles stored in the shape's geometry.

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`shape_is_nil`

SEE ALSO

Fonts are discussed in the chapter "Font Objects" in this book.

GXSetShapeFont

You can use the `GXSetShapeFont` function to alter the font of the style object associated with a particular shape.

```
void GXSetShapeFont(gxShape target, gxFont aFont);
```

`target` A reference to the shape whose font you want to alter.

`aFont` The new font.

DESCRIPTION

The `GXSetShapeFont` function sets the font of the style object associated with the shape specified by the `target` parameter. If you want the default font, pass `nil` in the `aFont` parameter. If a style object is shared among shapes, `GXSetShapeFont` copies the style object so that only the shape in the target parameter is affected by the changes to the font.

This function provides a convenient way to change the font of a shape without having to call the `GXGetShapeStyle` function to obtain a reference to the shape's style object.

The function specifies the font for the style object associated with the shape object. However, the glyph and layout shapes may have arrays of styles in their geometries and therefore do not necessarily use the style object attached to the shape object. In this case, you should use the `GXGetStyleFont` (page 6-40) and `GXGetStyleFont` (page 6-40) functions to get and set the fonts of the styles stored in the shape's geometry.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`
`illegal_font_parameter`

Notices (debugging version)

`font_already_set`

SEE ALSO

Fonts are discussed in the chapter "Font Objects" in this book.

Getting and Setting the Text Face

The text face property of a style object specifies an algorithmic typestyle that you want to apply to text. You use the `gxTextFace` data structure, which is described on page 6-36, when retrieving or setting text face information.

You can use the `GXGetStyleFace` function to retrieve the text face information from a style object and the `GXSetStyleFace` function to specify the text face information for a style object.

The GXGetShapeFace and GXSetShapeFace functions provide a way to retrieve and specify the text face information for the style object associated with a particular shape.

GXGetStyleFace

You can use the GXGetStyleFace function to get the text face currently set in a style object.

```
long GXGetStyleFace(gxStyle source, gxTextFace *face);
```

source	A reference to the style object whose text face you want to determine.
face	The text face set in the style object, returned by the function. This can be nil, which just returns the layer count.
function result	The number of layers in the text face. If there is no text face, the function returns –1.

DESCRIPTION

The GXGetStyleFace function returns the number of layers in the text face. The function also returns, in the face parameter, the text face used by the style object.

Note
Your application must allocate enough memory to store the text face and all of the face layers of that text face. See “Creating Text Faces” on page 6-17. u

ERRORS, WARNINGS, AND NOTICES

Errors
out_of_memory
style_is_nil

GXSetStyleFace

You can use the GXSetStyleFace function to set or change the text face of a style object.

```
void GXSetStyleFace(gxStyle target, const gxTextFace *face);
```

target	A reference to the style object whose text face you want to change.
face	The new text face for the style object. If nil, it will remove an existing face.

DESCRIPTION

The `GXSetStyleFace` function sets the text face of the style object specified by target to the text face specified in the `face` parameter.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`	
`style_is_nil`	
`parameter_out_of_range`	(debugging version)
`inverseFill_must_set_cliplayer_flag`	(debugging version)
`style_wrong_type`	(debugging version)
`layer_style_cannot_contain_face`	(debugging version)
`transform_wrong_type`	(debugging version)

Notices (debugging version)

`face_already_set`

GXGetShapeFace

You can use the `GXGetShapeFace` function to determine the text face set for the style object of a particular QuickDraw GX typographic shape.

```
long GXGetShapeFace(gxShape source, gxTextFace *face);
```

`source` A reference to the shape whose text face you want to determine.

`face` The text face of the shape, returned by the function.

function result The number of layers in the text face. If there is no text face, the function returns –1.

DESCRIPTION

The `GXGetShapeFace` function returns the number of layers in the text face named by the `face` parameter. The function also returns the text face itself.

The function returns information about the text face setting of the style object associated with the shape object. However, the glyph and layout shapes may have arrays of styles in their geometries and therefore do not necessarily use the style object attached to the shape object. In this case, you should use the `GXGetStyleFace` function, described on page 6-43 and the `GXSetStyleFace` function, described on page 6-43, to determine the text faces for the styles stored in the shape's geometry.

Note

Your application must allocate enough memory to store the text face and all of the face layers of that text face. See "Creating Text Faces" on page 6-17. u

ERRORS, WARNINGS, AND NOTICES

Errors
`out_of_memory`
`shape_is_nil`

GXSetShapeFace

You can use the `GXSetShapeFace` function to alter the text face of the style object associated with a particular shape.

```
void GXSetShapeFace(gxShape target, const gxTextFace *face);
```

`target`	A reference to the shape whose text face you want to alter.
`face`	The new text face.

DESCRIPTION

The `GXSetShapeFace` function sets the text face of the style object associated with the shape specified by the `target` parameter. If a style object is shared among shapes, `GXSetShapeFace` copies the style object so that only the shape in the parameter is affected by the changes to the text face.

This function provides a convenient way to change the text face of a shape without calling the `GXGetShapeStyle` function to obtain a reference to the shape's style object.

You can use this function in combination with the `GXGetShapeFace` function, described on page 6-44, to set or clear single style attributes.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`	
`style_is_nil`	
`parameter_out_of_range`	(debugging version)
`inverseFill_must_set_cliplayer_flag`	(debugging version)
`style_wrong_type`	(debugging version)
`layer_style_cannot_contain_face`	(debugging version)
`transform_wrong_type`	(debugging version)

Notices (debugging version)
`face_already_set`

Getting and Setting the Text Size of a Style Object

The text size property of style objects specifies the size, in typographic points, of the text of a style object.

You can use the `GXGetStyleTextSize` function to retrieve the text size from a style object and the `GXSetStyleTextSize` function to specify the text size of a style object.

The `GXGetShapeTextSize` and `GXSetShapeTextSize` functions provide a way to retrieve and specify the text size for the style object associated with a particular shape.

GXGetStyleTextSize

You can use the `GXGetStyleTextSize` function to determine the text size of a style object.

```
Fixed GXGetStyleTextSize(gxStyle source);
```

`source` A reference to the style object whose text size you want to determine.

function result The text size, in typographic points, of the style object named by the `source` parameter.

DESCRIPTION

The `GXGetStyleTextSize` function returns the text size, in typographic points, of the style object named by the `source` parameter.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`style_is_nil`

GXSetStyleTextSize

You can use the `GXSetStyleTextSize` function to set or change the text size used by a style object.

```
void GXSetStyleTextSize(gxStyle target, Fixed size);
```

`target` A reference to the style object whose text size you want to set.

`size` The new text size, in points. This parameter can be any positive value, including fractional sizes. A value of 0 resets the text size to the point size natural for the current script. For Roman scripts, this is 12 points.

DESCRIPTION

The `GXSetStyleTextSize` function sets the text size, in typographic points, of the style object.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`style_is_nil`
`parameter_out_of_range` (debugging version)

Notices (debugging version)

`text_size_already_set`

GXGetShapeTextSize

You can use the `GXGetShapeTextSize` function to determine the text size, in typographic points, set for the style object of a particular QuickDraw GX typographic shape.

```
FixedGXGetShapeTextSize(gxShape source);
```

`source` A reference to the shape whose text size you want to determine.

function result The text size, in typographic points, of the style object of the shape named by the `source` parameter.

DESCRIPTION

The `GXGetShapeTextSize` function returns the text size, in typographic points, of the style object of the shape named by the `source` parameter.

The function returns information about the text size setting of the style object associated with the shape object. However, the glyph and layout shapes may have arrays of styles in their geometries and therefore do not necessarily use the style object attached to the shape object. In this case, you should use the `GXGetStyleTextSize` (page 6-46) and `GXGetStyleTextSize` (page 6-46) functions to set the sizes of the styles stored in the shape's geometry.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`style_is_nil`

GXSetShapeTextSize

You can use the `GXSetShapeTextSize` function to alter the text size of the style object associated with a particular shape.

```
void GXSetShapeTextSize(gxShape target, Fixed size);
```

`target` A reference to the shape whose text size you want to change.

`size` The new text size, in points. This parameter can be any positive value, including fractional sizes. A value of 0 resets the text size to the point size natural for the current script. For Roman scripts, this is 12 points.

DESCRIPTION

The `GXSetShapeTextSize` function sets the text size, in typographic points, of the style object associated with the shape specified by the `target` parameter. If a style object is shared among shapes, `GXSetShapeTextSize` copies the style object so that only the shape in the `target` parameter is affected by the changes to the text size.

This function provides a convenient way to change the text size of a shape without calling the `GXGetShapeStyle` function to obtain a reference to the shape's style object.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`style_is_nil`
`parameter_out_of_range` (debugging version)

Notices (debugging version)

`text_size_already_set`

Getting and Setting the Alignment of a Style Object

The alignment property of style objects specifies the type of alignment used in a style object. For more information, see "Alignment" on page 6-11. Possible values for alignment are described in "Alignment Values" on page 6-38.

You can use the `GXGetStyleJustification` function to retrieve the alignment amount from a style object and the `GXSetStyleJustification` function to specify the alignment for a style object.

The `GXGetShapeJustification` and `GXSetShapeJustification` functions provide a way to retrieve and specify the alignment of the style object associated with a text or glyph shape.

The justification value accounts for the difference between ideal and device metrics when the text is measured or drawn. For more control over how justification is used in your text, see the chapter "Layout Line Control" in this book.

GXGetStyleJustification

You can use the `GXGetStyleJustification` function to determine the alignment set by a style object.

```
fract GXGetStyleJustification(gxStyle source);
```

`source` A reference to the style object whose alignment you want to determine.

function result The alignment set by a style object

DESCRIPTION

The `GXGetStyleJustification` function returns the alignment set by a style object. Possible values for the style's alignment are discussed in "Alignment Values" on page 6-38.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`style_is_nil`

GXSetStyleJustification

You can use the `GXSetStyleJustification` function to set or change the alignment used by a style object.

```
void GXSetStyleJustification(gxStyle target, fract justify);
```

`target` A reference to the style object whose alignment you want to change.

`justify` The alignment you want to set in the style object. Possible values are discussed in "Alignment Values" on page 6-38.

DESCRIPTION

The `GXSetStyleJustification` function sets the alignment used by the style object named by the `target` parameter.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`style_is_nil`
`parameter_out_of_range` (debugging version)

Notices (debugging version)

`justification_already_set`

GXGetShapeJustification

You can use the `GXGetShapeJustification` function to determine the alignment set for the style object of a particular QuickDraw GX typographic shape.

```
fractGXGetShapeJustification(gxShape source);
```

`source` A reference to the shape whose set alignment you want to determine.

function result The alignment set in the style object used by the specified shape.

DESCRIPTION

The `GXGetShapeJustification` function returns the alignment set in the style object associated with the shape specified by `source`.

The function returns information about the alignment setting of the style object associated with the shape object. However, the glyph and layout shapes may have arrays of styles in their geometries and therefore do not necessarily use the style object attached to the shape object. In this case, you should use the `GXGetStyleJustification` (page 6-49) and `GXGetStyleJustification` (page 6-49) functions to determine the alignments of the styles stored in the shape's geometry.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
style_is_nil
```

GXSetShapeJustification

You can use the `GXSetShapeJustification` function to alter the alignment of the style object associated with a particular shape.

```
void GXSetShapeJustification(gxShape target, fract justify);
```

`target` A reference to the shape whose alignment you want to alter.

`justify` The alignment you want to set in this shape. Possible values for this parameter are described in "Alignment Values" on page 6-38.

DESCRIPTION

The `GXSetShapeJustification` function sets the alignment of the style object associated with the shape specified by the `target` parameter. If a style object is shared among shapes, `GXSetShapeJustification` copies the style object so that only the shape in the `target` parameter is affected by the changes to the alignment value.

This function provides a convenient way to change the alignment of a shape without calling the `GXGetShapeStyle` function to obtain a reference to the shape's style object.

The function returns information about the alignment setting of the style object associated with the shape object. However, the glyph and layout shapes may have arrays of styles in their geometries and therefore do not necessarily use the style object attached to the shape object. In this case, you should use the `GXGetStyleJustification` (page 6-49) and `GXSetStyleJustification` (page 6-49) functions to set the alignment of the styles stored in the shape's geometry.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
style_is_nil
parameter_out_of_range
```

Getting and Setting the Font Variations of a Style Object

The font variations property of style objects specifies a particular variation coordinate to be used with the style's font. You use the `gxFontVariation` data type, which is described in the chapter "Fonts Objects" of this book, when retrieving or setting font variations for the user.

You can use the `GXGetStyleFontVariations` function to retrieve the font variations from a style object and the `GXSetStyleFontVariations` function to specify the font variations for a style object.

The `GXGetShapeFontVariations` and `GXSetShapeFontVariations` functions provide a way to retrieve and specify the font variations for the style object associated with a particular shape.

You should use the font variation functions described in the chapter "Fonts Objects" to retrieve, add, or change font variation data in the font.

GXGetStyleFontVariations

You can use the `GXGetStyleFontVariations` function to determine the font variations set by a style object.

```
long GXGetStyleFontVariations(gxStyle source,
                              gxFontVariation variations[]);
```

`source` A reference to the style object whose font variations you want to determine.

`variations` The font variations set by the style object, returned by the function, allocated by the application.

function result The number of font variations in the style object. The function returns 0 if there are no font variations.

DESCRIPTION

The `GXGetStyleFontVariations` function returns the number of font variations associated with the style object. It returns the font variations in the `variations` parameter, if it is not set to `nil`.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
style_is_nil
```

SEE ALSO

Font variations are described in the chapter “Fonts Objects” in this book.

GXSetStyleFontVariations

You can use the `GXSetStyleFontVariations` function to set or change the font variation used by a style object.

```
void GXSetStyleFontVariations(gxStyle target, long count,
                              const gxFontVariation variations[]);
```

`target`	A reference to the style object whose font variation you want to change.
`count`	The number of font variations you are specifying.
`variations`	The new font variation for the style object.

DESCRIPTION

The `GXSetStyleFontVariations` function sets the font variation of a style object.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
style_is_nil
parameter_out_of_range                    (debugging version)
```

Notices (debugging version)

```
font_variations_already_set
```

SEE ALSO

Font variations are described in the chapter “Fonts Objects” in this book.

GXGetShapeFontVariations

You can use the `GXGetShapeFontVariations` function to determine the font variations set for the style object of a particular QuickDraw GX typographic shape.

```
long GXGetShapeFontVariations(gxShape source,
                              gxFontVariation variations[]);
```

`source` A reference to the shape whose font variations you want to determine.

`variations` The font variations of the shape, returned by the function if it is not `nil`.

function result The number of font variations in the style object associated with the shape object. The function returns 0 if there are no font variations.

DESCRIPTION

The `GXGetShapeFontVariations` function returns the number of font variations associated with the style object of the shape object. The function also returns the font variations.

The function returns information about the font variation setting of the style object associated with the shape object. However, the glyph and layout shapes may have arrays of styles in their geometries and therefore do not necessarily use the style object attached to the shape object. In this case, you should use the `GXGetStyleFontVariations` (page 6-51) and `GXSetStyleFontVariations` (page 6-52) functions to get the font variation of the styles stored in the shape's geometry.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`style_is_nil`

SEE ALSO

Font variations are described in the chapter "Fonts Objects" in this book.

GXSetShapeFontVariations

You can use the GXSetShapeFontVariations function to alter the font variations associated with a particular shape.

```
void GXSetShapeFontVariations(gxShape target, long count,
                              const gxFontVariation variations[]);
```

target	A reference to the shape whose font variations you want to alter.
count	The number of font variations you are specifying.
variations	The new font variations.

DESCRIPTION

The GXSetShapeFontVariations function sets the font variations of the style object associated with the shape specified by the target parameter. If a style object is shared among shapes, GXSetShapeFontVariations copies the style object so that only the shape in the target parameter is affected by the changes to the font variations.

This function provides a convenient way to change the font variations of a shape without calling the GXGetShapeStyle function to obtain a reference to the shape's style object.

The function specifies the font variation setting of the style object associated with the shape object. However, the glyph and layout shapes may have arrays of styles in their geometries and therefore do not necessarily use the style object attached to the shape object. In this case, you should use the GXGetStyleFontVariations (page 6-51) and GXSetStyleFontVariations (page 6-52) functions to set the font variations of the styles stored in the shape's geometry.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
style_is_nil
parameter_out_of_range                    (debugging version)
```

Notices (debugging version)

```
font_variations_already_set
```

SEE ALSO

Font variations are described in the chapter "Fonts Objects" in this book.

Retrieving the Elements in a Font Variation Suite

QuickDraw GX allows you to retrieve all the elements in a font variation suite. The font variation suite contains complete information on the font variations specified for a shape or style object. You can

- retrieve the number of elements in a font variation suite for a style object with the `GXGetStyleFontVariationSuite` function
- retrieve the font variation suite for the style object associated with a specified shape with the `GXGetShapeFontVariationSuite` function

The font variation suite is described in “Font Variations” on page 6-13. Font variations are described in the chapter “Font Objects” in this book.

GXGetStyleFontVariationSuite

You can use the `GXGetStyleFontVariationSuite` function to retrieve the font variation suite for a style object.

```
long GXGetStyleFontVariationSuite(gxStyle source,
                                  gxFontVariation variations[]);
```

`source` A reference to the style object whose font variation suite you want to retrieve.

`variations` An array of font variation structures. On return this array contains the font variations for the style.

function result The number of elements in the font variation suite, which is equal to the number of variation axes in a font.

DESCRIPTION

The `GXGetStyleFontVariationSuite` function returns the number of elements in the font variation suite for the font specified by the style associated with the source object. It returns the variations themselves in the `variations` parameter. If you pass `nil` for the `variations` parameter, this function returns a valid result but returns no array. You typically call this function twice, first with a `nil` value for `variations` in order to get the right size of array to allocate and the second time to retrieve the array itself.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`shape_is_nil`

SEE ALSO

To get the font variation suite for a shape object, use the `GXGetShapeFontVariationSuite` function, described in the next section.

Font Variations, including `GXCountFontVariations`, are described in the chapter "Font Objects" in this book.

GXGetShapeFontVariationSuite

You can use the `GXGetShapeFontVariationSuite` function to retrieve the font variation suite for the style object associated with a specified shape.

```
long GXGetShapeFontVariationSuite(gxShape source,
                                  gxFontVariation variations[]);
```

`source` A reference to the shape object whose style object's font variation suite you want to retrieve.

`variations` An array of font variation structures. On return this array contains the font variations for the style.

function result The number of elements in the font variation suite. The function returns 0 if there are no font variations.

DESCRIPTION

The `GXGetShapeFontVariationSuite` function returns the number of elements in the font variation suite for the font specified by the style object associated with the source shape. It returns the variations themselves in the `variations` parameter. If you pass `nil` for the `variations` parameter, this function returns a valid result but returns no array. You typically call this function twice, first with a `nil` value for `variations` in order to get the right size of array to allocate and the second time to retrieve the array itself.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
shape_is_nil
```

SEE ALSO

To get the font variation suite from a shape object, use the `GXGetStyleFontVariationSuite` function, described on page 6-55.

Font Variations are described in the chapter "Font Objects" in this book.

Retrieving Font Metrics

QuickDraw GX allows you to retrieve font metrics for an object in several ways. You can

- retrieve the font metrics for a style object, including line spacing and caret angle, with the `GXGetStyleFontMetrics` function
- retrieve the font metrics for a style object associated with a shape with the `GXGetShapeFontMetrics` function
- retrieve the font metrics for a style object associated with a shape, taking into account the shape's transform, with the `GXGetShapeLocalFontMetrics` function
- retrieve the font metrics for a style object associated with a shape, taking into account the mappings on a specified view port and view device, with the `GXGetShapeDeviceFontMetrics` function

Font metrics are described in "Font Metrics" on page 6-14.

GXGetStyleFontMetrics

You can use the `GXGetStyleFontMetrics` function to retrieve the font metrics for a style object, including line spacing and caret angle.

```
void GXGetStyleFontMetrics(gxStyle sourceStyle, gxPoint* before,
                           gxPoint* after, gxPoint* caretAngle,
                           gxPoint* caretOffset);
```

`sourceStyle`
A reference to the style object whose font metrics you need.

`before` A pointer to a `gxPoint` structure. On return, the point specifies the distance and direction to the previous line of text. For horizontal text, this corresponds to the font's ascent.

`after` A pointer to a `gxPoint` structure. On return, the point specifies the distance and direction to the next line of text. For horizontal text, this corresponds to the font's descent.

`caretAngle` A pointer to a `gxPoint` structure. On return, the point specifies the direction for the text selection caret.

`caretOffset`
A pointer to a `gxPoint` structure. On return, the point specifies the direction and distance relative to the origin, where the text selection caret should intersect the baseline of the text.

DESCRIPTION

The `GXGetStyleFontMetrics` function returns the font metrics of a style object, including its line spacing and its caret angle. These values are returned as points representing the vectors. The values of the vectors differentiate between horizontal and vertical text. This function takes into account the point size, variations, and the settings of the `gxNoMetricsGridText` and `gxVerticalText` text attributes.

ERRORS, WARNINGS, AND NOTICES

Errors

`illegal_type_for_shape (if not typographic)` (debugging version)
`shape_is_nil`

SEE ALSO

The `GXGetShapeFontMetrics` function is described in the next section. The `GXGetShapeLocalFontMetrics` function is described on page 6-59.
The `GXGetShapeDeviceFontMetrics` function is described on page 6-60.

GXGetShapeFontMetrics

You can use the `GXGetShapeFontMetrics` function to retrieve the font metrics for the style object associated with a shape.

```
void GXGetShapeFontMetrics(gxShape source, gxPoint* before,
                           gxPoint* after, gxPoint* caretAngle,
                           gxPoint* caretOffset);
```

`source` A reference to the shape object whose font metrics you need.

`before` A pointer to a `gxPoint` structure. On return, the point specifies the distance and direction to the previous line of text. For horizontal text, this corresponds to the font's ascent.

`after` A pointer to a `gxPoint` structure. On return, the point specifies the distance and direction to the next line of text. For horizontal text, this corresponds to the font's descent.

`caretAngle` A pointer to a `gxPoint` structure. On return, the point specifies the direction for the text selection caret.

`caretOffset`
A pointer to a `gxPoint` structure. On return, the point specifies the direction and distance relative to the shape's origin, where the text-selection caret should intersect the baseline of the text.

DESCRIPTION

The `GXGetShapeFontMetrics` function returns the font metrics of the `style` object associated with the specified shape, including its line spacing and its caret angle. These values are returned as points representing vectors. The values of the vectors differentiate between horizontal and vertical text. This function is equivalent to calling the `GXGetStyleFontMetrics` function with the result of `GXGetShapeStyle`.

ERRORS, WARNINGS, AND NOTICES

Errors

`illegal_type_for_shape (if not typographic)` (debugging version)
`shape_is_nil`

SEE ALSO

The `GXGetStyleFontMetrics` function is described on page 6-57. The `GXGetShapeLocalFontMetrics` function is described in the next section. The `GXGetShapeDeviceFontMetrics` function is described on page 6-60.

GXGetShapeLocalFontMetrics

You can use the `GXGetShapeLocalFontMetrics` function to retrieve the font metrics for the style object associated with a shape, taking into account the shape's transform. The results are expressed in local coordinates.

```
void GXGetShapeLocalFontMetrics(gxShape sourceShape,
                                gxPoint* before,
                                gxPoint* after,
                                gxPoint* caretAngle,
                                gxPoint* caretOffset);
```

`sourceShape`
A reference to the shape object whose font metrics you need.

`before` A pointer to a `gxPoint` structure. On return, the point specifies the distance and direction to the previous line of text. For horizontal text, this corresponds to the font's ascent. If the parameter is `nil`, it is ignored.

`after` A pointer to a `gxPoint` structure. On return, the point specifies the distance and direction to the next line of text. For horizontal text, this corresponds to the font's descent. If the parameter is `nil`, it is ignored.

`caretAngle` A pointer to a `gxPoint` structure. On return, the point specifies the direction for the text-selection caret. If the parameter is `nil`, it is ignored.

`caretOffset`
A pointer to a `gxPoint` structure. On return, the point specifies the direction and distance relative to the shape's origin, where the text selection caret should intersect the baseline of the text. If the parameter is `nil`, it is ignored.

DESCRIPTION

The `GXGetShapeLocalFontMetrics` function returns the font metrics for the style object associated with the source shape, in local space. These values are returned as points that represent vectors. The values of the vectors differentiate between horizontal and vertical text and account for any mapping that may be applied to the shape through its transform object.

ERRORS, WARNINGS, AND NOTICES

Errors

`illegal_type_for_shape (if not typographic)` (debugging version)
`shape_is_nil`

SEE ALSO

The `GXGetStyleFontMetrics` function is described on page 6-57. The `GXGetShapeFontMetrics` function is described on page 6-58.
The `GXGetShapeDeviceFontMetrics` function is described in the next section.

GXGetShapeDeviceFontMetrics

You can use the `GXGetShapeDeviceFontMetrics` function to retrieve the font metrics for the style object associated with a shape, taking into account the mappings on the specified view port and view device. The result is expressed in device coordinates.

```
void GXGetShapeDeviceFontMetrics(gxShape sourceShape,
                                 gxViewPort port,
                                 gxViewDevice device,
                                 gxPoint* before,
                                 gxPoint* after,
                                 gxPoint* caretAngle,
                                 gxPoint* caretOffset);
```

`sourceShape`
: A reference to the shape object whose font metrics you need.

`port`
: The view port whose mappings you need to take into account.

`device`
: The view device whose mappings you need to take into account.

`before`
: A pointer to a `gxPoint` structure. On return, the point specifies the distance and direction to the previous line of text. For horizontal text, this corresponds to the font's ascent.

`after`
: A pointer to a `gxPoint` structure. On return, the point specifies the distance and direction to the next line of text. For horizontal text, this corresponds to the font's descent.

`caretAngle` A pointer to a `gxPoint` structure. On return, the point specifies the direction for the text-selection caret.

`caretOffset`
A pointer to a `gxPoint` structure. On return, the point specifies the direction and distance relative to the shape's origin, where the text-selection caret should intersect the baseline of the text.

DESCRIPTION

The `GXGetShapeDeviceFontMetrics` function returns the font metrics for the style object associated with the source shape in device space. These values are returned as points that represent vectors. The values of the vectors differentiate between horizontal and vertical text and account for any mapping that may be on the shape's transform as well as the specified view port and view device.

ERRORS, WARNINGS, AND NOTICES

Errors

`illegal_type_for_shape (if not typographic)` (debugging version)
`shape_is_nil`

SEE ALSO

The `GXGetStyleFontMetrics` function is described on page 6-57. The `GXGetShapeFontMetrics` function is described on page 6-58.
The `GXGetShapeLocalFontMetrics` function is described on page 6-59.

Getting and Setting the Encoding of a Style Object

The encoding in a style is the combination of the platform, script, and language. Platforms, scripts, and languages are described in the chapter "Fonts Objects" in this book.

You can use the `GXGetStyleEncoding` function to retrieve the platform, script, and language information from a style object and the `GXSetStyleEncoding` function to specify the platform, script, and language of a style object.

The `GXGetShapeEncoding` and `GXSetShapeEncoding` functions provide a way to retrieve and specify the platform, script, and language for the style object associated with a particular shape.

Text shapes contain only one encoding value. Glyphs shapes and layout shapes may contain multiple encoding values, because these shapes may contain several different styles in the styles arrays in the shapes' geometries. Each style can contain a separate encoding value.

GXGetStyleEncoding

You can use the `GXGetStyleEncoding` function to determine the platform, script, and language of a style object.

```
gxFontPlatform GXGetStyleEncoding(gxStyle source,
                                  gxFontScript *script,
                                  gxFontLanguage *language);
```

`source` A reference to the style object whose platform, script, or language you want to determine.

`script` The style object's script, returned by the function.

`language` The style object's language, returned by the function.

function result The platform of the style object.

DESCRIPTION

The `GXGetStyleEncoding` function returns the style's platform, script, and language.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`style_is_nil`

SEE ALSO

The `gxFontPlatform`, `gxFontScript`, and `gxFontLanguage` data types are discussed in the chapter "Fonts Objects" of this book.

GXSetStyleEncoding

You can use the `GXSetStyleEncoding` function to set or change the platform, script, and language of a style object.

```
void GXSetStyleEncoding(gxStyle target, gxFontPlatform platform,
                        gxFontScript script,
                        gxFontLanguage language);
```

`target` A reference to the style object whose platform, script, or language you want to change.

`platform` The new platform value.

`script`	The new script value.
`language`	The new language value.

DESCRIPTION

The `GXSetStyleEncoding` function sets the platform, script, and language of the style object. Default values for the platform, script, and language are `gxMacintoshPlatform`, `gxRomanScript`, and `gxNoLanguage`, respectively.

The `GXSetStyleEncoding` function does not change or translate the character codes in the shape. You cannot use `GXSetStyleEncoding` to convert a particular character from one encoding to another, because a character found on one platform may not be represented on another platform.

If you set the platform to the value `gxGlyphPlatform`, you should set the values of the `script` and `language` parameters to `gxNoScript` and `gxNoLanguage`.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`style_is_nil`
`inconsistent_parameters` (debugging version)

Notices (debugging version)

`encoding_already_set`

SEE ALSO

The `gxFontPlatform`, `gxFontScript`, and `gxFontLanguage` data types are discussed in the chapter “Fonts Objects” of this book.

GXGetShapeEncoding

You can use the `GXGetShapeEncoding` function to determine the encoding set for the style object of a particular QuickDraw GX typographic shape.

```
gxFontPlatform GXGetShapeEncoding(gxShape source,
                                  gxFontScript *script,
                                  gxFontLanguage *language);
```

`source`	A reference to the shape whose platform you want to determine.
`script`	The script of the style object, returned by the function.
`language`	The language of the style object, returned by the function.

function result The platform of the style object used by the specified shape.

DESCRIPTION

The `GXGetShapeEncoding` function returns the platform, script, and language of the style object associated with the shape object.

The function returns information about the platform, script, and language settings of the style object associated with the shape object. However, the glyph and layout shapes may have arrays of styles in their geometries and therefore do not necessarily use the style object attached to the shape object. In this case, you should use the `GXGetStyleEncoding` (page 6-62) and `GXSetStyleEncoding` (page 6-62) functions to determine the platform, scripts, and language the styles stored in the shape's geometry.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`style_is_nil`

SEE ALSO

The `gxFontPlatform`, `gxFontScript`, and `gxFontLanguage` data types are discussed in the chapter "Fonts Objects" of this book.

GXSetShapeEncoding

You can use the `GXSetShapeEncoding` function to alter the encoding of the style object associated with a particular shape.

```
void GXSetShapeEncoding(gxShape target, gxFontPlatform platform,
                        gxFontScript script,
                        gxFontLanguage language);
```

`target`	A reference to the shape whose platform you want to alter.
`platform`	The platform of the style object. The `gxFontPlatform` data type is discussed in the chapter "Fonts Objects" of this book.
`script`	The script of the style object. The `gxFontScript` data type is discussed in the chapter "Fonts Objects" of this book.
`language`	The language of the style object. The `gxFontLanguage` data type is discussed in the chapter "Fonts Objects" of this book.

DESCRIPTION

The `GXSetShapeEncoding` function sets the style's platform, script, and language values for the shape named by the `target` parameter. The encoding specifies the way the font interprets the text stream into a series of character codes. A font may support one or more encodings.

If a style object is shared by more than one shape, `GXSetShapeEncoding` copies the style object so that only the shape in the `target` parameter is affected by the changes to the platform, script, and language values.

The function specifies the platform, script, and language settings for the style object associated with the shape object. However, the glyph and layout shapes may have arrays of styles in their geometries and therefore do not necessarily use the style object attached to the shape object. In this case, you should use the `GXGetStyleEncoding` (page 6-62) and `GXSetStyleEncoding` (page 6-62) functions to set the encoding of the styles stored in the shape's geometry.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
out_of_memory
inconsistent_parameters                    (debugging version)
```

Notices (debugging version)

```
style_platform_already_set
```

SEE ALSO

The `gxFontPlatform`, `gxFontScript`, and `gxFontLanguage` data types are discussed in the chapter "Fonts Objects" of this book.

Getting and Setting the Text Attributes of a Style Object

The text attributes property of style objects specifies how QuickDraw GX alters glyph outlines or sets text to be horizontal or vertical. You use the `gxTextAttributes` data type, which is described on page 6-38, when retrieving or setting text attributes.

You can use the `GXGetStyleTextAttributes` function to retrieve the text attributes from a style object and the `GXSetStyleTextAttributes` function to specify the text attributes of a style object.

The `GXGetShapeTextAttributes` and `GXSetShapeTextAttributes` functions provide a way to retrieve and specify the text attributes for the style object associated with a particular shape.

GXGetStyleTextAttributes

You can use the `GXGetStyleTextAttributes` function to determine which text attributes are set for a particular style object.

```
gxTextAttribute GXGetStyleTextAttributes(gxStyle source);
```

`source` A reference to the style object whose attributes you want to determine.

function result The text attributes of the style object.

DESCRIPTION

The `GXGetStyleTextAttributes` function returns the text attributes of the style object specified by the `source` parameter.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`out_of_memory`

SEE ALSO

See “Text Attributes” on page 6-38 for a listing of text attributes.

GXSetStyleTextAttributes

You can use the `GXSetStyleTextAttributes` function to set or change the text attributes of a style object.

```
void GXSetStyleTextAttributes(gxStyle target,
                              gxTextAttribute attributes);
```

`target` A reference to the style object whose text attributes you want to change.

`attributes` The new text attributes for the style.

DESCRIPTION

The `GXSetStyleTextAttributes` function sets the text attributes of the style object specified by the `target` parameter to those specified in the `attributes` parameter.

You should always get the current settings of the text attributes before setting any of them. The `GXSetStyleTextAttributes` function replaces all of the attributes currently associated with the shape; if you want any attributes to remain the same, you must include them in the call, unless you want to set them all explicitly.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
out_of_memory
parameter_out_of_range                    (debugging version)
```

Notices (debugging version)

```
text_attributes_already_set
```

SEE ALSO

Text attributes are described on page 6-38.

GXGetShapeTextAttributes

You can use the `GXGetShapeTextAttributes` function to determine which text attributes are set for the style object of a particular QuickDraw GX typographic shape.

```
gxTextAttribute GXGetShapeTextAttributes(gxShape source);
```

`source` A reference to the shape whose text attributes you want to determine.

function result The text attributes of the style object attached to the `source` specified by shape.

DESCRIPTION

The `GXGetShapeTextAttributes` function returns the text attributes of the style object associated with the shape specified by the `source` parameter.

This function provides a convenient way to determine the text attributes of a shape without calling the `GXGetShapeStyle` function to obtain a reference to the shape's style object. However, the glyph and layout shapes may have arrays of styles in their geometries and therefore do not necessarily use the style object attached to the shape object. In this case, you should use the `GXGetStyleTextAttribute` and `GXSetStyleTextAttribute` functions to determine the attributes of the styles stored in the shape's geometry.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
out_of_memory
```

SEE ALSO

See "Text Attributes" on page 6-38 for a listing of text attributes.

GXSetShapeTextAttributes

You can use the GXSetShapeTextAttributes function to alter the text attributes of the style object associated with a particular shape.

```
void GXSetShapeTextAttributes(gxShape target,
                              gxTextAttribute attributes);
```

`target` A reference to the shape whose text attributes you want to alter.

`attributes` The new set of text attributes.

DESCRIPTION

The GXSetShapeTextAttributes function sets the text attributes of the style object associated with the shape specified by the `target` parameter. If a style object is shared among shapes, GXSetShapeTextAttributes copies the style object so that only the shape in the target parameter is affected by the changes to the text attributes.

This function provides a convenient way to set the style attributes of a shape without calling the GXGetShapeStyle function to obtain a reference to the shape's style object.

You can use this function in combination with the GXGetShapeTextAttributes function (page 6-67) to set or clear single style attributes. However, the glyph and layout shapes may have arrays of styles in their geometries and therefore do not necessarily use the style object attached to the shape object. In this case, you should use the GXGetStyleTextAttribute (page 6-66) and GXSetStyleTextAttribute (page 6-66) functions to set the attributes of the styles stored in the shape's geometry.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`out_of_memory`
`parameter_out_of_range` (debugging version)

Notices (debugging version)

`text_attributes_already_set`

SEE ALSO

Text attributes are described on page 6-38.

Summary of Typographic Styles

Constants and Data Types

```
enum gxTextAttributes {
   gxAutoAdvanceText    = 0x0001,
   gxNoContourGridText  = 0x0002,
   gxNoMetricsGridText  = 0x0004,
   gxAnchorPointsText   = 0x0008,
   gxVerticalText       = 0x0010,
   gxNoOpticalScaleText = 0x0020
} ;

typedef long gxTextAttribute;

#define gxLeftJustify 0
#define gxCenterJustify (fract1/2)
#define gxRightJustify fract1
#define gxFillJustify -1

enum gxLayerFlags;{
   gxUnderlineAdvanceLayer = 0x0001,
      /* a gxLine is drawn through the advances */
   gxSkipWhiteSpaceLayer = 0x0002,
      /* except characters describing white space */
   gxUnderlineIntervalLayer = 0x0004,
      /* (+ gxStringLayer) a gxLine is drawn through the gaps
      between advances */
   gxUnderlineContinuationLayer = 0x0008,
      /* (+ gxStringLayer) join this underline with
      another face */
   gxWhiteLayer = 0x0010,
      /* the layer draws to white instead of black */
   gxClipLayer = 0x0020,
      /* the characters define a clip */
   gxStringLayer = 0x0040
      /* all characters in run are combined */
}

typedef long gxLayerFlag;
```

```
typedef struct {
   gxShapeFill outlineFill;/* outline framed or filled */
   gxLayerFlag flags;      /* various additional effects */
   gxStyle  outlineStyle;/* outline */
   gxTransform outlineTransform;/* italic, condense, extend */
   gxPoint  boldOutset;/* bold */
} gxFaceLayer;

typedef struct {
   long        faceLayers;/* layer to implement shadow */
   gxMapping   advanceMapping;/* algorithmic change to advance
                           width */
   gxFaceLayer gxFaceLayer[gxAnyNumber];/* zero or more face
                           layers describing the face */
} gxTextFace;
```

Functions

Getting and Setting the Font of a Style Object

```
gxFont GXGetStyleFont       (gxStyle source);
void GXSetStyleFont         (gxStyle target, gxFont aFont);
gxFont GXGetShapeFont       (gxShape source);
void GXSetShapeFont         (gxShape target, gxFont aFont);
```

Getting and Setting the Text Face

```
long GXGetStyleFace         (gxStyle source, gxTextFace *face);
void GXSetStyleFace         (gxStyle target, const gxTextFace *face);
long GXGetShapeFace         (gxShape source, gxTextFace *face);
void GXSetShapeFace         (gxShape target, const gxTextFace *face);
```

Getting and Setting the Text Size of a Style Object

```
Fixed GXGetStyleTextSize    (gxStyle source);
void GXSetStyleTextSize     (gxStyle target, Fixed size);
Fixed GXGetShapeTextSize    (gxShape source);
void GXSetShapeTextSize     (gxShape target, Fixed size);
```

Getting and Setting the Alignment of a Style Object

```
fract GXGetStyleJustification
                            (gxStyle source);
void GXSetStyleJustification
                            (gxStyle target, fract justify);
```

```
fract GXGetShapeJustification
                            (gxShape source);
void GXSetShapeJustification(gxShape target, fract justify);
```

Getting and Setting the Style Font Variations of a Style Object

```
long GXGetStyleFontVariations
                            (gxStyle source, gxFontVariation variations[]);
void GXSetStyleFontVariations
                            (gxStyle target, long count,
                             const gxFontVariation variations[]);
long GXGetShapeFontVariations
                            (gxShape source, gxFontVariation variations[]);
void GXSetShapeFontVariations
                            (gxShape target, long count,
                             const gxFontVariation variations[]);
```

Retrieving the Elements in a Font Variation Suite

```
void GXGetStyleFontVariationSuite
                            (gxStyle source, gxFontVariation variations[]);
void GXGetShapeFontVariationSuite
                            (gxShape source, gxFontVariation variations[]);
```

Retrieving Font Metrics

```
void GXGetStyleFontMetrics  (gxStyle sourceStyle, gxPoint* before,
                             gxPoint* after, gxPoint* caretAngle,
                             gxPoint* caretOffset);
void GXGetShapeFontMetrics  (gxShape source, gxPoint* before, gxPoint* after,
                             gxPoint* caretAngle, gxPoint* caretOffset);
void GXGetShapeLocalFontMetrics
                            (gxShape sourceShape, gxPoint* before,
                             gxPoint* after, gxPoint* caretAngle,
                             gxPoint* caretOffset);
void GXGetShapeDeviceFontMetrics
                            (gxShape sourceShape, gxViewPort port,
                             gxViewDevice device, gxPoint* before,
                             gxPoint* after, gxPoint* caretAngle,
                             gxPoint* caretOffset);
```

Getting and Setting the Encoding of a Style Object

```
gxFontPlatform GXGetStyleEncoding
                            (gxStyle source, gxFontScript *script,
                             gxFontLanguage *language);
void GXSetStyleEncoding     (gxStyle target, gxFontPlatform platform,
                             gxFontScript script, gxFontLanguage language);
gxFontPlatform GXGetShapeEncoding
                            (gxShape source, gxFontScript *script,
                             gxFontLanguage *language);
void GXSetShapeEncoding     (gxShape target, gxFontPlatform platform,
                             gxFontScript script, gxFontLanguage language);
```

Getting and Setting the Text Attributes of a Style Object

```
gxTextAttribute GXGetStyleTextAttributes
                             (gxStyle source);
void GXSetStyleTextAttributes
                            (gxStyle target, gxTextAttribute attributes);
gxTextAttribute GXGetShapeTextAttributes
                            (gxShape source);
void GXSetShapeTextAttributes
                            (gxShape target, gxTextAttribute attributes);
```

CHAPTER 7

Font Objects

Contents

This chapter describes font objects and the functions you can use to manipulate them. Read this chapter if you use any kind of font object for the QuickDraw GX shapes you create.

Before reading this chapter, you should be familiar with the information in the chapter "Introduction to QuickDraw GX Typography" in this book. You should also be familiar with the information discussed in *Inside Macintosh: QuickDraw GX Objects*.

This chapter introduces QuickDraw GX font objects and describes their properties. It then shows you how to create, dispose of, and manipulate these objects to

- gain access to the system's font list
- specify in a style object how a font should be used—for example, specify encodings, variations, and font features
- edit a font and add your own fonts

This chapter also lists and cross-references font-related QuickDraw GX functions that are described elsewhere in this book and in other parts of *Inside Macintosh*.

About Font Objects

Fonts are represented in QuickDraw GX as **font objects.** To draw or print text, you must reference or use a font object.

Fonts come in a variety of formats and can be stored in many different ways. In QuickDraw GX, fonts are consolidated into a single object type that hides the complexity of the font data from your application. With QuickDraw GX, you can have a single object type, as well as a single set of methods of accessing the font data.

Each QuickDraw GX font object encapsulates a particular style of text—for example, Times Bold or Times Italic. The same font object is used for all point sizes.

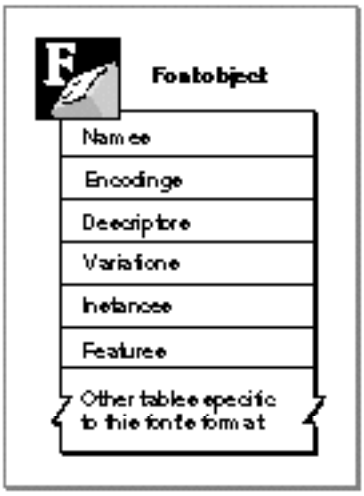
A collection of QuickDraw GX font objects that share a common design is known as a **font family**—for example, Times, which is composed of Times Roman, Times Bold, Times Italic, and Times Bold Italic. By grouping all such fonts into a font family, QuickDraw GX allows your application to display only family names in the Font menu, thereby simplifying for the user the process of choosing a font.

The interface to each QuickDraw GX font object is entirely procedural—you cannot access any information in the fonts directly. To manipulate the pieces of information in the font object—for example, its properties—you must use QuickDraw GX functions.

Font Object Properties

QuickDraw GX font objects have six accessible properties, as shown in Figure 7-1. Note that, because a font is an object and not a data structure, the order of the properties as shown in Figure 7-1 is completely arbitrary.

Figure 7-1 The QuickDraw GX font object and its accessible properties



The six accessible properties include

- **Names.** Text in a font that contains information about the font.
- **Encodings.** Internal conversion tables for interpreting a specific character set.
- **Descriptors.** Identifying characteristics used to measure the stylistic attributes of one font in comparison to another font—for example, whether one font is “bolder” than another.
- **Variations.** A setting along an axis. This can be the range, from minimum to maximum, of each variation axis in a font.
- **Instances.** A named font variation coordinate.
- **Features.** A type that holds information about a font feature.

Every QuickDraw GX font object contains font names and encodings. The other properties shown in Figure 7-1 offer additional functionality but may not be present in every font. In addition to these six accessible properties, a QuickDraw GX font object contains other properties specific to the font format, depending on whether the format is TrueType GX, Type 1 GX, or 'NFNT', or another format.

QuickDraw GX provides functions for manipulating each of these font object properties. These functions are described in the section “Font Objects Reference” beginning on page 7-21.

Names

Each QuickDraw GX font object contains a set of font names. **Font names** provide specific information about a font, such as its family name, style, copyright date, version, and manufacturer. Some font names can be used to build menus in your application—for example, family and style—whereas other names are used to identify the font

uniquely—for example, the unique and PostScript names. Here are some examples of font names with predefined selectors:

- **Font family.** This name is shared by all styles in a family. An example is “New York”.
- **Style.** This stylistic variation distinguishes a font from other members of the same family. Examples are “Regular”, “Italic”, “Bold”, or “Black”.
- **Unique name.** This is the manufacturer’s name for the font. An example is “Apple Computer New York Black 3.0 8/10/92”.
- **Full font name.** An example is “New York Black”.
- **Copyright.** This is the manufacturer’s copyright notice. An example of this is “© Apple Computer, Inc. 1993.”
- **Version.** This is the font manufacturer’s version number. An example is “3.0.”
- **PostScript name.** This is the PostScript-legible name of the font. An example is “NewYork-Black”.
- **Trademark.** This the trademark holder’s name. An example is “Palatino is a registered trademark of Linotype AG”.
- **Manufacturer.** This is the font manufacturer’s name. An example is “Apple Computer, Inc.”

Note

In addition to predefined name selectors, a font may contain other names that describe parts of the font, such as ligatures or swashes. However, these names do not have predefined name selectors. You can gain access to these names through QuickDraw GX features and variations, which are discussed later in this section. u

Font names are identified by a font’s platform, script, and language, which are discussed in the next section.

A QuickDraw GX font object can have font names in any of the languages specified by the `gxFontLanguage` enumeration. For example, a font manufacturer can store the English name of a font as “Geneva Bold” and the French name of the font as “Geneva Gras.” The currently defined languages are listed in “Languages” beginning on page 7-28.

Encodings

Each QuickDraw GX font object contains a certain number of **character encodings.** Each encoding is an internal conversion table for interpreting a specific character set—that is, a way to map a character code to a glyph code for that font.

Like font names, font encodings are also identified by platform, script, and language. A **platform** describes what standard the character set was designed for, such as the Macintosh computer or the Unicode standard. Possible platform values are listed in “Font Platforms” on page 7-25. A custom platform is one whose encoding does not correspond to a specific standard. A **script** is a writing system, or a set of characters with basic rules of use in creating a visual depiction of one or many languages. For example, the Arabic script can be used to depict written Arabic, Egyptian, or classical Arabic. Each script has a

unique set of attributes. Roman script has a general left-to-right direction of text, whereas Arabic script has a general right-to-left direction of text. Printed Roman characters are relatively independent of each other; Arabic glyphs change shape depending on the glyphs that surround them. Some writing systems, such as Roman and Cyrillic, are basically **alphabetic**: glyphs symbolize discrete phonemic elements in the language. Other writing systems, including Japanese Kana, are **syllabic**: the glyphs stand for syllables in the language. Still other writing systems—namely, Japanese Kanji, Chinese Hanzi, and Korean Hanja—include **ideographic** glyphs. In these systems, glyphs do not represent pronunciation alone but are related to the component meanings of words. A typical character set for an ideographic writing system can be quite large, ranging from 7000 to more than 30,000 characters. A written **language** refers to the whole body of written words—and methods of combining words to create meaning—used by a particular group of people.

Figure 7-2 shows examples of alphabetic, syllabic, and ideographic representations of characters.

Figure 7-2 Words with alphabetic, syllabic, and ideographic characters

In general, the encoding of a font won't be identified by a specific language because platform and script are enough to identify a character encoding. The exception to this rule occurs in a few Macintosh scripts, where two or more languages in the same script represent two different character sets. For example, Turkish and Croatian are both languages in the Roman script but have different encodings.

Character Code Sizes

QuickDraw GX supports character code sizes larger than 7-bit ASCII codes (0-127). These include

- 16-bit character codes, as used by Microsoft and Unicode, in which each character is stored as a 16-bit number
- a mixed 8-/16-bit encoding, as used by Chinese, Japanese, and Korean scripts, in which certain byte values are set aside to signal the first byte of a 2-byte character. The value of the first byte specifies whether you include the value of the next byte as well.
- 8-bit character codes, where each character is stored as an 8-bit number

You can use a font's encoding to find out what character code size it uses. Because each encoding table in a font is identified by platform and script, each combination of platform and script specifies the size of the character code that the table can translate. Depending on platform and script, your application may need to generate character codes of different sizes.

Table 7-1 enumerates the platforms and scripts that QuickDraw GX supports in relation to character code sizes. Note that all are either 8- or 16-bit, but some scripts—such as Chinese, Japanese, and Korean—use a mixed 8-/16-bit encoding.

Table 7-1 Character code sizes among various platforms and scripts

Character code size (bits)	Platform	Script
16	Unicode	(All Unicode versions)
16	Custom	Custom 16-bit script
16	Microsoft	(All Microsoft scripts)
mixed 8/16	Macintosh	Chinese
mixed 8/16	Macintosh	Korean
mixed 8/16	Macintosh	Japanese
mixed 8/16	Macintosh	Simplified Chinese
mixed 8/16	Custom	Custom 8-bit/16-bit script
8	Macintosh	(Any Macintosh script not already listed previously)
8	Custom	Custom 8-bit script

To find out what encodings a font supports, you use the `GXGetFontEncoding` function, described on page 7-44. To search for all the fonts that support a given encoding, you use the `GXFindFonts` function, described on page 7-33.

Font Descriptors

Each font object within a family has a certain number of identifiable characteristics called **font descriptors,** and these define such attributes in a font as its weight, width, italic slant and optical point size—that is, the point size for which the font was designed. A font descriptor is a data structure that allows your application to read and measure the stylistic attributes of one font versus another font—to determine, for example, if one font is "bolder" than another.

Font descriptors give numerical values for these stylistic attributes.

Table 7-2 lists some predefined descriptors. For more information about ornamental sets, see the chapter "Layout Styles" in this book.

Table 7-2 A list of predefined font descriptors

Descriptor	Descriptor tag	Description
Weight	`'wght'`	The thickness of the line used to draw a glyph of the font. A plain font might have a value of 1.0.
Width	`'wdth'`	The width or narrowness of the glyphs. A plain font might have a value of 1.0.
Italic slant	`'slnt'`	The number of degrees, from –90 to 90, by which the glyphs lean. A plain font might have a value of 0 degrees.
Optical point size	`'opsz'`	The point size for which the font was designed. A common value might be 12.0 points.
Nonalphabetic	`'nalf'`	The value corresponding to a nonalphabetic form provided by a font, The values are identical to those for ornamental sets: dingbats equal 1, Pi characters equal 2, fleurons equal 3, decorative borders equal 4, international symbols equal 5, and math symbols equal 6. An undefined value or a value of 0 indicates that the font is alphabetic.

Font Variations

For most fonts, a single list of descriptors is enough to describe its appearance. However, some fonts are capable of generating a wide range of stylistic changes. Such a font contains **font variation** axes, each of which describes a particular stylistic attribute and the range of values that the font can use.

Font variation axes are named by the same tags used by descriptors—for example, `'wght'`. Each axis has a minimum, maximum, and default value. The minimum and maximum values determine the range of values that the variation axis covers. If a font contains both a font descriptor and a font variation with the same tag, the default value of the variation is equal to the descriptor.

In Figure 7-3, the variation axis `'wght'` has a default value of 1.0, and minimum and maximum values of 0.62 and 1.3. This font can create a range of glyphs of varying thicknesses—from light to bold—that your application can draw.

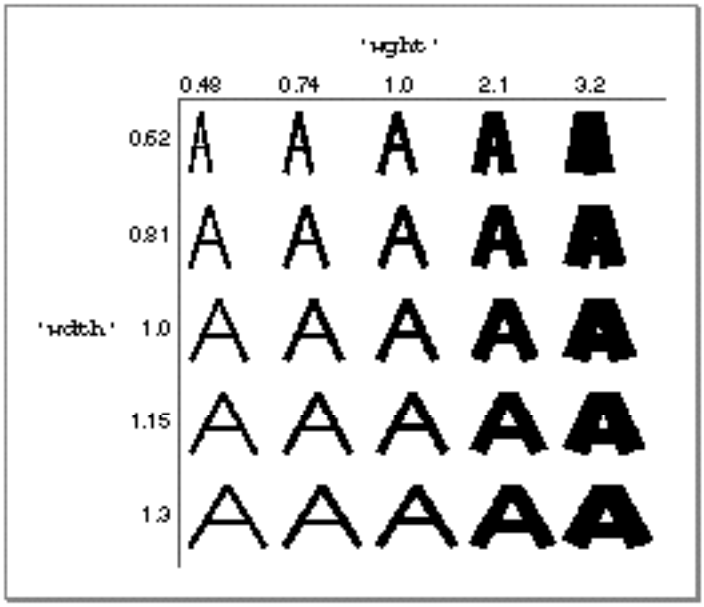
Figure 7-3 Font variations along the `'wght'` variation axis

A font may also name specific values along a variation axis as font instances, described in the next section.

In addition, your application can combine multiple style coordinates. For example, a font may have a `'wght'` axis and a `'wdth'` axis. The user can then select any combination of bold and condensed values—such as 75 percent bold and 50 percent condensed.

Figure 7-4 shows an example of a font variation with two axes. In Figure 7-4, the weight axis has a minimum value of 0.48 and a maximum of 3.2, and the width axis has a minimum of 0.62 and a maximum of 1.3.

Figure 7-4 Font variations for the `'wght'` and `'wdth'` axes



QuickDraw GX provides your application with functions that allow you to count the number of font variation axes in a particular font and to specify a font variation by index or by tag (such as `'wdth'`) in the font's list of font variations.

Font Instances

A **font instance** is a named font variation coordinate. It is a setting identified by a type designer that includes a value for each variation axis in the font. A font instance provides that set of values with a name, which is stored in the names property of the font object. For example, if a font has a weight variation axis whose default is 1.0 and whose maximum is 1.5, a font instance might be "Demibold" with a `'wght'` axis value of 1.2.

Your application can make font instances available to the user as part of the user's font and style selection mechanism, along with other font and style selections.

Font Features

QuickDraw GX fonts can include **font features—**changes to the selection of glyphs in the font, such as automatic ligature formation and cursive connections between glyphs. Some of these changes are essential for writing systems such as Arabic or Hindi that require changes to glyphs and words depending on context; some are useful simply to enhance the appearance of text. Apple Computer, Inc. has defined a standard set of font features.

Features are grouped into **types;** each feature type has some number of **feature settings.** For example, in the lettercase feature type, the settings would represent all lowercase, uppercase, or small caps. This means that in a style or shape object, you can specify features with particular settings that modify what gets drawn.

Font features allow your application to draw different versions of a letter—for example, small caps, inferiors, and ligatures.

The feature types and their feature settings are described in the chapter “Layout Styles” in this book.

Note

Feature types and feature selectors are defined and listed in the **QuickDraw GX Font Feature Registry,** a document maintained by Apple Computer, Inc. To obtain the latest version of the feature registry, please contact Apple Computer at the AppleLink address FONTREGISTRY. u

QuickDraw GX Font Formats

QuickDraw GX provides a single, consistent application programming interface (API) for all GX font objects. It also provides you with functions for finding a certain font and then accessing its properties rather than requiring your application to interpret the font’s data directly.

At the same time, the API gives you flexibility in your implementation of a font object—for example, in your choice of font format and methods of storing and referencing fonts. Font objects in QuickDraw GX can represent a variety of font formats, including but not limited to

- TrueType GX
- Type 1 GX
- `'NFNT'`

Your application can use a font in any of these font formats. Because all fonts are accessible through the same set of functions, your application should not need to know the font’s format.

How Font Objects Are Stored and Referenced

In the QuickDraw GX architecture, each font object has a **storage type,** describing the method used to store the font. A font is stored in one of four ways:

- with a resource ID
- with a memory handle
- with a file specification
- as a `'FOND'` resource

Along with its storage type, each font has a storage reference, which specifies the instance of the storage type it uses.

You can determine how a font is stored simply by checking its associated storage type, which is of the `gxFontStorageTag` type. Table 7-3 shows the storage types and what each means.

Table 7-3 QuickDraw GX storage types

Storage tag	Storage reference
`gxResourceFontStorage`	Resource handle
`gxHandleFontStorage`	Handle
`gxFileFontStorage`	File reference
`gxNfntFontStorage`	A 32-bit value with the `txFace` in the high-order 16 bits and the `txFont` in the low-order 16 bits.

Note

When QuickDraw GX is initialized, it searches the Fonts Folder for resources of type `'sfnt'` and `'FOND'`. Then `GXNewFont` is called with the `'sfnt'` resources—after which, `GXNewFont` is called with each `'FOND'` resource that does not reference any `'sfnt'` resources. In this way, when your application uses QuickDraw GX, all available `'sfnt'` resources and bitmap-only `'FOND'` resources are already instantiated as font objects. u

QuickDraw GX creates a list of fonts currently available in the system and identifies each font object by a unique 4-byte ID. This font ID is the font object reference found in the font property of the style object; it is also the same as the `fontID` parameter used in the functions described in the “Font Objects Reference” section of this chapter. It is not, however, the same as the storage reference in Table 7-3.

IMPORTANT

Although this font ID is a unique reference to a font, it is *not* guaranteed to be unique across different applications, because it is regenerated each time an application is launched. Therefore, your application should never save this ID in a document; instead, it should use the font’s unique name. s

Font Attributes

Each font has a set of **font attributes,** which are a group of flags that modify the behavior or identity of the font. These flags are defined in the `gxFontAttributes` enumeration.

Font Embedding

Font embedding refers to the technique of storing the font object's binary data in a document. The advantage of this technique is that the text in the document always displays the correct font, even if it is moved to another computer. The disadvantage is that this technique increases the size of the document.

Two QuickDraw GX functions implement font embedding: `GXNewFont` and `GXFlattenFont`.

The `GXNewFont` function, described on page 7-64, takes the binary data of a font and returns the font object. The `GXFlattenFont` function, described on page 7-65, takes a font object and returns its binary data.

Font Tables

The information in a font object is organized into a series of tables, each identified by a 4-byte tag. Table 7-4 shows some of the standard tables.

Table 7-4 Font tables and their contents

Storage table	Attributes
`'name'`	Font names
`'cmap'`	Font encodings
`'fdsc'`	Font descriptors

QuickDraw GX provides functions for looking for a specific table, iterating through the tables, and changing or deleting a table. These functions are described in "Advanced Font Functions" beginning on page 7-63.

There are other tables that are private to that font's format.

IMPORTANT

You should never alter the data in a font, including the font tables, unless your application is a font editor or otherwise needs to change the data in or a name of a font. Most applications, such as word processors, do not need to change the data in a font. s

The tables of a TrueType GX font are described in *QuickDraw GX Font Formats*, available from APDA. In most cases, a general-purpose application does not need to know or use the formats of the font's tables.

The List of Available Fonts

When you initialize a QuickDraw GX graphics client, QuickDraw GX creates font objects for all of the fonts in the system Font folder. These become available as a list to the application. You can access and retrieve this list of available fonts by calling the `GXFindFonts` function, described in the following section "Using Font Objects" and on page 7-33.

The Default Font

QuickDraw GX provides you with functions to manipulate the default font. When you first create a typographic shape, for example, its style is initialized to have a font reference of `nil`. If QuickDraw GX is passed a `nil` for the font parameter, it substitutes the default font—initially, Helvetica, if it is available—for that parameter. To manipulate the default font and change from Helvetica to a different font, you can use the `GXGetDefaultFont` and `GXSetDefaultFont` functions, described on page 7-35 and page 7-36, respectively.

Using Font Objects

In QuickDraw GX, you can perform a variety of operations with font objects. These operations can be grouped into the following categories:

- getting information about the available fonts in the system
- gaining access to the properties of fonts
- changing or deleting information stored in the tables of a font—but only for fonts created by your application, not those stored in the system

Note

QuickDraw GX functions that begin with *GXGet* and *GXFind* are similar in that they retrieve roughly the same information. However, the *GXGet* functions find the information by its index in the font's list of that type of information and return or name a tag or a font name, whereas the *GXFind* functions use a tag to find the information and return its index. u

Getting Information About Available Fonts

If you want to get the list of available fonts, you call `GXFindFonts` first to get the number of fonts available, as shown in Listing 7-1. The function returns the number of fonts that meet the search criteria. The `GXFindFonts` function, described on page 7-33, allows you to get a list of available fonts or any subset of available fonts—from a single font to a list of font families—in the system.

The function copies the fonts it finds into the final parameter, which serves as the buffer references for your application. You can then create a buffer large enough to store the font references and call `GXFindFonts` again, passing it the buffer (`fontList`) to get the list.

Listing 7-1 Obtaining a list of available fonts in the system

```
/* return the number of fonts */
long count = GXFindFonts(nil, 0, 0, 0, 0, 0, nil, 1,
                         gxSelectToEnd, nil);
/* copy the fonts into the font list */
gxFont* fontList = (gxFont*)NewPtr(count * sizeof(gxFont));
GXFindFonts(nil, 0, 0, 0, 0, 0, nil, 1, count, fontList);
```

To get a reference to the second available font, you can call

```
GXFindFonts(nil, 0, 0, 0, 0, 0, nil, 0, 2, 1, &myFont);
```

To get the number of font families available, use

```
GXFindFonts(nil, gxFamilyFontName, 0, 0, 0, 0, nil, 0, 1,
            gxSelectToEnd, nil);
```

To get a reference to the second font family in the list of available font families, use

```
GXFindFonts(nil, gxFamilyFontName, 0, 0, 0, 0, nil, 0, 2, 1,
            &myFont)
```

To get information about the number of fonts available in a single font family, you call

```
count = GXFindFonts(myFontFamily, 0, 0, 0, 0, 0, nil, 0, 1,
                    gxSelectToEnd, nil);
```

where `myFontFamily` is a reference to the font whose family you want information about.

Likewise, if you want a reference to the second font in a particular font family, call

```
GXFindFonts(myFontFamily, 0, 0, 0, 0, 0, nil, 0, 2, 1, &myFont);
```

Drawing With a Specific Font

To find a font by name, you use `GXFindFonts`. You call this function when you open a document that specifies a font. For example, if you want a reference to the font Chicago, use

```
GXFindFonts(nil, gxFullFontName, gxMacintoshPlatform,
            gxRomanScript, gxEnglishLanguage, strlen("Chicago"),
            "Chicago", 1, 1, &myFont);
```

Once you have found a specific font, you can use the following code to associate the font object with the shape object, so that a particular shape uses that font:

```
GXSetShapeFont(myShape, myFont);
```

Now you can draw the shape.

```
GXDrawShape(myShape);
```

Gaining Access to Font Properties

This section describes how you can gain access to and manipulate various font properties.

Getting a Font Name

If your application needs the name of a font (for example, to display it in a menu), you can use the `GXGetFontName` function.

If you want to get a font name by index (for example, if you are iterating through all font names in the font), you can also call the `GXGetFontName` function to get the size of a font name and then call it a second time to retrieve the name, as shown in Listing 7-2.

Note
If you are trying to get a font name by its meaning, platform, script, and language, use `GXFindFontName`. u

Listing 7-2 Using the `GXGetFontName` function

```
long length = GXGetFontName(myFont, index, nil, nil, nil, nil,
                            nil);
unsigned char* name = NewPtr(length);
GXGetFontName(myFont, index, nil, nil, nil, nil, name);
```

To return the font's full name as a C string, you can write the `FindFontFullName` library function, as shown in Listing 7-3.

Listing 7-3 Extracting a full name as a C string

```
/* return the font's full name as a C string */
short GXFindFontFullName(gxFont fontID, char name[])
{
   short length

   length = FindFontName(fontID, gxFullFontName,
                         gxMacintoshPlatform, gxRomanScript,
                         gxEnglishLanguage,
                         (unsigned char*)name, nil);
   name[length] = 0;
   return length;
}
```

Adding a Font Instance

If you add an attribute—for example, a font instance—to a font that has a font name, you must also add the name for that attribute. To add the attribute name, you must get an unused font name for the new name by using the `GXNewFontNameID` function.

Listing 7-4 shows one method of adding a new font instance and a font name for that instance to a font. The function takes a font, a font name, and coordinates for the new font instance. It gets a new `gxFontName` from the `GXNewFontNameID` function, adds the new font name to the font using the `GXSetFontName` function, and then adds the font instance to the font, using the `GXSetFontInstance` function to coordinate the new font instance with its associated font name.

Listing 7-4 Adding a new font name to a font

```
void AddFontInstance(gxFont theFont, const char instanceName[],
                     const gxFontVariation coord[])
{
   gxFontName nameID;

   nameID = GXNewFontNameID(theFont);
   GXSetFontName(theFont, nameID, gxMacintoshPlatform,
                 gxRomanScript, gxEnglishLanguage,
                 strlen(instanceName),
                 (unsigned char*)instanceName);
   GXSetFontInstance(theFont, 0, nameID, coord);
}
```

The GXNewFontNameID function is described on page 7-40. The GXSetFontName function is described on page 7-41. The GXSetFontInstance function is described on page 7-57.

Retrieving Font Features

This section describes the code you can use for retrieving font features. This technique is useful if your application needs to put in menus for features and instances. Listing 7-5 shows how you can retrieve an array of font features—in this case, ligature settings available in the font.

Listing 7-5 Retrieving an array of ligature settings

```
/*
Return the array of ligature settings available
in this font, or nil if they are not available.
*/
gxFontFeatureSetting* ReturnLigatureSettings(gxFont fontID)
{

   long index;
   gxFontName nameID;
   gxFontFeatureSetting* settings;
   settings = nil;
   nameID = GXFindFontFeature(fontID, ligaturesType, nil, &count,
                              nil, &index);
   if (nameID != 0)
   {
   settings = (gxFontFeatureSetting*)NewPtr(count *
               sizeof(gxFontFeatureSetting));
               GXGetFontFeature(fontID, index, nil, nil,
               settings, nil);
   }
   return settings;
}
```

The GXFindFontFeature function is described on page 7-62. The GXGetFontFeature function is described on page 7-61.

Determining Font Variations

Listing 7-6 show you how to determine whether a font has a variation axis. This technique is useful if your application needs to put in sliders for font variations.

Listing 7-6 Determining font variations

```
/*
Determines whether font has a "weight" variation axis; if it does,
returns its maximum value.
*/
Fixed ReturnMaxWeightVariation(gxFont fontID)
{
   long index;
   Fixed max;
   index = GXFindFontVariation(fontID, 'wght', nil, nil, &max,
                               nil);
   if (index == 0)
      max = 0;// no 'wght' axis available
   return max
}
```

The `GXFindFontVariation` function is described on page 7-55.

Retrieving Language-Specific Font Lists

Listing 7-7 shows you how to retrieve a language-specific font list. This technique is useful, for example, if you need to add just Japanese fonts in a Font menu. Listing 7-7 shows an example of how you can return all the fonts that support Japanese characters.

Listing 7-7 Retrieving all fonts that support Japanese characters

```
/* return all of the fonts that support Japanese characters*/
long FindJapaneseFonts(gxFont fontList[])
{
   long numJapaneseFonts = GXFindFonts(nil, gxNoFontName,
gxMacintoshPlatform, gxJapaneseScript, gxNoLanguage,
                     0, nil, 1, gxSelectToEnd, fontList);
   return numJapaneseFonts;
}
```

The `GXFindFonts` function is described on page 7-33.

Manipulating Font Tables

To get the number of font tables in a font, you can use the `GXCountFontTables` function, described on page 7-70. To get an entire font table that you specify by its index in the font's list of font tables, you use the `GXGetFontTable` function, described on page 7-71. To get part of a font table that you specify by index, you use the `GXGetFontTableParts` function, described on page 7-72.

To get an entire font table that you specify by its table tag, use `GXFindFontTable`. To get part of a font table that you specify by table tag from a font, you use `GXFindFontTableParts`.

To change part of an existing font table or add a new font table, use `GXSetFontTable`. To delete a font table from a font permanently, you use `GXDeleteFontTable`.

IMPORTANT

You should never alter the data in a font, including the font tables, unless your application is a font editor or otherwise needs to change the data in a font. Most applications, such as word processors, do not need to examine or change the data in a font. s

The tables of a TrueType GX font are described in *QuickDraw GX Font Formats*, available from APDA.

If you need to take advantage of some of QuickDraw GX's advanced font functions—for example, if your application is a font editor—you can use the `GXGetFontTable` function. This function allows you to retrieve the size of a particular font table and then call it a second time to retrieve the actual data from the table, as shown in Listing 7-8.

Listing 7-8 Using the `GXGetFontTable` function to retrieve a table

```
long size = GXGetFontTable(fontID, index, nil, nil);
void* tableData = NewPtr(size);
GXGetFontTable(fontID, index, tableData, nil);
```

The `GXGetFontTable` function is described on page 7-71.

Font Objects Reference

This section describes the enumerations, constants, data types, and functions that are specific to the QuickDraw GX font object.

"Basic Constants and Data Types" and "Advanced Constants and Data Types" list the enumerated types and structures that provide information about fonts.

"Basic Font Functions" beginning on page 7-32 lists the QuickDraw GX functions you use to manipulate QuickDraw GX fonts. "Advanced Font Functions" beginning on page 7-63 lists the QuickDraw GX functions you use to change the information a font contains.

Basic Constants and Data Types

This section describes the constants and data types that you can use to provide basic information about and retrieve basic information from QuickDraw GX font objects. The data types discussed in this section include

- the `gxFont` type, which you use to reference a font
- the `gxFontVariation` type, which you use to reference font variation and font instance information
- the `gxFontDescriptorTag` type, which you use to reference a font descriptor
- the `gxFontVariationTag` type, which you use to reference a variation axis
- the `gxFontName` type which distinguishes the types of font names available in a font
- the `gxFontFeatureFlag` enumeration, which you use to get information about a particular font feature in the font—for example, whether the settings of that feature are mutually exclusive
- the `gxFontFeature` type, which is a reference to a specific font feature
- the `gxFontFeatureSetting` type, which contains a setting of a font feature and the name ID in the name table of that setting
- the `gxFontPlatform` type, which identifies the platform for a name or encoding
- the `gxFontScript` type, which specifies the script used by a platform
- the `gxFontLanguage` type, which specifies the language used by a script

The Font Object

To gain access to font objects in QuickDraw GX, you use always functions. Your application never gets a pointer or handle to the actual font data. To allow type checking, QuickDraw GX defines the `gxFont` data type as a pointer to a structure.

```
typedef struct gxPrivateFontRecord *gxFont;
```

To obtain or alter information in a font object, you pass `gxFont` to a function. In general, you should never need to change or gain access to the data in a font object directly.

Font Variations, Instances, and Descriptors

The `gxFontVariation` structure describes font variations, instances, and descriptors. The `gxFontVariationTag` type identifies a variation axis.

```
struct gxFontVariation {
   gxFontVariationTag   name;
   Fixed                value;
} ;
typedef struct gxFontVariation gxFontDescriptor;
```

Field descriptions

`name`	The name of the variation axis, in a four-character tag. For example, `'wght'` is the name of the weight variation axis.
`value`	A coordinate along the variation axis specified by name.

You can get the minimum and maximum values for an axis using the `GXGetFontVariation` function, described on page 7-54, or the `GXFindFontVariation` function, described on page 7-55.

The `gxFontDescriptor` structure is identical to the `gxFontVariation` structure. The `gxFontDescriptorTag` identifies a font descriptor. Note that descriptors are used to identify the style of a simple font—that is, a font that does not have variations.

```
typedef long gxFontDescriptorTag
```

Font variations and instances, which use the variation axes, are discussed in "Font Variations" on page 7-10.

The possible values for the `name` field of the `gxFontVariation` structure, whether it describes a variation, instance, or descriptor, are described in "Font Variations" on page 7-10.

Font Names

Font names are values in a font that contain information about the font. Each font name is distinguished by a value from the `gxFontName` type.

```
enum gxFontNames {
   gxNoFontName,
   gxCopyrightFontName,
   gxFamilyFontName,
   gxStyleFontName,
   gxUniqueFontName,
   gxFullFontName,
   gxVersionFontName,
   gxPostscriptFontName,
   gxTrademarkFontName,
   gxManufacturerFontName
};

typedef long gxFontName;
```

Constant descriptions

`gxNoFontName`	The font name is not specified. You can use this value with such functions as `GXFindFonts` (page 7-33) to indicate that, during a search for a font using other specific criteria, any font name constitutes a match.

`gxCopyrightFontName`
: The manufacturer's copyright.

`gxFamilyFontName`
: The font family name, such as "Geneva".

`gxStyleFontName`
: The font style, such as "Bold".

`gxUniqueFontName`
: The manufacturer's unique name for the font, such as "Apple Computer Geneva Bold 3.0".

`gxFullFontName` The full font name, such as "Geneva Bold".

`gxVersionFontName`
: The manufacturer's version number, such as "3.0". (The name does not need to include the word "version.")

`gxPostscriptFontName`
: The PostScript-legible name of the font, such as "Geneva-Bold".

`gxTrademarkFontName`
: The trademark holder's name.

`gxManufacturerFontName`
: The name of the font's manufacturer.

New font names are registered in *QuickDraw GX Font Feature Registry*, described on page 7-12. Font names are described in "Names" on page 7-6.

Font Features

The `gxFontFeature` type identifies information about a specific font feature. The possible values for a font feature are described in the chapter "Layout Shapes" in this book.

```
typedef long gxFontFeature;
```

The `gxFontFeatureFlag` type defines font feature flags.

```
typedef long gxFontFeatureFlag;
```

Font feature flags provide information about a particular font feature, such as whether the settings of the feature can be combined or are mutually exclusive.

```
#define gxMutuallyExclusiveFeature 0x8000
```

Flag descriptions

`gxMutuallyExclusiveFeature`
: If this flag is set, the settings for this font feature in the font are mutually exclusive. If this flag is not set, the settings can be combined with each other. For example, a lettercase is exclusive, whereas ligatures are nonexclusive.

You can determine the font feature flags of a font feature using the `GXGetFontFeature` function, described on page 7-61 or the `GXFindFontFeature` function, described on page 7-62.

A font feature setting structure contains one setting of a font feature and its associated name ID. A font feature may have one or more settings associated with it.

The `GXGetFontFeature` and `GXFindFontFeature` functions use the `gxFontFeatureSetting` structure.

```
struct gxFontFeatureSetting {
   unsigned short setting;
   unsigned short nameID;
} ;
typedef long gxFontFeatureFlag;
```

Field descriptions

`setting` The font feature setting. The values for this field are described in the chapter “Layout Shapes” in this book.

`nameID` A reference to the font name for this setting in the font.

To find a name for the font feature setting, you can use the `GXGetFontName` function, described on page 7-38.

Font Platforms

A font platform marks the class of encoding table and character code set the font uses. A font may contain multiple encoding tables. Each platform is distinguished by a value from the `gxFontPlatform`s enumeration.

```
enum gxFontPlatforms {
   gxGlyphPlatform = -1,
   gxNoPlatform,
   gxUnicodePlatform,
   gxMacintoshPlatform,
   gxReservedPlatform,
   gxMicrosoftPlatform,
   gxCustomPlatform
} ;

typedef long gxFontPlatform;
```

Constant descriptions

`gxGlyphPlatform` This is a reserved value used by the QuickDraw GX graphics system. A font never uses this setting. The graphics system uses this for identifiers that store glyph codes rather than character codes.

`gxNoPlatform` The platform is not specified. You can use this value with such functions as `GXFindFonts`, described on page 7-33, to indicate that, during a search for a font using other specific criteria, any type of platform constitutes a match.

`gxUnicodePlatform`
: The platform uses the Unicode character code specifications. For more information about the Unicode encodings, see *The Unicode Standard: Worldwide Character Encoding*, volumes 1 and 2, available from Addison-Wesley.

`gxMacintoshPlatform`
: The platform uses one of the Macintosh character code sets.

`gxReservedPlatform`
: The platform is reserved for future use.

`gxMicrosoftPlatform`
: This platform uses one of the Microsoft character code sets.

`gxCustomPlatform`
: This is a nonstandard platform, specific to the font.

QuickDraw GX Macintosh Scripts

Script values, together with platform values and optionally language values, identify the character encoding for a font. Each platform may define a set of script codes. For more information about scripts and how they work, see “Encodings” on page 7-7.

The Macintosh platform defines a number of script codes for scripts used around the world. The `gxMacintoshScripts` enumeration defines values for these script codes used with the `gxMacintoshScript` type.

```
enum gxMacintoshScripts {
   gxNoScript,
   gxRomanScript,
   gxJapaneseScript,
   gxTraditionalChineseScript,
   gxChineseScript = gxTraditionalChineseScript,
   gxKoreanScript,
   gxArabicScript,
   gxHebrewScript,
   gxGreekScript,
   gxCyrillicScript,
   gxRussianScript = gxCyrillicScript,
   gxRSymbolScript,
   gxDevanagariScript,
   gxGurmukhiScript,
   gxGujaratiScript,
   gxOriyaScript,
   gxBengaliScript,
   gxTamilScript,
   gxTeluguScript,
   gxKannadaScript,
   gxMalayalamScript,
   gxSinhaleseScript,
```

```
   gxBurmeseScript,
   gxKhmerScript,
   gxThaiScript,
   gxLaotianScript,
   gxGeorgianScript,
   gxArmenianScript,
   gxSimpleChineseScript,
   gxTibetanScript,
   gxMongolianScript,
   gxGeezScript,
   gxEthiopicScript = gxGeezScript,
   gxAmharicScript = gxGeezScript,
   gxSlavicScript,
   gxEastEuropeanRomanScript = gxSlavicScript,
   gxVietnameseScript,
   gxExtendedArabicScript,
   gxSindhiScript = gxExtendedArabicScript,
   gxUninterpretedScript
} ;

typedef long gxFontScript;
```

The `gxNoScript` value indicates that no particular script is specified. You can use this value with such functions as `GXFindFonts`, described on page 7-33, to indicate that, during a search for a font using other specific criteria, any type of script constitutes a match. All other values in the `gxMacintoshScripts` enumeration refer to the names of script systems from around the world.

The `gxFontScript` type identifies the script for a name or encoding. Each platform has a corresponding enumeration of legal scripts.

The `gxCustomScripts` enumeration defines values for script codes used with the `gxCustomPlatform` type. It is not related to other scripts.

```
enum gxCustomScripts {
   gxCustom8bitScript =1
   gxCustom816bitScript,
   gxCustom16bitScript
};
typedef long gxFontScript;
```

The `gxMicrosoftScripts` enumeration defines values for script codes used with the `gxMicrosoftPlatform` type.

```
enum gxMicrosoftScripts {
   gxMicrosoftSymbolScript =1,
   gxMicrosoftStandardScript
};
```

Languages

The `gxFontLanguage` type names the language of a particular font name. It is used primarily for names but also appears as the identifying block for an encoding.

The `gxMacintoshLanguages` enumeration lists the language values that can be used with the `gxFontLanguage` type.

```
enum gxMacintoshLanguages {
   gxNoLanguage,
   gxEnglishLanguage,
   gxFrenchLanguage,
   gxGermanLanguage,
   gxItalianLanguage,
   gxDutchLanguage,
   gxSwedishLanguage,
   gxSpanishLanguage,
   gxDanishLanguage,
   gxPortugueseLanguage,
   gxNorwegianLanguage,
   gxHebrewLanguage,
   gxJapaneseLanguage,
   gxArabicLanguage,
   gxFinnishLanguage,
   gxGreekLanguage,
   gxIcelandicLanguage,
   gxMalteseLanguage,
   gxTurkishLanguage,
   gxCroatianLanguage,
   gxTradChineseLanguage,
   gxUrduLanguage,
   gxHindiLanguage,
   gxThaiLanguage,
   gxKoreanLanguage,
   gxLithuanianLanguage,
   gxPolishLanguage,
   gxHungarianLanguage,
   gxEstonianLanguage,
   gxLettishLanguage,
   gxLatvianLanguage = gxLettishLanguage,
   gxSaamiskLanguage,
   gxLappishLanguage = gxSaamiskLanguage,
   gxFaeroeseLanguage,
   gxFarsiLanguage,
   gxPersianLanguage = gxFarsiLanguage,
```

```
    gxRussianLanguage,
    gxSimpChineseLanguage,
    gxFlemishLanguage,
    gxIrishLanguage,
    gxAlbanianLanguage,
    gxRomanianLanguage,
    gxCzechLanguage,
    gxSlovakLanguage,
    gxSlovenianLanguage,
    gxYiddishLanguage,
    gxSerbianLanguage,
    gxMacedonianLanguage,
    gxBulgarianLanguage,
    gxUkrainianLanguage,
    gxByelorussianLanguage,
    gxUzbekLanguage,
    gxKazakhLanguage,
    gxAzerbaijaniLanguage,
    gxAzerbaijanArLanguage,
    gxArmenianLanguage,
    gxGeorgianLanguage,
    gxMoldavianLanguage,
    gxKirghizLanguage,
    gxTajikiLanguage,
    gxTurkmenLanguage,
    gxMongolianLanguage,
    gxMongolianCyrLanguage,
    gxPashtoLanguage,
    gxKurdishLanguage,
    gxKashmiriLanguage,
    gxSindhiLanguage,
    gxTibetanLanguage,
    gxNepaliLanguage,
    gxSanskritLanguage,
    gxMarathiLanguage,
    gxBengaliLanguage,
    gxAssameseLanguage,
    gxGujaratiLanguage,
    gxPunjabiLanguage,
    gxOriyaLanguage,
    gxMalayalamLanguage,
    gxKannadaLanguage,
    gxTamilLanguage,
```

```
    gxTeluguLanguage,
    gxSinhaleseLanguage,
    gxBurmeseLanguage,
    gxKhmerLanguage,
    gxLaoLanguage,
    gxVietnameseLanguage,
    gxIndonesianLanguage,
    gxTagalogLanguage,
    gxMalayRomanLanguage,
    gxMalayArabicLanguage,
    gxAmharicLanguage,
    gxTigrinyaLanguage,
    gxGallaLanguage,
    gxOromoLanguage = gxGallaLanguage,
    gxSomaliLanguage,
    gxSwahiliLanguage,
    gxRuandaLanguage,
    gxRundiLanguage,
    gxChewaLanguage,
    gxMalagasyLanguage,
    gxEsperantoLanguage,
    gxWelshLanguage = 129,
    gxBasqueLanguage,
    gxCatalanLanguage,
    gxLatinLanguage,
    gxQuechuaLanguage,
    gxGuaraniLanguage,
    gxAymaraLanguage,
    gxTatarLanguage,
    gxUighurLanguage,
    gxDzongkhaLanguage,
    gxJavaneseRomLanguage,
    gxSundaneseRomLanguage
} ;
typedef long gxFontLanguage;
```

The `gxNoLanguage` value indicates that no particular language is specified. You can use this value with such functions as `GXFindFonts`, described on page 7-33, to indicate that, during a search for a font using other specific criteria, any type of language constitutes a match.

All other values in the `gxMacintoshLanguages` enumeration refer to the English names of languages from around the world.

Advanced Constants and Data Types

This section describes constants and data types that you can use for advanced operations on QuickDraw GX font objects.

- The `gxFontStorageTag` enumeration lists the ways you can store a QuickDraw GX font.
- The `gxFontStorageReference` type references the font from its particular method of storage.
- The `gxFontFormatTag` identifier specifies the particular format, and therefore the particular font scaler, of the QuickDraw GX font.
- The `gxFontTableTag` type references the names of font tables.
- The `gxFontAttribute` enumeration specifies whether a font is a system font or an application-specific font and whether it can be edited.

Font Storage Tags

Fonts can be stored as resources, handles, or files. You can determine how a font is stored by checking its associated storage type, which is of the `gxFontStorageTag` type.

```
#define gxResourceFontStorage 0x72737263  /* 'rsrc' */
#define gxHandleFontStorage   0x686e646c  /* 'hndl' */
#define gxFileFontStorage     0x62617373  /* 'bass' */
#define gxNfntFontStorage     0x6e666e74  /* 'nfnt' */

typedef long gxFontStorageTag;
```

Constant descriptions

`gxResourceFontStorage`
: The font is stored in an `'sfnt'` resource. The resource does not need to be loaded.

`gxHandleFontStorage`
: The font is stored in a nonpurgeable handle.

`gxFileFontStorage`
: The font is stored in an open file.

`gxNfntFontStorage`
: The font is stored in a `'FOND'` resource that references only `'FONT'` and `'NFNT'` resources.

The `gxFontStorageReference` type contains the reference to the resource, handle, or file where the resource is stored. If the font is stored in a resource, the reference is a handle to the resource; if the font is stored in a handle, the reference is the handle itself; and if the font is stored in a file, the reference is the file reference number. If the font is stored in an `'NFNT'` resource, the reference is `styleBits` * 65536 + `FONDResourceID`.

```
typedef void *gxFontStorageReference;
```

Font Table Tags

A font table tag is a 4-byte code that describes the type of table. For example, the table tag `'kern'` means the table is a kerning table. For portability, when your application calls routines such as `gxFindFontTable`, you should specify tags in the hexadecimal notation `0x6B65726E`, because of byte-ordering concerns.

```
typedef long gxFontTableTag;
```

Font Attributes

Each font has a set of **font attributes,** which are a group of flags that modify the behavior or identity of the font. These flags are defined in the `gxFontAttributes` enumeration.

```
enum gxFontAttributes {
   gxSystemFontAttribute = 0x0001
   gxReadOnlyFontAttribute = 0x0002
};

typedef long gxFontAttribute;
```

Constant descriptions

`gxSystemFontAttribute`
: The font object was not created by the application but was created by QuickDraw GX.

`gxReadOnlyFontAttribute`
: The font object cannot be passed as a parameter to any of the font-editing functions, for example, `GXDeleteFontName`.

Fonts created by calling `GXNewFont` are marked as nonsystem fonts. Fonts stored in `'NFNT'` resources are always marked read-only and cannot be edited in QuickDraw GX.

To determine the font attributes of a specified font, you can use the `GXGetFont` (page 7-67) or `GXFindFont` (page 7-67) function.

Basic Font Functions

This section describes the functions that access, retrieve, change, or delete information about the fonts in the system. With these functions, you can

- get a complete list of available fonts, or any subset of the list, such as a list only of font families
- determine what the default font for your application is and change it
- retrieve, add, or delete font names
- retrieve information about font encoding, features, and variations, without editing the font
- get and alter information about font descriptors and instances

These functions describe ways of altering the tables of the font. The advanced functions are useful mainly for font-related applications, such as font editors. In most cases, general-purpose applications, such as word processors, do not need the advanced functions.

Note
QuickDraw GX functions that begin with *GXGet* and *GXFind* are similar in that they retrieve roughly the same information. However, the *GXGet* functions find the information by its index in the font's list of that type of information and return a tag or a font name, whereas the *GXFind* functions use a tag to find the information and return its index. u

Getting the List of Available Fonts

You can get the list of available fonts, or specified subsets of fonts or font families, using `GXFindFonts`.

GXFindFonts

You can use the `GXFindFonts` function to get the list of available fonts or a list of fonts that match a particular set of criteria.

```
long GXFindFonts(gxFont family, gxFontName meaning,
                 gxFontPlatform platform, gxFontScript script,
                 gxFontLanguage language, long nameLength,
                 const unsigned char name[], long index,
                 long count, gxFont fonts[]);
```

`family` A reference to the font whose font family determines the search range `GXFindFonts` uses. If this value is `nil`, then `GXFindFonts` applies the search criteria to all fonts, without regard to their families. If it is not `nil`, the search is limited to members of this family.

`meaning` The kind of font name you want `GXFindFonts` to search for.

`platform` The platform of the font. The `platform`, `script`, and `language` parameters identify the character set and language of the name.

`script` The script of the font.

`language` The language of the font.

`nameLength`
A byte count for the data stored in the `name` parameter.

`name` The actual name of the font you are searching for. If you are searching for a particular font, you can include its name, such as "Geneva," here.

`index` The number of the matching font to begin counting. A value of 1 means `GXFindFonts` should start with the first matching font it finds.

`count`	The maximum number of matches you want `GXFindFonts` to return. The function returns the actual number of matching fonts, which may be less than or equal to the value of `count`. If you want all possible matches, use the value `gxSelectToEnd`.
`fonts`	A pointer to a buffer your application provides, into which `GXFindFonts` copies the font references that match the given criteria. The function returns, in the `count` parameter, the number of fonts that matched. If the value of `fonts` is `nil`, `GXFindFonts` returns nothing in this parameter.
function result	The number of fonts that match the search criteria.

DESCRIPTION

The `GXFindFonts` function takes search criteria (specified by the `family`, `meaning`, `name`, `platform`, `script`, `language`, and `index` parameters) and returns both the number of fonts that match those criteria and, if there are any matching fonts and the value of `fonts` is not `nil`, the font references of the fonts that match.

The values of the `nameLength` and `name` parameters specify the actual data for the font being searched for. Note that `nameLength` contains a byte count, which may not equal the number of characters in the name.

ERRORS, WARNINGS, AND NOTICES

Errors

`count_is_less_than_one`	(debugging version)
`index_is_less_than_one`	(debugging version)
`inconsistent_parameters`	(debugging version)
`parameter_out_of_range`	(debugging version)
`out_of_memory`	
`internal_font_error`	
`illegal_font_parameter`	

SEE ALSO

Various ways of using the `GXFindFonts` function are described in “Getting Information About Available Fonts” on page 7-15.

Font names are discussed in “Names” on page 7-6.

Platforms, scripts, languages, and encoding tables are discussed in “Encodings” on page 7-7.

Counting Glyphs in a Font

If your application needs to know how many glyphs are in a font (for example, to make waterfalls), it can count them with the `GXCountFontGlyphs` call.

GXCountFontGlyphs

You can use the `GXCountFontGlyphs` function to count the number of glyphs in a font.

```
long GXCountFontGlyphs (gxFont fontID);
```

`fontID` A reference to the font whose glyphs you want to count.

function result The number of glyphs in the font.

DESCRIPTION

The `GXCountFontGlyphs` function retrieves the number of glyphs present in the font referenced in the `fontID` parameter. Valid glyph codes for the font range from 0 to this function's result minus 1. Glyph codes can be set directly into a typographic shape and then drawn.

ERRORS, WARNING, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

Getting and Setting the Default Font

When you first create a typographic shape, its style is initialized to have a font reference of `nil`. When QuickDraw GX is passed `nil` for the font parameter, it substitutes the default font for that parameter. To manipulate the default font, you can use `GXGetDefaultFont` and `GXSetDefaultFont`. QuickDraw GX initializes this to Helvetica, if it is available.

GXGetDefaultFont

You can use the `GXGetDefaultFont` function to retrieve the current default font for your application.

```
gxFont GXGetDefaultFont(void);
```

function result A reference to the current application's default font.

DESCRIPTION

The `GXGetDefaultFont` function returns a reference to the application's default font. QuickDraw GX initializes it to a font, so the application does not need to set it.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
```

SEE ALSO

You can change the default font with `GXSetDefaultFont`, described next.

GXSetDefaultFont

You can use the `GXSetDefaultFont` function to set the default font for your application.

```
gxFont GXSetDefaultFont(gxFont fontID);
```

`fontID` A reference to the font you want as the default font of your application.

function result A reference to the previous default font.

DESCRIPTION

The `GXSetDefaultFont` function changes your application's default font to the font specified by `fontID`. Because there is no system-wide default font, `GXSetDefaultFont` affects only the default font for the current application.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

SEE ALSO

To determine the current default font for your application, use `GXGetDefaultFont` (page 7-35).

Manipulating Font Names

QuickDraw GX allows you to work with font names in several ways. You can

- count the number of font names in a particular font (`GXCountFontNames`)
- specify a font name by index (`GXGetFontName`)
- specify a font name by type, platform, script, and language (`GXFindFontName`)
- find an available ID for a new font name (`GXNewFontID`)
- edit an existing font name, or create a new one in the font (`GXSetFontName`)
- delete a font name permanently from a font (`GXDeleteFontName`)

GXCountFontNames

You can use the `GXCountFontNames` function to determine the number of font names in a font.

```
long GXCountFontNames(gxFont fontID);
```

`fontID` A reference to the font whose font names you want to count.

function result The total number of entries in the names property of the font object. These entries include strings such as the names of features, which just identify specific aspects of the font.

DESCRIPTION

The `GXCountFontNames` function returns the total number of names in the font referenced by `fontID`. This total includes each occurrence of each name, including repetitions of the same name in different platforms, languages, or scripts, and other strings such as the names of features. You can use this number to iterate through the names in a font using the `GXGetFontName` function (described next).

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

SEE ALSO

Font names are discussed in “Names” on page 7-6.

GXGetFontName

You can use the GXGetFontName function to find a font name by its index.

```
long GXGetFontName(gxFont fontID, long index, gxFontName *name,
                   gxFontPlatform *platform, gxFontScript *script,
                   gxFontLanguage *language, unsigned char text[]);
```

fontID	A reference to the font from which you want to extract font names.
index	A value between 1 and the number of font names. (The number of font names is returned by the GXCountFontNames function.)
name	A pointer to the type of font name.
platform	A pointer to the platform of the font name.
script	A pointer to the script of the font name.
language	A pointer to the language of the font name.
text	On return, the text of the font name. If text is set to nil, the function ignores it.
function result	The byte length of the font name found. If no font name is found, the function returns 0.

DESCRIPTION

The GXGetFontName function takes a font reference and an index in the list of font names in the font and returns the font name and information about the name—its type, platform, script, and language—if those parameters are not nil.

If the function returns 0, GXGetFontName did not find a font name or modify any of the parameters.

Your application is responsible for allocating the memory for the text parameter.

ERRORS, WARNINGS, AND NOTICES

Errors

index_out_of_range	(debugging version)
out_of_memory	
internal_font_error	
illegal_font_parameter	

SEE ALSO

The GXCountFontNames function is described on page 7-37.

To search for a font name by its type, platform, script, or language, use the GXFindFontName function (described on page 7-39).

Font names are discussed in “Names” on page 7-6.

Platforms, scripts, encoding tables, and languages are discussed in “Encodings” on page 7-7.

GXFindFontName

You can use the `GXFindFontName` function to find a font name by its type, platform, script, and language.

```
long GXFindFontName(gxFont fontID, gxFontName name,
                    gxFontPlatform platform, gxFontScript script,
                    gxFontLanguage language, unsigned char text[],
                    long *index);
```

`fontID`	A reference to the font whose font names you want to search.
`name`	The type of font name you are searching for.
`platform`	The platform of the font name you are searching for.
`script`	The script of the font name you are searching for.
`language`	The language of the font name you are searching for.
`text`	On return, the text of the font name. If you pass `nil` for this parameter, the function ignores it.
`index`	On return, a pointer to the index of the font name in the font’s list of font names. If you pass `nil` for this parameter, the function ignores it.

function result The byte length of the font name. If no font name is found, the function returns 0.

DESCRIPTION

The `GXFindFontName` function searches the specified font for a font name that matches the values of the `name`, `platform`, `script`, and `language` parameters. If it finds a font name, `GXFindFontName` fills out the `text` and `index` parameters with the font name and its index in the font’s list of font names, if those parameters are not set to `nil`.

Your application is responsible for allocating the storage for the `text` parameter.

ERRORS, WARNINGS, AND NOTICES

Errors

`inconsistent_parameters` (debugging version)
`out_of_memory`
`internal_font_error`
`illegal_font_parameter`

Warnings

`index_out_of_range`

SEE ALSO

To search for a font name by its index, use the `GXGetFontName` function, described on page 7-38.

Font names are discussed in “Names” on page 7-6.

Platforms, scripts, languages, and encoding tables are discussed in “Encodings” on page 7-7.

GXNewFontNameID

You can use the `GXNewFontNameID` function to get a font name ID that is currently unused in a specific font. However, calling this function does not actually change the font.

```
gxFontName GXNewFontNameID(gxFont fontID);
```

`fontID` A reference to the specific font for which you want to find a font name ID.

function result A font name ID that is currently available.

DESCRIPTION

The `GXNewFontNameID` function finds an available font name ID for the font you specify. The function doesn’t reserve the font name ID for you; if you call the function twice in succession, it may return the same font name ID both times. You can use this font name ID with the `GXSetFontName` function (described next) to add a new font name that does not have a conflicting font name ID.

One use for this function is adding a new font instance, which requires specifying a variation coordinate and a name ID.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

GXSetFontName

You can use the GXSetFontName function to add a new font name to or change an existing font name in a font.

```
long GXSetFontName(gxFont fontID, gxFontName name,
                   gxFontPlatform platform, gxFontScript script,
                   gxFontLanguage language, long nameLength,
                   const unsigned char text[]);
```

fontID	A reference to the font for which you want to add or substitute a font name.
name	The type of font name.
platform	The platform of the font name. The platform, script, and language parameters identify the character set and language of the name.You must include values for these parameters, because a font can store multiple versions of the same name, each one in a different platform, script, and language.
script	The script of the font name.
language	The language of the font name.
nameLength	The byte count of the text in the name parameter, which may be different from the character count, depending on the type of text.
text	A pointer to the name you want to add to the font or substitute for another font name. The text parameter can point to a list of 1-byte character codes or 2-byte character codes for Kanji or Unicode.

function result The index of the name added or changed in the font's list of font names.

DESCRIPTION

The GXSetFontName function adds a new font name or replaces an existing font name with the text in the name parameter. The function replaces the font name only if name, platform, script, and language match an existing name in the font; otherwise, GXSetFontName adds a new entry to the font.

If you add font names to a font, QuickDraw GX enlarges the font data accordingly.

IMPORTANT

The GXSetFontName function permanently changes the data in the font. In general, you should only call this function for fonts created by your application using GXNewFont and not for a system font. s

ERRORS, WARNINGS, AND NOTICES

Errors

`parameter_out_of_range`	(debugging version)
`inconsistent_parameters`	(debugging version
`out_of_memory`	
`internal_font_error`	
`illegal_font_parameter`	
`font_cannot_be_changed`	

SEE ALSO

The `GXNewFont` function is described on page 7-64.

To retrieve a font name by its index, use the `GXGetFontName` function, described on page 7-38. To retrieve a font name by its platform, script, or language, use the `GXFindFontName` function, described on page 7-39.

Font names are discussed in "Names" on page 7-6.

Platforms, scripts, languages, and encoding tables are discussed in "Encodings" on page 7-7.

GXDeleteFontName

You can use the `GXDeleteFontName` function to delete a font name from a font.

```
long GXDeleteFontName(gxFont fontID, long index,
                      gxFontName name, gxFontPlatform platform,
                      gxFontScript script,
                      gxFontLanguage language);
```

`fontID`	A reference to the font from which you want to delete a font name.
`index`	The index of the font name in the font's list of font names. If this value is greater than 0, it represents the indexed location of the font name in the font; if this value is 0, `GXDeleteFontName` identifies the font name by the values of the `meaning`, `platform`, `script`, and `language` parameters.
`name`	The type of font name you want to delete.
`platform`	The platform of the font name. The `platform`, `script`, and `language` parameters uniquely identify the character set and language of the font name.You must include values for these parameters, because a font can store multiple versions of the same name, each one in a different platform, script, and language.
`script`	The script of the font name.
`language`	The language of the font name.
function result	The index of the name deleted. If the function does not delete a name, it returns 0.

DESCRIPTION

The `GXDeleteFontName` function permanently deletes a font name from the font referenced by the `fontID` parameter. You can identify the font name you want deleted either by its index, or if `index` is equal to 0, by its `meaning`, `platform`, `script`, and `language` parameters.

If you delete font names from a font, QuickDraw GX decreases the font data accordingly.

IMPORTANT

The `GXDeleteFontName` function permanently changes the data in the font. In general, you should only call this function for fonts created by your application using `GXNewFont` and never for a system font. s

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`internal_font_error`
`illegal_font_parameter`
`font_cannot_be_changed`
`inconsistent_parameters` (debugging version)

Warnings

`font_table_not_found`

SEE ALSO

The `GXNewFont` function is described on page 7-64.

Font names are discussed in “Names” on page 7-6.

Platforms, scripts, languages, and encoding tables are discussed in “Encodings” on page 7-7.

Manipulating Font Encodings

QuickDraw GX allows you to work with font encodings in several ways. You can

- count the number of encoding tables in a font (`GXCountFontEncodings`)
- specify by index the platform, script, and language values for an encoding table (`GXGetFontEncoding`)
- obtain the index for an encoding table that you specify by its platform, script, and language values (`GXFindFontEncoding`)
- obtain the glyph codes of a specific platform encoding that correspond to a string of character codes (`GXApplyFontEncoding`)

GXCountFontEncodings

You can use the `GXCountFontEncodings` function to retrieve the number of encoding tables in a font.

```
long GXCountFontEncodings(gxFont fontID);
```

`fontID` A reference to the font whose encoding tables you want to count.

function result The number of supported encoding tables in the font.

DESCRIPTION

The `GXCountFontEncodings` function returns the number of supported encoding tables in the font.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

GXGetFontEncoding

You can use the `GXGetFontEncoding` function to identify an encoding table in a font by its index.

```
gxFontPlatform GXGetFontEncoding(gxFont fontID, long index,
                                 gxFontScript *script,
                                 gxFontLanguage* language);
```

`fontID` A reference to the font you want to search for a specific encoding table.

`index` A value between 1 and the number of supported encoding tables. (The number of encoding tables is returned by `CountFontEncodings`.)

`script` On return, the script of the specified encoding table.

`language` On return, the language of the specified encoding table.

function result The table's platform value.

DESCRIPTION

The `GXGetFontEncoding` function identifies an encoding table in a font by its index. If the `script` parameter is not set to `nil`, `GXGetFontEncoding` returns the script value for that encoding table. If the `language` parameter is not set to `nil`, `GXGetFontEncoding` returns the language value for that encoding table.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

Warnings

```
index_out_of_range
```

SEE ALSO

The `GXCountFontEncodings` function is described on page 7-44.

To search for an encoding table specified by platform, script, and language, use the `GXFindFontEncoding` function, described next.

Platforms, scripts and encoding tables are discussed in "Encodings" on page 7-7.

GXFindFontEncoding

You can use the `GXFindFontEncoding` function to find the index of an encoding table specified by platform, script, and language values.

```
long GXFindFontEncoding(gxFont fontID, gxFontPlatform platform,
                        gxFontScript script,
                        gxFontLanguage language);
```

`fontID`	A reference to the font you want to search for a specific encoding table.
`platform`	The platform of the encoding table.
`script`	The script of the encoding table.
`language`	The language of the encoding table.

function result The table's index in the font's list of encoding tables. If the function does not find the specified encoding table, it returns 0.

DESCRIPTION

The `GXFindFontEncoding` function searches the specified font for an encoding table that supports the specified platform, script, and language. If you specify a language, you must also specify a script, although you can specify a script without a language.

If you specify a language without naming a script or specify a script without naming a platform, `GXFindFontEncoding` returns the error `inconsistent_parameters`.

ERRORS, WARNINGS, AND NOTICES

Errors

`inconsistent_parameters` (debugging version)
`out_of_memory`
`internal_font_error`
`illegal_font_parameter`

Warnings

`font_language_not_found`

SEE ALSO

To search for an encoding table specified by index, use the `GXGetFontEncoding` function, described on page 7-44.

Platforms are discussed in "Encodings" on page 7-7. The `gxFontPlatform` data type is discussed in "Font Platforms" on page 7-25.

Scripts and encoding tables are discussed in "Encodings" on page 7-7. The `gxFontScript` data type is discussed in "QuickDraw GX Macintosh Scripts" on page 7-26.

GXApplyFontEncoding

You can use the `GXApplyFontEncoding` function to translate a list of character codes to the glyph indices that correspond to a particular encoding table.

```
long GXApplyFontEncoding(gxFont fontID, long index, long* length,
                         const unsigned char text[], long count,
                         unsigned short glyphs[], char was16Bit[]);
```

`fontID` A reference to the font containing the encoding table you want to search.

`index` The index of the encoding table to be used.

`length` On entry, the maximum number of bytes of text you want the function to process. On return, this parameter specifies the number of bytes in the `text` parameter actually processed. This is useful for scripts such as Kanji, which contain mixed 8-bit and 16-bit text. If you pass `nil` for this parameter, the function ignores it.

`text`	An array of character codes that `GXApplyFontEncoding` should translate into glyph indices from the specified encoding table.
`count`	The maximum number of glyphs in the `glyphs` parameter.
`glyphs`	On return, an array of the resulting glyph codes. Your application must maintain a buffer to hold this array.
`was16bit`	An array of Boolean values, one for each glyph. Each value indicates whether the glyph is a translation of an 8-bit or 16-bit character code. The value `true` means the corresponding character code is 16-bit. If this parameter is `nil`, the function ignores it.

function result The number of glyphs processed by the function.

DESCRIPTION

The `GXApplyFontEncoding` function takes a font reference and an array of character codes from the `text` parameter and puts the array of corresponding glyph codes from the specified encoding table in the `glyphs` parameter. Typographic shapes perform this function automatically; thus, your application may never need to call this function directly.

The `GXApplyFontEncoding` function returns glyph codes that are zero-based. Glyph code 0 is the missing glyph.

ERRORS, WARNINGS, AND NOTICES

Errors

```
unknown_font_table_format
out_of_memory
internal_font_error
illegal_font_parameter
length_is_less_than_zero                    (debugging version)
```

Warnings

```
index_out_of_range
```

SEE ALSO

To find an encoding table by its index, use the `GXGetFontEncoding` function, described on page 7-44.

To find an encoding table by its platform, script, and language values, use the `GXFindFontEncoding` function, described on page 7-45.

Manipulating Font Descriptors

QuickDraw GX allows you to work with font descriptors in several ways. You can

- count the number of font descriptors in a particular font (`GXCountFontDescriptors`)
- get a font descriptor that you specify by its index (`GXGetFontDescriptor`)
- get a font descriptor that you specify by its tag, such as `'wdth'` (`GXFindFontDescriptor`)
- add new information about a font descriptor, or create a new font descriptor (`GXSetFontDescriptor`)
- delete a font descriptor permanently from a font (`GXDeleteFontDescriptor`)

Font descriptors are described in “Font Descriptors” on page 7-9.

GXCountFontDescriptors

You can use the `GXCountFontDescriptors` function to get the number of descriptors in a font.

```
long GXCountFontDescriptors(gxFont fontID);
```

`fontID` A reference to the font whose descriptors you want to count.

function result The number of descriptors in the font.

DESCRIPTION

The `GXCountFontDescriptors` function returns the number of different descriptors available in the font. Each descriptor consists of a descriptor tag and a value.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

GXGetFontDescriptor

You can use the `GXGetFontDescriptor` function to return the font descriptor you specify by index.

```
gxFontDescriptorTag GXGetFontDescriptor(gxFont fontID, long index,
                                        Fixed* descriptorValue);
```

`fontID` A reference to the font from which you want to get the font descriptor information.

`index` A value between 1 and the number of descriptors in a font. (The number of descriptors is returned by `GXCountFontDescriptors`.)

`descriptorValue`
On return, the value assigned to this descriptor.

function result The descriptor tag of the specified descriptor.

DESCRIPTION

The `GXGetFontDescriptor` function returns the tag for the font descriptor you specify by index. The function then copies the descriptor's value into the `descriptorValue` parameter, if `descriptorValue` is not set to `nil`.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

Warnings

```
index_out_of_range
```

SEE ALSO

The `GXCountFontDescriptors` function is described on page 7-48.

To retrieve a font descriptor specified by descriptor tag, use the `GXFindFontDescriptor` function, described next.

GXFindFontDescriptor

You can use the `GXFindFontDescriptor` function to return the index and value for a font descriptor you specify by tag.

```
long GXFindFontDescriptor(gxFont fontID,
                          gxFontDescriptorTag descriptorTag,
                          Fixed* descriptorValue);
```

`fontID` A reference to the font from which you want to get font descriptor information.

`descriptorTag`
The descriptor tag you are searching for.

`descriptorValue`
On return, the value assigned to this descriptor.

function result The index of the specified descriptor. If the function does not find the descriptor, it returns 0.

DESCRIPTION

The `GXFindFontDescriptor` function returns the index for the font descriptor you specify by tag. If the function returns a positive index, it also returns the descriptor's value in the `descriptorValue` parameter, if `descriptorValue` is not set to `nil`.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

SEE ALSO

You can retrieve a font descriptor specified by index using the `GXGetFontDescriptor` function, described on page 7-49.

GXSetFontDescriptor

You can use the `GXSetFontDescriptor` function to add a new descriptor to or change an existing descriptor in a font.

```
long GXSetFontDescriptor(gxFont fontID, long index,
                         gxFontDescriptorTag descriptorTag,
                         Fixed descriptorValue);
```

`fontID` A reference to the font you want to add a font descriptor to or change a font descriptor in.

`index` The index of the descriptor in the font's list of descriptors, if this value is greater than 0. If it is 0, then `GXSetFontDescriptor` uses the `descriptorTag` parameter.

`descriptorTag`
The tag of the descriptor you want to add or change. If the value of `index` is greater than 0, this parameter should be set to 0.

`descriptorValue`
The value you want assigned to this descriptor.

function result The index of the descriptor that was changed or added.

DESCRIPTION

The `GXSetFontDescriptor` function changes the value of the descriptor in the font you specify by tag. If no matching descriptor is found, `GXSetFontDescriptor` adds a new descriptor to the font with the given descriptor tag and value.

If you add font descriptors to a font, QuickDraw GX is responsible for enlarging the font data accordingly.

IMPORTANT

The `GXSetFontDescriptor` function permanently changes the data in the font. In general, you should only call this function for fonts created by your application using `GXNewFont` and never for a system font. s

ERRORS, WARNINGS, AND NOTICES

Errors

```
illegal_font_parameter
font_cannot_be_changed
inconsistent_parameters                    (debugging version)
out_of_memory
internal_font_error
```

Warnings

```
index_out_of_range
```

GXDeleteFontDescriptor

You can use the `GXDeleteFontDescriptor` function to delete a font descriptor you specify by index or tag.

```
long GXDeleteFontDescriptor(gxFont fontID, long index,
                            gxFontDescriptorTag descriptorTag);
```

`fontID` A reference to the font from which you want to delete a font descriptor.

`index` The index of the descriptor you want to delete. If this value is greater than 0 and the `descriptorTag` parameter is set to 0, the value of `index` is the index of the descriptor in the font's descriptor list. If this parameter is set to 0 and the `descriptorTag` parameter contains a value other than `nil`, `GXDeleteFontDescriptor` uses the value of the `descriptorTag` parameter to search for the descriptor.

`descriptorTag`
The tag you want to delete.

function result The index of the deleted descriptor. If the function does not delete a descriptor, it returns 0.

DESCRIPTION

The `GXDeleteFontDescriptor` function permanently deletes the specified descriptor from the font. You can specify the descriptor either by the value of its index in the font's list of descriptors, or by its tag in the `descriptorTag` parameter, if the `index` parameter is set to 0.

If you delete font descriptors from a font, QuickDraw GX is responsible for decreasing the font data accordingly.

IMPORTANT

The `GXDeleteFontDescriptor` function permanently changes the data in the font. In general, you should only call this function for fonts created by your application using `GXNewFont` and never for a system font. s

ERRORS, WARNINGS, AND NOTICES

Errors

```
illegal_font_parameter
font_cannot_be_changed
inconsistent_parameters                    (debugging version)
out_of_memory
internal_font_error
```

Warnings

```
font_table_not_found
index_out_of_range
```

Manipulating Font Variations

QuickDraw GX allows you to work with font variations in several ways. You can

- count the number of font variation axes in a particular font (`GXCountFontVariations`)
- get a font variation that you specify by its index in the font's list of font variations (`GXGetFontVariation`)
- get a font variation that you specify by its tag (such as `'wdth'`) in the font's list of font variations (`GXFindFontVariation`)

You cannot add variations to or delete variations from a font because the format of a font variation varies from one font format to another.

Font variations are described in "Font Variations" on page 7-10.

GXCountFontVariations

You can use the `GXCountFontVariations` function to get the number of variation axes available for a font.

```
long GXCountFontVariations(gxFont fontID);
```

`fontID` A reference to the font containing the variations.

function result The number of variations in the font.

DESCRIPTION

The `GXCountFontVariations` function returns the number of different variation axes available from the font.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

GXGetFontVariation

You can use the GXGetFontVariation function to get the tag for a specific variation axis in a font you specify by index.

```
gxFontVariationTag GXGetFontVariation(gxFont theFont, long index,
                                      Fixed* minValue,
                                      Fixed* defaultValue,
                                      Fixed* maxValue,
                                      gxFontName* nameID);
```

theFont	A reference to the font containing the font variation you want to find.
index	The index of the variation you want in the list of variations in the font.
minValue	On return, the minimum value for the variation.
defaultValue	On return, the default value for the variation. This value is exactly the same as the font's descriptor value for this tag.
maxValue	On return, the maximum value for the variation.
nameID	On return, the name ID for this variation in the font.

function result The tag for the variation specified.

DESCRIPTION

The GXGetFontVariation function returns the tag for the variation specified by the index parameter. The function also returns the minimum value, default value, maximum value, and name ID of the variation, if those parameters are not set to nil. The name ID identifies the name (for instance, "weight") of the specified variation axis.

You can use the font name ID returned by GXGetFontVariation with the GXFindFontName function (page 7-39) to retrieve the name for a given variation from the font names property of the font object.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

Warnings

```
index_out_of_range
```

SEE ALSO

To search for a specific variation by its tag, use the GXFindFontVariation function, described next.

GXFindFontVariation

You can use the `GXFindFontVariation` function to find a variation in a font you specify by its tag.

```
long GXFindFontVariation(gxFont theFont,
                         gxFontTableTag variationTag,
                         Fixed* minValue, Fixed* defaultValue,
                         Fixed* maxValue, gxFontName* nameID);
```

`theFont` A reference to the font containing the font variation you want to find.

`variationTag` The tag of the variation you are searching for.

`minValue` On return, the minimum value for the variation.

`defaultValue` On return, the default value for the variation.

`maxValue` On return, the maximum value for the variation.

`nameID` On return, the ID of the name in the font for this variation.

function result The index of the variation specified.

DESCRIPTION

The `GXFindFontVariation` function returns the index for the variation specified by `variationTag`. The function also returns the minimum value, default value, maximum value, and name ID of the variation, if those parameters are not set to `nil`.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

SEE ALSO

To retrieve a specific variation by its index, use the `GXGetFontVariation` function (page 7-54).

You can use the font name ID returned by `GXFindFontVariation` with the `GXFindFontName` function (page 7-39) to retrieve the name for a given variation from the names property of the font object.

Manipulating Font Instances

QuickDraw GX allows you to work with font instances in several ways. You can

- count the number of font instances in a particular font (`GXCountFontInstances`)
- get a specific font instance in the font's list of font instances (`GXGetFontInstance`)
- set information about a font instance, or create a new one in the font (`GXSetFontInstance`)
- delete a font instance permanently from a font (`GXDeleteFontInstance`)

Font instances are described in "Font Instances" on page 7-11.

GXCountFontInstances

You can use the `GXCountFontInstances` function to retrieve the number of font instances available in a font.

```
long GXCountFontInstances(gxFont theFont);
```

`theFont` A reference to the font whose font instances you want to count.

function result The number of font instances in the font.

DESCRIPTION

The `GXCountFontInstances` function returns the number of font instances available in the font.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

GXGetFontInstance

You can use the `GXGetFontInstance` function to get the font name for a font instance you specify by index.

```
gxFontName GXGetFontInstance(gxFont theFont, long index,
                             gxFontVariation variation[]);
```

`theFont` A reference to the font whose font instance you want to retrieve.

`index` The index number of the font instance you want.

`variation` On return, an array of the variation data for this instance.

function result The font name for the font instance specified.

DESCRIPTION

The `GXGetFontInstance` function returns the font name for the font instance specified by the value of the `index` parameter. Then `GXGetFontInstance` copies the font variation settings for that instance into the `variation` parameter, if `variation` is not set to `nil`.

Each instance is always identified by a complete set of font variations. A font instance contains a value for each font variation available, even if that value is only the default font variation setting.

Your application must allocate enough memory in the `variation` parameter to store as many font variations as are available; you can determine this number by calling `GXCountFontVariations`. You can pass the `variation` parameter to `GXSetStyleFontVariation` or `GXSetShapeFontVariation` if you want to draw text with this font instance.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

Warnings

```
index_out_of_range
```

SEE ALSO

See the chapter “Typographic Styles” in this book for information on the `GXSetStyleFontVariation` or `GXSetShapeFontVariation` function, if you want to draw text with this font instance.

GXSetFontInstance

You can use the `GXSetFontInstance` function to add a font instance to a font or change an existing one.

```
long GXSetFontInstance(gxFont fontID, long index,
                       gxFontName name,
                       const gxFontVariation variation[]);
```

`fontID` A reference to the font in which to add or change a font instance.

`index`	The index of the instance you want to change. If the value of `index` is 0 and the value of `name` does not match any existing instance, `GXSetFontInstance` creates a new instance. If the value of `index` is greater than 0 or the value of `name` matches an existing instance, `GXSetFontInstance` replaces the instance with the values of the `name` and `variation` parameters.
`name`	The font name of the instance you want to change.
`variation`	An array of the variation data for the instance that you want to add to the font.

function result	The index of the new or changed font instance.

DESCRIPTION

The `GXSetFontInstance` function adds a new font instance to a font or replaces the name and data for an existing instance.

The `GXSetFontInstance` function does not create the actual name for the instance; it only stores the ID of the font name for that instance. You must also call the `GXSetFontName` function if you are either changing the name of an existing instance or adding a new instance.

Each instance must have a complete set of font variations. A font instance contains a value for each font variation available, even if that value is only the default font variation setting.

IMPORTANT

The `GXSetFontInstance` function permanently changes the data in the font. In general, you should only call this function on fonts created by your application using `GXNewFont`. s

ERRORS, WARNINGS, AND NOTICES

Errors

`illegal_font_parameter`	
`font_cannot_be_changed`	
`parameter_out_of_range`	(debugging version)
`out_of_memory`	
`internal_font_error`	
`inconsistent_parameters`	(debugging version)

Warnings

`font_table_not_found`

SEE ALSO

If you call `GXSetFontInstance` to create a font instance, you should then call `GXSetFontName` (page 7-41) to create the name for the instance. To get an unused name ID for a new instance, call `GXNewFontNameID` page 7-40.

GXDeleteFontInstance

You can use the GXDeleteFontInstance function to delete a font instance from a font.

```
long GXDeleteFontInstance(gxFont fontID, long index,
                          gxFontName nameID);
```

fontID	A reference to the font from which you want to delete a font instance.
index	The index of the font instance you want to delete. If this value is greater than 0, it specifies the instance to be deleted. If it is 0, then GXDeleteFontInstance deletes the instance that matches the value of nameID.
nameID	The font name of the font instance you want to delete.
function result	The index of the deleted instance. If the function does not delete a font instance, it returns 0.

DESCRIPTION

The GXDeleteFontInstance function permanently removes the specified font instance from the font and changes the index values for the following font instances.

IMPORTANT

The GXDeleteFontInstance function permanently changes the data in the font. In general, you should only call this function for fonts created by your application using GXNewFont. s

The GXDeleteFontInstance function does not delete the name of the instance. If you call GXDeleteFontInstance, you should then call GXDeleteFontName to delete the actual name.

ERRORS, WARNINGS, AND NOTICES

Errors

```
illegal_font_parameter
font_cannot_be_changed
inconsistent_parameters                (debugging version)
parameter_out_of_range                 (debugging version)
out_of_memory
internal_font_error
```

Warnings

```
font_table_not_found
```

Manipulating Font Features

QuickDraw GX allows you to work with font features in several ways. You can

- determine the number of font features available in a font by (`GXCountFontFeatures`)
- get the name of a specific font feature by specifying its index in the font's list of features (`GXGetFontFeature`)
- get a specific font feature from a font by specifying its feature type (`GXFindFontFeature`)

You cannot add font features to or delete font features from a font.

Font features are described in the chapter "Layout Styles" in this book.

GXCountFontFeatures

You can use the `GXCountFontFeatures` function to determine the number of font features available in a font.

```
long GXCountFontFeatures(gxFont fontID);
```

`fontID` A reference to the font whose features you want to count.

function result The number of font features types available in the font.

DESCRIPTION

You can use the `GXCountFontFeatures` function to determine the number of font features available in a font.

Note
These are feature types and not feature type / feature selector pairs. Thus, a font supporting only common and rare ligatures would return 1 (for ligature type) and not 2 for the selectors. u

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

SEE ALSO

To iterate through all of the font features in a font, use `GXGetFontFeature`, described next.

To search for a specific font feature, use the `GXFindFontFeature` function, described on page 7-62.

GXGetFontFeature

You can use the `GXGetFontFeature` function to get the name of a specific font feature by specifying its index in the font's list of features.

```
gxFontName GXGetFontFeature(gxFont fontID, long index,
                            gxFontFeatureFlag* flags,
                            long* settingCount,
                            gxFontFeatureSetting settings[],
                            gxFontFeature* feature);
```

`fontID` A reference to the font from which you want to retrieve a font feature name.

`index` The index of the font feature you are searching for.

`flags` Flags associated with the font feature specified. This is returned to the caller if not `nil`.

`settingCount` The number of settings for this font feature. This is returned to the caller if not `nil`.

`settings` The settings of this font feature. This is returned to the caller if not `nil`. Your application is responsible for the storage.

`feature` The font feature. Returned to the caller if not `nil`.

function result The name ID of the font feature.

DESCRIPTION

The `GXGetFontFeature` function takes a font and an index in the font's list of font features and returns the name ID of the feature. You can use this function to build a menu of font features.

If the function result is not `nil`, you can use it to get the text of the feature's name using the `GXFindFontName` function. This function returns the text of the name of the feature in a readable string, possibly in one or several encodings.

The application is responsible for allocating sufficient space for the settings variable, based on the settings count.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

Warnings

```
index_out_of_range
```

SEE ALSO

The `GXFindFontName` function is described on page 7-39.

To search for a specific font feature, use the `GXFindFontFeature` function, described next.

To get the number of font features available in the font, use the `GXCountFontFeatures` function, described on page 7-60.

GXFindFontFeature

You can use the `GXFindFontFeature` function to get a specific font feature from a font by specifying its feature type.

```
gxFontName GXFindFontFeature(gxFont fontID, gxFontFeature feature,
                             gxFontFeatureFlag* flags,
                             long* settingCount,
                             gxFontFeatureSetting settings[],
                             long* index);
```

`fontID` A reference to the font from which you want to retrieve a font feature name.

`feature` The feature type of the desired feature.

`flags` On return, the feature's flags. This is returned to the caller if not `nil`.

`settingCount` The number of settings for the specified feature. This is returned to the caller if not `nil`.

`settings` The settings for the specified feature. This is returned to the caller if not `nil`.

`index` The index of the feature in the font. This is returned to the caller if not `nil`.

function result The name ID of the feature in the font. If the font does not contain the desired feature, the function returns 0.

DESCRIPTION

The `GXFindFontFeature` function takes a font ID and feature type, and returns the name ID of the specified feature in the font.

You can use the `GXFindFontName` function with the function result to get a readable string that is the name of the feature.

The application is responsible for allocating sufficient space for the settings variable, based on the settings count.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

SEE ALSO

The `GXFindFontName` function is described on page 7-39.

To iterate through all of the font features in a font, use the `GXGetFontFeature` function, described on page 7-61.

To get the number of font features available in the font, use the `GXCountFontFeatures` function, described on page 7-60.

Advanced Font Functions

The functions described in this section allow you to alter the basic tables of the font or manipulate the font feature data in the font. These advanced font functions are useful primarily for font-related applications, such as font editors. In most cases, general-purpose applications, such as word processors, do not use these functions.

Adding, Removing, and Flattening Fonts

You can add a new font to the list of available fonts using `GXNewFont` and remove it using `GXDisposeFont`. You can also flatten the shape of a font using `GXFlattenFont`.

GXNewFont

You can use the `GXNewFont` function to create a font object and add it to the list of available fonts.

```
gxFont GXNewFont(gxFontStorageTag storage,
                 gxFontStorageReference reference,
                 gxFontAttribute attributes);
```

`storage` The storage type of the font you are adding. Possible storage type values are listed in "Font Storage Tags" on page 7-31.

`reference` The storage reference to the font data, usually an `'sfnt'` resource.

`attributes` The attribute for the font you are adding.

function result A reference to the font you are adding.

DESCRIPTION

The `GXNewFont` function adds a font to the list of registered fonts. `GXNewFont` does not make a copy of the font's data or create a new resource, file, or handle. The font's data is specified in the `reference` parameter.

You must balance a call to `GXNewFont` with a call to `GXDisposeFont`, described next, when your application no longer needs to use that font.

ERRORS, WARNINGS, AND NOTICES

Errors

`illegal_font_storage_type`	(debugging version)
`illegal_font_storage_reference`	(debugging version)
`illegal_font_attributes`	(debugging version)
`out_of_memory`	
`internal_font_error`	

SEE ALSO

Storage types and storage references are discussed in "How Font Objects Are Stored and Referenced" on page 7-13.

GXDisposeFont

You can use the `GXDisposeFont` function to remove a font from the list of available fonts.

```
void GXDisposeFont(gxFont fontID);
```

`fontID` A reference to the font you want to remove from the list of registered fonts.

DESCRIPTION

The `GXDisposeFont` function removes a font from the list of available fonts. The font must be one you have added to the list.

IMPORTANT

You should *not* call `GXDisposeFont` for a system font. You should call it only for fonts you have created or added by means of `GXNewFont`. s

The `GXDisposeFont` function frees private storage and caches allocated by the system. Your application is responsible for disposing the actual font data.

ERRORS, WARNINGS, AND NOTICES

Errors

`illegal_font_parameter`

GXFlattenFont

You use the `GXFlattenFont` function to flatten all or part of a font object. You flatten a font to be embed it into a spool file.

```
void GXFlattenFont(gxFont source, struct scalerStream* stream,
                   struct gxSpoolBlock* block);
```

`source` A reference to the font you want to flatten.

`stream` The scaler stream you want for the font.

`block` The spool block.

DESCRIPTION

The `GXFlattenFont` function takes a font, a `gxScalerStream` structure, and spool block and flattens the font so that you can include the flattened font with a flattened shape. The `scalerStream` structure is used only by font-scaling programs and is not described here.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`	
`internal_font_error`	
`illegal_font_parameter`	
`unflattening_interrupted_by_client`	
`null_font_scaler_context`	
`null_font_scaler_input`	
`invalid_font_scaler_context`	
`invalid_font_scaler_input`	
`invalid_font_scaler_font_data`	
`font_scaler_newblock_failed`	
`font_scaler_getfonttable_failed`	
`font_scaler_bitmap_allocation_failed`	
`font_scaler_outline_allocation_failed`	
`required_font_scaler_table_missing`	
`unsupported_font_scaler_outline_format`	
`unsupported_font_scaler_stream_format`	
`unsupported_font_scaler_font_format`	
`font_scaler_hinting_error`	
`font_scaler_rasterizer_error`	
`font_scaler_internal_error`	
`font_scaler_invalid_matrix`	
`font_scaler_fixed_overflow`	
`font_scaler_api_version_mismatch`	
`font_scaler_streaming_aborted`	
`unknown_font_scaler_error`	
`spoolProcedure_is_nil`	(debugging version)
`parameter_out_of_range`	(debugging version)
`inconsistent_parameters`	(debugging version)

SEE ALSO

The `gxSpoolBlock` structure is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects.*

Getting and Setting Basic Font Storage Information

You can get a font’s storage type, storage reference, and font attributes using the `GXGetFont` function. You can then use these values to get the font ID using the `GXFindFont` function. In addition, you can change a font’s storage type, storage reference, and font attribute using the `GXSetFont` function.

GXGetFont

You can use the `GXGetFont` function to get a font's storage type, storage reference, and font attributes.

```
gxFontStorageTag GXGetFont(gxFont fontID,
                           gxFontStorageReference *reference,
                           gxFontAttribute *attributes);
```

`fontID` A reference to the font whose storage reference you want.

`reference` On return, a reference to the data of the font you specify. If you set this parameter to `nil`, the function ignores it.

`attributes` The font attributes, returned by the function. If you set this parameter to `nil`, the function ignores it.

function result The storage type of the font specified by `fontID`.

DESCRIPTION

The `GXGetFont` function returns the storage reference of the data storage type, and attributes of the font you specify. You should call this function if your application needs to know the font's storage type or storage reference.

ERRORS, WARNINGS, AND NOTICES

Errors
`illegal_font_parameter`

GXFindFont

You can use the `GXFindFont` function to get a single font with a particular storage type and storage reference.

```
gxFont GXFindFont(gxFontStorageTag storage,
                  gxFontStorageReference reference,
                  gxFontAttribute* attributes);
```

`storage` The storage type of the font you want to find.

`reference` The storage reference of the font you want to find.

`attributes` On return, the font attributes. If you set this parameter to `nil`, the function ignores it.

function result A reference to the font that matches the values given in the `storage` and `reference` parameters.

DESCRIPTION

The `GXFindFont` function returns the font with the storage type and reference you specify. If the `attribute` parameter is not `nil`, `GXFindFont` copies the font's attributes into it. If no matching font is found, `GXFindFont` returns `nil`.

ERRORS, WARNINGS, AND NOTICES

Errors

`illegal_font_storage_type`	(debugging version)
`out_of_memory`	
`illegal_font_parameter`	

GXSetFont

You can use the `GXSetFont` function to change a font's storage type, storage reference, and font attributes.

```
void GXSetFont(gxFont fontID, gxFontStorageTag storage,
               gxFontStorageReference reference,
               gxFontAttribute attributes);
```

`fontID`	A reference to the font whose storage information you want to change.
`storage`	The storage type you want to assign to the font.
`reference`	The storage reference you want to assign to the font.
`attributes`	On return, the font attributes. If you set this parameter to `nil`, the function ignores it.

DESCRIPTION

The `GXSetFont` function changes a font's storage type and storage reference to the values you provide. If you want to change only one of these values, you should call `GXGetFont` to get the original storage reference or storage type values and include it in the proper parameter of `GXSetFont`.

SPECIAL CONSIDERATIONS

You should not use the `GXSetFont` function on a system font object, because you should not change the information of any font object other than those created by your application.

ERRORS, WARNINGS, AND NOTICES

Errors

`illegal_font_storage_type`	(debugging version)
`illegal_font_attributes`	(debugging version)
`illegal_font_storage_reference`	(debugging version)
`illegal_font_parameter`	

SEE ALSO

The GXGetFont function is described on page 7-67.

GXGetFontFormat

You can use the `GXGetFontFormat` function to determine the internal format of a font.

```
gxFontFormatTag GXGetFontFormat(gxFont fontID);
```

`fontID` A reference to the font whose format you want.

function result The type of font scaler the font uses, in the form of a 4-byte tag.

DESCRIPTION

The `GXGetFontFormat` function returns a tag that specifies the font scaler of the font. This tag corresponds to the type of font scaler QuickDraw GX uses when drawing this font. Here are possible values:

Scaler tag	Font scaler
`'true'`	TrueType
`'typ1'`	Adobe Type 1
`'nfnt'`	`'NFNT'` bitmapped font

Other tags may refer to other font scalers.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`internal_font_error`

Manipulating Font Tables

QuickDraw GX allows you to work with font tables in several ways. You can

- get the number of font tables present in a particular font (`GXCountFontTables`)
- retrieve part or all of a font table that you specify by index (`GXGetFontTable`, `GXGetFontTable`Parts)
- retrieve part or all of a font table that you specify by table tag (`GXFindFontTable`, `GXFindFontTable`Parts)
- add or change part or all of font table in a font (`GXSetFontTable`, `GXSetFontTable`Parts)
- retrieve part or all of a font table (`GXDeleteFontTable`)

GXCountFontTables

You can use the `GXCountFontTables` function to get the number of font tables present in a particular font.

```
long GXCountFontTables(gxFont fontID);
```

`fontID` A reference to the font whose font tables you want to count.

function result The number of tables in the font.

DESCRIPTION

The `GXCountFontTables` function returns the number of tables in the font named by the `fontID` parameter.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

SEE ALSO

To retrieve the size and data of an entire font table that you specify by index, use the `GXGetFontTable` function, described next.

To retrieve the size and data of part of a font table that you specify by index, use `GXGetFontTableParts` function, described on page 7-72.

To retrieve the size and data of an entire font table that you specify by table tag, use the `GXFindFontTable` function, described on page 7-73.

To retrieve the size and data of part of a font table that you specify by table tag, use the The GXFindFontTableParts function, described on page 7-74.

GXGetFontTable

You can use the GXGetFontTable function to retrieve an entire font table that you specify by index.

```
long GXGetFontTable(gxFont fontID, long index, void* tableData,
                    gxFontTableTag* tableTag);
```

fontID	A reference to the font from which you want to retrieve the font table.
index	The font table's index in the font's list of tables. The number of font tables in the font is returned by GXCountFontTables.
tableData	On return, the data of the specified font table, if this parameter is not nil. Your application is responsible for allocating the memory for this parameter.
tableTag	On return, the tag of the table you are searching for, if this parameter is not nil.
function result	The size of the font table, in bytes. If the table is 0 bytes long, or if the index is out of range, GXGetFontTable returns 0.

DESCRIPTION

The GXGetFontTable function takes an index between the values of 1 and the number of tables in the font and returns the size of the font table.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

Warnings

```
font_table_index_out_of_range
```

SEE ALSO

The GXCountFontTables function is described on page 7-70.

To retrieve the size and data of part of a font table that you specify by index, use the GXGetFontTableParts function, described on page 7-72.

To retrieve the size and data of an entire font table that you specify by table tag, use the GXFindFontTable function, described on page 7-73.

To retrieve the size and data of part of a font table that you specify by table tag, use the `GXFindFontTableParts` function, described on page 7-74.

GXGetFontTableParts

You can use the `GXGetFontTableParts` function to retrieve part of a font table you specify by index.

```
long GXGetFontTableParts(gxFont fontID, long index, long offset,
                         long length, void* tableData,
                         gxFontTableTag* tableTag);
```

`fontID` A reference to the font from which you want to retrieve part of a font table.

`index` The font table's index in the font's list of tables. The number of font tables in the font is returned by `GXCountFontTables`.

`offset` The starting position in the table of the data you want to retrieve.

`length` The portion of the table you want to retrieve, in bytes.

`tableData` On return, the data of the specified font table, if this parameter is not `nil`. Your application is responsible for allocating the memory for this parameter.

`tableTag` The tag of the table you are searching for, returned by the function, if this parameter is not `nil`.

function result The number of bytes returned.

DESCRIPTION

The `GXGetFontTableParts` function returns part of the table you specify using an offset into the table and the number of bytes you want to retrieve. You can use `GXGetFontTableParts` to read part of a font table that is too large to fit entirely into memory.

If the value of the `offset` parameter added to the value of the `length` parameter is greater than the total length of the table, `GXGetFontTableParts` returns only the data from the `offset` value to the end of the table.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

Warnings

```
font_table_index_out_of_range
```

SEE ALSO

The `GXCountFontTables` function is described on page 7-70.

To retrieve the size and data of an entire font table that you specify by index, use the `GXGetFontTable` function, described on page 7-71.

To retrieve the size and data of an entire font table that you specify by table tag, use the `GXFindFontTable` function, described next.

To retrieve the size and data of part of a font table that you specify by table tag, use the `GXFindFontTableParts` function, described on page 7-74.

GXFindFontTable

You can use the `GXFindFontTable` function to retrieve an entire font table that you specify by table tag.

```
long GXFindFontTable(gxFont fontID, gxFontTableTag tableTag,
                     void* tableData, long* index);
```

`fontID`	A reference to the font from which you want to retrieve the font table.
`tableTag`	The tag of the table you want to retrieve.
`tableData`	On return, the table is copied into this buffer.
`index`	On return, the index of the table in the font.

function result The size of the table, in bytes. If no table has a tag that matches the value of `tableTag`, then the function returns 0.

DESCRIPTION

The `GXFindFontTable` function takes a font table tag and returns the size of that table, in bytes. If the value of `tableData` is not `nil`, then `GXFindFontTable` copies the font table into it. Your application is responsible for maintaining this buffer. If the `index` parameter is not `nil`, then `GXFindFontTable` copies the table's index into it.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

SEE ALSO

To retrieve the size and data of part of a font table that you specify by table tag, use the `GXFindFontTableParts` function, described next.

To retrieve the size and data of an entire font table that you specify by index, use the `GXGetFontTable` function, described on page 7-71.

To retrieve the size and data of part of a font table that you specify by index, use the `GXGetFontTableParts` function, described on page 7-72.

GXFindFontTableParts

You can use the `GXFindFontTableParts` function to retrieve part of a font table you specify by table tag.

```
long GXFindFontTableParts(gxFont fontID, gxFontTableTag tableTag,
                          long offset, long length,
                          void* tableData, long* index);
```

`fontID`	A reference to the font from which you want to retrieve part of a font table.
`tableTag`	The tag of the table you want to retrieve.
`offset`	The starting position in the table of the data you want to retrieve, in bytes.
`length`	The amount of the table you want to retrieve, in bytes.
`tableData`	On return, the table is copied into this buffer.
`index`	On return, the index of the table in the font.
function result	The number of bytes processed by the function.

DESCRIPTION

The `GXFindFontTableParts` function retrieves the part of the font table you specify by table tag. The function result is the number of bytes processed. You can use this function to read a table that is too large to fit in available memory.

If the value of `offset` added to the value of `length` exceeds the length of the table, then only the bytes from the `offset` value to the end of the table are processed. If the `index` parameter is not `nil`, then `GXFindFontTableParts` copies the table's index into it.

ERRORS, WARNINGS, AND NOTICES

Errors

```
out_of_memory
internal_font_error
illegal_font_parameter
```

SEE ALSO

To retrieve the size and data of an entire font table that you specify by table tag, use the `GXFindFontTable` function, described page 7-73.

To retrieve the size and data of an entire font table that you specify by index, use the `GXGetFontTable` function, described on page 7-71.

To retrieve the size and data of part of a font table that you specify by index, use the `GXGetFontTableParts` function, described on page 7-72.

GXSetFontTable

You can use the `GXSetFontTable` function to add or change an entire font table in a font.

```
long GXSetFontTable(gxFont fontID, long index,
                    gxFontTableTag tableTag,
                    long length, const void* tableData);
```

`fontID`	A reference to the font in which you want to add or replace a table.
`index`	The number of the table in the font's list of tables. (The number of font tables in the font is returned by `GXCountFontTables`.)
`tableTag`	The tag of the table you want to add or replace.
`length`	The length of the table you want to add or replace.
`tableData`	A pointer to the new data for the table.

function result The index of the table added or changed.

DESCRIPTION

The `GXSetFontTable` function replaces the existing table with the data in the `tableData` parameter or adds a new table. If the value of `index` is greater than 0 or the value of `tableTag` matches the table tag for an existing table, that table's data is resized to the value of the `length` parameter and replaced with the data in the `tableData` parameter. If the value of `index` is 0 and the value of `tableTag` does not match an existing table tag, `GXSetFontTable` adds a new table to the font with the tag specified in `tableTag` and the data from the `tableData` parameter.

If you enlarge or decrease a font table, QuickDraw GX is responsible for enlarging or decreasing the data in the font.

IMPORTANT

The `GXSetFontTable` function permanently changes the data in the font. In general, you should only call this function for fonts created by your application using `GXNewFont`. s

ERRORS, WARNINGS, AND NOTICES

Errors

`inconsistent_parameters` (debugging version)
`illegal_font_parameter`
`font_cannot_be_changed`
`out_of_memory`
`internal_font_error`

Warnings

`font_table_index_out_of_range`

SEE ALSO

The `GXCountFontTable` function is described on page 7-70.

The `GXNewFont` function is described on page 7-64.

To change part of an existing font table instead of the entire table, use the `GXSetFontTableParts` function described next.

GXSetFontTableParts

You can use the `GXSetFontTableParts` function to change part of a font table by adding or replacing a part of the table.

```
long GXSetFontTableParts(gxFont fontID, long index,
                         gxFontTableTag tableTag, long offset,
                         long oldLength, long newLength,
                         const void* tableData);
```

`fontID`	A reference to the font whose font table is to be affected.
`index`	The index of the table in the list of font tables, if this value is greater than 0. If it is 0, `GXSetFontTableParts` searches by the value of the `tableTag` parameter.
`tableTag`	The tag of the table you want to add to or change.
`offset`	The number of bytes from the beginning of the table to the place where you want to begin replacing data in the table.
`oldLength`	The number of bytes you want to replace.
`newLength`	The number of bytes you want to add to the table.
`tableData`	The new data for the table.

function result The index of the table added or changed.

DESCRIPTION

The GXSetFontTableParts function replaces part of the table you specify using an offset from the beginning of the table, replacing the number of bytes in oldLength with the number of bytes in newLength.

If you enlarge or decrease a font table, QuickDraw GX enlarges or decreases the font.

IMPORTANT

The GXSetFontTableParts function permanently changes the data in the font. In general, you should only call this function for fonts created by your application using GXNewFont. s

ERRORS, WARNINGS, AND NOTICES

Errors

```
illegal_font_parameter
font_cannot_be_changed
inconsistent_parameters                    (debugging version)
out_of_memory
internal_font_error
```

Warnings

```
font_table_index_out_of_range
```

SEE ALSO

To change the contents of an entire existing font table instead of just part of a font table, use the GXSetFontTable function described on page 7-75.

GXDeleteFontTable

You can use the GXDeleteFontTable function to delete a table from a font.

```
long GXDeleteFontTable(gxFont fontID, long index,
                       gxFontTableTag tableTag);
```

fontID	A reference to the font from which you want to delete the font table.
index	The index of the table you want to delete, if this value is greater than 0. If this value is 0, GXDeleteFontTable uses the value of the tableTag parameter to determine the correct table.
tableTag	The table tag of the font table you want to delete, if the value of the index parameter is 0.
function result	The index of the table deleted. If the function does not delete a font table, it returns 0.

DESCRIPTION

The `GXDeleteFontTable` function deletes the table specified by `index`, if that value is greater than 0, or by `tableTag`, if the value of `index` is 0.

If you delete a font table, QuickDraw GX decreases the size of the font.

IMPORTANT

The `GXDeleteFontTable` function permanently changes the data in the font. In general, you should only call this function for fonts created by your application using `GXNewFont`. s

ERRORS, WARNINGS, AND NOTICES

Errors

```
illegal_font_parameter
font_cannot_be_changed
inconsistent_parameters                    (debugging version)
out_of_memory
internal_font_error
```

Warnings

```
font_table_index_out_of_range
```

Changing Font Data

If you have changed the data in a font without using one of the QuickDraw GX font functions described in this chapter, you must call `GXChangedFont` to alert the system that you have changed the font.

GXChangedFont

You can use the `GXChangedFont` function if you have changed a font's data and want to alert the system that its private caches are invalid.

```
void GXChangedFont(gxFont fontID);
```

`fontID` A reference to the font you have changed.

DESCRIPTION

If you have changed a font's data directly (without using one of the QuickDraw GX font functions), you should call `GXChangedFont` to alert the system that its private caches are invalid.

QuickDraw GX automatically calls `GXChangedFont` when you alter a font using one of the system's font-editing functions, such as `GXSetFontTable`.

ERRORS, WARNINGS, AND NOTICES

Errors

```
illegal_font_parameter
```

Summary of Font Objects

Basic Constants and Data Types

The Font Object

```
typedef struct gxPrivateFontRecord *gxFont;
```

Font Variations, Instances, and Descriptors

```
typedef long gxFontVariationTag;

typedef long gxFontDescriptorTag;

struct gxFontVariation {
   gxFontVariationTag name;
   Fixed    value;
} ;
typedef struct gxFontVariation gxFontDescriptor;
```

Font Names

```
enum gxFontNames {
   gxNoFontName,
   gxCopyrightFontName,
   gxFamilyFontName,
   gxStyleFontName,
   gxUniqueFontName,
   gxFullFontName,
   gxVersionFontName,
   gxPostscriptFontName,
   gxTrademarkFontName,
   gxManufacturerFontName,
   gxLastReservedFontName = 256
} ;
typedef long gxFontName;
```

Font Feature Flags

```
#define gxMutuallyExclusiveFeature 0x8000
```

Font Features

```
typedef long gxFontFeature;

struct gxFontFeatureSetting {
   unsigned short setting;
   unsigned short nameID;
}

typedef long gxFontFeatureFlag;
```

Font Platforms

```
enum gxFontPlatforms {
   gxGlyphPlatform = -1,
   gxNoPlatform,
   gxUnicodePlatform,
   gxMacintoshPlatform,
   gxReservedPlatform,
   gxMicrosoftPlatform,
   gxCustomPlatform,
} ;
typedef long gxFontPlatform;
```

QuickDraw GX Macintosh Scripts

```
enum gxUnicodeScripts {
   gxUnicodeDefaultSemantics = 1,
   gxUnicodeV1_1Semantics,
   gxISO10646_1993Semantics
} ;

enum gxMacintoshScripts {
   gxNoScript,
   gxRomanScript,
   gxJapaneseScript,
   gxTraditionalChineseScript,
   gxChineseScript = gxTraditionalChineseScript,
   gxKoreanScript,
   gxArabicScript,
   gxHebrewScript,
   gxGreekScript,
   gxCyrillicScript,
   gxRussianScript = gxCyrillicScript,
```

```
   gxRSymbolScript,
   gxDevanagariScript,
   gxGurmukhiScript,
   gxGujaratiScript,
   gxOriyaScript,
   gxBengaliScript,
   gxTamilScript,
   gxTeluguScript,
   gxKannadaScript,
   gxMalayalamScript,
   gxSinhaleseScript,
   gxBurmeseScript,
   gxKhmerScript,
   gxThaiScript,
   gxLaotianScript,
   gxGeorgianScript,
   gxArmenianScript,
   gxSimpleChineseScript,
   gxTibetanScript,
   gxMongolianScript,
   gxGeezScript,
   gxEthiopicScript = gxGeezScript,
   gxAmharicScript = gxGeezScript,
   gxSlavicScript,
   gxEastEuropeanRomanScript = gxSlavicScript,
   gxVietnameseScript,
   gxExtendedArabicScript,
   gxSindhiScript = gxExtendedArabicScript,
   gxUninterpretedScript
} ;
typedef long gxFontScript;

enum gxCustomScripts {
   gxCustom8bitScript =1,
   gxCustom816bitScript,
   gxCustom16bitScript
};


enum gxMicrosoftScripts {
   gxMicrosoftSymbolScript =1,
   gxMicrosoftStandardScript
};
```

QuickDraw GX Macintosh Languages

```
enum gxMacintoshLanguages {
   gxNoLanguage,
   gxEnglishLanguage,
   gxFrenchLanguage,
   gxGermanLanguage,
   gxItalianLanguage,
   gxDutchLanguage,
   gxSwedishLanguage,
   gxSpanishLanguage,
   gxDanishLanguage,
   gxPortugueseLanguage,
   gxNorwegianLanguage,
   gxHebrewLanguage,
   gxJapaneseLanguage,
   gxArabicLanguage,
   gxFinnishLanguage,
   gxGreekLanguage,
   gxIcelandicLanguage,
   gxMalteseLanguage,
   gxTurkishLanguage,
   gxCroatianLanguage,
   gxTradChineseLanguage,
   gxUrduLanguage,
   gxHindiLanguage,
   gxThaiLanguage,
   gxKoreanLanguage,
   gxLithuanianLanguage,
   gxPolishLanguage,
   gxHungarianLanguage,
   gxEstonianLanguage,
   gxLettishLanguage,
   gxLatvianLanguage = gxLettishLanguage,
   gxSaamiskLanguage,
   gxLappishLanguage = gxSaamiskLanguage,
   gxFaeroeseLanguage,
   gxFarsiLanguage,
   gxPersianLanguage = gxFarsiLanguage,
   gxRussianLanguage,
   gxSimpChineseLanguage,
   gxFlemishLanguage,
   gxIrishLanguage,
   gxAlbanianLanguage,
```

```
gxRomanianLanguage,
gxCzechLanguage,
gxSlovakLanguage,
gxSlovenianLanguage,
gxYiddishLanguage,
gxSerbianLanguage,
gxMacedonianLanguage,
gxBulgarianLanguage,
gxUkrainianLanguage,
gxByelorussianLanguage,
gxUzbekLanguage,
gxKazakhLanguage,
gxAzerbaijaniLanguage,
gxAzerbaijanArLanguage,
gxArmenianLanguage,
gxGeorgianLanguage,
gxMoldavianLanguage,
gxKirghizLanguage,
gxTajikiLanguage,
gxTurkmenLanguage,
gxMongolianLanguage,
gxMongolianCyrLanguage,
gxPashtoLanguage,
gxKurdishLanguage,
gxKashmiriLanguage,
gxSindhiLanguage,
gxTibetanLanguage,
gxNepaliLanguage,
gxSanskritLanguage,
gxMarathiLanguage,
gxBengaliLanguage,
gxAssameseLanguage,
gxGujaratiLanguage,
gxPunjabiLanguage,
gxOriyaLanguage,
gxMalayalamLanguage,
gxKannadaLanguage,
gxTamilLanguage,
gxTeluguLanguage,
gxSinhaleseLanguage,
gxBurmeseLanguage,
gxKhmerLanguage,
gxLaoLanguage,
```

```
   gxVietnameseLanguage,
   gxIndonesianLanguage,
   gxTagalogLanguage,
   gxMalayRomanLanguage,
   gxMalayArabicLanguage,
   gxAmharicLanguage,
   gxTigrinyaLanguage,
   gxGallaLanguage,
   gxOromoLanguage = gxGallaLanguage,
   gxSomaliLanguage,
   gxSwahiliLanguage,
   gxRuandaLanguage,
   gxRundiLanguage,
   gxChewaLanguage,
   gxMalagasyLanguage,
   gxEsperantoLanguage,
   gxWelshLanguage = 129,
   gxBasqueLanguage,
   gxCatalanLanguage,
   gxLatinLanguage,
   gxQuechuaLanguage,
   gxGuaraniLanguage,
   gxAymaraLanguage,
   gxTatarLanguage,
   gxUighurLanguage,
   gxDzongkhaLanguage,
   gxJavaneseRomLanguage,
   gxSundaneseRomLanguage
} ;
typedef long gxFontLanguage;

#define gxLastCustomLanguage 256
```

Font Format Tag

```
typedef long gxFontFormatTag;
```

Advanced Constants and Data Types

Font Storage Tags and References

```
#define gxResourceFontStorage 0x72737263/* 'rsrc' */
#define gxHandleFontStorage 0x686e646c/* 'hndl' */
#define gxFileFontStorage 0x62617373/* 'bass' */
#define gxNfntFontStorage 0x6e666e74/* 'nfnt' */

typedef long gxFontStorageTag;

typedef void *gxFontStorageReference;
```

Font Table Tags

```
typedef long gxFontTableTag;
```

Font Attributes

```
enum gxFontAttributes {
   gxSystemFontAttribute = 0x0001,
   gxReadOnlyFontAtttribute = 0x0002
} ;
typedef long gxFontAttribute;
```

Basic Font Functions

Getting the List of Available Fonts

```
long GXFindFonts            (gxFont family, gxFontName meaning,
                             gxFontPlatform platform, gxFontScript script,
                             gxFontLanguage language, long nameLength,
                             const unsigned char name[], long index,
                             long count, gxFont fonts[]);
```

Counting Glyphs in a Font

```
long GXCountFontGlyphs      (gxFont fontID);
```

Getting and Setting the Default Font

```
gxFont GXGetDefaultFont     (void);
gxFont GXSetDefaultFont     (gxFont fontID);
```

Manipulating Font Names

```
long GXCountFontNames       (gxFont fontID);
long GXGetFontName          (gxFont fontID, long index, gxFontName *name,
                             gxFontPlatform *platform, gxFontScript *script,
                             gxFontLanguage *language,
                             unsigned char text[]);
long GXFindFontName         (gxFont fontID, gxFontName name,
                             gxFontPlatform platform, gxFontScript script,
                             gxFontLanguage language, unsigned char text[],
                             long *index);
gxFontName GXNewFontNameID  (gxFont fontID);
long GXSetFontName          (gxFont fontID, gxFontName name,
                             gxFontPlatform platform, gxFontScript script,
                             gxFontLanguage language, long nameLength,
                             const unsigned char text[]);
long GXDeleteFontName       (gxFont fontID, long index, gxFontName name,
                             gxFontPlatform platform, gxFontScript script,
                             gxFontLanguage language);
```

Manipulating Font Encodings

```
long GXCountFontEncodings   (gxFont fontID);
gxFontPlatform GXGetFontEncoding
                            (gxFont fontID, long index,
                             gxFontScript *script,
                             gxFontLanguage* language);
long GXFindFontEncoding     (gxFont fontID, gxFontPlatform platform,
                             gxFontScript script,
                             gxFontLanguage language);
long GXApplyFontEncoding    (gxFont fontID, long index, long* length,
                             const unsigned char text[], long count,
                             unsigned short glyphs[], char was16Bit[]);
```

Manipulating Font Descriptors

```
long GXCountFontDescriptors (gxFont fontID);
gxFontDescriptorTag GXGetFontDescriptor
                            (gxFont fontID, long index,
                             Fixed* descriptorValue);
long GXFindFontDescriptor   (gxFont fontID, gxFontDescriptorTag
                             descriptorTag, Fixed* descriptorValue);
long GXSetFontDescriptor    (gxFont fontID, long index,
                             gxFontDescriptorTag descriptorTag,
                             Fixed descriptorValue);
```

```
long GXDeleteFontDescriptor (gxFont fontID, long index,
                             gxFontDescriptorTag descriptorTag);
```

Manipulating Font Variations

```
long GXCountFontVariations  (gxFont fontID);
gxFontVariationTag GXGetFontVariation
                            (gxFont theFont, long index, Fixed* minValue,
                             Fixed* defaultValue, Fixed* maxValue,
                             gxFontName* nameID);
long GXFindFontVariation    (gxFont theFont, gxFontTableTag variationTag,
                             Fixed* minValue, Fixed* defaultValue,
                             Fixed* maxValue, gxFontName* nameID);
```

Manipulating Font Instances

```
long GXCountFontInstances   (gxFont, theFont);
gxFontName GXGetFontInstance(gxFont theFont, long index,
                             gxFontVariation variation[]);
long GXSetFontInstance      (gxFont fontID, long index, gxFontName name,
                             const gxFontVariation variation[]);
long GXDeleteFontInstance   (gxFont fontID, long index, gxFontName nameID);
```

Manipulating Font Features

```
long GXCountFontFeatures    (gxFont fontID);
gxFontName GXGetFontFeature (gxFont fontID, long index,
                             gxFontFeatureFlag* flags, long* settingCount,
                             gxFontFeatureSetting settings[],
                             gxFontFeature* feature);
gxFontName GXFindFontFeature(gxFont fontID, gxFontFeature feature,
                             gxFontFeatureFlag* flags, long* settingCount,
                             gxFontFeatureSetting settings[], long* index);
```

Advanced Font Functions

Adding, Removing, and Flattening Fonts

```
gxFont GXNewFont            (gxFontStorageTag storage,
                             gxFontStorageReference reference,
                             gxFontAttribute attributes);
void GXDisposeFont          (gxFont fontID);
void GXFlattenFont          (gxFont source,
                             struct gxScalerStream* stream,struct
                             gxSpoolBlock* block);
```

Getting and Setting Basic Font Storage Information

```
gxFontStorageTag GXGetFont  (gxFont fontID,
                             gxFontStorageReference *reference,
                             gxFontAttribute *attributes);
gxFont GXFindFont           (gxFontStorageTag storage,
                             gxFontStorageReference reference,
                             gxFontAttribute* attributes);
void GXSetFont              (gxFont fontID, gxFontStorageTag storage,
                             gxFontStorageReference reference,
                             gxFontAttribute attributes);
gxFontFormatTag GXGetFontFormat
                            (gxFont fontID);
```

Manipulating Font Tables

```
long GXCountFontTables      (gxFont fontID);
long GXGetFontTable         (gxFont fontID, long index, void* tableData,
                             gxFontTableTag* tableTag);
long GXGetFontTableParts    (gxFont fontID, long index, long offset,
                             long length, void* tableData,
                             gxFontTableTag* tableTag);
long GXFindFontTable        (gxFont fontID, gxFontTableTag tableTag,
                             void* tableData, long* index);
long GXFindFontTableParts   (gxFont fontID, gxFontTableTag tableTag,
                             long offset, long length, void* tableData,
                             long* index);
long GXSetFontTable         (gxFont fontID, long index,
                             gxFontTableTag tableTag, long length,
                             const void* tableData);
long GXSetFontTableParts    (gxFont fontID, long index,
                             gxFontTableTag tableTag, long offset,
                             long oldLength, long newLength,
                             const void* tableData);
long GXDeleteFontTable      (gxFont fontID, long index,
                             gxFontTableTag tableTag);
```

Changing Font Data

```
void GXChangedFont          (gxFont fontID);
```

CHAPTER 8

Layout Styles

Contents

This chapter describes the properties and features of the style object that affect only layout shapes. By manipulating style-object properties, you can override much of the font-defined behavior of the glyphs in a style run of a layout shape. You can control glyph positioning and spacing in several ways, you can modify kerning behavior, you can substitute glyphs at the end of the layout process, and you can turn on or off a large variety of font features available in QuickDraw GX fonts.

Much of the information in this chapter is optional. If the default, font-specified layout behavior provided by QuickDraw GX is sufficient for your application's needs, you do not need to use the information or techniques given here, except possibly for the setting of certain run controls that affect caret display and justification. If you do not create layout shapes, you do not need the information in this chapter. Read this chapter only if you create layout shapes and need to override the default QuickDraw GX handling of text layout and display.

Before reading this chapter, you should be familiar with the information in the chapters "Introduction to QuickDraw GX Typography," "Typographic Shapes," "Typographic Styles," and "Layout Shapes" in this book. You should also be familiar with the general concepts of QuickDraw GX objects, as described in *Inside Macintosh: QuickDraw GX Objects*.

Some layout-related properties of the style object are not discussed in this chapter. Style-object properties related to the display and positioning of carets are discussed in the chapter "Layout Carets, Highlighting, and Hit-Testing" in this book. Properties related to the justification of layout shapes, because they are commonly used to affect an entire line at a time, are discussed in the chapter "Layout Line Control" in this book. All other style-object properties that affect only layout shapes are described here.

This chapter starts by describing the layout-related properties of the style object. It then describes how to use QuickDraw GX functions to

- modify behavior by modifying the run controls structure
- override kerning behavior
- substitute individual glyphs when text of a style run is drawn
- select font features for the style run

About Layout Styles

The general features of layout shapes are described in the chapter "Layout Shapes" in this book. Because layout shapes consist of one or more style runs, each layout must have one or more style objects associated with it. The properties of the style object that are common to all typographic shapes—font, text size, text attributes, and so on—are described in the chapter "Typographic Styles" in this book. The properties that are specific to layout shapes are, for the most part, described here.

Much of the text-layout behavior in a style run is controlled by tables in the font for that run. By manipulating the layout shape–related properties of the style object, you can override that font-specified behavior for special purposes. This section discusses the

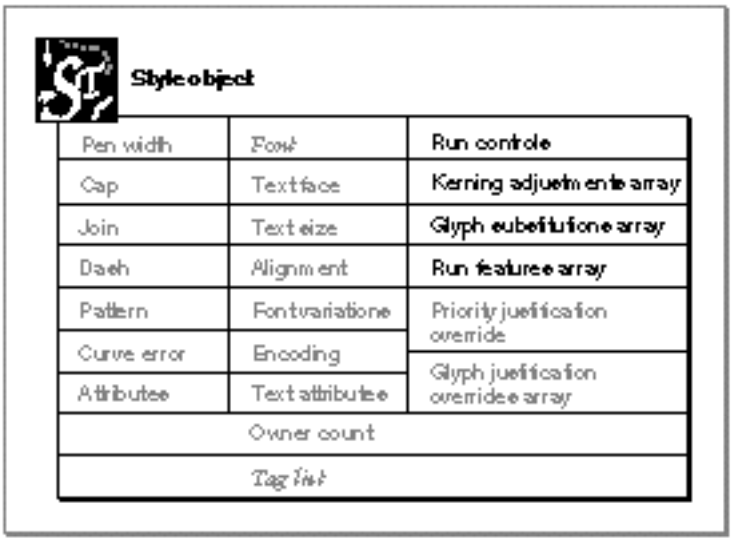
style-run controls (known simply as run controls), kerning adjustments array, glyph substitutions array, and run-features array properties and describes how they give you the ability to modify layout behavior.

Style-Object Properties Used by Layout Shapes

Every layout shape references one or more style objects, one for each style run in the shape. If a layout shape has more than one style run, the style objects for each run are specified in the style list, an array of object references that is part of the layout shape's geometry. If the layout shape has only a single style run, its style object reference can be either in a (one-element) style list or in the regular style reference that is a property of any shape object. The shape's style can also be used in the multiple-run case by specifying a `nil` on the entry in the style list.

Figure 8-1 shows the properties of a style object. Properties in the left column of the diagram are used primarily by the style objects of geometric shapes. Properties in the center column are used by the style objects of all typographic shapes, including layout shapes. Properties in the right column are used by the style objects of layout shapes only. (The two properties across the bottom are used by the style objects of all shapes.)

Figure 8-1 Layout-specific properties of the style object discussed in this chapter



Each style run in a layout shape can have its own values for all of the above properties. The upper four properties in the right column of the style object in Figure 8-1 (emphasized in black) are meaningful only within the context of an individual style run, and are therefore described in this chapter:

- **Run controls.** A structure that controls a variety of formatting and display features for a style run. See the section "Run Controls" beginning on page 8-5 for more information.

- **Kerning adjustments array.** An array that alters, for any number of glyph pairs, the kerning behavior that would otherwise be used automatically in a style run. See the section “Kerning Adjustments” beginning on page 8-16 for more information.
- **Glyph substitutions array.** An array that allows you final control over glyph selection during layout. You can specify any number of glyph pairs so that, wherever one glyph of a pair would appear in a style run, QuickDraw GX substitutes the other glyph for it. See the section “Glyph Substitutions” beginning on page 8-18 for more information.
- **Run-features array.** An array that specifies whether to employ, and at what level, various special typographic features provided by the font for a particular style run. See the section “Font Features” beginning on page 8-18 for more information.

The remaining two properties, although defined independently for each style run, are related to justification of entire lines of text. Because their effects are usually considered in that context, rather than in the context of a single style run, they are not described in this chapter:

- Priority justification override. Each entry in the priority justification override structure alters the standard justification behavior for all glyphs of a given justification priority. For more information, see the discussions of priority justification overrides in the chapter “Layout Line Control” in this book.
- Glyph justification overrides array. The glyph justification overrides array alters the standard justification behavior of one or more individual glyphs. For more information, see the discussions of glyph justification overrides in the chapter “Layout Line Control” in this book.

The rest of this section discusses run controls, kerning adjustments, glyph substitution, and run features, also known as *font features*. Priority justification overrides and glyph justification overrides are not discussed further in this chapter.

Run Controls

The run controls for a given style run are a collection of values and settings that control various aspects of the layout process. The following run-control features are described in this chapter:

- **with-stream** and **cross-stream shift,** a uniform shifting of the positions of all glyphs by the same amount, either parallel or perpendicular to the baseline
- **with-stream** and **cross-stream kerning,** the automatic adjustment of the relative positions of individual pairs or sets of glyphs; the adjustment can be parallel or perpendicular to the baseline
- **tracking,** the selection of a font-provided setting for “looseness” or “tightness” in the spacing of glyphs, which may vary in a complex way with point size
- **optical alignment,** the fine adjustment of glyph position at the line ends to give a more even visual appearance to margins
- **hanging glyphs,** a set of glyphs, usually punctuation, that typically extend beyond the left and right margins of the text area and whose widths are not counted when line length is measured
- **imposed width,** the forcing of a specific width onto the glyphs of a style run, regardless of its text content or other style properties

The following features are also defined by the run controls, although they are principally described in other chapters in this book:

- **Caret angle** is the specification of the text caret (insertion-point marker) or the edges of a highlight to be either always perpendicular to the baseline or always parallel to the slant of the style run's text. Caret angle is discussed in the chapter "Layout Carets, Highlighting, and Hit-Testing" in this book.
- **Ligature splitting** is the division of a ligature for hit-testing purposes into regions corresponding to each of its component glyphs. For illustrations of how ligature splitting affects display and highlighting, see the chapter "Layout Carets, Highlighting, and Hit-Testing" in this book.
- **Baseline type** is the specification of the fundamental baseline (such as Roman, hanging, or ideographic centered) that the text of this style run is to use. Baseline types are discussed in the chapter "Layout Line Control" in this book.
- **Direction override** is the imposition of a left-to-right or right-to-left direction onto the glyphs in this style run, regardless of their natural direction as specified in the font. Glyph direction and direction overrides are discussed in the chapter "Layout Line Control" in this book.
- **Postcompensation action** is a set of processes (such as glyph stretching and ligature decomposition) that occur at the end of the justification process, after glyph positions have been calculated. You can prevent postcompensation action from occurring; see the discussion of justification in the chapter "Layout Line Control" in this book.
- **Ligature decomposition** is the breaking up of a ligature into its component glyphs during justification, so that the individual glyphs may more evenly occupy the space allotted to the ligature. Ligature decomposition occurs at a font-specified threshold that you can change. See the discussion of justification in the chapter "Layout Line Control" in this book.

The run controls for a style run are contained in the run controls structure, described on page 8-57. The rest of this section discusses the run controls that affect shifting, kerning, tracking, optical alignment, hanging glyphs, and imposed width.

With-Stream Shift and Cross-Stream Shift

Your application can specify two types of positional shifts that apply equally to all glyphs in a style run. **With-stream shift** adds or removes space before or after each glyph in the run, and can be used for manual kerning or letterspacing. **Cross-stream shift** raises or lowers the entire style run (or shifts it sideways if it's vertical text), and can be used for superscript and subscript effects.

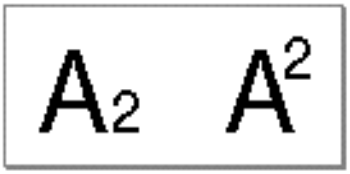
You can apply with-stream shift before (to the left of) or after (to the right of) the glyphs of the style run, or both. A shift may be either positive or negative. Positive with-stream shift moves the glyphs farther apart; negative shift moves them closer together. Positive cross-stream shift moves the glyphs upward from the baseline (as in superscripts); negative shift moves them downward (as in subscripts).

Figure 8-2 shows an example of a negative and a positive with-stream shift applied before (to the left of) the glyphs of a style run. The glyphs "c" and "d" constitute a single style run. The line is drawn first with no shift, then with a large negative with-stream shift for that style run, and finally with an even larger positive with-stream shift.

Figure 8-2 Negative and positive with-stream shift

Figure 8-3 illustrates the simultaneous use of with-stream and cross-stream shift. The layout consists of two style runs of a single glyph each. On the left the layout is drawn with no shifting. On the right, a negative with-stream shift is applied before the “2”, and a positive cross-stream shift is applied to the “2”. The net result is a well-proportioned and well-placed superscript. (There are other ways to make superscripts, including the use of superiors; see Table 8-10 on page 8-32.)

Figure 8-3 Combining with-stream and cross-stream shift



When text is shifted in a with-stream direction, the boundary (caret position) between pairs of glyphs is adjusted to be halfway between the advance width of the earlier glyph and the origin of the later glyph, as shown in Figure 8-4.

Figure 8-4 Caret position between with-stream shifted glyphs

A sample function that makes use of both with-stream and cross-stream shift is shown in Listing 8-2 on page 8-43.

Using with-stream and cross-stream shifts can give your application a full manual letter-spacing capability. This kind of positioning control, however, works differently than the automatic letterspacing capabilities of QuickDraw GX (kerning and tracking, described in the following sections).

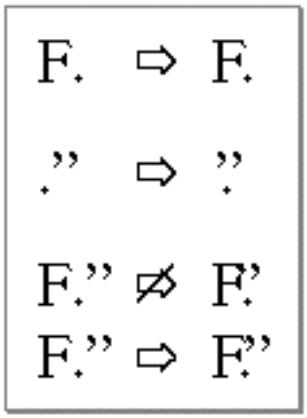
With-Stream Kerning and Cross-Stream Kerning

Kerning is an automatic, fine adjustment to normal spacing that QuickDraw GX applies to specific pairs or groups of glyphs, or to glyphs in specific contexts. It is commonly used to increase the overlap between glyphs that "fit together" naturally. Unlike manual shifting, kerning does not apply evenly to all glyphs in a style run. Also, kerning does *not* refer to the apparent overlap that can be caused by glyphs that overhang their bounds (glyphs that extend beyond their leading or trailing edges defined by the character origin and advance width), as shown in Figure 8-5.

Figure 8-5 Apparent kerning caused by a glyph that extends beyond its advance width

QuickDraw GX uses information in font tables to determine how much to increase or decrease the space between glyphs. In the general case, this amount can depend on more than just the two adjacent glyphs: it can also depend on preceding or following glyphs, or even on glyphs in other parts of the line. For example, the two pairs of glyphs in Figure 8-6 might kern, but the triple would not--at least not in the same manner as the two separate pairs.

Figure 8-6 When kerning can and cannot occur

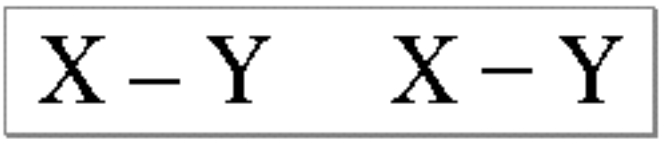


For determining caret positions, kerning offset is effectively split between glyphs in a kerned pair. The example on the right in Figure 8-7 shows where the caret would appear between the two kerned glyphs.

Figure 8-7 Caret position between two kerned glyphs

Cross-stream kerning allows the automatic movement of glyphs perpendicular to the line orientation of the text. (For horizontal text, the automatic movement is vertical.) For example, Figure 8-8 (right) shows how a hyphen between two uppercase glyphs could be raised to reflect the centers of those characters.

Figure 8-8 Cross-stream kerning



Cross-stream kerning is required for scripts like Taliq (used in Urdu). It can also be used to assist in the creation of automatic fractions. (See page 8-32 for additional discussion of automatic fractions.)

Kerning Inhibit

Kerning inhibit is a feature of QuickDraw GX that allows you to partially or fully defeat the font-specified kerning for all glyphs of a style run. You can specify that only a percentage (from 100 percent down to 0) of the font-defined with-stream kerning amount is to

be applied when QuickDraw GX draws the glyphs of a style run. Figure 8-9 shows the same phrase written three times, first with normal kerning (top), then with only half the normal kerning amount, and finally with no kerning at all.

Figure 8-9 Partially and fully inhibiting kerning

You also can completely override the font-specified kerning for individual pairs of glyphs in a style run, giving them any value you wish. See “Kerning Adjustments” beginning on page 8-16.

Tracking

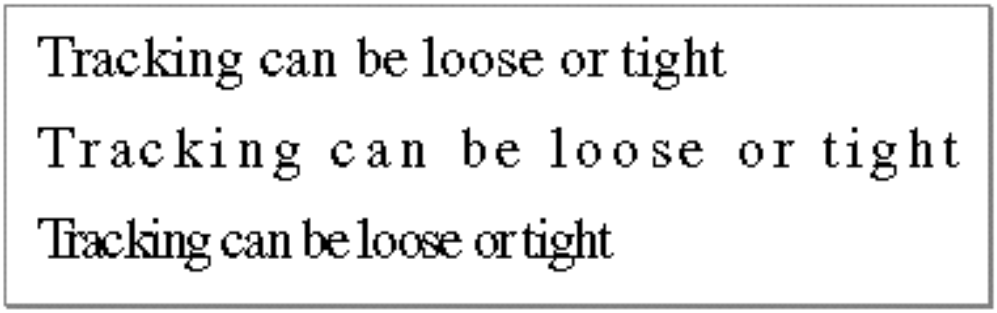
Tracking is another glyph-positioning adjustment you can control. You can expand or contract the spacings of all glyphs in a style run by applying a value to that style run. This value, called the **track setting,** uses information defined by the font to specify whether interglyph spacing is to be tightened or loosened.

Tracking is different from with-stream shifting because the actual amount of space added or removed is controlled by the font, not by your application. The positional shifts are the result of two-dimensional interpolation based on the track setting, the text size in points, and the threshold values present in the font’s tracking table. These threshold values are used to permit nonlinear tracking amounts: for example, a single track setting can specify different sets of spacings for text below 8 points, from 8 to 12 points, from 12 to 15, from 15 to 36, and over 36 points, if the font designer wishes it.

Specifying a track setting of 0 means “space normally” according to the specifications of the font designer. That does *not* necessarily mean that no adjustment to spacing occurs. The font designer may decide that “normal spacing” includes some spacing adjustment in certain point size ranges.

Figure 8-10 shows the same phrase written three times in a particular font. At the top the application specified a track setting of 0; in the middle, it specified a large positive track setting (+2); and at the bottom it specified a large negative track setting (–2).

Figure 8-10 Tracking with track settings



The sample function that generated Figure 8-10 is shown in Listing 8-3 on page 8-45.

Optical Alignment

In multiline text, glyphs may seem to line up incorrectly at the margins. This is accounted for by two factors. First, glyph advance widths contain a certain amount of extra white space (**side bearing**) to account for the normal interglyph spacing, as shown in Figure 8-11.

Figure 8-11 Advance widths, including side bearings to allow for interglyph spacing

This produces certain anomalies at line margins, because the side bearing varies with font size. For example, if different sizes of a single glyph from the same font are left-aligned, they may not line up exactly, as Figure 8-12 shows.

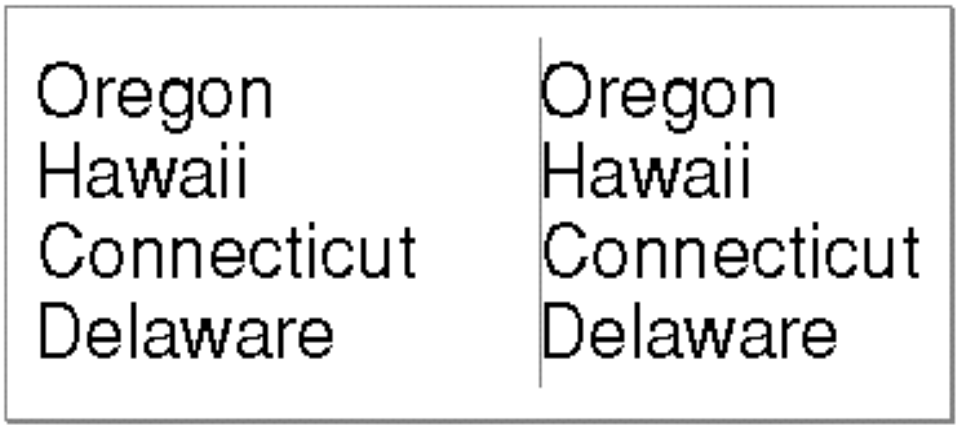
Figure 8-12 Misalignment caused by advance widths that vary with glyph size

The second problem is that due to optical effects, curved lines do not appear to line up properly with straight lines. To make them appear to line up, some compensation must occur. On baselines, for example, curved letters such as “C” or “S” are generally designed to extend slightly below the baseline, so that they appear to line up with straight letters such as “H”, as in Figure 8-13.

Figure 8-13 How curved letters extend below the baseline to align with straight letters

This same effect should happen at the edges of lines. On the left side of Figure 8-14, the “O” in “Oregon” and the “C” in “Connecticut” appear to be indented compared to the “H” and “D” glyphs. However, as shown by the vertical line on the right, the outlines of the four glyphs are exactly aligned. The apparent indentation is an optical effect.

Figure 8-14 Apparent misalignment of curved letters



To compensate for these effects, QuickDraw GX can apply optical alignment information contained in the font. When determining the leading and trailing edges of a line of text, QuickDraw GX uses the optical leading and trailing edges.

In Figure 8-15, the black arrow spans the distance from the origin to the advance width of the glyph, defining the width of the glyph plus side bearings. The spaces between the solid lines represent the side bearings, and the dashed lines represent the **optical edges** of the glyph. Note that on each side the optical edge is further inset from the standard edge by more than the amount of the side bearing.

Figure 8-15 The optical edges of a glyph

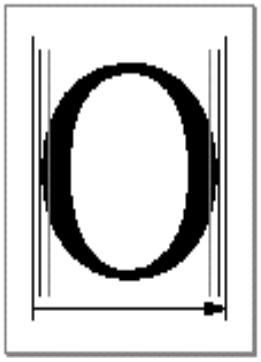
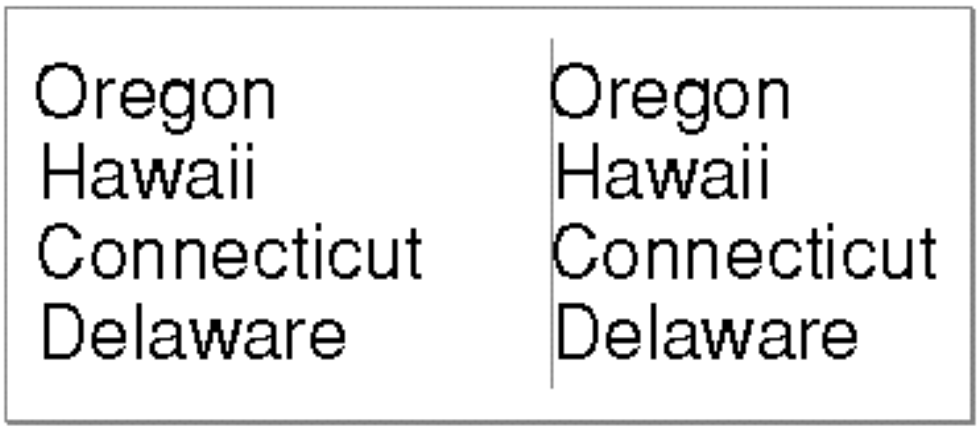


Figure 8-16 shows the same text as Figure 8-14, except that on the left in Figure 8-16, the glyphs appear to be aligned. However, as shown by the vertical line on the right, the outlines of the four glyphs are not exactly aligned; the glyphs have been shifted to compensate for optical effect.

Figure 8-16 Optical alignment at line edges

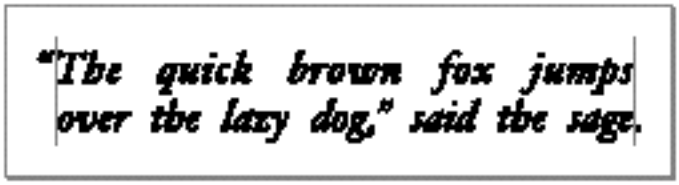


QuickDraw GX applies optical alignment by default. Your application can suppress it by setting the flag `gxNoOpticalAlignment` in the run controls structure. The `gxNoOpticalAlignment` flag is described on page 8-61.

Hanging Glyphs

One of the properties that QuickDraw GX understands about a glyph is whether it is permitted to “hang” off one or both ends of a line. This property is font-specified, and is usually true for “lightweight” punctuation, such as quotation marks or periods. This permits automatic alignment of text lines such as that shown in Figure 8-17.

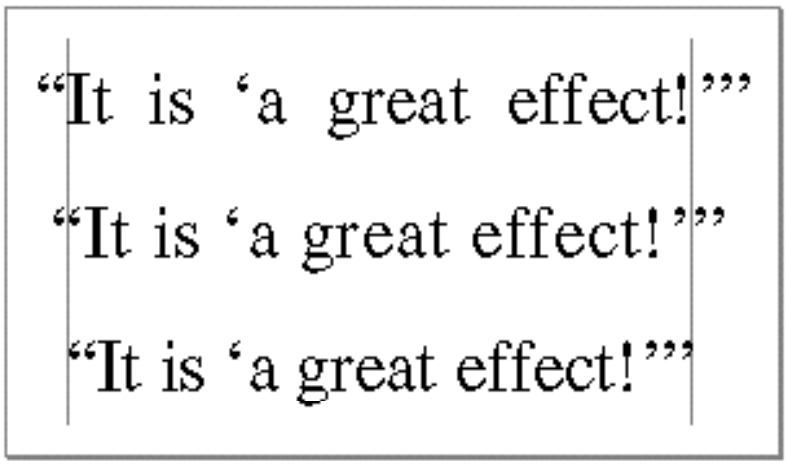
Figure 8-17 Automatic hanging punctuation



By default, QuickDraw GX uses this font-specific information to automatically hang punctuation where appropriate. Your application has the ability to control the degree to which this happens (or whether it happens at all). You can use **hanging inhibit** to control the degree to which the hanging punctuation glyphs in a style run hang. A value of 0 (the

default) indicates that the glyph should hang by the normal amount. A positive nonzero value lessens the amount of hanging proportionally, down to a value of 1, which means "no hanging at all." Figure 8-18 shows the same line of (justified) text laid out with no hanging inhibit (top), with an inhibit of 0.5 (middle), and with full inhibit (bottom).

Figure 8-18 Effects of hanging inhibit factor



You can also specify that all glyphs of a particular style run are to be hanging glyphs, whether or not the font designer intended them to be. Figure 8-19 shows a line in which the question mark, which is not normally a hanging glyph, is in its own style run and is defined as hanging; it therefore extends beyond the margin.

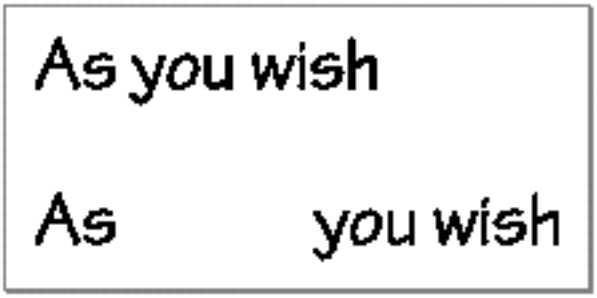
Figure 8-19 Defining a nonhanging glyph as a hanging glyph

Imposed Width

If a picture or other graphic is to be embedded in a line of text, your application can create a gap at a specific point in that line by using a single whitespace character as its own style run and imposing a width on that style run, as shown in Figure 8-20. The specified glyph always has the imposed width, regardless of the point size of the text, to within a single pixel in device resolution.

Figure 8-20 shows a layout shape in which one of the style runs consists of only the whitespace character between the words “As” and “you”. The shape is drawn twice; first with no imposed width and then with an imposed width on the whitespace character. By imposing a width on the style run, you can make the gap between the words as large as you wish.

Figure 8-20 A style run with an imposed width in a line of text



The sample function that generated Figure 8-20 is shown in Listing 8-6 on page 8-48.

Kerning Adjustments

As described in the section “With-Stream Kerning and Cross-Stream Kerning” beginning on page 8-8, QuickDraw GX uses font information to automatically kern pairs and groups of glyphs contextually, and you can partially or fully inhibit that kerning by setting a field in the run controls structure.

The kerning specified by QuickDraw GX-compatible fonts is sophisticated and should be sufficient in nearly all situations. However, your application can exert additional influence on kerning behavior if necessary. Using the kerning adjustments array in the style object, you can override the normal kerning for individual pairs of glyphs.

Adjustments to kerning involve both a **point-size factor** and a **scale factor.** The adjustment is $ax + b$ where x is the automatic kerning value specified in the font, a is the scale factor, and b is the product of the point-size factor (b') and the run’s point size (s). (The value (b') is the value included in the data structure.) The adjustment is always *added* to the normal, font-specified kerning in a given situation.

To apply the kerning adjustments, QuickDraw GX performs the following calculations. Here x' represents the new kerning value after adjustments have been applied:

```
x' = x + (ax + b)
```

where

```
b = b' * s
```

So, with a scale factor of 1, you would get

```
x' = x + (x + b)
```

which means that the final kerning value would be (2*x* + *b*). Therefore, to simply double the font-specified kerning, use a scale factor of 1 and a point-size factor of 0. In general, to multiply the font-specified kerning by any factor *n*, use a scale factor of (*n* -1) and a point-size factor of 0.

With a scale factor of –1, you would get

```
x' = x + (-x + b)
```

which means that the final kerning value would be (*b*). Therefore, to replace the font-specified kerning with your own constant value, use a scale factor of –1 and a point-size factor that, when multiplied by the point size, yields the total kerning amount that you want.

With a scale factor of 0, you would get

```
x' = x + (0x + b)
```

which means the final result will be (*x* + *b*). Therefore, to make a constant adjustment to the kerning value, use a scale factor of 0 and a point-size factor that, when multiplied by the point size, yields the additional kerning amount that you want.

Figure 8-21 shows an example in which the kerning for the “W A” glyph pair is adjusted twice. The upper line is drawn with normal kerning. The middle line is drawn with a scale factor of –0.5 and a point-size factor of 0, giving a result of half the normal kerning. The bottom line is drawn with a scale factor of –1 and a point-size factor of +0.5, which removes the normal kerning and adds a large constant value to the spacing.

Figure 8-21 Application-specified kerning adjustments

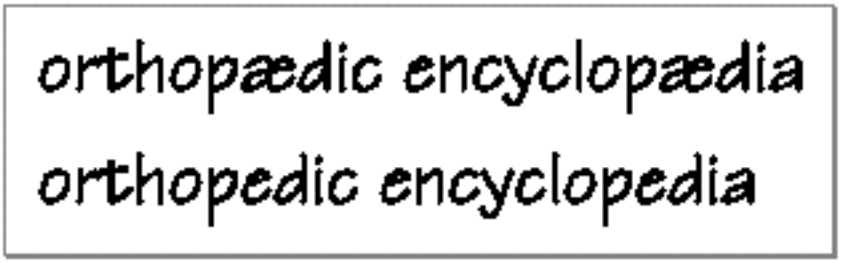
The sample function that generated Figure 8-21 is shown in Listing 8-7 on page 8-50.

Glyph Substitutions

Your application can have final control over the glyphs chosen by QuickDraw GX when it lays out a line. You can specify, in the glyph substitutions array of the style object, specific glyphs that QuickDraw GX is to substitute for other specific glyphs before drawing. Your application has the last say, because glyph substitutions specified in this way always occur after all automatic glyph substitutions that QuickDraw GX performs (except for substitutions that may occur during postcompensation action—see the discussion of justification in the chapter “Layout Line Control” in this book).

Figure 8-22 shows a simple example for demonstration purposes, in which all lowercase “æ” glyphs are replaced with “e” using glyph substitutions. More realistically, you might use glyph substitutions to allow users to apply specific swash variants of glyphs, in situations not provided for through other methods such as the smart-swash font features described in Table 8-8 on page 8-30.

Figure 8-22 Application-controlled glyph substitution



The sample function that generated Figure 8-22 is shown in Listing 8-8 on page 8-52.

Font Features

Much of the text-layout capability of QuickDraw GX happens automatically. Tables in QuickDraw GX–compatible fonts control the layout process in many ways. Thus, applications can get sophisticated linguistic and layout behavior without having to specify parameters to control it, and without having to implement it themselves. As has been shown in earlier sections in this chapter, automatic kerning, optical alignment, and placement of hanging punctuation are examples of font-specified layout capabilities that occur without your application’s intervention.

Another large category of layout capabilities controlled by QuickDraw GX–compatible fonts is called *font features*. **Font features** are typographic and layout capabilities that can be selected or deselected by an application, and that control many aspects of glyph selection, ordering, and positioning. Font features include fundamental controls such as whether or not contextual forms are to be used, and details of appearance such as whether or not alternate forms of glyphs are to be used at the beginnings of words. To a large extent, the appearance of a layout shape is a function of the number and kinds of font features chosen.

Font features are applied to each style run based on font defaults, modified by values in the run-features array of the style object, if present. Because you specify these features on a run-by-run basis, you can customize the layout of text with dissimilar fonts or even different languages on a single line. There is no universally prescribed set of default font features; each font picks which features to support and which ones to "turn on" by default. But you can use the run-features array to turn on font features that are off by default, or to turn off font features that you do not want.

Font vendors create tables that implement a set of font features from which your application can pick and choose. The architecture of font features is open-ended; as font vendors create new kinds of features, QuickDraw GX automatically takes advantage of them. An initially defined standard set of features is described in this chapter; as new fonts add new features, the defined set of font features will be expanded to accommodate them. Your application can query fonts to determine the available set of features and their names, using QuickDraw GX functions such as `GXCountFontFeatures`, `GXGetFontFeature`, and `GXFindFontFeature`. See the chapter "Font Objects" in this book for more information.

Feature Types, Feature Selectors, and the Feature Registry

Font features are grouped into categories called **feature types,** within which individual **feature selectors** are used to define particular feature settings or selections. Feature types and feature selectors are defined and listed in the *QuickDraw GX Font Feature Registry*, a document maintained by Apple Computer, Inc. Although this chapter gives examples of feature types and feature selectors, the specific set of features and their names as given here may not be complete. The feature registry is the official document that defines them, and it is evolving as new fonts are created.

To obtain the latest version of the feature registry, please contact Apple Computer, Inc., at the AppleLink address FONTREGISTRY.

Table 8-1 lists examples of feature types that a font can support and that your application can choose among when laying out text.

Table 8-1 Examples of feature types

Constant	Explanation
`allTypographicFeaturesType`	Specifies whether or not any font features are to be applied at all. Table 8-2 on page 8-22 lists the feature selectors related to this feature type.
`ligaturesType`	Specifies the use of required ligatures and other categories of optional ligatures. Table 8-3 on page 8-24 lists the feature selectors related to this feature type.
`cursiveConnectionType`	Specifies whether or not cursive connections are to be used between glyphs. Table 8-4 on page 8-26 lists the feature selectors related to this feature type.

continued

Table 8-1 Examples of feature types (continued)

Constant	Explanation
`letterCaseType`	Specifies case changes, such as all uppercase, all lowercase, and small caps, for scripts in which case has meaning. Table 8-5 on page 8-26 lists the feature selectors related to this feature type.
`verticalSubstitutionType`	Allows substitution of vertical forms of particular glyphs (such as parentheses) in vertical runs of text. Table 8-6 on page 8-27 lists the feature selectors related to this feature type.
`linguisticRearrangementType`	Either permits or inhibits linguistic (Indic-style) rearrangement of glyphs. Table 8-7 on page 8-28 lists the feature selectors related to this feature type.
`smartSwashType`	Controls whether swash variants of glyphs are to be substituted in specific places in the text, such as at the beginnings or ends of words or lines. Table 8-8 on page 8-30 lists the feature selectors related to this feature type.
`diacriticsType`	Controls whether diacritical marks are shown or hidden. Table 8-9 on page 8-31 lists the feature selectors related to this feature type.
`verticalPositionType`	Controls the selection of superscript and subscript glyph sets. Table 8-10 on page 8-32 lists the feature selectors related to this feature type.
`fractionsType`	Controls automatic substitution or formation of fractions. Table 8-11 on page 8-33 lists the feature selectors related to this feature type.
`overlappingCharactersType`	Controls whether long tails on glyphs are permitted to collide with other glyphs. Table 8-12 on page 8-34 lists the feature selectors related to this feature type.
`characterShapeType`	Specifies for languages such as Chinese that have both sets whether traditional or simplified characters are to be used. Table 8-13 on page 8-35 lists the feature selectors related to this feature type.
`numberSpacingType`	Specifies whether to use fixed-width or proportional-width glyphs for numerals. Table 8-14 on page 8-35 lists the feature selectors related to this feature type.

continued

Table 8-1 Examples of feature types (continued)

Constant	Explanation
`numberCaseType`	Specifies whether to use numerals that do, or do not, extend below the baseline. Table 8-15 on page 8-36 lists the feature selectors related to this feature type.
`styleOptionsType`	Specifies any of several named alternative forms that may be available in the font, such as engraved or cursive. Table 8-16 on page 8-37 lists the feature selectors related to this feature type.
`typographicExtrasType`	Controls several effects, such as substitution of en dashes for hyphens, that are associated with sophisticated typography. Table 8-17 on page 8-37 lists the feature selectors related to this feature type.
`mathematicalExtrasType`	Controls several features, such as changing asterisks to multiplication symbols, used for typesetting mathematical expressions. Table 8-18 on page 8-38 lists the feature selectors related to this feature type.
`ornamentSetsType`	Specifies certain sets of non-alphanumeric glyphs, such as decorative borders or musical symbols. Table 8-19 on page 8-39 lists the feature selectors related to this feature type.
`characterAlternativesType`	Specifies, by number, any font-specific set of alternate glyph forms. Table 8-20 on page 8-40 lists the feature selector related to this feature type.
`designComplexityType`	Specifies an overall complexity of appearance, as defined by the font. Table 8-21 on page 8-40 lists the feature selectors related to this feature type.

Within some feature types, you can choose only one of the available feature selectors, such as whether numbers are to be proportional or fixed-width. With other feature types you can turn “on” or “off” any number of feature selectors at once; for example, under ligatures you can choose any combination of the available classes of ligatures that the font supports.

Your application can select a group of features, place selectors for them in a run-features array, and assign that array to a layout shape’s style object. QuickDraw GX will then use those features, plus any font-specified features not overridden by your feature selections, when it draws the layout shape.

You can also turn font features on or off. Table 8-2 lists the feature selectors for the `allTypographicFeaturesType` feature type; by specifying the selector `allTypeFeaturesOnSelector` or `allTypeFeaturesOffSelector` for that feature type, you can turn the entire set of features on or off. Note that if you turn all font features off this way, you turn off *all* font features, including all the font-specified defaults. (That may result in linguistically incorrect display.) If you turn font features on, you turn on the font-specified defaults, modified by whatever feature settings you have specified in the run-features array.

Table 8-2 Feature selectors for the `allTypographicFeaturesType` font feature type

Constant	Explanation
`allTypeFeaturesOnSelector`	Tells QuickDraw GX to use the font features specified in this style run's run-features array and the defaults specified by the font.
`allTypeFeaturesOffSelector`	Tells QuickDraw GX to ignore all font features specified either by the font or in this style run's run-features array.

The rest of this section gives examples of the kinds of feature selectors that may be available for some of the feature types listed in Table 8-1. Please consult the feature registry for more up-to-date information.

Contextual Font Features

One class of font features is contextual, meaning that how (or if) the feature is applied to a given glyph depends on the glyph's position compared to adjacent glyphs. Much of QuickDraw GX's text-layout power results from its ability to apply sophisticated contextual processing.

QuickDraw GX's ability to automatically substitute one or more glyphs for one or more other glyphs is called **automatic form substitution.** QuickDraw GX supports several kinds of automatic form substitution, including ligatures, cursive contextual forms, contextual case substitution, vertical substitution, rearrangement, automatic fraction generation, and others.

Automatic Ligature and Contextual Form Generation

A **ligature** is a rendering form that represents a combination of two or more individual characters. Examples include the "fi" ligature in English (Figure 8-23) and the miim-miim ligature in Arabic (Figure 8-24).

Figure 8-23 Ligatures in Roman text

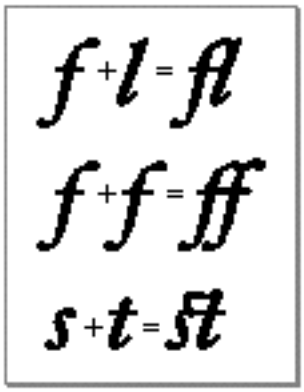
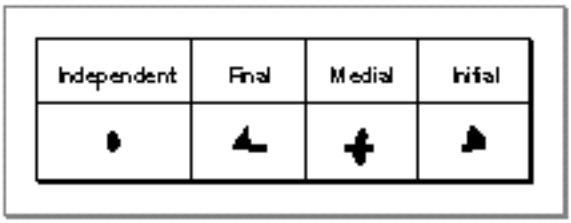


Figure 8-24 A ligature in Arabic text

A **contextual form** is an alternate appearance of a glyph that is used in certain contexts. Arabic, for example, has different contextual forms of characters, depending on whether they are at the beginning, the middle, or the end of a word. Figure 8-25 shows the forms of the Arabic letter “ha” that appear alone, at the beginning, middle, or end of a word. The same character code is used in each case; QuickDraw GX chooses the correct glyph when laying out the text.

Figure 8-25 Versions of the Arabic letter “ha”



Ligatures

If the font supports the ligatures feature type, you can select features related to ligature formation, such as those shown in Table 8-3.

Table 8-3 Feature selectors for the `ligaturesType` feature type

Constant	Explanation
`requiredLigaturesOnSelector` `requiredLigaturesOffSelector`	Allows or prevents the use of ligatures that the font designates as required by the language (such as certain Arabic ligatures).
`commonLigaturesOnSelector` `commonLigaturesOffSelector`	Allows or prevents the use of ligatures that the font designates as “common,” or normally used (such as the “fi” ligature in Roman text).
`rareLigaturesOnSelector` `rareLigaturesOffSelector`	Allows or prevents the use of ligatures that the font designates as “rare” (such as “ct” or “ss” ligatures).
`logosOnSelector` `logosOffSelector`	Allows or prevents the use of ligatures that the font designates as logotypes (typically used for trademarks or other special display text).
`rebusPicturesOnSelector` `rebusPicturesOffSelector`	Allows or prevents the use of rebuses (pictures that represent words or syllables).
`diphthongLigaturesOnSelector` `diphthongLigaturesOffSelector`	Specifies whether or not to replace diphthong sequences, such as “AE” and “oe”, with their equivalent ligatures (“Æ” and “œ” in this case).

Figure 8-26 shows several levels of ligature formation specified through ligature feature selectors. The sample function that generated Figure 8-26 is shown in Listing 8-9 on page 8-53.

Figure 8-26 Levels of ligature formation controlled with ligature feature selectors

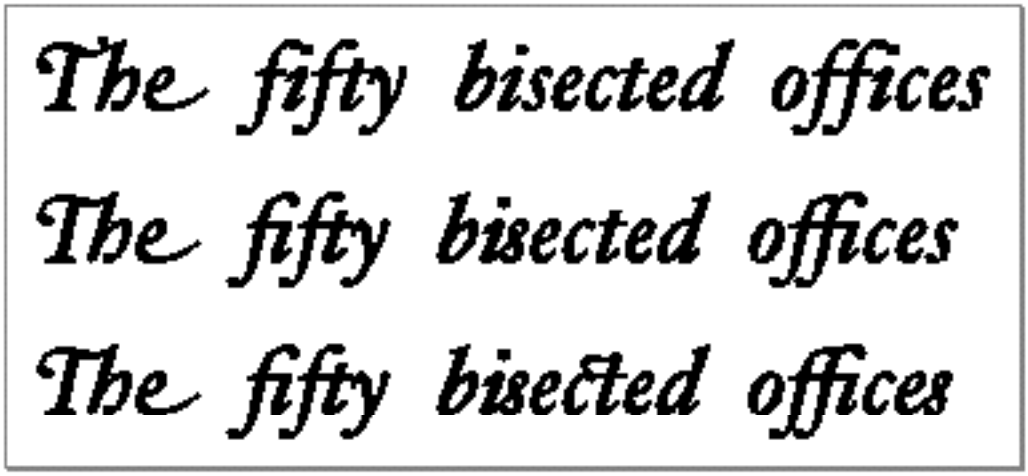
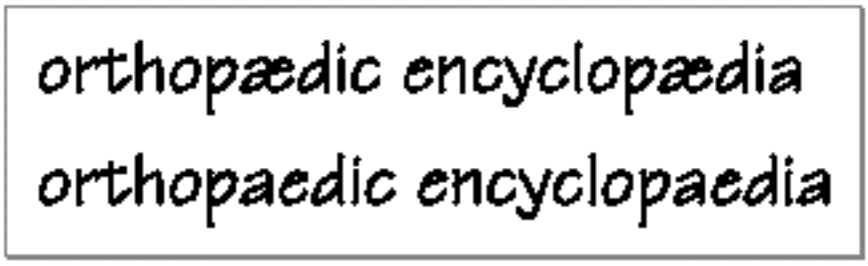


Figure 8-27 shows the results of selection (upper) and deselection (lower) of diphthong ligatures.

Figure 8-27 Use of diphthong ligatures



Cursive Connection

All Arabic fonts use cursive connection, and some Roman fonts may also support cursive connection. If a font supports the cursive connection feature type, you may be able to select features that either disable cursive connection completely, enable letterforms that connect in a noncontextual manner, or enable completely contextual, cursively connected letterforms (as in Arabic). Table 8-4 lists the feature selectors for cursive connection.

Table 8-4 Feature selectors for the `cursiveConnectionType` feature type

Constant	Explanation
`unconnectedSelector`	Disables cursive connection.
`partiallyConnectedSelector`	Specifies noncontextual cursive connection.
`cursiveSelector`	Specifies fully contextual cursive connection. For Arabic fonts, this selector is set by default.

Figure 8-28 shows an example of noncontextual cursive connection in a Roman font.

Figure 8-28 Noncontextual cursive connection in a Roman font

Letter Case

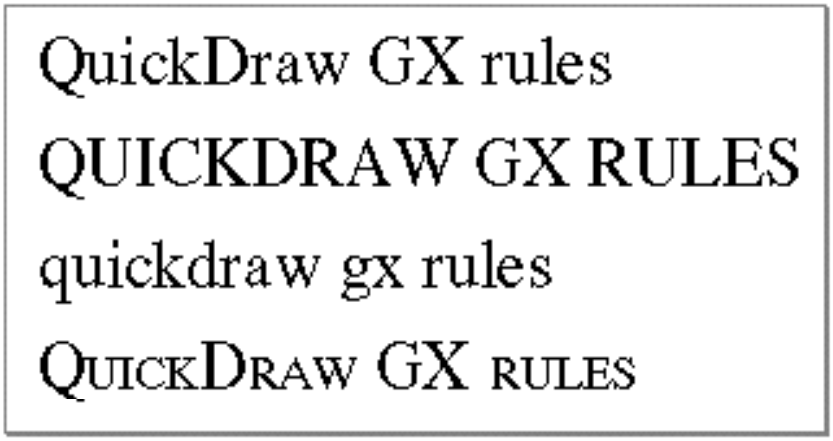
In fonts for languages in which case is significant, QuickDraw GX allows you to specify certain automatic case changes. If the font supports the letter case feature type, you can select features that specify case changes such as those shown in Table 8-5.

Table 8-5 Feature selectors for the `letterCaseType` feature type

Constant	Explanation
`upperAndLowerCaseSelector`	Specifies no case conversion.
`allCapsSelector`	Specifies conversion of all letters to uppercase. (This feature is noncontextual.)
`allLowerCaseSelector`	Specifies conversion of all letters to lowercase. (This feature is noncontextual.)
`smallCapsSelector`	Specifies conversion of all lowercase letters to small caps. (This feature is noncontextual.)
`initialCapsSelector`	Specifies conversion of all lowercase letters at the beginnings of words to uppercase. (This feature is contextual.)
`initialCapsAndSmallCapsSelector`	Specifies conversion of all lowercase letters at the beginnings of words to uppercase, and all other lowercase letters to small caps. (This feature is contextual.)

Figure 8-29 shows a phrase that is first drawn with no case conversion (`upperAndLowerCaseSelector`), and then with the selectors `allCapsSelector`, `allLowerCaseSelector`, and `smallCapsSelector`, respectively.

Figure 8-29 Case conversion



Note
Contrary to common perception, the small caps style is not simply the use of capital letters in a smaller point size. If the font contains true small caps glyphs, you can specify them with a letter case feature selector, and QuickDraw GX will use them. u

Vertical Substitution

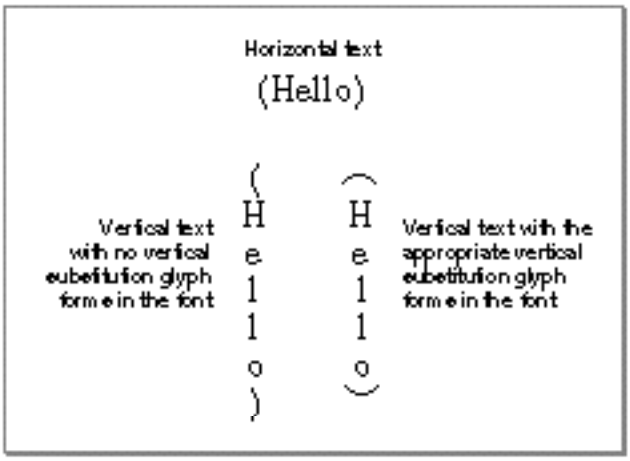
Vertical substitution is a glyph substitution in which the glyph for a given glyph code is replaced by an alternate form in a vertical line. (This is not the same as rotating the glyph.) Table 8-6 shows the feature selectors for vertical substitution.

Table 8-6 Feature selectors for the `verticalSubstitutionType` feature type

Constant	Explanation
`substituteVerticalFormsOnSelector` `substituteVerticalFormsOffSelector`	Allows or prevents the substitution of alternate glyph forms in vertical lines.

Figure 8-30 illustrates how vertical substitution works.

Figure 8-30 Vertical substitution forms in a font



For vertical substitution to happen, the vertically rotated forms must exist in the font and must be indicated as such in the font's tables; otherwise, no characters are substituted. If the font supports the vertical substitution feature type, its default behavior is to perform such substitutions; you may either prevent the substitution or allow it to occur.

Linguistic Rearrangement

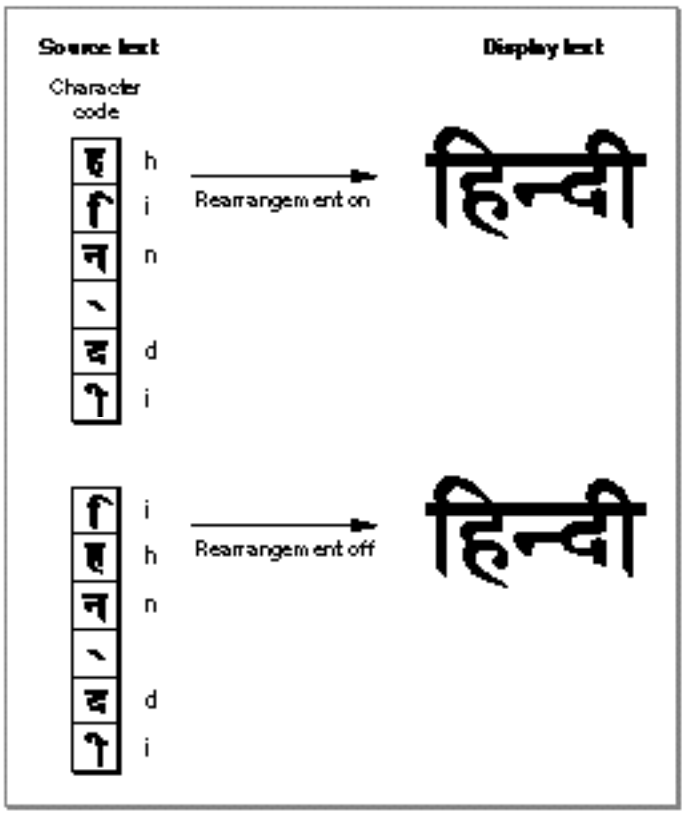
Linguistic (Indic-style) rearrangement is a standard feature of Devanagari and other South Asian scripts. However, users may not always want it to occur, preferring instead to enter characters in an "already reversed" order. If a font supports the rearrangement feature type, you can either allow the default behavior (which is to perform rearrangement) or you can prevent it. Table 8-7 shows the feature selectors for rearrangement.

Table 8-7 Feature selectors for the `linguisticRearrangementType` feature type

Constant	Explanation
`linguisticRearrangementOnSelector` `linguisticRearrangementOffSelector`	Allows or prevents the automatic rearrangement of certain glyphs as required by language rules.

Figure 8-31 shows two examples of the display of the word “hindi”, first with linguistic rearrangement on and then with it off. Note that when rearrangement is off, the storage order of the character codes in the source text must reflect display order, rather than normal input order.

Figure 8-31 The word “hindi” drawn with rearrangement turned on (upper) and off (lower)



Swashes and Smart Swashes

A **swash** is a variation, often ornamental, of an existing glyph. Using font tables, QuickDraw GX can identify and automatically substitute swashes for existing glyphs. Alternatively, your application can allow the user to choose swash forms at the time the layout is created.

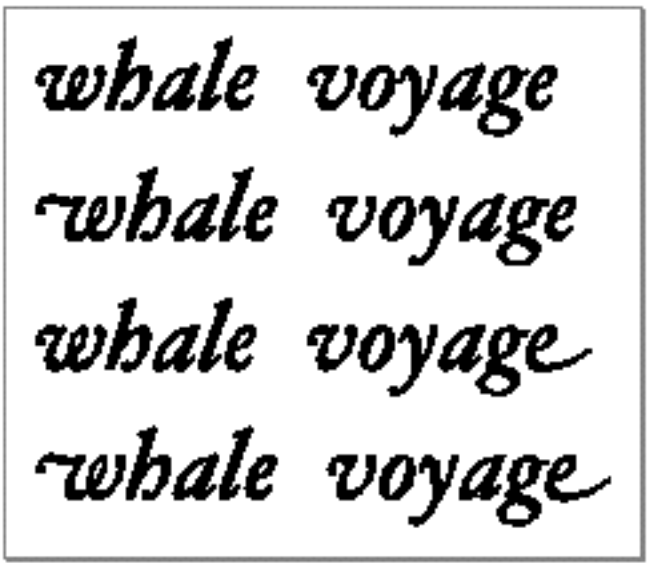
Collections of swash forms called **smart swashes** can be designated by the font designer and put in swash tables. Smart swashes are contextual and swashes are not. If the font supports the smart swashes feature type, you can select features that allow you to specify sets of swashes, such as shown in Table 8-8.

Table 8-8 Feature selectors for the `smartSwashType` feature type

Constant	Explanation
`wordInitialSwashesOnSelector` `wordInitialSwashesOffSelector`	Allows or prevents the substitution of swash variants that begin words.
`wordFinalSwashesOnSelector` `wordFinalSwashesOffSelector`	Allows or prevents the substitution of swash variants that end words.
`lineInitialSwashesOnSelector` `lineInitialSwashesOffSelector`	Allows or prevents the substitution of swash variants that begin lines.
`lineFinalSwashesOnSelector` `lineFinalSwashesOffSelector`	Allows or prevents the substitution of swash variants that end lines.
`nonFinalSwashesOnSelector` `nonFinalSwashesOffSelector`	Allows or prevents the substitution of swash variants that can occur at the beginnings or interiors of words

Figure 8-32 shows the same phrase written four times: first without swash variants, then with line initials, then with line finals, and finally with both line initials and line finals.

Figure 8-32 Specifying different swashes with feature selectors



The sample function that generated Figure 8-32 is shown in Listing 8-10 on page 8-55.

Note
If you want your application to define its own set of swashes, it can use glyph substitutions to replace the QuickDraw GX glyph choices with its own. See the section "Glyph Substitutions" beginning on page 8-18. u

Diacritical Marks

A glyph with a diacritical mark is a form of ligature. For fonts whose glyphs can take diacritical marks, QuickDraw GX allows you several display options. If the font supports the diacritical marks feature type, you can specify that QuickDraw GX should show, hide, or decompose diacritical marks, as shown in Table 8-9.

Table 8-9 Feature selectors for the `diacriticsType` feature type

Constant	Explanation
`showDiacriticsSelector`	Specifies that QuickDraw GX is to form accent ligatures on the glyphs they apply to.
`hideDiacriticsSelector`	Specifies that QuickDraw GX is not to form any accent ligatures.
`decomposeDiacriticsSelector`	Specifies that QuickDraw GX is to display marked glyphs as unmarked, followed by the accent ligatures as stand-alone glyphs.

For Roman fonts the default setting is to show diacritical marks. In text for scripts in which vowel marks are not normally shown, you can specify that marks be visible in certain instances, such as for children's text, or for pronunciation guides on rare words. Figure 8-33 shows an example of Hebrew text drawn with and without its diacritical marks.

Figure 8-33 Hebrew text with diacritical marks shown (upper) and hidden (lower)

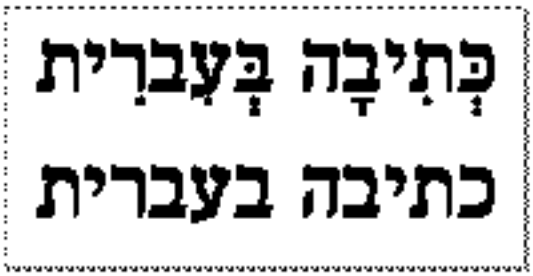
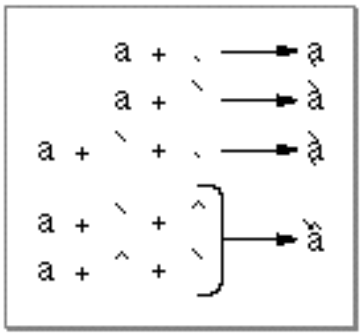


Figure 8-34 shows an example of text drawn with and without its accents.

Figure 8-34 Accented forms



Vertical Position

For fonts that support the vertical position feature type, you can select features that allow you to specify glyph variants related to vertical position, as shown in Table 8-10.

Table 8-10 Feature selectors for the `verticalPositionType` feature type

Constant	Explanation
`normalPositionSelector`	Specifies use of normally positioned glyph set.
`superiorsSelector`	Specifies use of superiors: glyph variants that are positioned above the baseline, used typically for superscripts.
`inferiorsSelector`	Specifies use of inferiors: glyph variants that are positioned below the baseline, used typically for subscripts.
`ordinalsSelector`	Specifies contextual substitution of glyphs that replace ordinal designations attached to numerals (such as "1st" substituting for "1st").

Fractions

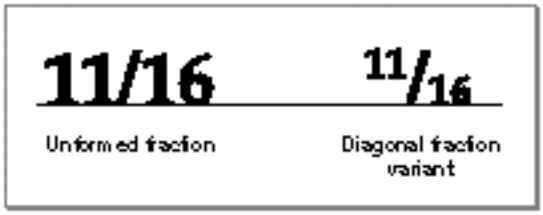
There are several ways to generate fractions with QuickDraw GX. For a font that supports the fractions feature type, you may be able to select between two different types of automatic fraction generation, as shown in Table 8-11.

Table 8-11 Feature selectors for the `fractionsType` feature type

Constant	Explanation
`noFractionsSelector`	Specifies no substitution or construction of fractions.
`verticalFractionsSelector`	Specifies replacement of slash-separated numeric sequences with predrawn fraction glyphs, if present in the font.
`diagonalFractionsSelector`	Specifies replacement of slash-separated numeric sequences with pre-drawn fraction glyphs, or else construction of fractions with numerators and denominators, or superiors and inferiors.

Figure 8-35 shows the same fraction, drawn first with `noFractionsSelector` and then with `diagonalFractionsSelector`.

Figure 8-35 Fractions



Note
To use the automatic fraction-generation capability, make sure that the slash separating the numerator and denominator is the fraction slash (character code 0xDA in the Standard Roman character set), not the normal slash character (0x2F). Automatic fraction generation does not occur unless the slash is a fraction slash. u

Prevention of Glyph Overlap

Some glyphs, especially certain initial swashes, have parts that extend well beyond their advance widths. An initial “Q”, for example, may have a tail that extends underneath the following “u”.

For fonts that support the glyph overlap feature type, you can specify that no glyph may overlap the outline of the following glyph. If it does, a non-overlapping form of the glyph is substituted. Table 8-12 lists the selectors for this feature.

Table 8-12 Feature selectors for the `overlappingCharactersType` feature type

Constant	Explanation
`preventOverlapOnSelector` `preventOverlapOffSelector`	Prevents or allows the collision of an extended part of one glyph with an adjacent glyph.

In the case of Figure 8-36, for example, preventing glyph overlap means that the script "Q" can remain because the following "u" has no descender to collide with it, whereas the script "L" is replaced with a simpler form to avoid collision with the "y".

Figure 8-36 Allowing and preventing glyph overlap

Noncontextual Font Features

Noncontextual font features include the selection of alternate glyph sets to give text a different appearance, and glyph substitution for purposes of mathematical typesetting or enhancing typographic sophistication.

Character Shape

The Chinese language can be represented with both a traditional and a simplified character set, as shown in Figure 8-37. Chinese fonts that support the character shape feature type allow you to select either set.

Figure 8-37 Traditional and simplified versions of a Chinese character

Note
Historically on the Macintosh, the difference has been handled by having separate script systems for traditional Chinese and simplified Chinese; while that is still the case, this font feature makes it possible to have both glyph repertoires present in a single font. u

Table 8-13 lists the selectors for this feature.

Table 8-13 Feature selectors for the `characterShapeType` feature type

Constant	Explanation
`traditionalCharactersSelector`	Specifies the use of traditional characters.
`simplifiedCharactersSelector`	Specifies the use of simplified characters.

Number Width

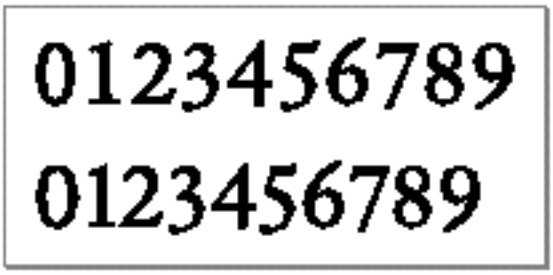
Many fonts support both proportional-width and fixed-width numerals, as shown in Figure 8-38. In proportional-width numerals the “1” is narrower than the “0”, whereas in fixed-width numerals they (and all the other numerals) have identical widths. Fixed-width numerals are also called *columnating* because they align well in text that consists of columns of numerical data. For fonts that support the number spacing feature type, you can select either fixed-width or proportional-width numerals. Table 8-14 lists the selectors for this feature.

Table 8-14 Feature selectors for the `numberSpacingType` feature type

Constant	Explanation
`monospacedNumbersSelector`	Specifies the use of fixed-width (columnating) numerals.
`proportionalNumbersSelector`	Specifies the use of proportional-width numerals.

Figure 8-38 shows both kinds of numerals.

Figure 8-38 Fixed-width and proportional-width numerals



Number Case

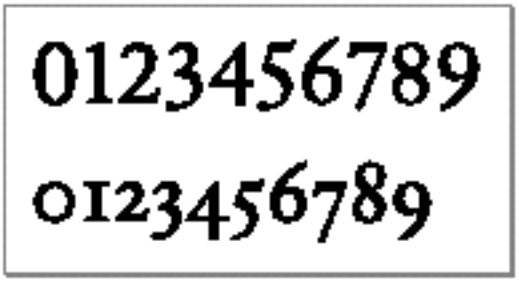
Some fonts support both lowercase (also called traditional or old-style) numerals, in which some glyphs extend below the baseline, and uppercase (also called *lining*) numerals, in which no glyphs extend below the baseline. For fonts that support the number case feature type, you can select either kind of numeral. Table 8-15 lists the selectors for this feature.

Table 8-15 Feature selectors for the `numberCaseType` feature type

Constant	Explanation
`lowerCaseNumbersSelector`	Specifies the use of lowercase (old-style) numerals.
`upperCaseNumbersSelector`	Specifies the use of uppercase (lining) numerals.

Figure 8-39 shows both kinds of numerals.

Figure 8-39 Uppercase and lowercase numerals



Style Options

A QuickDraw GX–compatible font may offer named sets of noncontextual glyph substitutions that give the text a specific style or appearance. You can select among sets, using selectors such as those listed in Table 8-16.

Table 8-16 Feature selectors for the `styleOptionsType` feature type

Constant	Explanation
`noStyleOptionsSelector`	Specifies the use of the standard glyph set.
`displayTextSelector`	Specifies the use of a glyph set that is designed for best display at large sizes (over 24 point).
`engravedTextSelector`	Specifies the use of a glyph set that has contrasting strokes parallel to the main stroke, giving an engraved effect.
`illuminatedCapsSelector`	Specifies the use of a glyph set with complex decoration surrounding the glyphs of capital letters.
`titlingCapsSelector`	Specifies the use of a glyph set in which capital letters have a special form for display in titles.
`tallCapsSelector`	Specifies the use of a glyph set in which capital letters have a taller form than is typical.

You may be able to select more than one feature at a time from the list of alternate forms. For example, a font may offer display, engraved, and engraved-display style options.

Typographic Extras

Fonts that support the typographic extras feature type allow you to specify certain small-scale typographic conventions, using selectors such as those shown in Table 8-17.

Table 8-17 Feature selectors for the `typographicExtrasType` feature type

Constant	Explanation
`hyphensToEmDashOnSelector` `hyphensToEmDashOffSelector`	Allows or prevents the automatic replacement of two adjacent hyphens with an em dash.
`hyphenToEnDashOnSelector` `hyphenToEnDashOffSelector`	Allows or prevents the automatic replacement of the sequence space-hyphen-space (or the hyphen in the sequence numeral-hyphen-numeral) with an en-dash.
`unslashedZeroOnSelector` `unslashedZeroOffSelector`	Allows or prevents the forced use of the unslashed zero glyph, regardless of whether the font specifies the slashed zero as the default.

continued

Table 8-17 Feature selectors for the `typographicExtrasType` feature type (continued)

Constant	Explanation
`formInterrobangOnSelector` `formInterrobangOffSelector`	Allows or prevents the automatic replacement of the sequence "?!" or "!?" with the font's interrobang glyph.
`smartQuotesOnSelector` `smartQuotesOffSelector`	Allows or prevents the automatic contextual replacement of straight quotation marks with curly ones.

Mathematical Extras

Fonts that support the mathematical extras feature type allow you to specify certain math-formatting conventions, using selectors such as those shown in Table 8-18.

Table 8-18 Feature selectors for the `mathematicalExtrasType` feature type

Constant	Explanation
`hyphenToMinusOnSelector` `hyphenToMinusOffSelector`	Allows or prevents the automatic replacement of the sequence space-hyphen-space (or the hyphen in the sequence numeral-hyphen-numeral) with a minus sign glyph (–).
`asteriskToMultiplyOnSelector` `asteriskToMultiplyOffSelector`	Allows or prevents the automatic replacement of the sequence space-asterisk-space (or the asterisk in the sequence numeral-asterisk-numeral) with a multiplication sign glyph (×).
`slashToDivideOnSelector` `slashToDivideOffSelector`	Allows or prevents the automatic replacement of the sequence space-slash-space (or the slash in the sequence numeral-slash-numeral) with a division sign glyph (÷).
`inequalityLigaturesOnSelector` `inequalityLigaturesOffSelector`	Allows or prevents the automatic replacement of sequences such as ">=" and "<=" with equivalent ligatures "≥" and "≤".
`exponentsOnSelector` `exponentsOffSelector`	Allows or prevents the automatic replacement of the sequence *exponentiation glyph*—numerals with the superior forms of the numerals. An example of an exponentiation glyph is "^".

Note

By convention, specifying the `hyphenToMinusOnSelector` in the mathematical extras feature type overrides specifying the `hyphenToEnDashOnSelector` in the typographic extras feature type. u

Ornament Sets

Fonts may include ornamental, nonalphabetic glyph sets used for various purposes. With a font that supports the ornament set feature type, you may be able to select among those glyph sets, using selectors such as those shown in Table 8-19.

Table 8-19 Feature selectors for the `ornamentSetsType` feature type

Constant	Explanation
`noOrnamentsSelector`	Specifies the use of no ornamental glyph sets.
`dingbatsSelector`	Specifies the use of dingbats: arrows, stars, bullets, and so on.
`piCharactersSelector`	Specifies the use of pi characters: related nonalphabetic symbols, such as musical notation glyphs.
`fleuronsSelector`	Specifies the use of fleurons: ornaments such as flowers, vines, and leaves.
`decorativeBordersSelector`	Specifies the use of decorative borders: glyphs used in interlocking patterns to form text borders.
`internationalSymbolsSelector`	Specifies the use of international symbols, such as the barred circle representing “no”.
`mathSymbolsSelector`	Specifies the use of mathematical symbols.

Figure 8-40 shows an example of glyphs from an ornamental set.

Figure 8-40 Ornamental glyphs

Character Alternates

This feature type gives a font a very general way to provide different sets of glyphs. Sets are numbered sequentially. For a font that supports the character alternates feature type, you can select by number any of the sets it provides.

For example, a font with 20 ampersands could place them in 20 selectors under this feature type. In general, however, named glyph sets provided through the `styleOptionsType` feature type are preferable. Table 8-20 lists the only defined selector for this feature.

Table 8-20 Feature selectors for the `characterAlternativesType` feature type

Constant	Explanation
`noAlternatesSelector`	Specifies the use of no character alternatives. This is the first (default) setting for this feature type; others are specified by number only.

Design Complexity

Some fonts may have several glyph sets that represent different designs from the same font-family, such as "plain" or "fancy." For a font that supports the design complexity feature type, design levels are numbered, and you can select any available level by number or by selectors such as those shown in Table 8-21.

Table 8-21 Feature selectors for the `designComplexityType` feature type

Constant	Explanation
`designLevel1Selector`	Specifies the basic glyph set.
`designLevel2Selector`	Specifies an alternate glyph set.
`designLevel3Selector`	Specifies an alternate glyph set.
`designLevel4Selector`	Specifies an alternate glyph set.
`designLevel5Selector`	Specifies an alternate glyph set.

Using Layout Styles

This section describes how to get special layout effects by manipulating these properties of the style object:

- run controls structure
- kerning adjustments array
- glyph substitutions array
- run features array

Not all style-object properties are demonstrated here. For examples of the use of decomposition adjustment factors, baseline-type specification, and direction overrides, see the chapter "Layout Line Control" in this book. For examples of the use of caret-angle speci-

fication and ligature splitting for caret positioning, see the chapter “Layout Carets, Highlighting, and Hit-Testing” in this book.

Initializing Style-Run Properties

Listing 8-1 is a sample library function that sets up a style object for use by a layout shape. It uses another library function, `SetStyleNamedFont`, and a library-defined data structure, `StyleRunOverrides`, that incorporates all override features.

Listing 8-1 Setting up a style object for a layout shape

```
void SetLayoutStyle(gxStyle s, char *gxfontName, Fixed textSize,
                    gxTextAttribute attr,
                    gxRunControls *runControls,
                    gxRunFeature runFeatures[],
                    long runFeaturesCount,
                    StyleRunOverrides *overrides)
{
   /* assign values to the style object that was passed in */
   SetStyleNamedFont(s, (unsigned char*)gxfontName);
   GXSetStyleTextSize(s, textSize);
   GXSetStyleTextAttributes(s, attr);

   /* if no run controls exist, assign them */
   if (runControls) GXSetStyleRunControls(s, runControls);

   /* if run features are passed in, assign them to the style */
   if (runFeatures) GXSetStyleRunFeatures(s, runFeaturesCount,
                                          runFeatures);

   /* assign all style-run overrides that have been passed in */
   if (overrides)
   {
      if (overrides->glyphSubstitutions)
         GXSetStyleRunGlyphSubstitutions(s,
                              overrides->glyphSubstitutionsCount,
                              overrides->glyphSubstitutions);
      if (overrides->kerningAdjustments)
         GXSetStyleRunKerningAdjustments(s,
                              overrides->kerningAdjustmentsCount,
                              overrides->kerningAdjustments);
```

```
      if (overrides->glyphJustOverrides)
         GXSetStyleRunGlyphJustOverrides(s,
                              overrides->glyphJustOverridesCount,
                              overrides->glyphJustOverrides);
      if (overrides->priorityJustOverride)
         GXSetStyleRunPriorityJustOverride(s,
                              overrides->priorityJustOverride);
   }
}
```

The `GXSetStyleRunControls` function is described on page 8-67. The `GXSetStyleRunFeatures` function is described on page 8-82. The `GXSetStyleRunGlyphSubstitutions` function is described on page 8-77. The `GXSetStyleRunKerningAdjustments` function is described on page 8-72. The `GXSetStyleRunGlyphJustOverrides` function and the `GXSetStyleRunPriorityJustOverride` function are described in the chapter "Layout Line Control" in this book.

Manipulating Run Controls

If a style run in your layout shape does not need to use run controls, it does not have to include a run controls structure at all. Having no run controls structure is equivalent to having one in which all values are set to 0.

If you do need to use run controls, then you should allocate a run controls structure, initialize all its values to 0, assign whatever individual nonzero values you need, and then attach it to a style object.

Using With-Stream and Cross-Stream Shift

You apply manual shifting to the glyphs of a style run by setting the `beforeWithStreamShift`, `afterWithStreamShift`, or `crossStreamShift` fields of the run controls structure for that style run. A value of 0 in any of the fields indicates that no shifting is to be performed in that direction.

Listing 8-2 is a partial listing of a sample routine that creates a line of text and displays it three times, with various values for with-stream and cross-stream shifting applied to one of its style runs.

The layout shape created in this routine is `layout`; the length of the text string `myString` is `len`; and the layout is first drawn at the location `myPoint`. This routine uses the library function `NewLayoutStyle` to create and initialize its style objects, which it stores in the array `styleList`. The run control's style-run lengths are contained in the array `runLengths`.

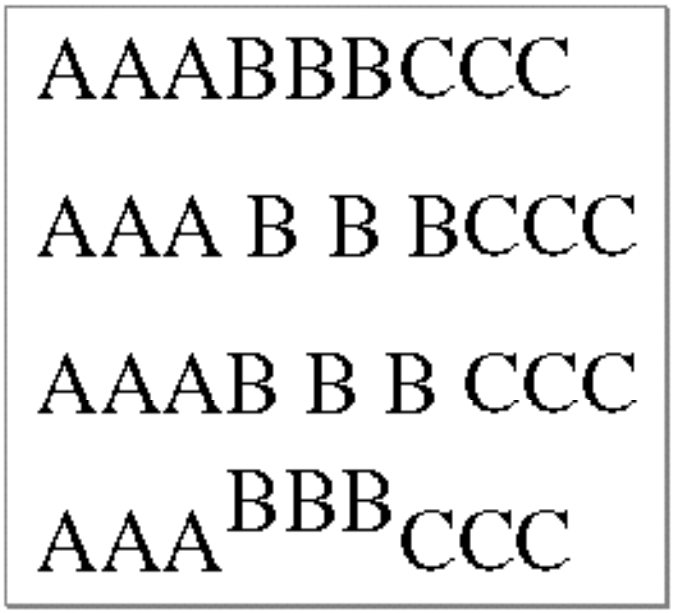
Listing 8-2 A sample that specifies with-stream and cross-stream shifting

```
void LetterSpacing(void)
{
   char           *myString = "AAABBBCCC";
   .
   .
   .
   /* set up the style runs and style objects for the shape */
   runLengths[0] = runLengths[1] = runLengths[2] = 3;
   regularStyle = NewLayoutStyle((char *) "\pTimes Roman",
                                 ff(36), 0, nil, nil, 0, nil);
   tweakedStyle = NewLayoutStyle((char *) "\pTimes Roman",
                                 ff(36), 0, nil, nil, 0, nil);

   styleList[0] = styleList[2] = regularStyle;
   styleList[1] = tweakedStyle;

   /* create the layout shape without run controls,and draw */
   layout = GXNewLayout(1, &len, (void *) &myString,
                        3, runLengths, styleList,
                        0, nil, nil,
                        nil, &myPoint);
   GXDrawShape(layout);

   /* give 2nd style run a left-side with-stream shift,and draw */
   InitializeRunControls(&runControls);
   runControls.beforeWithStreamShift = ff(15);
   GXSetStyleRunControls(tweakedStyle, &runControls);
   GXMoveShape(layout, 0, ff(75));
   GXDrawShape(layout);

   /* give the style run a right-side with-stream shift,and draw */
   runControls.beforeWithStreamShift = 0;
   runControls.afterWithStreamShift = ff(15);
   GXSetStyleRunControls(tweakedStyle, &runControls);
   GXMoveShape(layout, 0, ff(75));
   GXDrawShape(layout);

   /* give the style run a cross-stream shift,and draw */
   runControls.afterWithStreamShift = 0;
   runControls.crossStreamShift = ff(15);
   GXSetStyleRunControls(tweakedStyle, &runControls);
```

```
    GXMoveShape(layout, 0, ff(75));
    GXDrawShape(layout);
    .
    .
    .
}
```

Figure 8-41 shows the results of executing the code in Listing 8-2. The `GXSetStyleRunControls` function is described on page 8-67.

Figure 8-41 Result of with-stream and cross-stream shift applied to a style run



Specifying Tracking Values

You can specify a general “looseness” or “tightness” for the text of a style run by putting a value in the `track` field of the style object’s run controls structure. The actual amount of spreading or compression is controlled by the font, and can vary nonlinearly with point size.

Listing 8-3 is a sample routine that creates a line of text and displays it three times, with three different values for tracking. It defines and sets up variables in a similar manner to Listing 8-2 on page 8-43, so those parts of this routine are not repeated here. The style object used by the layout shape in this listing is `myStyle`; the length of the text string `myString` is `len`; and the layout is first drawn at the location `myPoint`.

Listing 8-3 Using track settings to spread or compress text

```
void Tracking(void)
{
   char  *myString = "Tracking can be loose or tight";
   .
   .
   .
   /* create and draw layout with default tracking value */
   layout = GXNewLayout(1, &len, (void *) &myString,
                        1, &len, &myStyle,
                        0, nil, nil, nil, &myPoint);
   GXDrawShape(layout);

   /* give it a track value of 2 (very loose),and redraw */
   InitializeRunControls(&runControls);
   runControls.track = ff(2);
   GXSetStyleRunControls(myStyle, &runControls);
   GXMoveShape(layout, 0, ff(50));
   GXDrawShape(layout);

   /* give it a track value of –2 (very tight), & redraw */
   runControls.track = -ff(2);
   GXSetStyleRunControls(myStyle, &runControls);
   GXMoveShape(layout, 0, ff(50));
   GXDrawShape(layout);
   .
   .
   .
}
```

Figure 8-10 on page 8-11 shows the results of executing the code in Listing 8-3. The `GXSetStyleRunControls` function is described on page 8-67.

Preventing Optical Alignment

QuickDraw GX automatically adjusts the positions of glyphs at line ends in order to improve the apparent alignment of columns and multiple lines of text. You can prevent that adjustment by setting the `gxNoOpticalAlignment` flag in the run controls structure of the style object for the style run at the end of the line.

Listing 8-4 is a partial listing of a sample routine that draws a line of fully justified text twice, once normally and once with optical alignment prevented. The layout shape created in this routine is `layout`; it uses the layout options structure `layoutOptions`

and the run controls structure `runControls`. The style object used by the layout shape is `myStyle`; the length of the text string `myString` is `len`; and the layout is first drawn at the location `myPoint`.

Listing 8-4 Preventing optical alignment

```
void OpticalAlignment(void)
{
   char     *myString = "OHIO";
   .
   .
   .
   /* set the width and justification of the layout shape */
   layoutOptions.width = ff(250);
   layoutOptions.just = fract1;

   /* draw lines at the margins, to better show the alignment */
   myLine.first.x = myLine.last.x = myPoint.x;
   myLine.first.y = 0;
   myLine.last.y = ff(1000);
   GXDrawLine(&myLine);
   myLine.first.x = myLine.last.x = myPoint.x+layoutOptions.width;
   GXDrawLine(&myLine);

   /* create and draw the layout shape, with default alignment */
   layout = GXNewLayout(1, &len, (void *) &myString,
                        1, &len, &myStyle, 0, nil, nil,
                        &layoutOptions, &myPoint);
   GXDrawShape(layout);

   /* prevent optical alignment, then redraw the shape */
   InitializeRunControls(&runControls);
   runControls.flags = gxNoOpticalAlignment;
   GXSetStyleRunControls(myStyle, &runControls);
   GXMoveShape(layout, 0, ff(85));
   GXDrawShape(layout);
   .
   .
   .
}
```

The `GXSetStyleRunControls` function is described on page 8-67.

Inhibiting Hanging Glyphs

Where alignment or justification of a line causes certain punctuation glyphs to be at a line margin, QuickDraw GX by default places those glyphs outside the margin, and does not account for their presence in calculating line length. Those glyphs are called *hanging glyphs*; each font defines the set of glyphs that are allowed to hang outside the margins.

You can partially or fully inhibit hanging behavior by placing a value in the `hangingInhibitFactor` field of the run controls structure of the style object for the style run at the end of the line. A value of 0 allows hanging to occur normally; a value of 1 completely prevents hanging. Values in between mean that a proportional fraction of the width of the hanging glyph is allowed to extend beyond the margin.

Listing 8-5 is a partial listing of a sample routine that draws a line of fully justified text three times, first with punctuation hanging normally, then with it partially inhibited, and finally with it fully inhibited.

The layout shape created in this routine is `layout`; it uses the layout options structure `layoutOptions` and the run controls structure `runControls`. The style object used by the layout shape is `myStyle`; the length of the text string `myString` is `len`; and the layout is first drawn at the location `myPoint`.

Listing 8-5 Inhibiting hanging punctuation

```
void HangingPunctuation(void)
{
   char     *myString = "“It is ‘a great effect!’”";
   .
   .
   .
   /* set the width and justification of the layout shape */
   InitializeLayoutOptions(&layoutOptions);
   layoutOptions.width = ff(360);
   layoutOptions.just = fract1;

   /* draw lines at the margins, to better show the hanging */
   myLine.first.x = myLine.last.x = myPoint.x;
   myLine.first.y = 0;
   myLine.last.y = ff(800);
   GXDrawLine(&myLine);
   myLine.first.x = myLine.last.x = myPoint.x+layoutOptions.width;
   GXDrawLine(&myLine);

   /* create and draw the layout shape, with normal hanging */
   layout = GXNewLayout(1, &len, (void *) &myString,
                        1, &len, &myStyle, 0, nil, nil,
                        &layoutOptions, &myPoint);
   GXDrawShape(layout);
```

```
   /* partially inhibit hanging, then redraw */
   InitializeRunControls(&runControls);
   runControls.hangingInhibitFactor = fract1 / 2;
   GXSetStyleRunControls(myStyle, &runControls);
   GXMoveShape(layout, 0, ff(54));
   GXDrawShape(layout);

   /* fully inhibit hanging, then redraw */
   runControls.hangingInhibitFactor = fract1;
   GXSetStyleRunControls(myStyle, &runControls);
   GXMoveShape(layout, 0, ff(54));
   GXDrawShape(layout);
   .
   .
   .
}
```

Figure 8-18 on page 8-15 shows the results of executing the code in Listing 8-5. The `GXSetStyleRunControls` function is described on page 8-67.

Imposing a Width on a Style Run

You can use imposed width to allow embedding of a picture or other graphic item within a line of text. Imposing a width on the glyphs of a style run forces them to each have a specific width, regardless of their font and point size. You would most typically use a style run consisting of a single whitespace glyph, and place an appropriate width value (in points) in the `imposedWidth` field of the run controls structure of the style object. Then you would set the `gxImposeWidth` bit in the flags field.

Listing 8-6 is a partial listing of a sample routine that draws a line of text containing three style runs. The middle run consists of a single whitespace glyph, but the run has an imposed width that forces it to take up a specific amount of space.

The layout shape created in this routine is `layout`; it uses the run controls structure `runControls`. The style objects used by the layout shape are `regularStyle` and `imposedStyle`, created with the library routine `NewLayoutStyle`. There are three style runs, whose lengths are specified in the `runLengths` array and whose style objects are specified in the `styleList` array. The length of the text string `myString` is `len`; and the layout is drawn at the location `myPoint`.

Listing 8-6 Creating a line containing a style run with an imposed width

```
void ImposedWidth(void)
{
   char  *myString = "As you wish";
   .
   .
   .
```

```
    /* set up style-run lengths, create “regular” style object */
    runLengths[0] = 2;
    runLengths[1] = 1;
    runLengths[2] = len - (runLengths[0] + runLengths[1]);
    regularStyle = NewLayoutStyle((char *) "\pTekton Plus Regular",
                            ff(36), 0, nil, nil, 0, nil);

    /* create “imposed” style object (without imposed width yet) */
    InitializeRunControls(&runControls);
    imposedStyle = NewLayoutStyle((char *) "\pTekton Plus Regular",
                            ff(36), 0, &runControls, nil, 0, nil);

    /* assign a style object to each style run */
    styleList[0] = styleList[2] = regularStyle;
    styleList[1] = imposedStyle;

    /* create and draw the layout the first time */
    layout = GXNewLayout(1, &len, (void *) &myString,
                         3, runLengths, styleList,
                         0, nil, nil,
                         nil, &myPoint);
    GXDrawShape(layout);

    /* impose a width on second style run, move & redraw layout */
    runControls.flags = gxImposeWidth;
    runControls.imposedWidth = ff(144);
    GXSetStyleRunControls(imposedStyle,&runControls);
    GXMoveShape(layout, ff(0), ff(60));
    GXDrawShape(layout);
    .
    .
    .
}
```

Figure 8-20 on page 8-16 shows the results of executing the code in Listing 8-6.

Using Kerning Adjustment Factors

You can adjust the kerning that occurs between specific pairs of glyphs in a style run by filling out kerning adjustment structures for those glyph pairs, and placing an array of such structures in the kerning adjustments array of the style object for that style run. Kerning adjustment involves both a scale factor and a point size factor, as discussed in the section “Kerning Adjustments” beginning on page 8-16.

Listing 8-7 is a partial listing of a sample routine that draws a line of text three times. The first time it draws the text normally; the second time it changes just the scale factor for

kerning between “A” and “W”; the third time it changes both the scale factor and the point size factor for that glyph pair.

The layout shape created in this routine is `layout`; it uses the kerning adjustment structure `kerningAdjustment`. The style object used by the layout shape is `myStyle`; the length of the text string `myString` is `len`; and the layout is first drawn at the location `myPoint`. The routine uses the functions `GXGetLayoutGlyphs` and `GXGetOffsetGlyphs` to determine the glyph codes for the glyph pair, and it uses the glyph codes array `glyphcodes`, as well as the parameters `offsetState`, `firstGlyph`, and `secondGlyph` to manipulate those glyph codes.

Listing 8-7 Adjusting the kerning amount for a pair of glyphs

```
void KerningAdjustments(void)
{
   char  *myString = "WAVE AWAY.";
   .
   .
   .
   /*create and draw the layout the first time */
   layout = GXNewLayout(1, &len, (void *) &myString,
                        1, &len, &myStyle,
                        0, nil, nil,
                        nil, &myPoint);
   GXDrawShape(layout);

   /*determine the glyph codes for the “A” and the “W” */
   GXGetLayoutGlyphs(layout, glyphcodes, nil, nil, nil, nil, nil, nil);
   GXGetOffsetGlyphs(layout, 1, false, &offsetState,
                     &firstGlyph, &secondGlyph);
   /*
      Place the glyph codes of the first and second glyphs of the layout
      shape into the kerning adjustment structure. (The ‘-1’s in the
      following lines convert from the 1-based space returned by
      GXGetOffsetGlyphs to the zero-based space needed for array references.)
   */
   kerningAdjustment.firstGlyph = glyphcodes[firstGlyph - 1];
   kerningAdjustment.secondGlyph = glyphcodes[secondGlyph - 1];

   /* first define a with-stream scale factor of –0.5, and nothing else */
   kerningAdjustment.crossStreamFactors.scaleFactor = 0;
   kerningAdjustment.crossStreamFactors.adjustmentPointSizeFactor = 0;
   kerningAdjustment.withStreamFactors.scaleFactor = -(fract1 / 2);
   kerningAdjustment.withStreamFactors.adjustmentPointSizeFactor = 0;
```

```
    /* assign the adjustment to the layout, and redraw */
    GXSetStyleRunKerningAdjustments(myStyle, 1, &kerningAdjustment);
    GXMoveShape(layout, 0, ff(75));
    GXDrawShape(layout);

    /* define a with-stream scale factor of –1, point-size factor of +0.5 */
    kerningAdjustment.withStreamFactors.scaleFactor = -fract1;
    kerningAdjustment.withStreamFactors.adjustmentPointSizeFactor = fixed1 /2;

    /* assign the new adjustment to the layout, and redraw */
    GXSetStyleRunKerningAdjustments(myStyle, 1, &kerningAdjustment);
    GXMoveShape(layout, 0, ff(75));
    GXDrawShape(layout);
    .
    .
    .
}
```

Figure 8-21 on page 8-17 shows the results of executing the code in Listing 8-7. The `GXSetStyleRunKerningAdjustments` function is described on page 8-72.

Substituting Glyphs

You can force QuickDraw GX to substitute any specific glyph for any other specific glyph in a style run when it draws a layout shape. The substitution occurs near the end of the layout process, after any automatic substitution QuickDraw GX may otherwise have performed (except for substitutions that may occur during postcompensation action). (For more information on postcompensation action, see the chapter “Layout Line Control” in this book.) You do this by specifying (by glyph code) one or more substitution pairs in the glyph substitutions array of the style object for that style run.

Listing 8-8 on page 8-52 is a partial listing of a sample routine that draws a line of text, and specifies that the glyph “æ” be replaced everywhere by the glyph “e”, and draws the line once again.

The layout shape created in this routine is `layout`; it uses the glyph substitution structure`glyphSubst`. The style object used by the layout shape is `myStyle`; the length of the text string `myString` is `len`; and the layout is first drawn at the location `myPoint`. The routine uses the function `GXGetLayoutGlyphs` to determine the glyph codes for the substitution pair, using the string `myTrialString` (of length `len0`) that contains the characters for the two glyphs; the routine places those glyph codes in the array `layoutGlyphs`. The trial string is used because the actual glyph code for the “æ” ligature cannot be assumed by the sample function. (Note that this function assumes that the font supports diphthong ligatures and that the font has an “æ” ligature).

Listing 8-8 Using glyph substitutions to replace one glyph with another

```
void GlyphSubstitutions(void)
{
   char                *myTrialString = "eae";
   char                *myString = "orthopaedic encyclopaedia";
   gxRunFeature        runFeature[1];
   short               len, len0;
   gxGlyphcode         layoutGlyphs[2];
   gxGlyphSubstitution glyphSubst;
   /* create a style object that specifies diphthong ligatures */
   runFeature[0].featureType = ligaturesType;
   runFeature[0].featureSelector = diphthongLigaturesOnSelector;
   myStyle = NewLayoutStyle((char*)
                       "\pTekton Plus Regular", ff(36),
                       0, nil, nil, 0, nil);
   GXSetStyleRunFeatures(myStyle, 1, runFeature);
   len0 = strlen(myTrialString);

   /* get substitution-pair glyph codes from trial string */
   layout = GXNewLayout(1, &len0, (void *) &myTrialString,
                       1, &len0, &myStyle,
                       0, nil, nil,
                       nil, nil);
   GXGetLayoutGlyphs(layout, layoutGlyphs,
                       nil, nil, nil, nil, nil, nil);
   glyphSubst.originalGlyph = layoutGlyphs[1];     /* the “æ” */
   glyphSubst.substituteGlyph = layoutGlyphs[0];   /* the “e” */
   GXDisposeShape(layout);
   len = strlen(myString);

   /* create and draw the layout shape without substitution */
   layout = GXNewLayout(1, &len, (void *) &myString,
                       1, &len, &myStyle,
                       0, nil, nil,
                       nil, &myPoint);
   GXDrawShape(layout);

   /* apply the substitution, then redraw the layout */
   GXSetStyleRunGlyphSubstitutions(myStyle, 1, &glyphSubst);
   GXMoveShape(layout, 0, ff(54));
   GXDrawShape(layout);
   .
   .
   .
}
```

Figure 8-22 on page 8-18 shows the results of executing the code in Listing 8-8. The `GXSetStyleRunGlyphSubstitutions` function is described on page 8-77.

Using Font Features

You can modify layout behavior by choosing font features from the set of features supported by a given font. You do that by setting up a run-features array and assigning it to a style object. The run-features array contains pairs of feature types and feature selectors; each pair specifies a given setting for a given feature type.

Feature types and feature selectors are defined in the feature registry. See the section "Font Features" beginning on page 8-18 for more information.

Specifying Levels of Ligature Formation

You can use the `ligaturesType` feature type to select whether or not to draw ligatures when drawing text. You specify this feature type with `ligaturesType` and the desired feature selector in a run-feature structure in the run-features array of the style object for that style run.

Listing 8-9 is a partial listing of a sample routine that draws a line of text three times: once with no ligatures, once with required and common ligatures, and once with required, common, and rare ligatures.

The layout shape created in this routine is `layout`; it uses the run-features array `runFeature`. The style object used by the layout shape is `myStyle`; the length of the text string `myString` is `len`; and the layout is first drawn at the location `myPoint`. The routine uses the function `GXSetStyleRunFeatures` to assign the run-features array to the style object.

Listing 8-9 Specifying three levels of ligature formation

```
void Ligatures(void)
{
   char  *myString = "The fifty bisected offices";
.
.
.
   /* set up run-features array by turning off all ligatures */
   runFeature[0].featureType = ligaturesType;
   runFeature[0].featureSelector = requiredLigaturesOffSelector;
   runFeature[1].featureType = ligaturesType;
   runFeature[1].featureSelector = commonLigaturesOffSelector;
   runFeature[2].featureType = ligaturesType;
   runFeature[2].featureSelector = rareLigaturesOffSelector;
```

```
    /* create and draw the layout with no ligatures */
    GXSetStyleRunFeatures(myStyle, 3, runFeature);
    layout = GXNewLayout(1, &len, (void *) &myString,
                         1, &len, &myStyle,
                         0, nil, nil,
                         nil, &myPoint);
    GXDrawShape(layout);

    /* add required and common ligatures; redraw the layout */
    runFeature[0].featureSelector = ligatureRequiredOnSelector;
    runFeature[1].featureSelector = ligatureCommonOnSelector;
    GXSetStyleRunFeatures(myStyle, 3, runFeature);
    GXMoveShape(layout, 0, ff(75));
    GXDrawShape(layout);

    /* add rare ligatures; redraw the layout */
    runFeature[2].featureSelector = ligatureRareOnSelector;
    GXSetStyleRunFeatures(myStyle, 3, runFeature);
    GXMoveShape(layout, 0, ff(75));
    GXDrawShape(layout);
    .
    .
    .
}
```

Figure 8-26 on page 8-25 shows the results of executing the code in Listing 8-9. The `GXSetStyleRunFeatures` function is described on page 8-82.

Specifying Different Types of Swashes

You can use the smart swash feature type to select whether or not to use swash variants of glyphs when drawing the text of a style run and to indicate which collections of swashes to include. You specify this feature type with `smartSwashType` and the desired feature selector in a run-feature structure in the run-features array of the style object for that style run.

Listing 8-10 is a sample routine that draws a line of text four times: once with no swashes, once with word-initial swashes only, once with word-final swashes only, and once with both word-initial and word-final swashes.

The layout shape created in this routine is `layout`; it uses the run-features array `runFeature`. The style object used by the layout shape is `myStyle`; the length of the text string `myString` is `len`; and the layout is first drawn at the location `myPoint`. The routine uses the function `GXSetStyleRunFeatures` to assign the run-features array to the style object.

Listing 8-10 Specifying three different types of swashes

```
void SmartSwashes(void)
{
   char  *myString = "whale voyage";
   .
   .
   .
   /* create the layout shape, turn off swashes, and draw */
   layout = GXNewLayout(1, &len, (void *) &myString,
                        1, &len, &myStyle,
                        0, nil, nil,
                        nil, &myPoint);

   runFeature[0].featureType = smartSwashType;
   runFeature[0].featureSelector = wordFinalSwashesOffSelector;
   runFeature[1].featureType = smartSwashType;
   runFeature[1].featureSelector = wordInitialSwashesOffSelector;
   GXSetStyleRunFeatures(myStyle, 2, runFeature);
   GXDrawShape(layout);

   /* turn word-initial swashes on, and redraw */
   runFeature[1].featureSelector = wordInitialSwashesOnSelector;
   GXSetStyleRunFeatures(myStyle, 2, runFeature);
   GXMoveShape(layout, 0, ff(54));
   GXDrawShape(layout);

   /* turn initials off and finals on, and redraw */
   runFeature[0].featureSelector = wordFinalSwashesOnSelector;
   runFeature[1].featureSelector = wordInitialSwashesOffSelector;
   GXSetStyleRunFeatures(myStyle, 2, runFeature);
   GXMoveShape(layout, 0, ff(54));
   GXDrawShape(layout);

   /* finally, turn both initials and finals on, and redraw */
   runFeature[1].featureSelector = wordInitialSwashesOnSelector;
   GXSetStyleRunFeatures(myStyle, 2, runFeature);
   GXMoveShape(layout, 0, ff(54));
   GXDrawShape(layout);
   .
   .
   .
}
```

Figure 8-32 on page 8-30 shows the results of executing the code in Listing 8-10. The `GXSetStyleRunFeatures` function is described on page 8-82.

Specifying Different Kinds of Case Substitution

You can use the letter case feature type to select among several types of noncontextual and contextual case substitutions in the text of a style run. You specify this feature type with `letterCaseType` and the desired feature selector in a run-feature structure in the run-features array of the style object for that style run.

Listing 8-11 is a partial listing of a sample routine that draws a line of text four times: once with no case substitution, once with all-caps substitution, once with all lowercase, and once with small caps substituted for lowercase glyphs.

The layout shape created in this routine is `layout`; it uses the run-features array `runFeature`. The style object used by the layout shape is `myStyle`; the length of the text string `myString` is `len`; and the layout is first drawn at the location `myPoint`. The routine uses the function `GXSetStyleRunFeatures` to assign the run-features array to the style object.

Listing 8-11 Specifying three different kinds of case substitution

```
void CaseSubstitution(void)
{
   char     *myString = "QuickDraw GX rules";
   .
   .
   .
   /* create and draw the layout, with no case substitution */
   layout = GXNewLayout(1, &len, (void *) &myString,
                        1, &len, &myStyle,
                        0, nil, nil,
                        nil, &myPoint);
   GXDrawShape(layout);

   /* specify all uppercase glyphs; redraw */
   runFeature[0].featureType = letterCaseType;
   runFeature[0].featureSelector = allCapsSelector;
   GXSetStyleRunFeatures(myStyle, 1, runFeature);
   GXMoveShape(layout, 0, ff(50));
   GXDrawShape(layout);

   /* specify all lowercase glyphs; redraw */
   runFeature[0].featureSelector = allLowerCaseSelector;
   GXSetStyleRunFeatures(myStyle, 1, runFeature);
   GXMoveShape(layout, 0, ff(50));
   GXDrawShape(layout);
```

```
/* specify initial caps followed by small caps; redraw */
runFeature[0].featureSelector = smallCapsSelector;
GXSetStyleRunFeatures(myStyle, 1, runFeature);
GXMoveShape(layout, 0, ff(50));
GXDrawShape(layout);
.
.
.
}
```

Figure 8-29 on page 8-27 shows the results of executing the code in Listing 8-11. The `GXSetStyleRunFeatures` function is described on page 8-82.

Layout Styles Reference

This section describes the functions that give you access to the layout-shape–specific properties of the style object, and the constants and data structures used by those functions and properties.

Constants and Data Types

This section describes the following data structures, and the data structures and constants associated with them:

- run controls structure
- kerning adjustment structure
- glyph substitution structure
- run-feature structure

Each of the structures corresponds in some way to a property of the style object used only by layout shapes.

Run Controls Structure

The run controls structure (type `gxRunControls`) is a property of every style object, but it is used only by layout shapes. In layout shapes, the run controls structure for each style run controls various features associated with text in that run. Your application can fill out this structure and assign it directly or indirectly to a style object with the `GXSetStyleRunControls` and `GXSetShapeRunControls` functions.

```
struct gxRunControls {
   gxRunControlFlags    flags;
   Fixed                beforeWithStreamShift;
   Fixed                afterWithStreamShift;
   Fixed                crossStreamShift;
   Fixed                imposedWidth;
   Fixed                track;
   fract                hangingInhibitFactor;
   fract                kerningInhibitFactor;
   Fixed                decompositionAdjustmentFactor;
   gxBaselineType       baselineType;
} ;
```

Field descriptions

`flags` The run control flags for this style run. See "Run Control Flags" on page 8-60 for a complete description of each bit flag in the `gxRunControlFlags` value.

`beforeWithStreamShift`
The amount of space (in points, 72 per inch) that QuickDraw GX should add to the left (or top, for vertical text) edge of all glyphs in the style run. Positive values move the glyphs farther apart; negative values move the glyphs closer together. See "With-Stream Shift and Cross-Stream Shift" beginning on page 8-6 for more information.

`afterWithStreamShift`
The amount of space (in points, 72 per inch) that QuickDraw GX should add to the right (or bottom, for vertical text) edge of all glyphs in the style run. Positive values move the glyphs farther apart; negative values move the glyphs closer together. See "With-Stream Shift and Cross-Stream Shift" beginning on page 8-6 for more information.

`crossStreamShift`
The distance (in points, 72 per inch) by which QuickDraw GX moves glyphs perpendicular to the text stream—that is, vertically for horizontal text and horizontally for vertical text. Cross-stream shift can be used to create superscripts and subscripts. Positive values shift horizontal text upward and vertical text to the right; negative values shift downward or to the left. Each glyph in the style run is shifted by the same amount from the baseline. See "With-Stream Shift and Cross-Stream Shift" beginning on page 8-6 for more information.

`imposedWidth` The width (in points, 72 per inch) to be imposed on each glyph in this style run, if the `gxImposeWidth` flag is set in the `flags` field of this structure. This width is used regardless of the contents or other settings for the style run. You can use this feature with a single whitespace glyph when you want to have a gap of fixed width in the line (for example, to place a graphic on the line). See "Imposed Width" beginning on page 8-15 for more information.

`track` The number that controls tracking, a set of font-defined adjustments to interglyph positions. QuickDraw GX automatically uses tracking information provided by the font; if you do not want tracking to occur, specify the value `gxNoTracking` for this field. Specify normal tracking with a value of 0; specify positive numbers for looser tracking, and negative numbers for tighter tracking. For more information on tracking, see "Tracking" beginning on page 8-10.

`hangingInhibitFactor`
A value between 0 and 1 specifying the degree to which hanging punctuation glyphs in this style run extend beyond the text margin. A value of 0 (the default) indicates that hanging punctuation glyphs should hang by the normal amount. A positive nonzero value lessens the amount of hanging proportionally; a value of 1 means "no hanging at all." See "Hanging Glyphs" beginning on page 8-14 for more information.

`kerningInhibitFactor`
A value between 0 and 1 specifying the relative proportion of the font-specified kerning that is applied to this style run. A value of zero means "kern normally." A positive nonzero value proportionally lessens the amount of kerning; a value of 1 means "no kerning." See "With-Stream Kerning and Cross-Stream Kerning" beginning on page 8-8 for more information.

`decompositionAdjustmentFactor`
Additional control over the font-specified threshold at which ligature decomposition occurs during justification. Values between –1.0 and 1.0 are interpreted as a fractional adjustment. A value of 0 means no adjustment, 0.5 means to add an additional 50 percent to the font-specified threshold, –0.25 means to subtract 25 percent from that threshold, and so on. Values less than –1.0 are meaningless. For more information, see the discussion of ligature decomposition and justification in the chapter "Layout Line Control" in this book.

`baselineType` The baseline type to be assigned to this style run. Possible values for this field are given in the discussion of baseline types in the chapter "Layout Line Control" in this book. If you want the default behavior (derived from font-specified characteristics), use the value `gxNoOverrideBaseline`. A value of 0 specifies the Roman baseline.

The `GXSetStyleRunControls` function is described on page 8-67; the `GXSetShapeRunControls` function is described on page 8-69. To obtain the run control values for a style run, use the `GXGetStyleRunControls` function, described on page 8-66, or the `GXGetShapeRunControls` function, described on page 8-68.

Run Control Flags

The run control flags are bit flags that make up a value in the `flags` field of the run controls structure of the style object for each style run in a layout shape. The run control flags affect the behavior of various parts of the layout process.

QuickDraw GX provides constants for all defined flag values. Any of the flag constants may be combined to form a single value for the `flags` field. Note that most of the run control flags have "No" as part of their name, such as `gxNoOpticalAlignment`. What this means is that the default QuickDraw GX behavior is to optically align glyphs, and only by setting this flag can you prevent that behavior from occurring.

```
#define gxNoLigatureSplits        0x80000000
#define gxNoCaretAngle            0x40000000
#define gxImposeWidth             0x20000000
#define gxNoCrossKerning          0x10000000
#define gxNoOpticalAlignment      0x08000000
#define gxForceHanging            0x04000000
#define gxNoSpecialJustification  0x02000000
#define gxDirectionOverrideMask   0x00000003

typedef unsigned long gxRunControlFlags;
```

Flag descriptions

`gxNoLigatureSplits`
: Tells QuickDraw GX whether to treat ligatures as indivisible objects for caret positioning. The default is `false`, which means that the caret can occupy intermediate positions within the ligature that correspond to the boundaries of the individual characters that make up the ligature. If `gxNoLigatureSplits` is `true` and the caret position is adjacent to a ligature, QuickDraw GX considers the next valid caret position to be across the entire ligature rather than at any point within it. For more information, see the discussion of ligature splits and carets in the chapter "Layout Carets, Highlighting, and Hit-Testing" in this book.

`gxNoCaretAngle`
: Tells QuickDraw GX whether to make all carets perpendicular to the baseline, regardless of the slant of the text's glyphs. If set, this flag overrides the value of the `highlightType` parameter in functions such as `GXGetLayoutCaret` and `GXGetLayoutHighlight`. If this flag is not set, the angle of the caret (and the edges of highlighting areas) depends on the intrinsic angle of the font's glyphs and whether the highlight type is `gxHighlightAverageAngle` or `gxHighlightStraight`. For more information, see the discussion of caret angle in the chapter "Layout Carets, Highlighting, and Hit-Testing" in this book.

`gxImposeWidth`
: Tells QuickDraw GX whether to give each glyph in this style run a certain amount of space, regardless of the textual content or other layout effects. A style run with a single space glyph and with this bit set to `true` can make a gap in the line so that it can contain a

picture. If this flag is set to `true`, there should be a positive width value in the `imposedWidth` field of the run controls structure. See "Imposed Width" beginning on page 8-15 for more information.

`gxNoCrossKerning`
Tells QuickDraw GX whether to suppress automatic cross-stream kerning for this style run. This flag has no effect on manual cross-stream shifting specified in the `crossStreamShift` field in the run controls structure. If you do not set this flag, QuickDraw GX performs any normal automatic cross-stream kerning specified by the font. See "With-Stream Kerning and Cross-Stream Kerning" beginning on page 8-8 for more information.

`gxNoOpticalAlignment`
Tells QuickDraw GX whether to suppress normal line-edge optical alignment for this style run. Optical alignment gives a more correct visual appearance of the edges of a line of text, as aligned with other lines of text surrounding it; for examples, see "Optical Alignment" beginning on page 8-11. If you do not set this flag, QuickDraw GX performs optical alignment automatically.

`gxForceHanging` Tells QuickDraw GX whether to designate all glyphs in the style run as hanging punctuation characters, even if they wouldn't normally be. See "Hanging Glyphs" beginning on page 8-14 for more information.

`gxNoSpecialJustification`
Tells QuickDraw GX whether to turn off any special justification processes, known as **postcompensation action,** that are applied after glyph positioning. Examples of a postcompensation actions are ligature decomposition, addition of kashidas, and stretching of glyphs. For more information, see the discussion of postcompensation action and justification in the chapter "Layout Line Control" in this book.

`gxDirectionOverrideMask`
Two bits containing the direction override value for the style run. Nonzero values for these bits impose a direction onto all glyphs in this style run, overriding the fundamental glyph directions specified in the style run's font. Constants for the `gxDirectionOverrideMask` bits are defined in the `gxDirectionOverride` enumeration; see the next section, "Direction Overrides."

You can ensure that you are assigning a valid value to the run control flags by performing an AND operation on the value `gxAllRunControlFlags` and your value before assigning it to the `flags` field:

```
#define gxAllRunControlFlags (gxNoLigatureSplits|gxNoCaretAngle|
                              gxImposeWidth|gxNoCrossKerning|
                              gxNoOpticalAlignment|gxForceHanging|
                              gxNoSpecialJustification|
                              gxDirectionOverrideMask)
```

Direction Overrides

In general, your application can let QuickDraw GX determine the proper direction for any text within a style run. QuickDraw GX orders sequences of glyphs for display based on glyph direction as specified in the font. However, you can override the directional behavior of glyphs, on a style-run basis, for special effects. You specify the overriding direction in the `gxDirectionOverrideMask` flag in the `flags` field of the style object's run controls structure.

The `gxDirectionOverrides` enumeration provides constants for the defined values of the `gxDirectionOverrideMask` flag:

```
enum gxDirectionOverrides {
   gxNoDirectionOverride   = 0,
   gxImposeLeftToRight     = 1,
   gxImposeRightToLeft     = 2,
   gxImposeArabic          = 3
};
typedef unsigned short gxDirectionOverride;
```

Constant descriptions

`gxNoDirectionOverride`
: Instructs QuickDraw GX to use the normal direction of the text in the style run.

`gxImposeLeftToRight`
: Instructs QuickDraw GX to force the text to be treated as left-to-right.

`gxImposeRightToLeft`
: Instructs QuickDraw GX to force the text to be treated as right to left.

`gxImposeArabic`
: Instructs QuickDraw GX to force the text to be treated as Arabic letters. Numbers interacting with `gxImposeArabic` behave slightly differently from numbers interacting with letters in other right-to-left scripts, such as Hebrew. See the discussion of the Unicode reordering model in *The Unicode Standard: Worldwide Character Encoding, Version 1.0,* Volume 1, for more information.

The purpose of a direction override is to permit applications to perform special rendering effects, such as drawing Roman text right-to-left. It is not intended to control text direction in general or the placement of blocks of text with respect to one another.

In general, the font-specified glyph direction controls text direction for individual sequences of glyphs, and nested direction levels in the levels array of the layout shape control the relative placement of the glyph sequences. A direction override overrides glyph directions only, and affects an entire style run at a time. Glyph directions, direction levels, and the levels array are described in the chapter "Layout Line Control" in this book.

Kerning Adjustment Factors Structure

The kerning adjustment factors structure (type `gxKerningAdjustmentFactors`) specifies the amount of an adjustment to automatic kerning. It is used in the `withStreamFactors` and `crossStreamFactors` fields of the kerning adjustment structure.

```
struct gxKerningAdjustmentFactors{
   fract       scaleFactor;
   Fixed       adjustmentPointSizeFactor;
};
```

Field descriptions

`scaleFactor` The scale factor. The font-specified automatic kerning value is multiplied by this factor.

`adjustmentPointSizeFactor` The point-size adjustment. This factor is multiplied by the current point size and then added to the kerning value. This value can be either positive or negative.

The total kerning adjustment, therefore, is

ax + b

where *x* is the automatic kerning value as specified in the font, *a* is `scaleFactor`, and *b* is `adjustmentPointSizeFactor` multiplied by the style run's point size.

For more discussion of the kerning adjustment formula, see "Kerning Adjustments" beginning on page 8-16. For examples of the use of these factors, see "Using Kerning Adjustment Factors" beginning on page 8-49.

Kerning Adjustment Structure

If you want to provide alterations to the kerning values that would otherwise be automatically used in a run of text, use the kerning adjustment structure (type `gxKerningAdjustment`). The kerning adjustment structure modifies the kerning for an individual pair of glyphs.

```
struct gxKerningAdjustment {
   gxGlyphcode                          firstGlyph;
   gxGlyphcode                          secondGlyph;
   struct gxKerningAdjustmentFactors    withStreamFactors;
   struct gxKerningAdjustmentFactors    crossStreamFactors;
} ;
```

Field descriptions

`firstGlyph` The glyph code of the first glyph of the kerning pair.

`secondGlyph` The glyph code of the second glyph of the kerning pair.

`withStreamFactors`
Withstream adjustments to the kerning.

`crossStreamFactors`
Cross-stream adjustments to the kerning.

You can assign an array of kerning adjustment structures to a style object using the `GXSetStyleRunKerningAdjustments` function or the `GXSetShapeRunKerningAdjustments` function. If the specified pair already kerns (based on data in the font's kerning table), the specified adjustments are added to it.

A value of `gxResetCrossStreamFactor` in the `adjustmentPointSizeFactor` field of `crossStreamFactors` resets the cross-stream kerning to the baseline:

```
#define gxResetCrossStreamFactor gxNegativeInfinity
```

The `GXGetStyleRunKerningAdjustments` function is described on page 8-70. The `GXSetStyleRunKerningAdjustments` function is described on page 8-72. The `GXSetShapeRunKerningAdjustments` function is described on page 8-74.

Glyph Substitution Structure

Sometimes the glyph substitutions QuickDraw GX automatically performs on a layout shape may not be appropriate for your needs. You can use the glyph substitution structure (type `gxGlyphSubstitution`) to have final control over which glyphs appear in the line. Substitutions specified with this structure always occur after all other substitutions except for postcompensation action, so your application has the final say.

```
struct gxGlyphSubstitution {
   gxGlyphcode    originalGlyph;
   gxGlyphcode    substituteGlyph;
};
```

Field descriptions

`originalGlyph` The original glyph. This is the glyph that would result from the layout process, in the absence of glyph substitution.

`substituteGlyph`
The glyph QuickDraw GX is to substitute for the original glyph.

In a given style run, your application can use the glyph substitution structure to specify that, any time a particular glyph would appear, QuickDraw GX substitutes a different glyph for it. You do this by supplying an array of glyph substitution structures to the `GXSetStyleRunGlyphSubstitutions` function or the `GXSetShapeRunGlyphSubstitutions` function.

The `GXGetStyleRunGlyphSubstitutions` function is described on page 8-75. The `GXSetStyleRunGlyphSubstitutions` function is described on page 8-77. The `GXGetShapeRunGlyphSubstitutions` function is described on page 8-78. The `GXSetShapeRunGlyphSubstitutions` function is described on page 8-79.

Run-Feature Structure

You can use the run-feature structure (type `gxRunFeature`) to specify the degree to which a particular font feature is used within a style run.

```
struct gxRunFeature {
   gxRunFeatureType     featureType;
   gxRunFeatureSelector featureSelector;
};
```

Field descriptions

`featureType` The type of font feature to affect.

`featureSelector` The setting or selection for that feature type.

Font feature types include such categories as ligature formation and number style. Feature selectors within those types include settings such as “do not use rare ligatures,” and “use proportional-width numbers.” The `gxRunFeatureType` and `gxRunFeatureSelector` types are defined as follows:

```
typedef unsigned short gxRunFeatureType;

typedef unsigned short gxRunFeatureSelector;
```

In a given style run, your application uses the run-feature structure to specify both the type of font feature to employ and the level or style of employment (perhaps including suppressing it entirely). You do this by supplying an array of run-feature structures to the `GXSetStyleRunFeatures` function or the `GXSetShapeRunFeatures` function.

The `GXGetStyleRunFeatures` function is described on page 8-80. The `GXSetStyleRunFeatures` function is described on page 8-82. The `GXGetShapeRunFeatures` function is described on page 8-83. The `GXSetShapeRunFeatures` function is described on page 8-84. Constants for some of the defined values for the `featureType` and `featureSelector` fields are listed in the section “Font Features” beginning on page 8-18.

Note

Constants for all supported feature types and feature selectors, along with descriptions of the features, are found in the *QuickDraw GX Font Feature Registry*. Please contact Apple Computer, Inc., at AppleLink address FONTREGISTRY, for the most recent version of the feature registry. u

Functions

This section describes the functions that give you access to the layout-shape–specific properties of the style object by using these functions, and by specifying either a style object or a layout shape object, you can get or set

- the run controls structure
- the kerning adjustments array
- the glyph substitutions array
- the run-features array

Getting and Setting Run Controls

The functions in this section allow you to get or set the run controls property of a specified style object or of the style object associated with a specified layout shape.

GXGetStyleRunControls

You can use the `GXGetStyleRunControls` function to retrieve the run controls property from a style object.

```
long GXGetStyleRunControls(gxStyle source,
                           gxRunControls *runControls);
```

`source` A reference to the style object whose run controls you need.

`runControls`
A pointer to a run controls structure. On return, the structure contains the run controls information for the style object referenced in the `source` parameter. If the style object has no run controls structure, the structure pointed to by this parameter is not modified.

function result The number of run controls structures for this style object. This value can be 0 or 1 only, because a style object can have a maximum of one run controls structure. If the style object does not have one, the function result is 0.

DESCRIPTION

The `GXGetStyleRunControls` function retrieves the run controls structure, if any, from the source style object. If a style object has no run controls structure, which is the default state, its behavior is as if it had a run controls structure with values of 0 or `nil` for all fields and flags (except that `baselineType` = `gxRomanBaseline = 0`).

ERRORS, WARNINGS, AND NOTICES

Errors
`style_is_nil`

SEE ALSO

You assign a run controls structure to a style object using the `GXSetStyleRunControls` function, described next. You retrieve the run controls property from the style object associated with a shape object using the `GXGetShapeRunControls` function, described on page 8-68.

The run controls structure is described on page 8-57.

GXSetStyleRunControls

You can use the `GXSetStyleRunControls` function to assign values to the run controls property of a style object.

```
void GXSetStyleRunControls(gxStyle target,
                           const gxRunControls *runControls);
```

`target` A reference to the style object whose run controls you are assigning.

`runControls`
A pointer to a run controls structure containing the run controls you want to assign to the style object referenced in the `target` parameter. If you pass `nil` for this parameter, `GXSetStyleRunControls` removes all run controls information from the style object.

DESCRIPTION

The `GXSetStyleRunControls` function assigns the specified run controls structure to the target style object, replacing any existing run controls structure in the style.

If the run controls structure contains illegal values, such as an imposed width less than 0 or a baseline type beyond the defined range, `GXSetStyleRunControls` posts a `parameter_out_of_range` error.

You can ensure that you are assigning a valid value to the run control flags by performing an AND operation on the value `gxAllRunControlFlags` and your value before assigning it to the `flags` field of the run controls structure.

ERRORS, WARNINGS, AND NOTICES

Errors
`style_is_nil`
`parameter_out_of_range`

Notices (debugging version)
`attributes_already_set`

SEE ALSO

You retrieve the run controls property from a style object using the `GXGetStyleRunControls` function, described in the previous section. You assign a run controls structure to the style object associated with a shape object using the `GXSetShapeRunControls` function, described on page 8-69.

The run controls structure is described on page 8-57.

For an example of the use of this function, see Listing 8-2 on page 8-43. Other examples occur in Listing 8-3 through Listing 8-5.

GXGetShapeRunControls

You can use the `GXGetShapeRunControls` function to retrieve the run controls property from the style object associated with a shape.

```
long GXGetShapeRunControls(gxShape source,
                           gxRunControls *runControls);
```

`source` A reference to the shape object whose associated style object contains the run controls information you need.

`runControls`
A pointer to a run controls structure. On return, the structure contains the run controls information for the style object associated with the shape object referenced in the `source` parameter. If the style object has no run controls structure, the structure pointed to by this parameter is not modified.

function result The number of run controls structures for the style object associated with this shape object. This value can be 0 or 1 only, because a style object can have a maximum of one run controls structure. If the style object does not have one, the function result is 0.

DESCRIPTION

The `GXGetShapeRunControls` function retrieves the run controls structure from the style object associated with the source shape object. If a style object has no run controls structure, which is the default state, it behaves as if it had a run controls structure with values of 0 or `nil` for all fields and flags.

Calling this function for the layout shape `myLayout` is equivalent to making the following call:

```
GXGetStyleRunControls(GXGetShapeStyle(myLayout), &myRunControls);
```

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the `GXNewLayout` function, for example,

you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as `GXGetStyleRunControls`.

ERRORS, WARNINGS, AND NOTICES

Errors
`shape_is_nil`
`parameter_is_nil`

SEE ALSO

You assign a run controls structure to the style object associated with a shape object using the `GXSetShapeRunControls` function, described next. You retrieve the run controls property directly from a style object using the `GXGetStyleRunControls` function, described on page 8-66.

The run controls structure is described on page 8-57.

GXSetShapeRunControls

You can use the `GXSetShapeRunControls` function to assign values to the run controls property of a style object associated with a shape.

```
void GXSetShapeRunControls(gxShape target,
                           const gxRunControls *runControls);
```

`target` A reference to the shape object whose associated style object is to be assigned the run controls information.

`runControls`
A pointer to a run controls structure that contains the run controls you want to assign to the style object associated with the shape object referenced in the `target` parameter. If you specify `nil` for this parameter, `GXSetShapeRunControls` removes all run controls information from the style object.

DESCRIPTION

`GXSetShapeRunControls` assigns the specified run controls structure to the style object associated with the target shape, replacing any existing run controls structure in the style.

Calling this function for the layout shape `myLayout` is equivalent to making the following call:

```
GXSetStyleRunControls(GXGetShapeStyle(myLayout), &myRunControls);
```

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the `GXNewLayout` function, for example, you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as `GXSetStyleRunControls`.

If the run controls structure contains illegal values, such as an imposed width less than 0 or a baseline type beyond the defined range, `GXSetShapeRunControls` posts a `parameter_out_of_range` error.

You can ensure that you are assigning a valid value to the run control flags by performing an AND operation on the value `gxAllRunControlFlags` and your value before assigning it to the `flags` field of the run controls structure.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`parameter_out_of_range`

Notices (debugging version)

`attributes_already_set`

SEE ALSO

You retrieve the run controls property from the style object associated with a shape object using the `GXGetShapeRunControls` function, described in the previous section. You assign a run controls structure directly to a style object using the `GXSetStyleRunControls` function, described on page 8-67.

The run controls structure is described on page 8-57.

Customizing Kerning

The functions in this section allow you to get or set the kerning adjustments property of a specified style object or of the style object associated with a specified layout shape.

GXGetStyleRunKerningAdjustments

You can use the `GXGetStyleRunKerningAdjustments` function to retrieve the array of kerning adjustment structures from a style object.

```
long GXGetStyleRunKerningAdjustments(gxStyle source,
                    gxKerningAdjustment kerningAdjustments[]);
```

`source` A reference to the style object whose kerning adjustments array you need.

`kerningAdjustments`
An array of kerning adjustment structures. On return, the array contains the kerning adjustments for the style object referenced in the `source` parameter. If you specify `nil` for this parameter, no information is returned in it; however, the function result is still the correct number of kerning adjustment structures for the style.

function result The number of kerning adjustment structures in the style object. If the style object contains no kerning adjustment structures, the function returns 0.

DESCRIPTION

The `GXGetStyleRunKerningAdjustments` function retrieves the kerning adjustments array, if any, from the source style object. If the style has no kerning adjustment structures, QuickDraw GX uses only font-specified kerning behavior when drawing.

To get the kerning adjustments themselves, you need to allocate an array to pass in the `kerningAdjustments` parameter when calling this function. To get the right size for the array, you can first call the function with a value of `nil` for the `kerningAdjustments` parameter. Then use the function result to allocate an array of the proper size, and call `GXGetStyleRunKerningAdjustments` a second time, this time passing the array.

SPECIAL CONSIDERATIONS

Because QuickDraw GX can reorder the elements in a style object's kerning adjustments array, the order of elements returned in the `kerningAdjustments` parameter to this function may differ from the order in which they were originally assigned to the style.

ERRORS, WARNINGS, AND NOTICES

Errors
`style_is_nil`

SEE ALSO

You assign a kerning adjustments array to a style object using the `GXSetStyleRunKerningAdjustments` function, described next. You retrieve the kerning adjustments array from the style object associated with a shape object using the `GXGetShapeRunKerningAdjustments` function, described on page 8-73.

The kerning adjustment structure is described on page 8-63.

GXSetStyleRunKerningAdjustments

You can use the `GXSetStyleRunKerningAdjustments` function to assign an array of kerning adjustment structures to a style object.

```
void GXSetStyleRunKerningAdjustments(gxStyle target, long count,
                  const gxKerningAdjustment kerningAdjustments[]);
```

`target` A reference to the style object whose kerning adjustments array you are assigning.

`count` The number of kerning adjustment structures to assign; the number of elements in the kerning adjustments array.

`kerningAdjustments`
The array of kerning adjustment structures to assign to the style object. If you specify `nil` for this parameter and 0 for the `count` parameter, the function removes all kerning adjustment structures from the style object.

DESCRIPTION

The `GXSetStyleRunKerningAdjustments` function assigns the specified array of kerning adjustment structures to the target style object.

If `count` is 0 and `kerningAdjustments` is non-`nil`, or if `count` is nonzero and `kerningAdjustments` is `nil`, `GXSetStyleRunKerningAdjustments` posts an `inconsistent_parameters` error.

ERRORS, WARNINGS, AND NOTICES

Errors

```
style_is_nil
count_is_less_than_zero
inconsistent_parameters
```

Notices (debugging version)

```
attributes_already_set
```

SEE ALSO

You retrieve the kerning adjustments array from a style object using the `GXGetStyleRunKerningAdjustments` function, described in the previous section. You assign a kerning adjustments array to the style object associated with a shape object using the `GXSetShapeRunKerningAdjustments` function, described on page 8-74.

The kerning adjustment structure is described on page 8-63.

For an example of the use of this function, see Listing 8-7 on page 8-50.

GXGetShapeRunKerningAdjustments

You can use the GXGetShapeRunKerningAdjustments function to retrieve the array of kerning adjustment structures from the style object associated with a shape.

```
long GXGetShapeRunKerningAdjustments(gxShape source,
                        gxKerningAdjustment kerningAdjustments[]);
```

`source` A reference to the shape object whose associated style object contains the kerning adjustments array you need.

`kerningAdjustments`
An array of kerning adjustment structures. On return, the array contains the kerning adjustments for the style object associated with the shape referenced in the `source` parameter. If you specify `nil` for this parameter, no information is returned in it; however, the function result is still the correct number of kerning adjustment structures for the style.

function result The number of kerning adjustment structures in the style object associated with the shape. If the style object contains no kerning adjustment structures, the function returns 0.

DESCRIPTION

The GXGetShapeRunKerningAdjustments function retrieves the kerning adjustments array, if any, from the style object associated with the source shape. If the style has no kerning adjustment structures, QuickDraw GX uses only font-specified kerning behavior when drawing.

Calling this function for the layout shape `myLayout` is equivalent to making the following call:

```
myCount = GXGetStyleRunKerningAdjustments(
                    GXGetShapeStyle(myLayout), myAdjustsArray);
```

To get the kerning adjustments themselves, you need to allocate an array to pass in the `kerningAdjustments` parameter when calling this function. To get the right size for the array, you can first call the function with a value of `nil` for the `kerningAdjustments` parameter. Then use the function result to allocate an array of the proper size, and call GXGetShapeRunKerningAdjustments a second time, this time passing the array.

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the GXNewLayout function, for example, you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as GXGetStyleRunKerningAdjustments.

SPECIAL CONSIDERATIONS

Because QuickDraw GX can reorder the elements in a style object's kerning adjustments array, the order of elements returned in the `kerningAdjustments` parameter to this function may differ from the order in which they were originally assigned to the style.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`

SEE ALSO

You assign a kerning adjustments array to the style object associated with a shape object using the `GXSetShapeRunKerningAdjustments` function, described next. You retrieve the kerning adjustments array directly from a style object using the `GXGetStyleRunKerningAdjustments` function, described on page 8-70.

The kerning adjustment structure is described on page 8-63.

GXSetShapeRunKerningAdjustments

You can use the `GXSetShapeRunKerningAdjustments` function to assign an array of kerning adjustment structures to the style object associated with a shape.

```
void GXSetShapeRunKerningAdjustments(gxShape target, long count,
                  const gxKerningAdjustment kerningAdjustments[]);
```

`target` A reference to the shape object whose associated style object is to be assigned the kerning adjustments array.

`count` The number of kerning adjustment structures to assign; the number of elements in the kerning adjustments array.

`kerningAdjustments`
The array of kerning adjustment structures to assign to the style object associated with the shape referenced in the `target` parameter. If you specify `nil` for this parameter and 0 for the `count` parameter, the function removes all kerning adjustment structures from the style object.

DESCRIPTION

The `GXSetShapeRunKerningAdjustments` function assigns the specified array of kerning adjustment structures to the style object associated with the target shape.

Calling this function for the layout shape `myLayout` is equivalent to making the following call:

```
GXSetStyleRunKerningAdjustments(GXGetShapeStyle(myLayout),
                                myCount, myAdjustsArray);
```

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the `GXNewLayout` function, for example, you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as `GXSetStyleRunKerningAdjustments`.

If `count` is 0 and `kerningAdjustments` is non-`nil`, or if `count` is nonzero and `kerningAdjustments` is `nil`, `GXSetShapeRunKerningAdjustments` posts an `inconsistent_parameters` error.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
count_is_less_than_zero
inconsistent_parameters
```

Notices (debugging version)

```
attributes_already_set
```

SEE ALSO

You retrieve the kerning adjustments array from the style object associated with a shape object using the `GXGetShapeRunKerningAdjustments` function, described in the previous section. You assign a kerning adjustments array directly to a style object using the `GXSetStyleRunKerningAdjustments` function, described on page 8-72.

The kerning adjustment structure is described on page 8-63.

Customizing Glyph Substitution

The functions in this section allow you to get or set the glyph substitutions property of a specified style object or of the style object associated with a specified layout shape.

GXGetStyleRunGlyphSubstitutions

You can use the `GXGetStyleRunGlyphSubstitutions` function to retrieve the array of glyph substitution structures from a style object.

```
long GXGetStyleRunGlyphSubstitutions(gxStyle source,
                        gxGlyphSubstitution glyphSubstitutions[]);
```

`source` A reference to the style object whose glyph substitutions array you need.

`glyphSubstitutions`
An array of glyph substitution structures. On return, the array contains the glyph substitution information for the style object referenced in the `source` parameter. If you specify `nil` for this parameter, no information is returned in it; however, the function result is still the correct number of glyph substitution structures for the style.

function result The number of glyph substitution structures in the style object. If the style object contains no glyph substitution structures, the function returns 0.

DESCRIPTION

The `GXGetStyleRunGlyphSubstitutions` function retrieves the glyph substitutions array, if any, from the source style object. To get the glyph substitutions themselves, you need to allocate an array to pass in the `glyphSubstitutions` parameter when calling this function. To get the right size for the array, you can first call the function with a value of `nil` for the `glyphSubstitutions` parameter. Then use the function result to allocate an array of the proper size, and call `GXGetStyleRunGlyphSubstitutions` a second time, this time passing the array.

SPECIAL CONSIDERATIONS

Because QuickDraw GX can reorder the elements in a style object's glyph substitutions array, the order of elements returned in the `glyphSubstitutions` parameter to this function may differ from the order in which they were originally assigned to the style.

ERRORS, WARNINGS, AND NOTICES

Errors
`style_is_nil`

SEE ALSO

You assign a glyph substitutions array to a style object using the `GXSetStyleRunGlyphSubstitutions` function, described next. You retrieve the glyph substitutions array from the style object associated with a shape object using the `GXGetShapeRunGlyphSubstitutions` function, described on page 8-78.

The glyph substitution structure is described on page 8-64.

GXSetStyleRunGlyphSubstitutions

You can use the `GXSetStyleRunGlyphSubstitutions` function to assign an array of glyph substitution structures to a style object.

```
void GXSetStyleRunGlyphSubstitutions(gxStyle target, long count,
                  const gxGlyphSubstitution glyphSubstitutions[]);
```

`target` A reference to the style object whose glyph substitutions array you are assigning.

`count` The number of glyph substitution structures to assign; the number of elements in the glyph substitutions array.

`glyphSubstitutions`
The array of glyph substitution structures to assign to the style object referenced in the `target` parameter. If you specify `nil` for this parameter and 0 for the `count` parameter, the function removes all glyph substitution structures from the style object.

DESCRIPTION

The `GXSetStyleRunGlyphSubstitutions` function assigns the specified array of glyph substitution structures to the target style object.

If `count` is 0 and `glyphSubstitutions` is non-`nil`, or if `count` is nonzero and `glyphSubstitutions` is `nil`, `GXSetStyleRunGlyphSubstitutions` posts an `inconsistent_parameters` error.

ERRORS, WARNINGS, AND NOTICES

Errors

```
style_is_nil
count_is_less_than_zero
inconsistent_parameters
```

Notices (debugging version)

```
attributes_already_set
```

SEE ALSO

You retrieve the glyph substitutions array from a style object using the `GXGetStyleRunGlyphSubstitutions` function, described in the previous section. You assign a glyph substitutions array to the style object associated with a shape object using the `GXSetShapeRunGlyphSubstitutions` function, described on page 8-79.

The glyph substitution structure is described on page 8-64.

For an example of the use of this function, see Listing 8-8 on page 8-52.

GXGetShapeRunGlyphSubstitutions

You can use the GXGetShapeRunGlyphSubstitutions function to retrieve the array of glyph substitution structures from the style object associated with a shape.

```
long GXGetShapeRunGlyphSubstitutions(gxShape source,
                        gxGlyphSubstitution glyphSubstitutions[]);
```

`source` A reference to the shape object whose associated style object contains the glyph substitutions array you need.

`glyphSubstitutions`
An array of glyph substitution structures. On return, the array contains the glyph substitution information for the style object associated with the shape referenced in the `source` parameter. If you specify `nil` for this parameter, no information is returned in it; however, the function result is still the correct number of glyph substitution structures for the style.

function result The number of glyph substitution structures in the style object associated with the shape. If the style object contains no glyph substitution structures, the function returns 0.

DESCRIPTION

The GXGetShapeRunGlyphSubstitutions function retrieves the glyph substitutions array, if any, from the style object associated with the source shape.

Calling this function for the layout shape `myLayout` is equivalent to making the following call:

```
myCount = GXGetStyleRunGlyphSubstitutions(
                        GXGetShapeStyle(myLayout), mySubsArray);
```

To get the glyph substitutions themselves, you need to allocate an array to pass in the `glyphSubstitutions` parameter when calling this function. To get the right size for the array, you can first call the function with a value of `nil` for the `glyphSubstitutions` parameter. Then use the function result to allocate an array of the proper size, and call GXGetShapeRunGlyphSubstitutions a second time, this time passing the array.

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the GXNewLayout function, for example, you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as GXGetStyleRunGlyphSubstitutions.

SPECIAL CONSIDERATIONS

Because QuickDraw GX can reorder the elements in a style object's glyph substitutions array, the order of elements returned in the `glyphSubstitutions` parameter to this function may differ from the order in which they were originally assigned to the style.

ERRORS, WARNINGS, AND NOTICES

Errors
`shape_is_nil`

SEE ALSO

You assign a glyph substitutions array to the style object associated with a shape object using the `GXSetShapeRunGlyphSubstitutions` function, described next. You retrieve the glyph substitutions array directly from a style object using the `GXGetStyleRunGlyphSubstitutions` function, described on page 8-75.

The glyph substitution structure is described on page 8-64.

GXSetShapeRunGlyphSubstitutions

You can use the `GXSetShapeRunGlyphSubstitutions` function to assign an array of glyph substitution structures to the style object associated with a shape.

```
void GXSetShapeRunGlyphSubstitutions(gxShape target, long count,
                  const gxGlyphSubstitution glyphSubstitutions[]);
```

`target` A reference to the shape object whose associated style object is to be assigned the glyph substitutions array.

`count` The number of glyph substitution structures to assign; the number of elements in the glyph substitutions array.

`glyphSubstitutions`
The array of glyph substitution structures to assign to the style object associated with the shape referenced in the `target` parameter. If you specify `nil` for this parameter and 0 for the `count` parameter, the function removes all glyph substitution structures from the style object.

DESCRIPTION

The `GXSetShapeRunGlyphSubstitutions` function assigns the specified array of glyph substitution structures to the style object associated with the source shape.

Calling this function for the layout shape `myLayout` is equivalent to making the following call:

```
GXSetStyleRunGlyphSubstitutions(GXGetShapeStyle(myLayout),
                                myCount, mySubsArray);
```

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the `GXNewLayout` function, for example, you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as `GXSetStyleRunGlyphSubstitutions`.

If `count` is 0 and `glyphSubstitutions` is non-`nil`, or if `count` is nonzero and `glyphSubstitutions` is `nil`, `GXSetShapeRunGlyphSubstitutions` posts an `inconsistent_parameters` error.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
count_is_less_than_zero
inconsistent_parameters
```

Notices (debugging version)

```
attributes_already_set
```

SEE ALSO

You retrieve the glyph substitutions array from the style object associated with a shape object using the `GXGetShapeRunGlyphSubstitutions` function, described in the previous section. You assign a glyph substitutions array directly to a style object using the `GXSetStyleRunGlyphSubstitutions` function, described on page 8-77.

The glyph substitution structure is described on page 8-64.

Customizing Font Features

The functions in this section allow you to get or set the run-features property of a specified style object, or of the style object associated with a specified layout shape.

GXGetStyleRunFeatures

You can use the `GXGetStyleRunFeatures` function to retrieve the array of run-feature structures from a style object.

```
long GXGetStyleRunFeatures(gxStyle source,
                           gxRunFeature runFeatures[]);
```

`source` A reference to the style object whose run-features array you need.

`runFeatures`
: An array of run-feature structures. On return, the array contains the run-feature information for the style object referenced in the `source` parameter. If you specify `nil` for this parameter, no information is returned in it; however, the function result is still the correct number of run-feature structures for the style object.

function result
: The number of run-feature structures in the style object. If the style object contains no run-feature structures, the function returns 0.

DESCRIPTION

The `GXGetStyleRunFeatures` function retrieves the run-features array, if any, from the source style object. If the style has no run-features array, QuickDraw GX applies only those features specified as defaults by the font when drawing.

To get the run features themselves, you need to allocate an array to pass in the `runFeatures` parameter when calling this function. To get the right size for the array, you can first call the function with a value of `nil` for the `runFeatures` parameter. Then use the function result to allocate an array of the proper size, and call `GXGetStyleRunFeatures` a second time, this time passing the array.

SPECIAL CONSIDERATIONS

Because QuickDraw GX can reorder the elements in a style object's run-features array, the order of elements returned in the `runFeatures` parameter to this function may differ from the order in which they were originally assigned to the style.

ERRORS, WARNINGS, AND NOTICES

Errors

`style_is_nil`

SEE ALSO

You assign a run-features array to a style object using the `GXSetStyleRunFeatures` function, described next. You retrieve the run-features array from the style object associated with a shape object using the `GXGetShapeRunFeatures` function, described on page 8-83.

The run-feature structure is described on page 8-65.

GXSetStyleRunFeatures

You can use the `GXSetStyleRunFeatures` function to assign an array of run-feature structures to a style object.

```
void GXSetStyleRunFeatures(gxStyle target, long count,
                           const gxRunFeature runFeatures[]);
```

`target` A reference to the style object whose run-features array you are assigning.

`count` The number of run-feature structures to assign; the number of elements in the run-features array.

`runFeatures`
The array of run-feature structures to assign to the style object referenced in the `target` parameter. If you specify `nil` for this parameter and 0 for the `count` parameter, the function removes all run-feature structures from the style object.

DESCRIPTION

The `GXSetStyleRunFeatures` function assigns the specified array of run-feature structures to the target style object. Font features assigned through this function add to or override the font-specified default features on an individual basis; simply specifying a run-features array with a value other than `nil` does not remove the default features. Specifying a `nil` run-features array and a value of 0 for the `count` parameter restores the complete set of default features.

If `count` is 0 and `runFeatures` is non-`nil`, or if `count` is nonzero and `runFeatures` is `nil`, `GXSetStyleRunFeatures` posts an `inconsistent_parameters` error.

ERRORS, WARNINGS, AND NOTICES

Errors
`style_is_nil`
`count_is_less_than_zero`
`inconsistent_parameters`

Notices (debugging version)
`attributes_already_set`

SEE ALSO

You retrieve the run-features array from a style object using the `GXGetStyleRunFeatures` function, described in the previous section. You assign a run-features array to the style object associated with a shape object using the `GXSetShapeRunFeatures` function, described on page 8-84.

The run-feature structure is described on page 8-65.

For an example of the use of this function, see Listing 8-9 on page 8-53. Other examples occur also in Listing 8-10 and Listing 8-11.

GXGetShapeRunFeatures

You can use the GXGetShapeRunFeatures function to retrieve the array of run-feature structures from the style object associated with a shape.

```
long GXGetShapeRunFeatures(gxShape source,
                           gxRunFeature runFeatures[]);
```

`source` A reference to the shape object whose associated style object contains the run-features array you need.

`runFeatures`
An array of run-feature structures. On return, the array contains the run-feature information for the style object associated with the shape referenced in the `source` parameter. If you specify `nil` for this parameter, no information is returned in it; however, the function result is still the correct number of run-feature structures for the style object.

function result The number of run-feature structures in the style object associated with the shape. If the style object contains no run-feature structures, the function returns 0.

DESCRIPTION

The GXGetShapeRunFeatures function retrieves the run-features array, if any, from the style object associated with the source shape. If the style has no run-features array, QuickDraw GX applies only those features specified as defaults by the font when drawing.

Calling this function for the layout shape `myLayout` is equivalent to making the following call:

```
myCount = GXGetStyleRunFeatures(GXGetShapeStyle(myLayout),
                                myFeaturesArray);
```

To get the run features themselves, you need to allocate an array to pass in the `runFeatures` parameter when calling this function. To get the right size for the array, you can first call the function with a value of `nil` for the `runFeatures` parameter. Then use the function result to allocate an array of the proper size, and call GXGetShapeRunFeatures a second time, this time passing the array.

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the GXNewLayout function, for example, you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as GXGetStyleRunFeatures.

SPECIAL CONSIDERATIONS

Because QuickDraw GX can reorder the elements in a style object's run-features array, the order of elements returned in the `runFeatures` parameter to this function may differ from the order in which they were originally assigned to the style.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`

SEE ALSO

You assign a run-features array to the style object associated with a shape object using the `GXSetShapeRunFeatures` function, described next. You retrieve the run-features array directly from a style object using the `GXGetStyleRunFeatures` function, described on page 8-80.

The run-feature structure is described on page 8-65.

GXSetShapeRunFeatures

You can use the `GXSetShapeRunFeatures` function to assign an array of run-feature structures to the style object associated with a shape.

```
void GXSetShapeRunFeatures(gxShape target, long count,
                           const gxRunFeature runFeatures[]);
```

`target` A reference to the shape object whose associated style object is to be assigned the run-features array.

`count` The number of run-feature structures to assign; the number of elements in the run-features array.

`runFeatures`
The array of run-feature structures to assign to the style object associated with the shape referenced in the `target` parameter. If you specify `nil` for this parameter and 0 for the `count` parameter, the function removes all run-feature structures from the style object.

DESCRIPTION

The `GXSetShapeRunFeatures` function assigns the specified array of run-feature structures to the style object associated with the target shape. Font features assigned through this function add to or override the font-specified default features on an individual basis; simply specifying a run-features array with a value other than `nil` does not remove the default features. Specifying a `nil` run-features array (and a value of 0 for the `count` parameter) restores the complete set of default features.

Calling this function for the layout shape `myLayout` is equivalent to making the following call:

```
GXSetStyleRunFeatures(GXGetShapeStyle(myLayout), myCount,
                      myFeaturesArray);
```

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the `GXNewLayout` function, for example, you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as `GXSetStyleRunFeatures`.

If `count` is 0 and `runFeatures` is non-`nil`, or if `count` is nonzero and `runFeatures` is `nil`, `GXSetShapeRunFeatures` posts an `inconsistent_parameters` error.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`count_is_less_than_zero`
`inconsistent_parameters`

Notices (debugging version)

`attributes_already_set`

SEE ALSO

You retrieve the run-features array from the style object associated with a shape object using the `GXGetShapeRunFeatures` function, described in the previous section. You assign a run-features array directly to a style object using the `GXSetStyleRunFeatures` function, described on page 8-82.

The run-feature structure is described on page 8-65.

Summary of Layout Styles

Constants and Data Types

Run Controls Structure

```
struct gxRunControls {
   gxRunControlFlags    flags;
   Fixed                beforeWithStreamShift;
   Fixed                afterWithStreamShift;
   Fixed                crossStreamShift;
   Fixed                imposedWidth;
   Fixed                track;
   fract                hangingInhibitFactor;
   fract                kerningInhibitFactor;
   Fixed                decompositionAdjustmentFactor;
   gxBaselineType       baselineType;
};
```

Run Control Flags

```
#define gxNoLigatureSplits        0x80000000
#define gxNoCaretAngle            0x40000000
#define gxImposeWidth             0x20000000
#define gxNoCrossKerning          0x10000000
#define gxNoOpticalAlignment      0x08000000
#define gxForceHanging            0x04000000
#define gxNoSpecialJustification  0x02000000
#define gxDirectionOverrideMask   0x00000003

typedef unsigned long gxRunControlFlags;

#define gxAllRunControlFlags (gxNoLigatureSplits|gxNoCaretAngle|
                              gxImposeWidth|gxNoCrossKerning|
                              gxNoOpticalAlignment|gxForceHanging|
                              gxNoSpecialJustification|
                              gxDirectionOverrideMask)
```

Direction Overrides

```
enum gxDirectionOverrides{
   gxNoDirectionOverride   = 0,
   gxImposeLeftToRight     = 1,
```

```
   gxImposeRightToLeft     = 2,
   gxImposeArabic          = 3
};
typedef unsigned short gxDirectionOverride;
```

Kerning Adjustment Factors Structure

```
struct gxKerningAdjustmentFactors {
   fract       scaleFactor;
   Fixed       adjustmentPointSizeFactor;
};
```

Kerning Adjustment Structure

```
struct gxKerningAdjustment {
   gxGlyphcode                          firstGlyph;
   gxGlyphcode                          secondGlyph;
   struct gxKerningAdjustmentFactors    withStreamFactors;
   struct gxKerningAdjustmentFactors    crossStreamFactors;
};
```

Glyph Substitution Structure

```
struct gxGlyphSubstitution {
   gxGlyphcode    originalGlyph;
   gxGlyphcode    substituteGlyph;
};
```

Run-Feature Structure

```
struct gxRunFeature {
   gxRunFeatureType     featureType;
   gxRunFeatureSelector featureSelector;
};
```

Functions

Getting and Setting Run Controls

```
long GXGetStyleRunControls  (gxStyle source, gxRunControls *runControls);
void GXSetStyleRunControls  (gxStyle target,
                             const gxRunControls *runControls);
long GXGetShapeRunControls  (gxShape source, gxRunControls *runControls);
void GXSetShapeRunControls  (gxShape target,
                             const gxRunControls *runControls);
```

Customizing Kerning

```
long GXGetStyleRunKerningAdjustments
                            (gxStyle source,
                             gxKerningAdjustment kerningAdjustments[]);
void GXSetStyleRunKerningAdjustments
                            (gxStyle target, long count,
                             const gxKerningAdjustment
                             kerningAdjustments[]);
long GXGetShapeRunKerningAdjustments
                            (gxShape source,
                             gxKerningAdjustment kerningAdjustments[]);
void GXSetShapeRunKerningAdjustments
                            (gxShape target, long count,
                             const gxKerningAdjustment
                             kerningAdjustments[]);
```

Customizing Glyph Substitution

```
long GXGetStyleRunGlyphSubstitutions
                            (gxStyle source,
                             gxGlyphSubstitution glyphSubstitutions[]);
void GXSetStyleRunGlyphSubstitutions
                            (gxStyle target, long count,
                             const gxGlyphSubstitution
                             glyphSubstitutions[]);
long GXGetShapeRunGlyphSubstitutions
                            (gxShape source,
                             gxGlyphSubstitution glyphSubstitutions[])
void GXSetShapeRunGlyphSubstitutions
                            (gxShape target, long count,
                             const gxGlyphSubstitution
                             glyphSubstitutions[]);
```

Customizing Font Features

```
long GXGetStyleRunFeatures  (gxStyle source, gxRunFeature runFeatures[]);
void GXSetStyleRunFeatures  (gxStyle target, long count,
                             const gxRunFeature runFeatures[]);
long GXGetShapeRunFeatures  (gxShape source, gxRunFeature runFeatures[]);
void GXSetShapeRunFeatures  (gxShape target, long count,
                             const gxRunFeature runFeatures[]);
```

CHAPTER 9

Layout Line Control

Contents

This chapter describes those features of layout shapes that help you lay out and manipulate an entire line of text. Line span, line length and line breaking, text direction, and justification can affect the text of a whole line, regardless of the number or characteristics of the individual style runs making up the line.

Although it is possible to create and draw a layout shape based solely on information presented in the chapter "Layout Shapes" in this book, most applications need to use the information presented here to take advantage of the layout capabilities that QuickDraw GX provides. Read the information in this chapter if you create layout shapes and need to control line characteristics. If you do not create layout shapes, you do not need the information in this chapter.

Before reading this chapter, you should be familiar with the information in the chapters "Introduction to QuickDraw GX Typography," "Typographic Shapes," "Typographic Styles," and "Layout Shapes" in this book. You should also be familiar with the general concepts of QuickDraw GX objects, as described in *Inside Macintosh: QuickDraw GX Objects*.

Some of the information in this chapter concerns layout-related properties of the style object. Most layout-related style properties are discussed in the chapter "Layout Styles" in this book. Those discussed here, the justification-related properties, are presented in this chapter because they are typically manipulated in the context of the line as a whole.

This chapter presents detailed information on text direction and nested direction levels, even though the levels array of the layout shape is introduced earlier in this book, in the chapter "Layout Shapes." Text direction and nested direction levels can affect the entire line, and therefore the details of how to manipulate them are presented here.

The chapter starts by describing how QuickDraw GX defines baselines, line measurement, text direction, and justification. It then describes how to use QuickDraw GX functions to

- set baselines
- determine line lengths and line spans
- break lines
- manipulate text direction, including nested direction levels in mixed-direction text
- perform full and partial justification with a variety of justification techniques
- change the justification behavior of classes of glyphs
- change the justification behavior of individual glyphs

About Line Control and Line Measurement for Layout Shapes

Text-handling with layout shapes is line-based. The features of layout shapes are designed mainly to allow you to lay out individual lines of text that uses complex formatting. Although your application may work with paragraphs, sections, chapters, and other larger text units, each line of your documents is at some point manipulated as a separate layout shape.

This section describes how QuickDraw GX defines and allows you to manipulate the following line-based components in text layout:

- baselines
- line measurement and line breaking
- text direction
- justification

Baselines

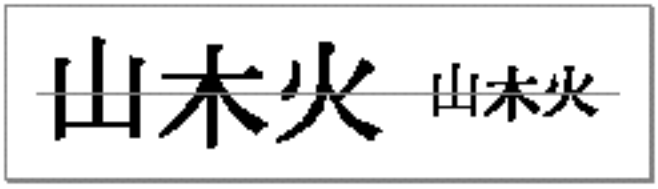
A **baseline** is an imaginary line that is used to align glyphs in a line of text. A baseline can coincide with various locations in a glyph, such as the bottom, middle, or top, depending on what type of baseline it is. The baseline represents a stable platform, giving a common point of alignment to glyphs of different shapes and sizes.

Baseline Types

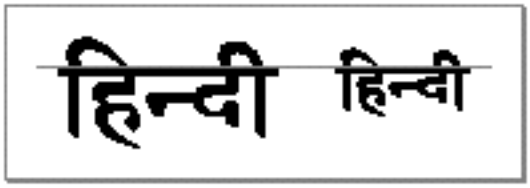
There are several common types of baselines used to lay out text:

- **Roman baseline.** The baseline used in most Roman scripts, as well as by Arabic and Hebrew. Most of the glyph appears above the Roman baseline, sometimes with portions below it, as with glyphs such as “y” or “j”. The baseline is near the bottom of the entire row of glyphs:

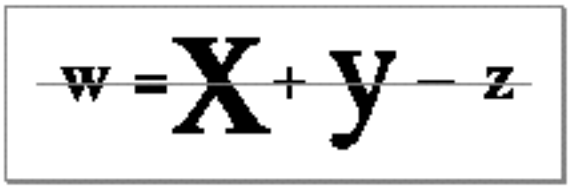
- **Ideographic centered baseline.** A baseline used by Chinese, Japanese, and Korean ideographic scripts; glyphs are centered halfway on the line height:



- **Hanging baseline.** The baseline used by Devanagari and similar scripts. Most of the glyph is below the baseline, sometimes with portions above it, and the baseline is near the top of the glyphs:



- **Math baseline**. A baseline used for setting mathematical expressions, centered on operators such as the minus sign. Such operators usually appear at half the x-height in a font:



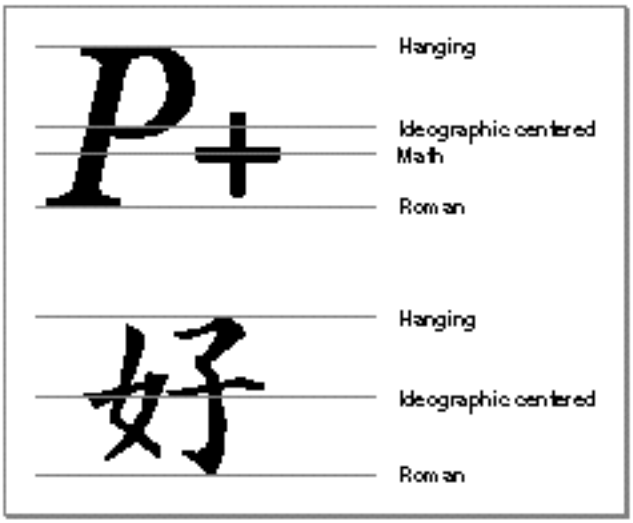
QuickDraw GX supports vertical text, although not by using vertical baselines. All baselines in QuickDraw GX are horizontal, but you can specify text characteristics such that the line is displayed properly when rotated to a vertical position. For more information, see “Baselines for Vertical Text” beginning on page 9-8. Also, see the introductory discussion of vertical text in the chapter “Layout Shapes” in this book.

QuickDraw GX supports a variety of baselines and allows you to mix text of various baselines and sizes on a single line. QuickDraw GX baseline types are defined in the `gxBaselineType` enumeration, described on page 9-58.

Font and Application Control Over Baselines

Each QuickDraw GX-compatible font specifies a baseline type for every glyph in the font. It also specifies the positions of each baseline in the font’s own overall coordinate system. This information is present only for the font as a whole; each glyph in the font shares the same set of baseline positions, although different glyphs in a font may use different baseline types within the set. Figure 9-1 shows where these baselines might be for two glyphs from two different fonts.

Figure 9-1 Baseline positions for two fonts



When you prepare to draw a line of text, you need to identify an overriding baseline (if desired) for each style run, and you also need to identify offsets from the y-coordinate of the layout shape's position for each of the baseline types. QuickDraw GX provides a routine to assist in this process. Figure 9-2 shows examples of a line with six style runs rendered with two different sets of baseline information. Assume for this example that the position of each shape is at the lower-left corner of the first 'A'. The first line shows the last three style runs having their baseline type overridden to the Roman baseline, where the Roman baseline has a delta of zero from the shape's position. The second line shows the baseline type of the first three style runs overridden to the hanging baseline, which has a nonzero delta from the shape's position.

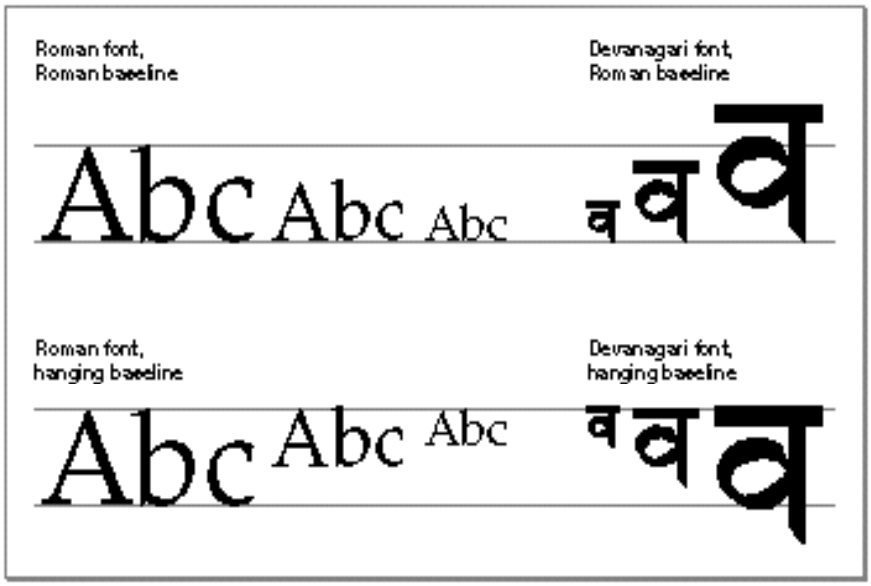
The information overriding the baseline for each style run is contained in the style's run controls, described on page 8-61. The information defining the baseline deltas is contained in the `gxBaselineDeltas` array, described on page 9-59.

Alignment of Multiple Baselines

This section describes how QuickDraw GX makes use of the baseline deltas to lay out text. If a layout shape comprises glyphs of only one baseline type, the alignment of the glyphs is obvious. But if the text in the shape is of different sizes or uses several different baseline types, it may not be obvious at first glance how best to line up the text:

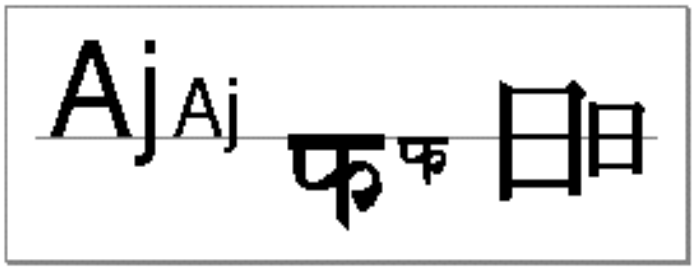
- One (incorrect) possibility might be to simply align all glyphs with a y-delta of 0. That strategy works for text whose default baseline happens to be y = 0, but it does not work for other text. Figure 9-2 shows the misalignment resulting from using a hanging baseline for mixed-size Roman text and a Roman baseline for Devanagari text.

Figure 9-2 How the same glyphs can align to different baselines



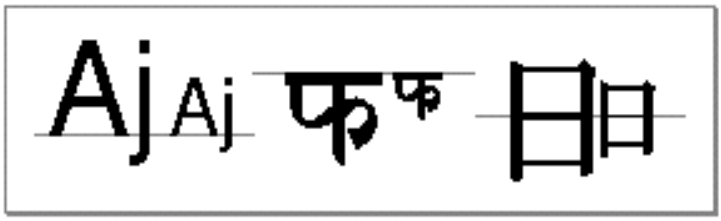
- Another (incorrect) possibility is to line up each style run's default baseline with the primary baseline you have chosen for the layout. Figure 9-3 shows an example in which the dominant style run is the leftmost and the default baseline type for the layout shape is Roman. The text within each style run aligns with its own baseline type: Roman text sits properly on its Roman baseline, Devanagari text hangs from its baseline, and the Chinese text straddles its ideographic centered baseline. Each style run is correct within itself, but the runs do not line up correctly because the baselines should not be aligned.

Figure 9-3 Text with multiple baselines aligned to y = 0



- The best results occur when the default baseline for each style run is aligned with the appropriate choice from the set of baselines as defined in the dominant style run. This is the method that QuickDraw GX is designed to support. Figure 9-4 shows the same text as Figure 9-3, but this time the Devanagari and Chinese text align with the hanging and ideographic centered baseline types defined for the Roman baseline of the Roman text.

Figure 9-4 Preferred alignment for multiple baselines



If you supply QuickDraw GX with the proper information about your text (what the deltas are between the other baseline types and what baseline type each of your style runs uses), it automatically aligns the baselines as shown in Figure 9-4, giving you the best results for multilanguage layout.

Drop capitals

Drop capitals are large uppercase letters that drop below the main line of text for aesthetic reasons. You can create a drop capital by setting the baseline type for individual runs of letters to the hanging baseline type in the run controls and by making sure the array of distances to the various baselines is set up, as shown in Figure 9-16 on page 9-30. u

See the section “Setting Baselines” beginning on page 9-27 for more information on baselines and how to control them.

Baselines for Vertical Text

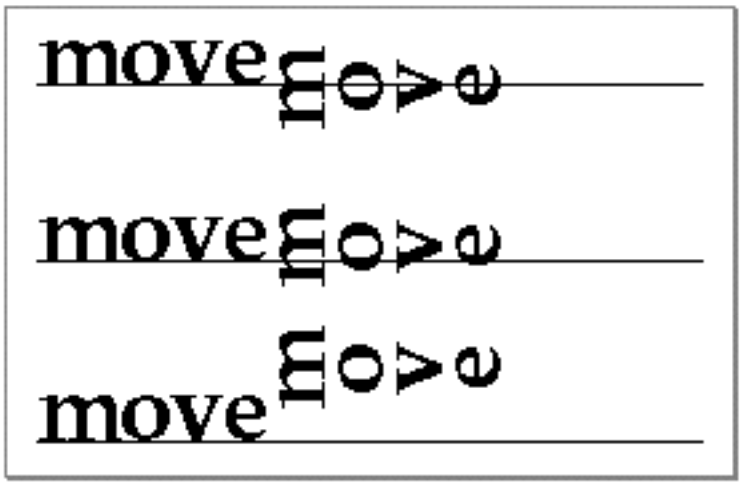
As noted previously, vertical text in a layout shape is just a special case of horizontal text. There is no vertical baseline and no vertical line direction in QuickDraw GX; a vertical line is calculated and laid out as if it were horizontal.

When QuickDraw GX creates text that is to be displayed vertically (meaning the `gxVerticalText` text attribute in its style object is set), it rotates the text’s individual glyphs 90 degrees counterclockwise. It then sets the glyphs on a baseline (which at this point is horizontal). If the glyphs do not have explicitly defined vertical metrics, QuickDraw GX synthesizes a centered vertical baseline and places the glyphs on it. Before drawing the shape, your application is responsible for setting its transform object’s mapping to rotate it 90 degrees (clockwise), thus making it vertical and restoring the glyphs to their proper orientation. You are also responsible for realigning the vertical glyphs, if necessary.

For runs of text that are vertical, baseline alignment takes on a slightly different meaning. Figure 9-5, for example, is a layout shape drawn three times, each consisting

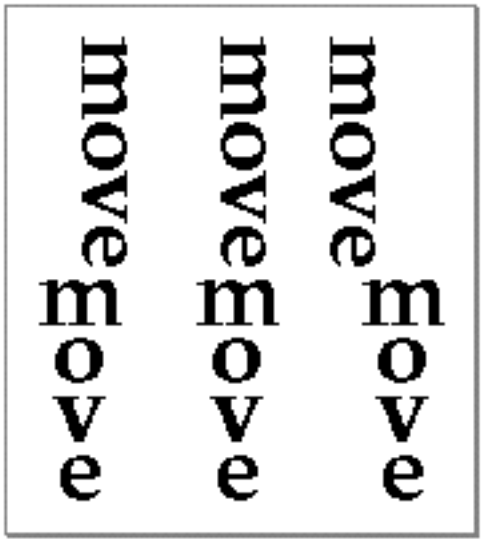
of two runs of text. The second run of text in each shape has the `gxVerticalText` text attribute set. The first line shows the result of setting `baselineType` to `gxRomanBaseline`, the second line shows the result of setting `baselineType` to `gxIdeographicCenterBaseline`, and the third line shows the result of setting `baselineType` to `gxHangingBaseline`. For all three lines, the baseline deltas were derived from the horizontal run of text using the `GXSetStyleBaselineDeltas` call.

Figure 9-5 Creating vertical text in a layout shape



The sample code used to generate Figure 9-5 is Listing 9-2 on page 9-31. If you rotated each shape produced by Listing 9-2 to a vertical position and then redrew it, you would get the text shown in Figure 9-6.

Figure 9-6 Rotating vertical text in a layout shape



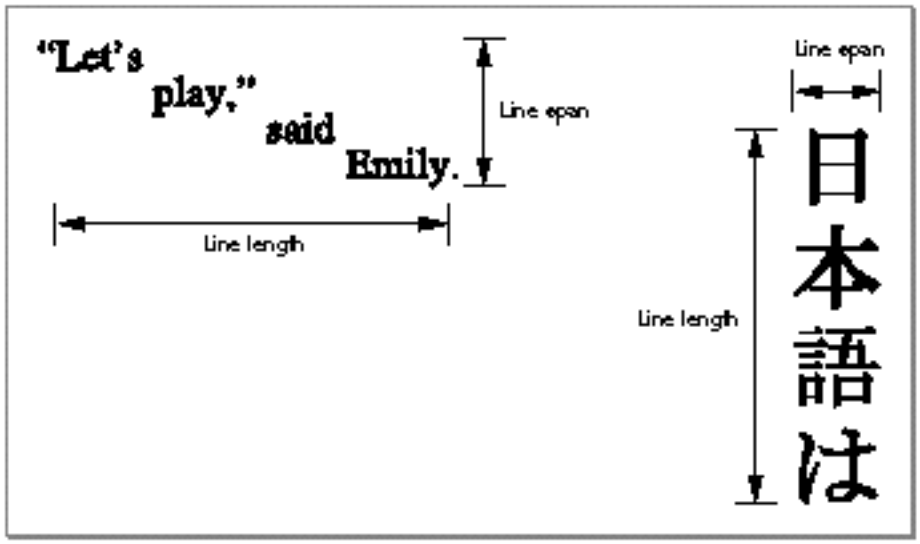
For more information, see “Drawing Vertical Text” beginning on page 9-30 and Listing 9-2 on page 9-31.

Line Measurement

Measuring the dimensions of text lines is a standard process in performing text layout. For QuickDraw GX layout shapes, line measurement can involve complex calculations.

For a horizontal line of text, line length is its width, and line span is its line height. For a vertical line of text, line length is a height measurement, and line span is a width measurement. Figure 9-7 illustrates length and span for both horizontal and vertical lines. Because QuickDraw GX treats vertical text as if it were horizontal, the subsequent examples in this section all assume horizontal lines (length is equivalent to width, and span is equivalent to height).

Figure 9-7 Line length and line span



Line Length

Determining the length of a line of text is fundamental to all layout, and especially for line breaking. You can use the width of the standard bounding rectangle or the typographic bounding rectangle of the layout shape to get two possibly slightly different measures of

its overall line length. QuickDraw GX also provides a function that gives you the length of any range of text within a layout shape, up to and including the entire shape.

See the section “Determining Line Lengths” beginning on page 9-32 for more information.

Line Span

Because the typographic capabilities of QuickDraw GX are limited to laying out and drawing single lines of text, any time you work with multiple lines you need to determine how far to separate the lines to avoid overlap, truncation, or excess leading. Whereas in simple text it is relatively easy to calculate line spacing based on a single text-size value, such calculations can be very complex in layout shapes. To space lines properly, you need to account for different text sizes in different style runs and the possibility of cross-stream kerning, cross-stream shifting, and shifted baselines.

You can use the height of the standard bounding rectangle or the typographic bounding rectangle of the layout shape to get two possibly slightly different measures of its line span. QuickDraw GX also provides a function that returns the line span for any layout shape, no matter how complex. QuickDraw GX uses that span when calculating the shape of the caret and the height of highlight areas; you can use it to determine where to place line starts.

QuickDraw GX also provides a function that allows you to set the line span for any layout shape, in case you need to force a specific line spacing or caret height.

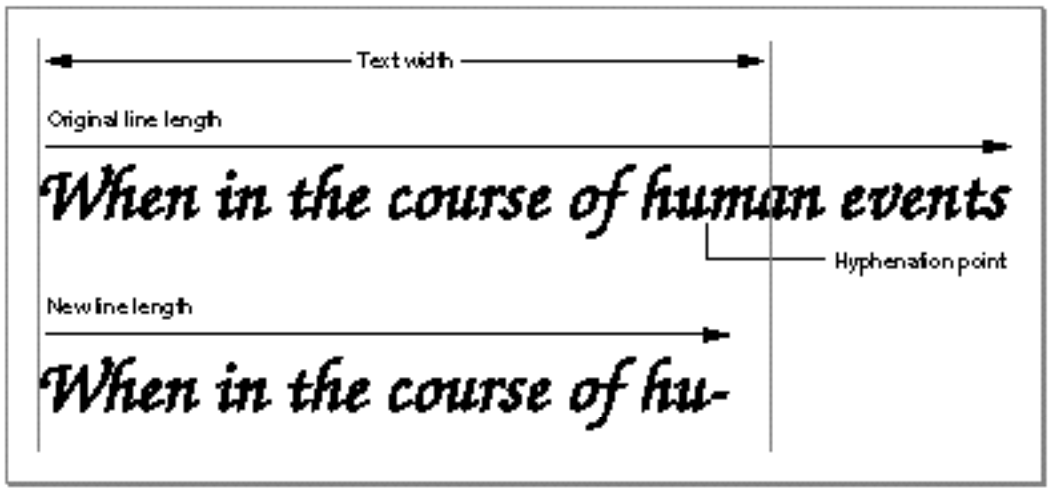
See the section “Determining Line Spans” beginning on page 9-33 for more information.

Line Breaking

If you work with text that can wrap to multiple lines, line breaking is a task of fundamental importance. Although QuickDraw GX provides several functions that help you with line breaking, an important point to remember is that *line-breaking decisions are still up to your application*. QuickDraw GX and the fonts it uses have no information about morphology, or the internal structure of words. Your application, perhaps with the help of the Macintosh WorldScript international resources, must decide where to end a line. What QuickDraw GX can do is provide fast ways to determine line length at potential break points, identify break points that are most efficient in terms of line layout, and create new layout shapes for each line.

Figure 9-8 shows the basic factors involved in the line-breaking decision. The **text width** is the area between the margins, within which all displayed text must fit. If the line length of your whole layout shape is less than the text width, there is no need to break the line at all. But if it is greater, you need to break the line so that it fits within the text width and also satisfies the requirements of the language.

Figure 9-8 Factors in line breaking



QuickDraw GX allows you to provide a set of **hyphenation points,** edge offsets in the source text at which it is appropriate to break the line. If you do so, QuickDraw GX uses that information in suggesting a line-break position. If you do not provide hyphenation points, QuickDraw GX calculates a break position at the last whole glyph that fits within the text width.

In returning a suggested break point, QuickDraw GX also notifies you of the closest positions to that break point that are staked. A **stake** is an edge offset in the source text at which point, if a decision is made to break the line there, the break will be “clean” in terms of layout processing—that is, not in the middle of a ligature, or inside a kerning pair, or between a pair of rearranged glyphs.

Staked offsets are separate from and largely independent of hyphenation points. The best hyphenation point may be within a ligature (the word “offload” would break within the “fl” ligature), so it may not be a staked position; a staked position (such as between the “l” and “o” in “offload”) may not be an acceptable hyphenation point. QuickDraw GX provides the information on staked positions to help you be most efficient in laying out text during line breaking; it does not take the place of the morphological information you must have to calculate proper hyphenation points.

See the section “Breaking Lines” beginning on page 9-33 for more information on line-breaking and examples of line-breaking algorithms. See the section “Using Macintosh WorldScript for Line Breaking” beginning on page 9-37 for information on using international resources to help with line-breaking decisions.

Text Direction

Displayed text always has a **direction,** which is the direction in which a person's eyes move when reading successive glyphs. Different languages have different directions, and some individual languages support more than one direction. Text of Roman languages typically has a left-to-right direction (although vertical text is used occasionally); text in Chinese and Japanese may have a vertical direction, although left-to-right text is also common (and right to left is possible). Arabic and Hebrew have a predominantly right-to-left direction, although some text in both languages is written left to right.

Vertical text

Throughout this section and in much of the rest of this chapter, the discussion is in terms of horizontal text lines. Because vertical text in QuickDraw GX can be thought of as horizontal text in which the individual glyphs are rotated 90 degrees, you can apply the discussions here to vertical text by substituting "top-to-bottom" for "left-to-right," "width" for "height," and "height" for "width." u

Direction is multileveled; horizontal text of one direction may have embedded text of the opposite direction, and the entire sequence may be embedded within a line of text that has either direction. To help sort out the complications, QuickDraw GX defines two direction-related concepts:

- **Glyph direction,** the most fundamental and smallest-scale directionality. It is applied to individual glyphs.
- **Dominant direction,** a broader direction control imposed on groups of glyphs. (The broadest control on text direction is **line direction,** and you can think of it as the dominant direction for an entire line of text.)

This section describes those concepts and shows you how to control them properly when laying out lines of mixed-direction text.

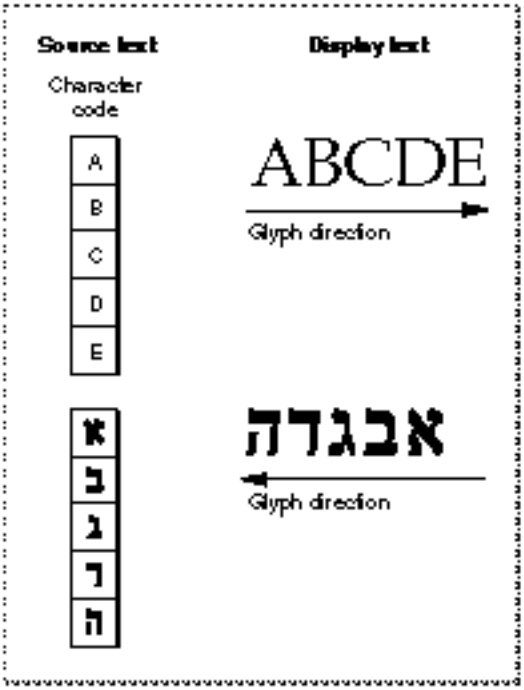
Glyph Direction

Every glyph in a font has a defined direction that determines how it is positioned in relation to the previous and subsequent (in reading order) glyphs. That direction is implied in the positions of the glyph's leading and trailing edges; for example, if a glyph's leading edge is on its left side, that glyph has a left-to-right direction.

When QuickDraw GX draws a sequence of glyphs, it takes glyph direction into account when calculating the order in which to display them. If a sequence of glyphs all have left-to-right direction, the left-to-right display order in which QuickDraw GX displays them matches the sequential order of their characters in the source text. If the sequence is

all right-to-left glyphs, QuickDraw GX displays them in an order that is opposite to the sequential order of their characters in the source text. Figure 9-9 shows examples of both kinds of sequences.

Figure 9-9 How glyph direction affects display order



QuickDraw GX always displays consecutive groups of glyphs that have a common direction in the sequence determined by that direction, regardless of other direction properties that may be imposed on the text. Your application has no control over this process, other than to override it globally for an entire style run. You can impose a strong right-to-left or left-to-right direction on all of the glyphs of a style run by setting a flag value in the run controls structure of the style object. The run controls structure is described in the chapter “Layout Styles” in this book.

Types of Glyph Direction

Because of the complications that arise in languages that support two directions and in single lines with text of different languages and directions, QuickDraw GX recognizes several types of directionality for glyphs. Most glyphs have a **strong type** of direction, meaning that they are always read only in the direction defined for them. Glyphs with strong left-to-right direction include most alphabetic, syllabic, and Han ideographic glyphs for most languages. Glyphs with strong right-to-left direction include Arabic and Hebrew alphabetic glyphs and punctuation.

Glyphs of numbers and their associated symbols are considered to have a **weak type** of direction. They typically are read left-to-right but are placed on the line as if they had the direction of the adjacent glyphs. (See the next section, “Dominant Direction” beginning on page 9-15, for a discussion of how glyph direction affects placement of blocks of glyphs on the line.)

Glyphs with weak direction include European and Arabic numbers, European number separators (such as the figure space, the period, and the slash), European number terminators (such as the plus and minus signs, the percent sign, and currency symbols), and the common number separators (the colon and the comma).

Neutral glyphs are glyphs without inherent direction. They generally take on the direction of the surrounding text. Neutral glyphs include block separators (such as the paragraph separator or line separator) and whitespace glyphs.

Normally, your application need not be concerned with the details of the type or class of direction associated with a particular glyph, although QuickDraw GX uses it in determining the reordering necessary to lay out a line of mixed-direction text. If you need to, you can override direction by setting a value in the run controls structure of the style object. Note, however, that the override applies to an entire style run, and that you can override into one of the strong types only. The run controls structure is described in the chapter “Layout Styles” in this book. For more information on direction classes and how they affect text layout, see *The Unicode Standard: Worldwide Character Encoding, Version 1.0*, Volume 1.

Dominant Direction

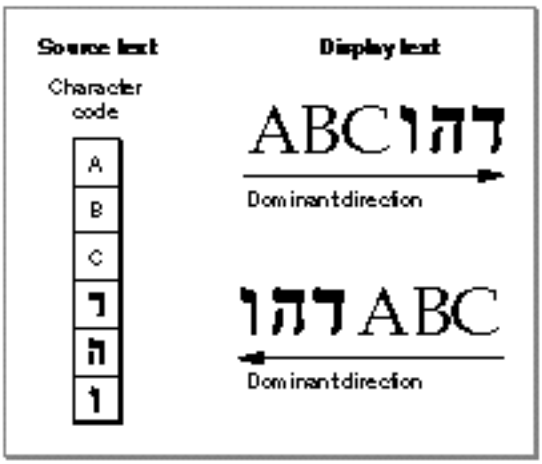
The **dominant direction** of a line (or any other text run) is the overall, controlling direction within which the individual glyph directions are set. Dominant direction does not reverse glyph direction; it is a higher-level effect. If a Hebrew word is embedded in a line of Roman text, the dominant direction for that line is left to right, but the Hebrew word is still laid out right to left, as expected. Conversely, Roman text embedded in a line of Hebrew, in which the dominant direction is right to left, is still displayed left to right.

For this reason, dominant direction has significance only in mixed-direction text. If all the text on a line has a uniform glyph direction, such as the text shown in Figure 9-9 on page 9-14, its display order is unchanged no matter what the dominant direction is. Changing the dominant direction has no effect on the display of either line of text in Figure 9-9.

In mixed-direction text, however, changing the dominant direction has a significant effect. Figure 9-10 shows a layout shape whose source text consists of three Roman characters followed by three Hebrew characters. In the upper display line, the dominant direction is specified as left to right, implying that the Hebrew letters are embedded in a line of Roman text. The Hebrew letters are laid out right to left, but the line as a whole (the groups of letters of a given direction) is laid out left to right.

The lower display line in Figure 9-10 shows the same text when the dominant direction is specified as right to left, implying that the Roman letters are embedded in a line of Hebrew text. The Roman letters are laid out left to right, but the line as a whole (the groups of letters of a given direction) is laid out right to left.

Figure 9-10 How dominant direction affects display order



The QuickDraw GX text layout model accounts for dominant direction as well as glyph direction, automatically performing any reordering needed for correct display of simple mixed-direction lines of text such as those shown in Figure 9-10. Furthermore, in QuickDraw GX the concept of dominant direction is not limited to line direction. Any section of text in a layout shape can have a dominant direction, and dominant directions exist in a nested hierarchy in which the line direction is simply the lowest nesting level.

This nesting of dominant directions allows you to preserve very complex multiple-language formatting. You specify the nested hierarchy of direction levels in the levels array of the layout shape. The levels array is introduced in the chapter “Layout Shapes” in this book. The next two sections show how to use the levels array to perform complex multilanguage formatting.

The Levels Array of the Layout Shape Object

Figure 9-11 shows the geometry of the layout shape. Part of the geometry is the levels array, represented by the properties level run count, level lengths array, and levels array.

It is in the levels array that you specify the dominant direction for the line of text in a layout shape. It is also in the levels array that you can specify additional nested levels of direction.

Figure 9-11 The levels array property of the layout shape

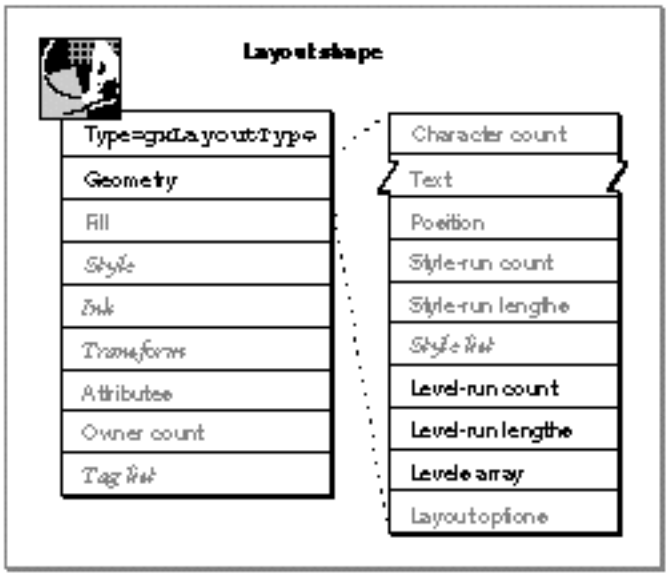
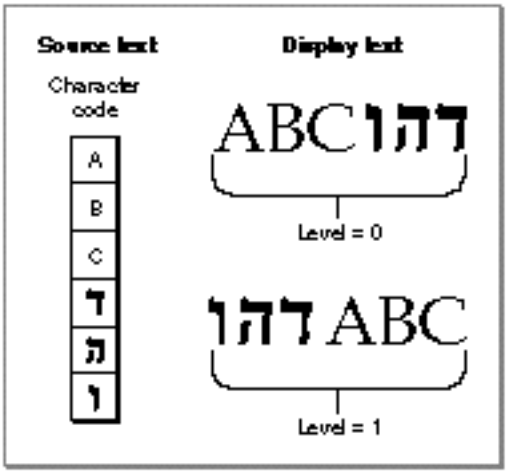


Figure 9-12 demonstrates the simplest relationship between nesting level and text direction. It shows the same layout shape as displayed in Figure 9-10 on page 9-16.

- If you specify a single direction-level run with a level of 0 for your entire layout shape (or if your levels array is `nil`, the default), you are specifying a left-to-right dominant direction for the line. The line is reordered and displayed as shown in the upper line of Figure 9-12 (identical to the upper line of Figure 9-10).
- If you specify a single direction-level run with a level of 1 for your entire layout shape, you are specifying a right-to-left dominant direction for the line. The line is reordered and displayed as shown in the lower line of Figure 9-12 (identical to the lower line of Figure 9-10).

Figure 9-12 How nesting level relates to text direction



In fact, if you specify any even number for a level, QuickDraw GX considers text of that level to have a dominant direction of left to right. Likewise, if you specify any odd number for a level, QuickDraw GX considers text of that level to have a dominant direction of right to left.

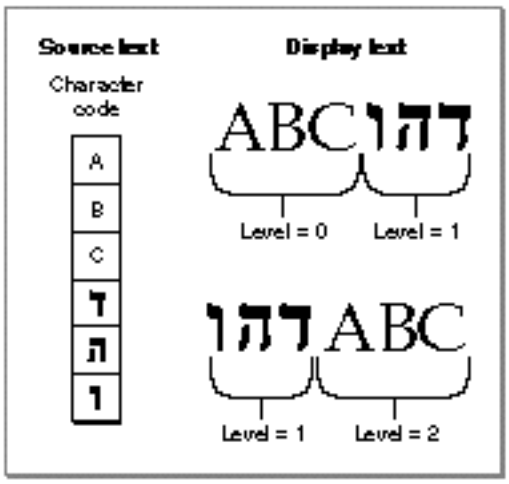
Note that a change in level need not accompany each change in glyph direction. QuickDraw GX automatically reorders glyphs appropriately according to their glyph directions.

Where a line contains more than one level of text, QuickDraw GX looks at the *lowest* level to determine the dominant direction for the line. Figure 9-13 shows alternative ways to specify the direction information shown in Figure 9-12 for the layout shape:

- In the upper line of Figure 9-13, each run of text in a given direction has its own level. The formatting of the display is identical to the upper line of Figure 9-12 because QuickDraw GX notes that the lowest level on the line is an even number and therefore orders the line from left to right.

- In the lower line of Figure 9-13, the runs of text are the same, except the Roman text has a level of 2, not 0. In this case, the lowest level on the line is an odd number, and QuickDraw GX therefore orders the line from right to left. The formatting of the display is identical to the lower line of Figure 9-12.

Figure 9-13 Multiple nesting direction levels in one line



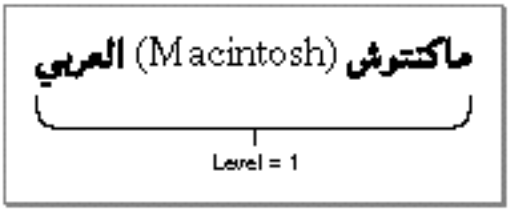
The use of levels other than 0 and 1 is unnecessary for this simple example, because you can specify a single level for the entire line. In most situations, even with mixed-direction text, you never need to use more than a single level run (with value 0 or 1) for your layout shape. The next section, however, shows how you can use multiple nested direction levels to prevent incorrect reordering in more complex situations.

Forced Reordering With Nested Direction Levels

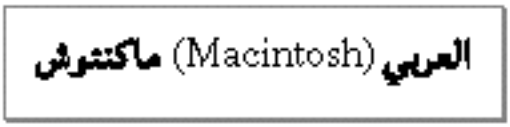
This section shows one example of when the explicit use of nesting levels may be necessary. Suppose, for example, that the following is a layout shape consisting of the Arabic-language equivalent of the phrase “Arabic Macintosh”:

In this case, all glyphs have a right-to-left direction, and specification of nesting levels (even for the dominant direction for the line) is unnecessary.

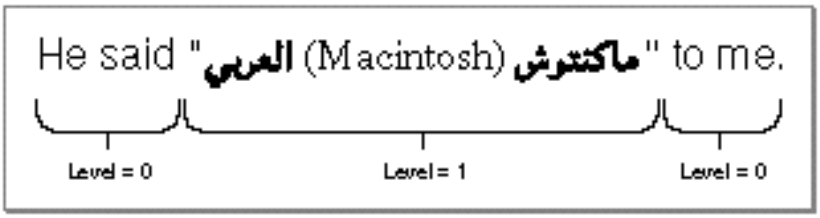
Now, however, assume that the English-language phrase “(Macintosh)” is added between the two Arabic words. To make sure that the line is ordered correctly when drawn, you must specify a right-to-left dominant direction for the line. You can do that by giving the entire layout a level of 1 (odd).The resulting layout is drawn correctly, as follows:



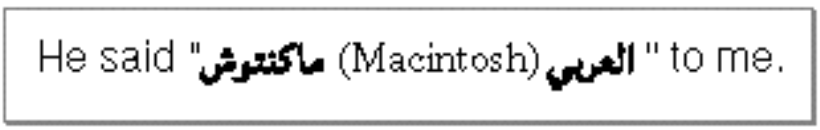
If you had not specified any level for the line, or if you had specified an even value, the two Arabic words would have been reversed— equivalent to saying “Macintosh Arabic” in English. The resulting layout would be drawn incorrectly, like this:



Finally, suppose that this predominantly Arabic phrase is made part of an English-language sentence. To preserve the overall right-to-left ordering of the phrase, you must give it an odd nesting level; to preserve the overall left-to-right ordering of the sentence itself, you must give the English parts a lower, even nesting level. The resulting layout is drawn correctly, as follows:



If you had simply given the entire sentence an even level (or specified no level at all), the Arabic words would once again have been reversed, and the sentence would have been incorrectly displayed, as follows:



Consider an even more complex example. Suppose you subsequently embedded the entire sentence in a line of Arabic text. You would then need to assign a level of 1 to those new (outer) Arabic parts and promote the levels 0 and 1 in the original sentence to levels 2 and 3. This is because QuickDraw GX looks at the lowest level to determine the overall line direction, which in this case would need be right to left (odd nesting level).

Your application can usually determine the intended dominant direction for a line when text is being entered, either from a system setting or from a user selection. However, it may not be able to generate these more complex nesting levels automatically without

user intervention. Therefore, to give users control over the orderings of complex phrases, you may want to allow them to set levels explicitly as they enter text, or to impose levels on the text they have previously created.

For more information and examples, see the section "Manipulating Nested Direction Levels" beginning on page 9-38.

Justification

Justification is the process of typographically fitting a line of text to a given width (or height, in the case of vertical text). In QuickDraw GX, some of the information that controls justification behavior is contained in the layout shape itself, whereas other information is contained in the style objects associated with the layout shape.

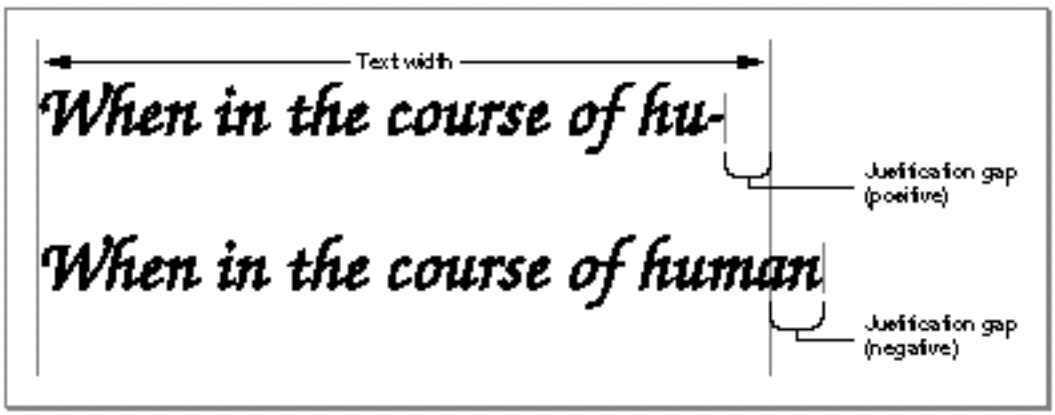
This section presents the QuickDraw GX justification model, notes where the information that controls it is stored, and discusses how to override aspects of it to produce special justification effects.

The Justification Model

The QuickDraw GX justification model is very powerful and completely multilingual. It supports the assignment of additional space to different classes of glyphs at different priority levels, and includes facilities for handling complex kashida-like justification such as is used in Arabic or script Roman. This section describes how the model works.

The **justification gap** is the difference in the length of a line of text before and after justification. For full justification, it is the difference between the text width and the length of the unjustified line (without considering hanging punctuation and optical effects, described in the chapter "Layout Styles" in this book). If the line must be stretched to fit the allotted space, the justification gap is positive and must be distributed among the glyphs of the line as extra width. If the line must be shrunk to fit, the justification gap is negative and that space must therefore be removed from the glyphs of the line. (See Figure 9-14.) QuickDraw GX can handle both positive and negative justification gaps and can even use different behaviors in the positive and negative cases.

Figure 9-14 Justification gap



Remember that, as noted in the chapter “Layout Shapes” in this book, justification is a continuous value that is specified in the `just` field of the layout options structure. Figure 9-14 illustrates the justification gap for a line that is to be fully justified (`just` = 1.0); the justification gap is the entire difference between line length and text width. If the line were to be only partially justified, the justification gap would be proportionally smaller. For example, if the line in Figure 9-14 were to be 50 percent justified (`just` = 0.5), the justification gap would be half the amount shown in the figure. QuickDraw GX supports partial or full justification, of both positive and negative justification gap, in all situations.

Justification Priority, Grow Limits, and Shrink Limits

Justification as performed by QuickDraw GX is a multistage process. QuickDraw GX does not have to, for example, assign intercharacter and interword white space in a fixed proportion when stretching a line. Instead, QuickDraw GX recalculates glyph positions in several passes, based on **justification priority.** Glyphs with higher priority are processed earlier. If, after processing all glyphs of a given priority, more justification gap remains, QuickDraw GX then processes glyphs of the next lower priority, and so on, until all the needed justification gap is taken up. The defined justification priorities are listed in Table 9-1.

Table 9-1 Justification priorities

Constant	Value	Explanation
`gxKashidaPriority`	0	The highest priority. Glyphs with this priority are adjusted first.
`gxWhiteSpacePriority`	1	Glyphs with this priority are adjusted after all glyphs with priority `gxKashidaPriority`.
`gxInterCharPriority`	2	Glyphs with this priority are adjusted after all glyphs with higher priority.
`gxNullJustificationPriority`	3	Glyphs with this priority have the lowest justification priority and are adjusted last.

Note

The justification priorities specify only the order in which glyphs participate in justification. The names of the priorities suggest the kinds of effects that are typical at each level, but the actual justification technique that QuickDraw GX applies—for example, addition of kashida extenders or white space—is defined for each glyph by the font. u

As an example of the steps involved in justification, in Roman fonts the whitespace glyphs typically have a higher justification priority than other glyphs. During justification, QuickDraw GX first assigns extra (interword) space to those glyphs alone, until

either the entire justification gap is accounted for or a specified **grow limit** for the justification of whitespace glyphs is reached. At that point, if more gap remains, QuickDraw GX then starts assigning extra (intercharacter) space to other glyphs. For this reason, intercharacter spacing need not occur as often as it does in proportional justification models (or even at all, if your application wishes).

Tables in the glyphs' font determine the assignment of justification priorities to glyphs and the specification of limits to the amount of width that can be added for each priority. Each glyph of a given priority can be stretched by a given amount on its right side and by a possibly different amount on its left side, after which processing passes to the next lower priority of glyph. (Note that "stretching" in this case can mean addition of white space, addition of connecting glyphs, such as kashidas, actual stretching of the glyph form itself, or other methods of taking up the space.) Likewise, each glyph of a given priority has **shrink limits,** used when the justification gap is negative. Shrink limits can be different on the left and right sides of a glyph and different from the grow limits.

When it assigns a certain amount of space to a glyph (or subtracts space), QuickDraw GX looks at the glyph's left and right grow limits (or shrink limits) and adds the space (or subtracts it) on each side, until those limits are reached.

Sometimes a justification gap remains even after QuickDraw GX has processed all priorities of glyphs on the line and has stretched each by the maximum amount. In this case, QuickDraw GX goes back to the highest priority glyphs on the line and distributes the remaining gap among them.

A font can specify that certain glyphs have **unlimited gap absorption,** meaning that, at the point during justification at which they are processed, all remaining justification gap is assigned to them.

The font also specifies, for each glyph, the actual technique of justification that must be applied to it. For example, most glyphs in most Roman fonts grow by the addition of white space to both sides of the glyph. Certain Arabic glyphs grow by the addition of kashidas (extender bars) to their right sides only. Other glyphs in some fonts grow by changing their shapes— for example, a hyphen might grow or shrink when the line it is part of is justified. For examples of each of these kinds of justification, see the section "Displaying Partial Justification" beginning on page 9-46.

Justification priorities are described further on page 9-60.

Postcompensation Action

Once QuickDraw GX has completed its processing and calculated the extra space to add to all glyphs, it may either draw the line as it is, or it may perform postcompensation action on it. **Postcompensation action** consists of addition, substitution, or modification of glyphs as needed to complete the justification. In many cases, postcompensation action is not needed; the existing glyphs are drawn in their revised positions, surrounded by whitespace. However, there are several cases in which it is either necessary or useful:

- **Addition of kashidas.** Justification in Arabic involves adding extension bars to the right sides of certain glyphs. The justification calculations create the proper amount of space to hold those extension bars; postcompensation action inserts the glyphs for them. A similar postcompensation process adds connectors between glyphs for justified Roman text using cursive fonts.

- **Changing glyph shape.** Certain glyphs in some fonts contribute to the justification of a line by actually deforming, rather than by having white space added on either side. In that case, postcompensation action consists of deforming the glyph to fill its new space. Deforming may be either by simple **glyph stretching,** which uses a text face mechanism, or by **glyph ductility,** which uses a font variation mechanism. Text faces are described in the chapter "Typographic Styles" in this book; font variations are described in the chapter "Font Objects" in this book.
- **Ligature decomposition.** Depending on the amount of white space surrounding a ligature, postcompensation action may replace that ligature with its component glyphs, after which QuickDraw GX recalculates the positions of all glyphs on the line.
- **Substitution of wider glyphs.** Some fonts have wider versions of certain glyphs, such as hyphens, that postcompensation action can substitute for the original glyphs.

Overriding Justification Behavior

If you want to provide custom justification behavior, your application can override any of the font-specified priorities and limits in a given style run, either for an individual glyph or for a whole priority of glyphs. Additionally, you can specify that a given glyph or a given priority of glyph is to have unlimited gap absorption; in other words, when that glyph or priority is processed during justification, all remaining justification gap must be assigned to it, regardless of its specified shrink or grow limits.

Your application cannot in general override the kind of justification that takes place for a given glyph—addition of white space, addition of a kashida, or stretching, or ductility. You can, however, prevent postcompensation action by setting the `gxNoSpecialJustification` flag in the run controls structure of the style run to which the text belongs. You can also override the threshold at which ligature decomposition starts by placing an appropriate value in the `decompositionAdjustmentFactor` field of the run controls structure of the style run to which the text belongs. The run controls structure is described in the chapter "Layout Styles" in this book.

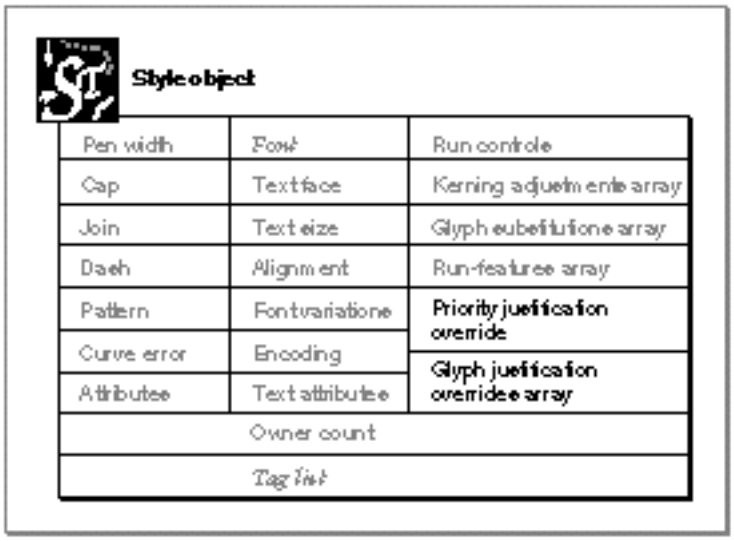
Justification Properties of the Shape Object and Style Object

Some of the information that controls justification behavior in QuickDraw GX is contained in the layout shape itself, whereas other information is contained in the style objects associated with that layout shape.

The `just` field of the layout options structure in the geometry of a layout shape contains a `fract` value between 0 and 1.0 that defines whether and to what extent a layout shape is to be justified. A value of 0 means no justification; a value of 1 means full justification; values in between specify varying degrees of justification. The layout options structure is described in the chapter "Layout Shapes" in this book.

Figure 9-15 shows the justification-related properties of the style object. Because each style run in a layout shape has its own style object, these properties, unlike the `just` value in the layout options structure, need not affect the entire layout shape.

Figure 9-15 Justification-related properties of the style object



There are three principal layout-specific properties of the style object that control justification behavior. Two of them are described in this chapter; both function as overrides to font-specified default behavior:

- **Priority justification override structure.** Each entry in this structure overrides the behavior of all glyphs of a given justification priority. If this property is set to `nil`, justification for each priority for this style run defaults to the font-specified behavior. See the next section, “Priority Justification Override,” for more information on this structure.
- **Glyph justification overrides array.** This array overrides the justification behavior of one or more individual glyphs. If this property is set to `nil`, justification for all glyphs in the style run defaults to the font-specified behavior. See the section “Glyph Justification Overrides” on page 9-26 for more information on this array.

Justification overrides basically have two effects: (1) changing the limits to which a given glyph or class of glyphs can be stretched (or shrunk) during justification, and (2) changing the justification priority of the glyph or class of glyphs. Overrides do not change the kind of justification that is applied—white space, kashida, glyph stretching, and so on.

One other layout-specific property of the style object, the run controls structure, contains one field and one flag that affect justification behavior. The field is the `decompositionAdjustmentFactor` field, and the flag is the `gxNoSpecialJustification` flag. Both affect postcompensation action, described on page 9-23. The run controls structure itself is described in the chapter “Layout Styles” in this book.

Priority Justification Override

The priority justification override structure specifies overriding justification behavior for specific classes of glyphs in a given style run. The structure is organized by priority; for each justification priority, it specifies the overriding behavior, if any. That behavior can include changes to the grow or shrink limits for that priority and even a change in priority value for that priority. The structure is a simple array of width delta structures, one for each defined justification priority.

```
typedef struct {
   gxWidthDeltaRecord   deltas[gxNumberOfJustificationPriorities];
} gxPriorityJustificationOverride;
```

Each width delta structure specifies, for both the grow and shrink cases, limits to the amount of space that can be added (or removed) from both the right and left sides of each of the glyphs of the given justification priority. This is its format:

```
typedef struct {
   Fixed                   beforeGrowLimit;
   Fixed                   beforeShrinkLimit;
   Fixed                   afterGrowLimit;
   Fixed                   afterShrinkLimit;
   gxJustificationFlags    growFlags;
   gxJustificationFlags    shrinkFlags;
} gxWidthDeltaRecord;
```

The `growFlags` and `shrinkFlags` fields control whether or not to apply the limits defined in the rest of the structure and whether or not to change other justification behavior, such as the priority itself. The flags also control whether or not unlimited gap absorption (see page 9-24) should be applied to the priority of glyphs specified in the structure. The fields of the width delta structure are described in more detail in the section “Width Delta Structure” beginning on page 9-61.

The priority justification override structure is described in more detail in the section “Priority Justification Override Structure” beginning on page 9-63.

Glyph Justification Overrides

The glyph justification overrides array specifies overriding justification behavior for individual glyphs in a given style run. It consists of an array of glyph justification override structures, one for each glyph whose behavior is to be overridden.

The glyph justification override structure assigns an overriding justification priority and behavior to a specific glyph in a style run. It contains a glyph code and a width delta structure.

```
typedef struct {
   gxGlyphcode          glyph;
   gxWidthDeltaRecord   override;
} gxGlyphJustificationOverride;
```

The width delta structure in this case specifies overrides and flags for all instances of a single glyph (specified by glyph code). The fields of the width delta structure are described in more detail in the section “Width Delta Structure” beginning on page 9-61.

The glyph justification override structure is described in more detail in the section “Glyph Justification Override Structure” beginning on page 9-64.

Using Line Control and Line Measurement With Layout Shapes

This section shows how to use QuickDraw GX functions and structures to measure and control the layout of text lines. In particular, it shows you how to

- set multiple baselines
- determine line lengths
- determine line spans
- break lines
- manipulate nested direction levels
- display continuous justification
- change the behavior of justification priorities
- change the justification behavior of glyphs

Setting Baselines

The section “Baselines” beginning on page 9-4 describes how QuickDraw GX uses font information to lay out multiple-baseline text. To take advantage of the capabilities of QuickDraw GX and thus get the best alignment when drawing a line of text that uses multiple baselines, you can use the following procedure:

1. Decide which baseline type to align the text of each style run to. For example, you may specify a Roman baseline for a Roman style run and a hanging baseline for a Devanagari style run on the same line. Set each baseline type in the `BaselineType` field of the run controls structure of the style object for that style run. (Actually, you need make no explicit assignments if you use the default baseline for the text of each style run.)
2. Determine the distances among baselines to be used for the entire line. Call the `GXGetStyleBaselineDeltas` function to get the distances (based on the font and text size of a style object, which may represent one of the style runs on the line). Store the baseline deltas information in the `baselineRec` field of the layout options structure.

When you draw the layout shape, QuickDraw GX aligns all baselines in all style runs according to the information in the baseline deltas structure.

Listing 9-1 is a partial listing of a function that aligns the hanging baselines of two different sizes of text to create drop capitals, for the string “Drop Caps”. The function creates two different style objects, one for the drop capitals and one for the lowercase letters. The function draws the line of text twice: once with the capitals at 70 points and once with capitals at 110 points (the body text is at 40 points in both cases).

Listing 9-1 makes use of application-defined functions `NewLayoutStyle`, `InitializeRunControls`, `InitializeLayoutOptions`, and `GXSetStyleTextSize` to initialize some of the objects and structures used in the listing. The length of the text string is `len`; the layout is drawn at the point `myPoint`. The function calls the QuickDraw GX function `GXGetStyleBaselineDeltas` to get the information needed for aligning the baselines.

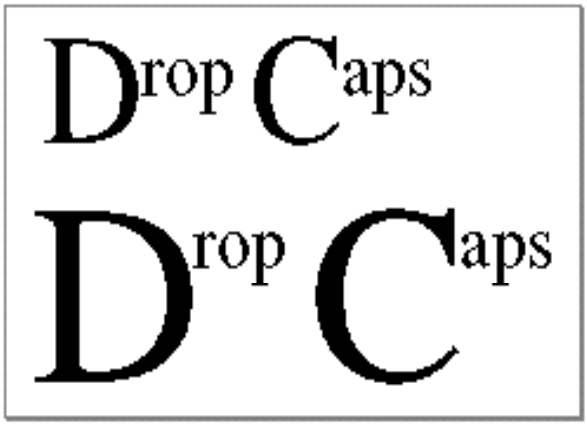
Listing 9-1 Aligning baselines to create drop capitals

```
void BaselineAlignment(WindowPtr sampleWindow)
{
   /* define and initialize variables */
   char                    *myString = "Drop Caps";
   gxLayoutOptions         layoutOptions;
   gxLineBaselineRecord    lineBaselineRecord;
   gxRunControls     runControls;
   gxShape           layout;
   short             runLengths[4];
   gxStyle           dropCapsStyle, regularStyle, styleArray[4];
   .
   .
   .
   /* set the size of the text and each of the style runs */
   runLengths[0] = runLengths[2] = 1;
   runLengths[1] = 4;
   runLengths[3] = 3;

   /* style for lowercase letters is 40-pt, no run controls */
   regularStyle = NewLayoutStyle((char *) "\pTimes Roman",
                                 ff(40), 0, nil, nil, 0, nil);

   /* set up hanging-baseline run controls for drop-cap style */
   InitializeRunControls(&runControls);
   runControls.baselineType = gxHangingBaseline;
```

```
   /* style for drop caps is 70-pt, with run controls */
   dropCapsStyle = NewLayoutStyle((char *) "\pTimes Roman",
                           ff(70), 0, &runControls, nil, 0, nil);

   /*
      Set up layout options structure for the layout shape; get
      baseline distances from Roman, based on lowercase style.
   */
   InitializeLayoutOptions(&layoutOptions);
   GXGetStyleBaselineDeltas(regularStyle, gxRomanBaseline,
                           lineBaselineRecord.deltas);
   layoutOptions.baselineRec = &lineBaselineRecord;

   /* assign styles to each style run */
   styleArray[0] = styleArray[2] = dropCapsStyle;
   styleArray[1] = styleArray[3] = regularStyle;

   /* create and draw a layout with the above text and styles */
   layout = GXNewLayout(1, &len, (void *) &myString,
                        4, runLengths, styleArray,
                        0, nil, nil,
                        &layoutOptions, &myPoint);
   GXDrawShape(layout);

   /* now modify the size of the capitals, but nothing else */
   GXSetStyleTextSize(dropCapsStyle, ff(110));

   /*move the shape and draw again; the drop caps still line up */
   GXMoveShape(layout, 0, ff(100));
   GXDrawShape(layout);
   .
   .
   .
}
```

Figure 9-16 shows the results of executing the function in Listing 9-1.

Figure 9-16 Drop capitals created by aligning baselines



The `GXGetStyleBaselineDeltas` function is described on page 9-66.

Drawing Vertical Text

Because there are no vertical baselines and no vertical line direction in QuickDraw GX, you create a vertical line by using the layout shape's transform object to rotate a horizontal line before drawing it.

To draw a layout shape as a line of vertical text, follow these steps:

1. Set the `gxVerticalText` text attribute for all style runs of the layout shape. Setting this text attribute has the effect of rotating each individual glyph by 90 degrees counterclockwise.
2. Lay out and measure the line as if it were horizontal. Caret positions, hit-testing, and measurements of line span and line length will be meaningful if you consider the line as horizontal, with rotated glyphs.
3. Call the `GXRotateShape` function or `GXRotateTransform` function to rotate the line 90 degrees clockwise when it is drawn. Because, for layout shapes, the `gxMapTransformShape` attribute is set, calling `GXRotateShape` does not affect the geometry of the layout itself; it changes only the mapping of the layout shape's transform object.
4. Draw the shape.

Listing 9-2 is a sample program that creates a layout shape several times, drawing it with and without the `gxVerticalText` text attribute set and using several baseline types.

Listing 9-2 Creating and drawing vertical text

```
   char *myString = “movemove”
   InitializeRunControls(&controls);
   controls.baselineType = gxRomanBaseline;

   myStyles[0] = NewLayoutStyle((char *) "\pHoefler Text", ff(48),
0,
            nil, nil, 0, nil);
   myStyles[1] = NewLayoutStyle((char *) "\pHoefler Text", ff(48),
            gxVerticalText, &controls, nil, 0, nil);
   GXGetStyleBaselineDeltas(myStyles[0], gxRomanBaseline,
            lbr.deltas);
   options.baselineRec = &lbr;
   myLens[0] = myLens[1] = 4;

   layout = GXNewLayout(
      1, &len, (void *) &myString,
      2, myLens, myStyles,
      0, nil, nil,
      &options, &myPoint);
   GXDrawShape(layout);

   controls.baselineType = gxIdeographicCenterBaseline;
   GXSetStyleRunControls(myStyles[1], &controls);
   GXMoveShape(layout, 0, ff(80));
   GXDrawShape(layout);

   controls.baselineType = gxHangingBaseline;
   GXSetStyleRunControls(myStyles[1], &controls);
   GXMoveShape(layout, 0, ff(80));
   GXDrawShape(layout);
```

For the results of Listing 9-2, see Figure 9-5 on page 9-9.

Some Asian languages use rotated Roman glyphs in the vertical text lines, as shown in Figure 9-17. To create such an effect, follow the above steps but put the Roman text in a separate style run and do not set the `gxVerticalText` text attribute for that style run. When you rotate and draw the shape, the Roman glyphs will then be rotated 90 degrees clockwise, as desired.

Figure 9-17 Rotated Roman glyphs in vertical text

Determining Line Lengths

You can determine the length of a line in a layout shape in at least three ways:

- Call the `GXGetShapeBounds` function to determine the standard bounding rectangle of the layout shape. This rectangle exactly encloses just the "inked" parts (the black pixels) of the displayed glyphs. The width of the rectangle is the length of the line, including any hanging punctuation and accounting for shifts due to optical alignment.
- Call the `GXGetShapeTypographicBounds` function to determine the typographic bounding rectangle of the layout shape. This rectangle spans the layout shape from the lowest descent line to the highest ascent line, regardless of whether any glyphs extend to those lines. The width of the rectangle extends from the origin of the first glyph through the advance width of the last glyph.
- Call the `GXGetLayoutRangeWidth` function, passing it a range that is the entire layout shape. The width it returns is equivalent to the width of the typographic bounds: it extends from the origin of the first glyph through the advance width of the last glyph.

The `GXGetLayoutRangeWidth` function is described on page 9-71. The `GXGetShapeBounds` function is described in the "Geometric Operations" chapter in *Inside Macintosh: QuickDraw GX Graphics*. The `GXGetShapeTypographicBounds` function is described in the chapter "Typographic Shapes" in this book.

Determining Line Spans

For drawing carets and highlighting areas, and for hit-testing, QuickDraw GX automatically calculates the proper line span (distance from the lowest descender to the highest ascender on the line) for a layout shape. Line span affects the height of caret shapes and highlight areas that QuickDraw GX calculates and returns to you, through functions such as `GXGetLayoutCaret` and `GXGetLayoutHighlight`. It also affects the area sensitive to hits for purposes of hit-testing.

QuickDraw GX provides the `GXGetLayoutSpan` function so that you can determine how far apart to space the text lines your application draws. (Line span plus any leading, or extra gap, that you add equals the line-to-line distance in multiline text.) This function returns the correct span for any line that is a layout shape. QuickDraw GX calculates a line height for the given line according to all the information it has, including all text sizes, manual and automatic position shifts, multiple baselines, and so on.

You can also use the functions `GXGetShapeBounds` and `GXGetShapeTypographicBounds` to determine line span. The height of the rectangle returned by `GXGetShapeTypographicBounds` is equivalent to the values returned by `GXGetLayoutSpan`; the value returned by `GXGetShapeBounds` may be smaller, since the bounding rectangle encloses only the black pixels of the displayed glyphs.

If your application wishes to set the line span manually and thus affect these results, use the `GXSetLayoutSpan`. Note that if you alter the line span with `GXSetLayoutSpan`, `GXGetLayoutSpan` returns the line span that you have set. If you need to recover the line span as originally calculated by QuickDraw GX, you can call `GXSetLayoutSpan` with a line span of 0.

For an example of the use of the `GXGetLayoutSpan` function for positioning line starts, see Listing 9-3 on page 9-34.

The `GXGetLayoutSpan` function is described on page 9-67. The `GXSetLayoutSpan` function is described on page 9-68. The `GXGetShapeBounds` function is described in the chapter “Geometric Operations” in *Inside Macintosh: QuickDraw GX Graphics*. The `GXGetShapeTypographicBounds` function is described in the chapter “Typographic Shapes” in this book.

Breaking Lines

QuickDraw GX text formatting is line-based, not paragraph-based. To lay out multiple lines of text, you need to determine for each line where to break the text, and then create a layout shape for that line.

The QuickDraw GX functions most used for line breaking are `GXGetLayoutBreakOffset`, `GXGetLayoutRangeWidth`, and `GXNewLayoutFromRange`. The `GXGetLayoutBreakOffset` function tells you at which point you can break a line if you want it to fit within a given text length. The `GXGetLayoutRangeWidth` function gives you the length of a range of text within a layout shape. `GXNewLayoutFromRange` function creates a new layout shape from a range of text within an existing layout shape.

The typical procedure to follow for each line to be laid out is this:

1. Call `GXGetLayoutBreakOffset`, starting with the offset in the source text that starts the line, to see how much will fit between the margins you specify.
2. Use the results of `GXGetLayoutBreakOffset` to determine the exact point in the source text at which to break the line. You can use the results of the function directly, or you can do additional forward or backward processing in the text to determine the most meaningful break point. See, for example "Using Macintosh WorldScript for Line Breaking" on page 9-37.
3. Optionally, check on your results by calling `GXGetLayoutRangeWidth` to compare the display width of the new range of text with you margins. You can account for extra characters such as hyphens when you make this calculation.
4. Call `GXNewLayoutFromRange` to create a new layout shape that represents the line. The new shape can have extra display glyphs, such as hyphens, that are not part of the source text.

Listing 9-3 is a library function (`NewStyledParagraph`) that creates a set of individual layout shapes, each representing a single line, from a larger layout shape that represents an entire paragraph. It calls `GXGetLayoutBreakOffset` to get initial line breaks and then does some simple processing on the results to determine the actual breaks. Next it calls `GXNewLayoutFromRange` to create a new layout shape for each line. When drawing the lines, it calls `GXGetLayoutSpan` to determine how far apart to separate the lines. (Leading between the lines is specified by the `extraLineGap` variable.)

Listing 9-3 uses several other library-defined functions for moving back and forth in the source text, such as `GetTextPiecePtr`, `GetPreviousOffset`, and `GetNextOffset`. It also constructs a library-defined data structure called a `ParagraphRecord` that holds information about the paragraph, such as the number of lines and a reference to each layout shape making up the paragraph. Some variables used here, such as `lineStartsCount`, are global to this function.

Listing 9-3 Breaking a Roman layout shape into individual lines of a paragraph

```
ParagraphRecordHandle NewStyledParagraph(
                            long textRunCount,
                            const void *text[],
                            const short textRunLengths[],
                            long styleRunCount,
                            const gxStyle styles[],
                            const short styleRunLengths[],
                            long levelRunCount,
                            const short levels[],
                            const short levelRunLengths[],
                            long totalByteCount,
                            const gxLayoutOptions *layoutOptions,
                            Fixed lineHeight,
                            const gxPoint *firstOrigin)
```

```
{
   /* define & initialize variables */
   boolean                 startIsStaked;
   gxByteOffset            lineStarts[lineStartsCount],
                           newLineStart, nextStake,
                           nls2, priorStake, thisLineStart;
   char                    *pChar, *pSav;
   Fixed                   currLineDelta, lineAscent, lineDescent;
   gxLayoutOptions         specialOptions;
   ParagraphRecordHandle   paraHandle;
   gxShape                 bigLayout, thisLine;
   short                   i, level = 0, nextLineIndex;

   /* set up for left-aligned, unjustified text */
   specialOptions = *layoutOptions;
   specialOptions.just = specialOptions.flush = 0;
   specialOptions.width = 0;

   /* first, create a “big” layout for the whole paragraph */
   bigLayout = GXNewLayout(textRunCount, textRunLengths, text,
                           styleRunCount, styleRunLengths, styles,
                           levelRunCount, levelRunLengths, levels,
                           &specialOptions, firstOrigin);
   /*
      Next, compute all the line breaks for the paragraph
      and store their offsets in a temporary array.
   */
   thisLineStart = 0;
   nextLineIndex = 0;
   while (thisLineStart < totalByteCount &&
                           nextLineIndex < lineStartsCount - 1)
   {
      lineStarts[nextLineIndex++] = thisLineStart;

      /*
         There is no hyphenation array, so the break is marked at
         the last glyph that fits on the line, regardless of
         its position in a word.
      */
      newLineStart = GXGetLayoutBreakOffset(bigLayout,
                                       thisLineStart,
                                       layoutOptions->width,
                                       0, nil, &startIsStaked,
                                       &priorStake, &nextStake);
      if (newLineStart == totalByteCount) break;
```

```
      /*
         Backtrack to first prior space before end of line,
         to rebreak line at word boundary. Your application
         should substitute a more sophisticated line breaking
         algorithm here.
      */
      nls2 = newLineStart;
      pSav = pChar = GetTextPiecePtr(text, textRunLengths,
                     GetPreviousOffset(bigLayout, newLineStart));
      while (nls2 >= lineStarts[nextLineIndex-1] && *pChar != ' ')
         pChar = GetTextPiecePtr(text, textRunLengths,
                     nls2 = GetPreviousOffset(bigLayout, nls2));

      /* if we've backed all the way up to the beginning of the
         line, use the line break originally returned from
         GXGetLayoutBreakOffset. Otherwise, take the new offset.
      */
      if (nls2 <= lineStarts[nextLineIndex-1])
         thisLineStart = newLineStart;
      else
      {
         if (pSav != pChar) nls2 = GetNextOffset(bigLayout, nls2);
         thisLineStart = nls2;
      }
   }

   /* put the last line-start entry beyond the end of the text */
   lineStarts[nextLineIndex] = (short) totalByteCount;

   /* now allocate space for the ParagraphRecord */
   paraHandle = (ParagraphRecordHandle) NewHandle(
               (Size) (sizeof(ParagraphRecord) +
               nextLineIndex * sizeof(gxShape)));
   (*paraHandle)->nLayouts = nextLineIndex;  /* No. of lines */

   /*
      Now create layout shapes for each of the lines
      and put them in the ParagraphRecord.
   */
   specialOptions = *layoutOptions;
   currLineDelta = 0;
```

```
   for (i = 0; i < nextLineIndex; i++)
   {
      /* don't justify the last line */
      if (i == nextLineIndex - 1) specialOptions.just = 0;

      /* create the layout for this line */
      thisLine = GXNewLayoutFromRange(bigLayout, lineStarts[i],
                           lineStarts[i+1], &specialOptions, nil);

      /* move the line downward by the proper amount */
      if (currLineDelta) GXMoveShape(thisLine, 0, currLineDelta);

      /* add this line to the paragraph record */
      (*paraHandle)->layouts[i] = thisLine;

      /* calculate amount by which to move next line down */
      if (lineHeight) currLineDelta += lineHeight;
      else
      {
         GXGetLayoutSpan(thisLine, &lineAscent, &lineDescent);
         currLineDelta += lineAscent + lineDescent + extraLineGap;
      }
   }
   (*paraHandle)->totalHeight = currLineDelta;

   GXDisposeShape(bigLayout); /* get rid of the big layout */
   return paraHandle;         /* use the paragraph record */
}
```

The `GXGetLayoutBreakOffset` function is described on page 9-69. The `GXGetLayoutRangeWidth` function is described on page 9-71.
The `GXNewLayoutFromRange` function is described on page 9-72.

Using Macintosh WorldScript for Line Breaking

QuickDraw GX and the fonts it uses have no information on how to break words according to the rules of any language. Your application must make the decision on where to end a line meaningfully.

However, Macintosh system software includes a group of managers, extensions, and resources known collectively as **WorldScript.** WorldScript was created to allow multi-language text processing and includes the Script Manager, Text Utilities, Text Services Manager, international resources, and other components. The typographic capabilities of QuickDraw GX replace much of the functionality of WorldScript. Nevertheless, you may still find parts of WorldScript useful to help you with line breaking.

The key WorldScript components for line breaking are the Text Utilities `FindWordBreaks` procedure and the string-manipulation resource (type `'itl2'`), one of the WorldScript international resources. Every script system supplied with Macintosh computers includes a string-manipulation resource; that resource contains line-break information, specific to that script system, that the `FindWordBreaks` procedure uses when breaking lines.

To use WorldScript along with QuickDraw GX for line breaking, you can use a procedure like the following:

1. Use `GXGetLayoutBreakOffset` to determine the last glyph of the layout shape that fits into the available width.
2. Pass the edge offset equivalent to the trailing edge of that glyph to the `FindWordBreaks` procedure, passing also the script code that defines the script system to which the text in that style run belongs.
3. In response, `FindWordBreaks` returns the edge offsets of the nearest word boundaries before and after the offset you pass in, according to information in the given script's string-manipulation resource. Normally, you would use the "before" offset to define the end of your line, although the "after" offset is also possible, given the justification model's handling of the shrink case.

Another possible approach is to analyze the last style run in your line before calling `GXGetLayoutBreakOffset`. You could use `FindWordBreaks` repeatedly to build a hyphenation array and then pass that array to `GXGetLayoutBreakOffset`; in that case, `GXGetLayoutBreakOffset` will return an acceptable break point in the language of that text.

IMPORTANT

Script codes for WorldScript are different from the script codes used in the encoding property of QuickDraw GX style objects. For example, in WorldScript, 0 is Roman script and 1 is Japanese; for QuickDraw GX, 0 is no script, 1 is Roman, and 2 is Japanese. Do not use any routines from WorldScript that presuppose a GrafPort because QuickDraw GX does not directly use GrafPorts. s

The `FindWordBreaks` procedure is described in the chapter "Text Utilities" in *Inside Macintosh: Text*. The string-manipulation resource is described in the appendix "International Resources" in *Inside Macintosh: Text*.

Manipulating Nested Direction Levels

To force a dominant direction on a line of text or on a phrase within a line, you can assign it a direction level. You define a run for it and assign that run a level in the levels array of the layout shape geometry.

- For lines with uniform left-to-right text, or left-to-right text with isolated embedded phrases of right-to-left text, you can ignore the levels array completely. Alternatively, you can define one run for the entire layout and assign it any even value (0 is most efficient).

- For lines with uniform right-to-left text, or right-to-left text with isolated embedded phrases of left-to-right text, you can define one run for the entire layout and assign it any odd value (1 is most efficient).
- For lines with complex (mixed-direction) phrases of one dominant direction embedded in a line of opposite dominant direction, you need to define a direction run and level for each phrase whose dominant direction must be preserved. An even level causes the dominant direction for that phrase to be left to right; an odd level causes the dominant direction to be right to left. The dominant direction of the line as a whole is defined by the lowest level (odd or even) on the line.

QuickDraw GX permits up to 15 nested direction levels, although a line that uses more than 2 or 3 is quite rare.

Listing 9-4 is a partial listing showing a simple, static example that defines a levels array and assigns levels to it to preserve the proper ordering of a line of mixed-direction text. The line combines Arabic and English and contains five phrases. The line is drawn at the location `posn`.

Listing 9-4 Defining nested direction levels for a line of text

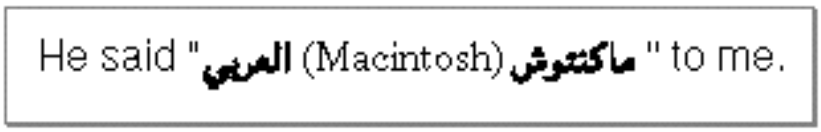
```
gxShape           layout;
gxStyle           timesStyle, helveticaStyle, baghdadStyle;
gxStyle           textStyles[5];
char              *textRuns[5];
short             textLengths[5], totalLength;
static short      levelRunLengths[3], levels[3];

/* define the 5 separate text strings for the line */
char *text1 = "He said “";
/*
   The following is "Macintosh" in Arabic:
   meem, alif, kaf,  noon, tah,  wau,  shin
*/
static char text2[] =
               {0xE5, 0xC7, 0xE3, 0xE6, 0xCA, 0xE8, 0xD4, 0};
char *text3 = " (Macintosh) ";
/*
   The following is "Arabic" in Arabic:
   alif, lam,  ein,  reh,  beh,  yeh
*/
static char text4[] = {0xC7, 0xE4, 0xD9, 0xD1, 0xC8, 0xEA, 0};
char *text5 = "” to me.";
.
.
.
```

```
/* Initialize the text runs to pass into GXNewLayout */
textRuns[0] = text1;
textRuns[1] = text2;
textRuns[2] = text3;
textRuns[3] = text4;
textRuns[4] = text5;

textLengths[0] = strlen (text1);
textLengths[1] = strlen (text2);
textLengths[2] = strlen (text3);
textLengths[3] = strlen (text4);
textLengths[4] = strlen (text5);

/*
   Initialize the direction levels arrays: here's where the
   lengths of the nested direction levels are defined.
   The first direction run is the first phrase; the second
   direction run is the second 3 phrases; and the third
   run is the last phrase.
*/
levelRunLengths[0] = textLengths[0];
levelRunLengths[1] = textLengths[1] + textLengths[2] +
                                          textLengths[3];
levelRunLengths[2] = textLengths[4];

totalLength = levelRunLengths[0] + levelRunLengths[1] +
                                        levelRunLengths[2];

/*
   Define the levels for each of the direction runs. The
   desired dominant direction is left to right; and so and the
   first and last runs have a level of 0; the central phrase
   must remain right to left, so it has a level of 1.
*/
levels[0] = 0;
levels[1] = 1;
levels[2] = 0;

/* define some styles; first is 36 pt. Helvetica */
helveticaStyle = NewLayoutStyle((char *) "\pHelvetica",
                                ff(36), 0, nil,
                                nil, 0, nil);
```

```
/* second is 36 pt. Baghdad Plain (Arabic) */
baghdadStyle = NewLayoutStyle((char *) "\pBaghdad Plain",
                              ff(36), 0, nil,
                               nil, 0, nil);

/* third is 36 pt. Times */
timesStyle = NewLayoutStyle((char *) "\pTimes Roman",
                              ff(36), 0, nil,
                              nil, 0, nil);

/* assign the styles to the style runs */
textStyles[0] = helveticaStyle;
textStyles[1] = baghdadStyle;
textStyles[2] = timesStyle;
textStyles[3] = baghdadStyle;
textStyles[4] = helveticaStyle;

/* now build and draw the layout */
layout = GXNewLayout(5, textLengths, (void *)textRuns,
                     5, textLengths, textStyles,
                     3, levelRunLengths, levels,
                     nil, &posn);
GXDrawShape (layout);

.
.
.
```

Figure 9-18 shows the results of executing the code in Listing 9-4. Compare this figure and the levels defined in Listing 9-4 with the phrases and levels shown in the section “Forced Reordering With Nested Direction Levels” beginning on page 9-19.

Figure 9-18 A text line with nested direction levels



The levels array is described in the chapter “Layout Shapes” in this book.

Overriding the Glyph Direction in a Style Run

Listing 9-5 shows an example of overriding the glyph direction in a style run, to make the individual glyphs in a sequence appear in reverse order. This changing of glyph direction is useful only for special effects; it is entirely different from imposing a dominant direction on a text run, which you do by assigning a direction level to it.

The function in Listing 9-5 creates a layout shape named `layout`; the shape uses a style object named `myStyle`. The function uses the library function `InitializeRunControls` to initialize a run controls structure, and the library function `NewLayoutStyle` to create and initialize a style object. The function initially draws the shape at the point `myPoint`. The length of the string composing the layout shape is `len`.

Listing 9-5 Overriding the glyph direction in a style run

```
{
char              *myString = "QuickDraw GX";
gxRunControls     runControls;
.
.
.
InitializeRunControls(&runControls);

/* create the style object and the layout shape */
myStyle = NewLayoutStyle((char *) "\pTimes Roman", ff(24),
                         0, nil, nil, 0, nil);

layout = GXNewLayout(1, &len, (void *) &myString,
                     1, &len, &myStyle,
                     0, nil, nil,
                     nil, &myPoint);
GXDrawShape(layout);

/* override the glyph direction in the run controls flags */
runControls.flags = gxImposeRightToLeft;
GXSetStyleRunControls(myStyle, &runControls);
GXMoveShape(layout, 0, ff(48));
GXDrawShape(layout);
.
.
.
}
```

Figure 9-19 shows the results of executing the code in Listing 9-5. Note that in this case the dominant direction for the line is still left-to-right, but the individual glyph directions are now all right to left.

Figure 9-19 Results of overriding glyph direction

Justifying Lines by Stretching and Shrinking

Listing 9-6 shows a simple example of justifying lines of different lengths to a single text width. The example first draws three unjustified lines. It then defines the text width to be equal to the width of the middle-length line, and specifies full justification for all lines. QuickDraw GX stretches or compresses the lines to fit.

The function in Listing 9-6 uses a single layout shape (`layout`) and a single style object (`myStyle`) for all three lines. It uses the layout options structure `layoutOptions`. It draws the first line at the location `myPoint`. It uses the function `GXGetShapeTypographicBounds` to determine the (unjustified) width of the middle text line.

Listing 9-6 A simple justification example

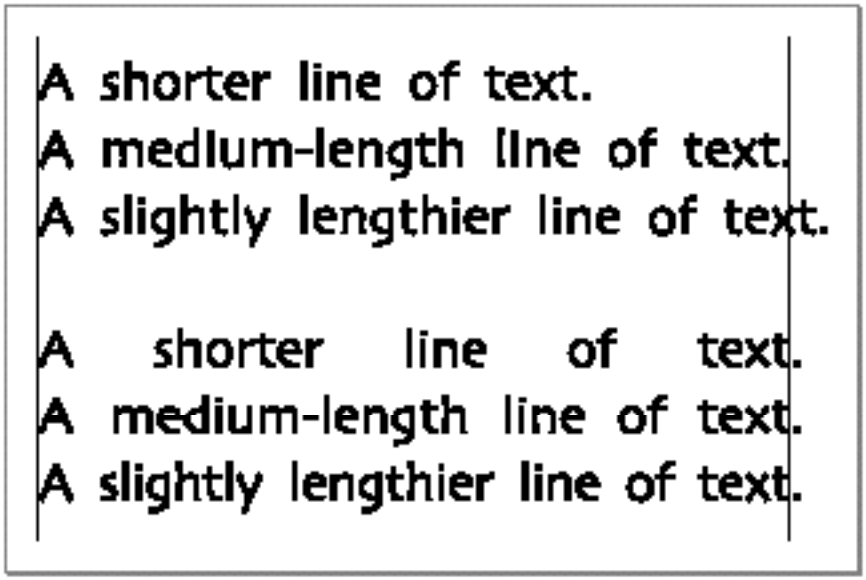
```
{
   char        *myString3 = "A shorter line of text";
   char        *myString = "A medium-length line of text";
   char        *myString2 = "A slightly lengthier line of text";
   gxLine      myLine;
   gxPoint     advance, myPoint;
   short       len;
   gxRectangle bounds;
   .
   .
   .
   /* draw left margin */
   myLine.first.x = myLine.last.x = myPoint.x;
   myLine.first.y = ff(0);
   myLine.last.y = ff(600);
   GXDrawLine(&myLine);
```

```
/* set up and draw the short line, unjustified */
myStyle = NewLayoutStyle((char *) "\pSkia Regular", ff(24),
                         0, nil, nil, 0, nil);
layoutOptions.just = 0;
len = strlen(myString3);
layout = GXNewLayout(1, &len, (void *) &myString3,
                         1, &len, &myStyle,
                         0, nil, nil,
                         &layoutOptions, &myPoint);
GXDrawShape(layout);

/* draw middle line, unjustified */
len = strlen(myString);
GXSetLayout(layout, 1, &len, (void *) &myString, 0,
            nil, nil, 0, nil, nil, nil, nil);
GXMoveShape(layout, 0, ff(36));
GXDrawShape(layout);

/* draw right margin at normal width of middle line */
GXGetShapeTypographicBounds(layout, &bounds);
myLine.first.x = myLine.last.x =
                     myPoint.x + bounds.right - bounds. left;
GXDrawLine(&myLine);

/* draw third line, unjustified */
len = strlen(myString2);
GXSetLayout(layout, 1, &len, (void *) &myString2, 0,
            nil, nil, 0, nil, nil, nil, nil);
GXMoveShape(layout, 0, ff(36));
GXDrawShape(layout);

/* set width and justification of layout options structure */
layoutOptions.just = fract1;
layoutOptions.width = bounds.right - bounds. left;

/* draw all three lines again, fully justified this time */
len = strlen(myString3);
GXSetLayout(layout, 1, &len, (void *) &myString3, 0,
            nil, nil, 0, nil, nil, &layoutOptions, nil);
GXMoveShape(layout, 0, ff(60));
GXDrawShape(layout);
```

```
len = strlen(myString);
GXSetLayout(layout, 1, &len, (void *) &myString, 0,
            nil, nil, 0, nil, nil, &layoutOptions, nil);
GXMoveShape(layout, 0, ff(36));
GXDrawShape(layout);

len = strlen(myString2);
GXSetLayout(layout, 1, &len, (void *) &myString2, 0,
            nil, nil, 0, nil, nil, &layoutOptions, nil);
GXMoveShape(layout, 0, ff(36));
GXDrawShape(layout);
.
.
.
}
```

Figure 9-20 shows the results of executing the code in Listing 9-6. The upper three lines are not justified; the lower three lines are fully justified, to a width that matches the unjustified length of the second line. In the lower three lines, the third line is compressed, and the first line is stretched, to accommodate the justification gap. The justified text in the lower three lines does not include the periods because a period is considered a hanging glyph in this font.

Figure 9-20 Unjustified (upper) and justified (lower) lines of different lengths



Displaying Partial Justification

Justification in QuickDraw GX is continuous, rather than just “on” or “off.” You specify the amount of justification in the `just` field of the layout options structure in the layout shape geometry. (Justification amounts between 0 and 100 percent are sometimes called *ragged justification*.)

This section illustrates several different kinds of justification. It shows how they function differently and how the appearance changes as justification increases from none to full. In each case the application controls only the amount of justification that is applied. Font settings control the kind of justification that is applied, and this justification happens automatically.

Justification With White Space

Listing 9-7 is a partial listing of a sample function that illustrates how continuous justification works in Roman text. It draws the same string of left-aligned text five times: once unjustified, once 25 percent justified, once 50 percent justified, once 75 percent justified and once fully justified. In this case, justification is achieved through addition of both intercharacter and interword white space.

Listing 9-7 uses the library function `NewLayoutStyle` to create and initialize a style object. It creates a layout shape named `layout` and a style object named `myStyle`. It draws the first line at the location `myPoint`. The text string in the shape has the length `len`.

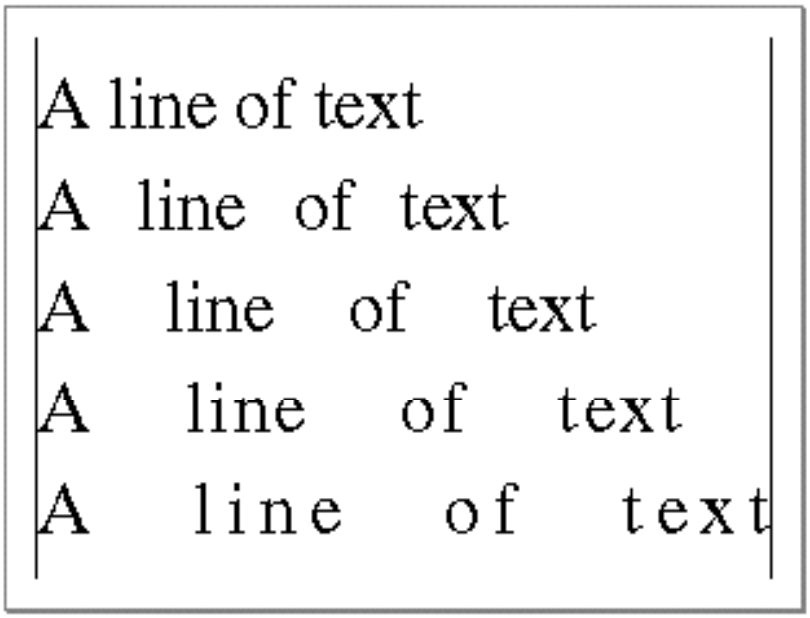
Listing 9-7 Displaying partial justification using white space

```
{
   /* define and initialize variables */
   char              *myString = "A line of text";
   gxLine            myLine;
   .
   .
   InitializeLayoutOptions(&layoutOptions);
   layoutOptions.width = ff(500);

   /* draw two vertical lines to mark the margins */
   myLine.first.x = myLine.last.x = myPoint.x;
   myLine.first.y = 0;
   myLine.last.y = ff(1000);
   GXDrawLine(&myLine);
   myLine.first.x = myLine.last.x = myPoint.x +
                                    layoutOptions.width;
   GXDrawLine(&myLine);
```

```
    /* create and draw the layout shape (unjustified) */
    myStyle = NewLayoutStyle((char *) "\pTimes Roman", ff(36), 0,
                              nil, nil, 0, nil);

    layout = GXNewLayout(1, &len, (void *) &myString,
                         1, &len, &myStyle,
                         0, nil, nil,
                         nil, &myPoint);
    GXDrawShape(layout);

    /* give the shape 25% justification and redraw */
    layoutOptions.just = fract1 / 4;
    GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
                &layoutOptions, nil);
    GXMoveShape(layout, 0, ff(54));
    GXDrawShape(layout);

    /* give the shape 50% justification and redraw */
    layoutOptions.just = fract1 / 2;
    GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
                &layoutOptions, nil);
    GXMoveShape(layout, 0, ff(54));
    GXDrawShape(layout);

    /* give the shape 75% justification and redraw */
    layoutOptions.just = 3 * (fract1 / 4);
    GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
                &layoutOptions, nil);
    GXMoveShape(layout, 0, ff(54));
    GXDrawShape(layout);

    /* give the shape 100% (full) justification and redraw */
    layoutOptions.just = fract1;
    GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
                &layoutOptions, nil);
    GXMoveShape(layout, 0, ff(54));
    GXDrawShape(layout);
    .
    .
    .
}
```

Figure 9-21 shows the results of executing the code in Listing 9-7.

Figure 9-21 Five degrees of justification with white space



Justification With Kashidas

Listing 9-8 shows a few fragments of a sample function that illustrates how continuous justification works in Arabic text. The function draws the same string of right-aligned text five times: once unjustified, once 25 percent justified, once 50 percent justified, once 75 percent justified and once fully justified. In this case, justification is achieved through addition of kashidas to certain glyphs.

The code in Listing 9-8 is essentially identical to that in Listing 9-7 on page 9-46, with the exception of the fragments listed here. No different coding is necessary to cause Arabic justification than is used to cause justification with white space.

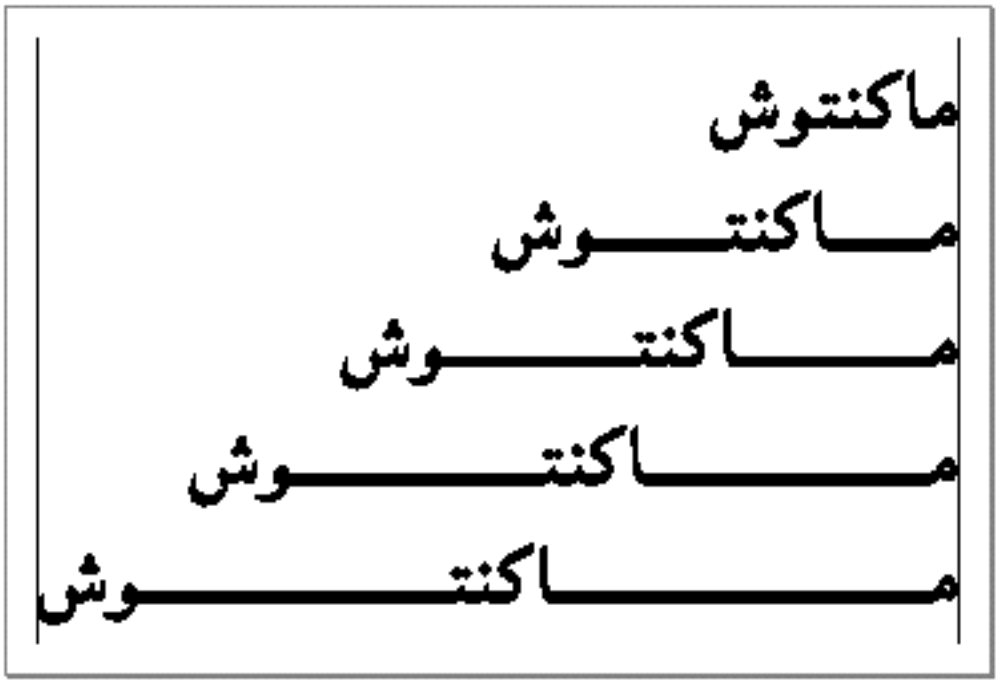
Listing 9-8 Displaying partial justification with kashidas

```
static char arabicString[8] = {'\xE5', '\xC7', '\xE3', '\xE6',
                               '\xCA', '\xE8', '\xD4', 0};
void ExtenderBars(WindowPtr sampleWindow)
{
   /* define and initialize variables */
   char             *myString = (char *) &arabicString[0];
   gxLine           myLine;
   .
   .
   .
```

```
    layoutOptions.flush = fract1; /* right-aligned */
    .
    .
    .
    /* create the style for the layout shape */
    myStyle = NewLayoutStyle((char *) "\pDiwan", ff(36),
                             noMetricsGridText, nil, nil, 0, nil);
    .
    .
    .
    /* create the shape and draw it at various justifications */
    .
    .
    .
}
```

Figure 9-22 shows the results of executing the function in Listing 9-8.

Figure 9-22 Five degrees of justification with kashidas

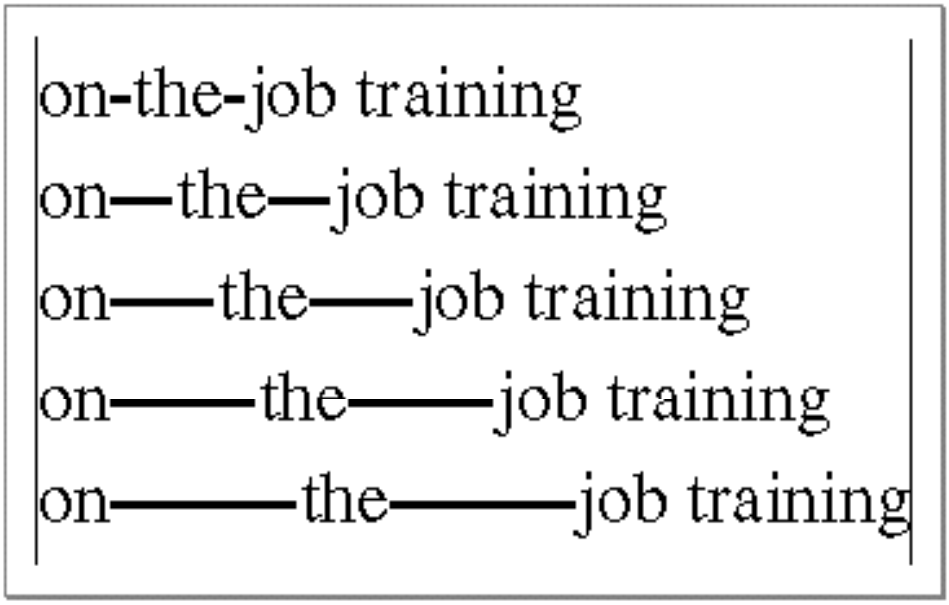


Justification With Glyph Deformation

Figure 9-23 shows the results of executing a sample program similar to the others in this section. The figure shows a line of text drawn with successively increasing justification values. In this case, all justification is by intercharacter space; however, one glyph (the hyphen) acts differently from the other glyphs. Instead of any glyphs gaining white space on either side, the hyphen gradually stretches to take up all of the justification gap.

The font in this example uses glyph stretching, which employs a text face mechanism to widen the glyph. Other fonts may use glyph ductility, which is a font-variation mechanism, to achieve similar results.

Figure 9-23 Glyph stretching during increasing justification



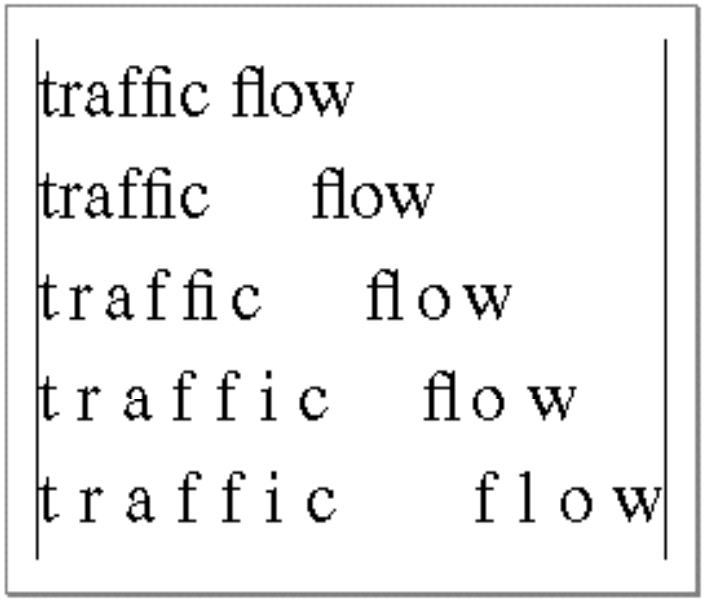
The application does not specifically request or control this behavior; it is controlled by the particular font used in this sample. However, because glyph stretching and glyph ductility are part of postcompensation action, you can prevent them from happening by setting the `gxNoSpecialJustification` flag in the run controls structure of the style run to which the text belongs. Postcompensation action is described on page 9-23; the run controls structure is described in the chapter “Layout Styles” in this book.

Justification and Ligature Decomposition

Figure 9-24 shows the results of executing another sample program that draws a line of text with successively increasing justification values. In this case, justification is by both interword and intercharacter space. But in addition, once the added white space reaches a certain threshold (equivalent to somewhere between 50 percent and 75 percent justifica-

tion in this example), the “fi” and “fl” ligatures decompose into their component glyphs, which then spread apart to take up the justification gap. Note that the “fi” ligature decomposes before the “fl” ligature in this font.

Figure 9-24 Ligature decomposition during increasing justification



The application does not specifically request this behavior; the font governs ligature decomposition during justification. However, you can control it in two ways. First, because ligature decomposition is part of postcompensation action, you can prevent it from happening by setting the `gxNoSpecialJustification` flag in the run controls structure of the style run to which the text belongs. Second, you can modify the threshold at which it occurs by placing a value in the `decompositionAdjustmentFactor` field of the run controls structure. For more information on the `decompositionAdjustmentFactor` field, see the chapter “Layout Styles” in this book.

Postcompensation action is described on page 9-23. The run controls structure is described in the chapter “Layout Styles” in this book.

The `just` field of the layout options structure is described in the chapter “Layout Shapes” in this book.

Changing the Behavior of Justification Priorities

A priority justification override structure contains the optional overrides for all priority levels that might occur within a single style run. The `deltas` array contains the width delta structures for each priority level. This is the primary input to the `GXSetStyleRunPriorityJustOverride` function.

For example, assume the user wants to override the normal (font-specified) justification behavior so that intercharacter spacing has the same priority as interword spacing when extra space must be distributed in the line. The following code fragment would produce the desired effect. Assume that there is a priority justification override structure called `overrides`, that the style to be affected is called `targetStyle`, and that the grow limits for glyphs with whitespace priority are `whiteSpaceBeforeLimit` and `whiteSpaceAfterLimit`.

```
myFlags = gxOverrideLimits | gxOverridePriority |
                                                gxWhiteSpacePriority;
overrides.deltas[gxInterCharPriority].growFlags = myFlags;
overrides.deltas[gxInterCharPriority].beforeGrowLimit =
                                                whiteSpaceBeforeLimit;
overrides.deltas[gxInterCharPriority].afterGrowLimit =
                                                whiteSpaceAfterLimit;
GXSetStyleRunPriorityJustOverride(targetStyle, &overrides);
```

As another example, Listing 9-9 is a partial listing of a sample function that demonstrates how to override justification priorities. It draws four strings, three of which are fully justified, as four separate layout shapes. (All four strings have nearly the same number of characters.) Two of the shapes have the default (font-specified) justification behavior. A third layout shape has a justification priority override that forces all justification adjustments to be made to interword spaces. A fourth has a justification priority override that forces all justification adjustments to be made to intercharacter spaces.

Listing 9-9 uses the run-controls structure `controls`, the layout-options structure `layoutOptions`, and the priority-justification override structures `allToSpace` and `allToChar`. It draws the text lines starting at the location `posn`.

Listing 9-9 Overriding justification priorities

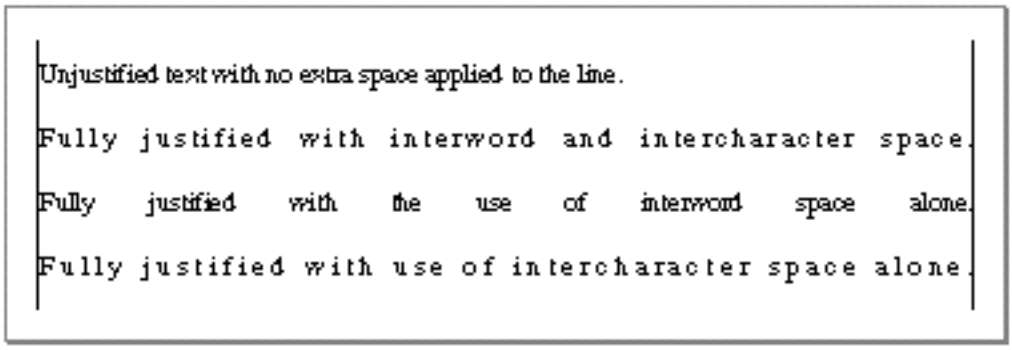
```
.
.
.
gxShape     layout1, layout2, layout3, layout4;
gxStyle     style1, style2, style3, style4;
gxPriorityJustificationOverride  allToSpace, allToChar;
char *text1 =
   "Unjustified text with no extra space applied to the line.";
char *text2 =
   "Fully justified with interword and intercharacter space.";
```

```
char *text3 =
   "Fully justified with the use of interword space alone.";
char *text4 =
   "Fully justified with use of intercharacter space alone.";
.
.
.
/*
   Set up the “all to space” override. Set unlimited gap
   absorption for the grow flag for white space priority, so
   that white space characters will absorb all justification
   gap.
*/
allToSpace.deltas[whiteSpacePriority].growFlags |=
                  (overrideUnlimited | unlimitedGapAbsorption);

/*
   Set up the “all to intercharacter” override. Set unlimited
   gap absorption for the grow flag for intercharacter
   priority,and then change the priority itself to kashida
   (the highest priority). This forces all justification
   gap to be distributed among intercharacter space, and not to
   white space.
*/
allToChar.deltas[interCharPriority].growFlags |=
               (overrideUnlimited | unlimitedGapAbsorption |
               overridePriority | kashidaPriority);

/* set up the positioning for the lines of text */
layoutOptions.width = ff(500);   /* extra width */
layoutOptions.just = fract1;     /* fully justified */

/* build a style object and a layout shape for each line */
len = strlen(text1);
style1 = NewLayoutStyle((char *) "\pTimes Roman", ff(14),
                        0, nil, nil, 0, nil);
layout1 = GXNewLayout(1, &len, (void *) &text1,
                        1, &len, &style1,
                        0, nil, nil,
                        nil, &posn);
```

```
posn.y += ff(30);
len = strlen(text2);
style2 = NewLayoutStyle((char *) "\pTimes Roman", ff(14),
                        0, nil, nil, 0, nil);
GXSetStyleRunControls(style2, &controls);
layout2 = GXNewLayout(1, &len, (void *) &text2,
                        1, &len, &style2,
                        0, nil, nil,
                        &layoutOptions, &posn);

posn.y += ff(30);
len = strlen(text3);
style3 = NewLayoutStyle((char *) "\pTimes Roman", ff(14),
                        0, nil, nil, 0, nil);
GXSetStyleRunPriorityJustOverride(style3, &allToSpace);
layout3 = GXNewLayout(1, &len, (void *) &text3,
                        1, &len, &style3,
                        0, nil, nil,
                        &layoutOptions, &posn);

posn.y += ff(30);
style4 = NewLayoutStyle((char *) "\pTimes Roman", ff(14),
                        0, nil, nil, 0, nil);
GXSetStyleRunPriorityJustOverride(style4, &allToChar);
layout4 = GXNewLayout(1, &len, (void *) &text4,
                        1, &len, &style4,
                        0, nil, nil,
                        &layoutOptions, &posn);

GXDrawShape (layout1);
GXDrawShape (layout2);
GXDrawShape (layout3);
GXDrawShape (layout4);
.
.
.
```

Figure 9-25 shows the results of executing the code in Listing 9-9.

Figure 9-25 Results of justification priority overrides on intercharacter and interword spacing



The `GXSetStyleRunPriorityJustOverride` function is described on page 9-75.

To retrieve the priority justification overrides for a style run, use the `GXGetStyleRunPriorityJustOverride` function, described on page 9-74. To retrieve the priority justification overrides for the style object associated with a particular shape, use the `GXGetShapeRunPriorityJustOverride` function, described on page 9-76. To set the priority justification overrides of the style object associated with a particular shape, use the `GXSetShapeRunPriorityJustOverride` function, described on page 9-77.

Changing Justification Behavior of Individual Glyphs

You can use one or more glyph justification override structures to assign an overriding justification priority and behavior to one or more specific glyphs in a style run. This section shows a single example, one involving the whitespace glyph.

Listing 9-10 is a partial listing of a sample function that illustrates how to use a glyph justification override to alter whitespace behavior. The function draws the same string of text three times: once unjustified, and twice fully justified. The first justified line shows the default (font-specified) justification behavior. In the second justified line, the justification of the whitespace glyph is overridden to force it to take up all justification gap.

The function in Listing 9-10 on page 9-56 creates a layout shape named `layout` and a style object named `myStyle`. It uses the layout-options structure `layoutOptions`. It draws the first line at the location `myPoint`. The text string in the shape has the length `len`.

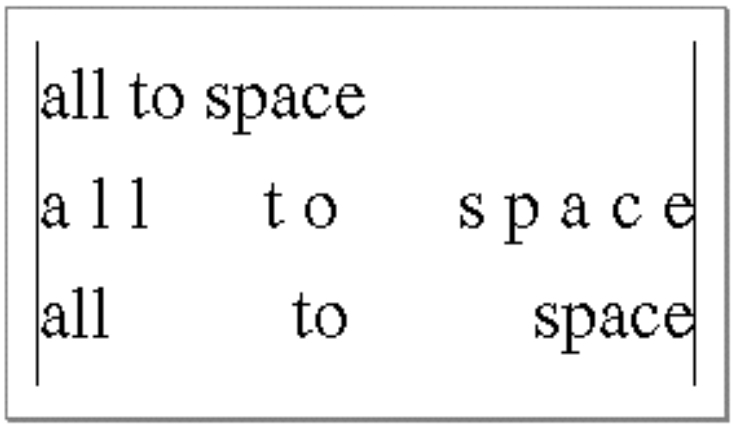
Listing 9-10 Overriding justification behavior of the whitespace glyph

```
void UnlimitedGapAbsorption(WindowPtr sampleWindow)
{
   char                          *myString = "all to space";
   gxGlyphcode                   glyphcodes[12];
   gxGlyphJustificationOverride  glyphJustOverride;
   gxLine                        myLine;
   unsigned short                firstGlyph, secondGlyph;
   .
   .
   .
   layoutOptions.width = ff(320);
   layoutOptions.just = 0;

   /* draw two vertical lines to mark the margins */
   myLine.first.x = myLine.last.x = myPoint.x;
   myLine.first.y = 0;
   myLine.last.y = ff(500);
   GXDrawLine(&myLine);

   myLine.first.x = myLine.last.x = myPoint.x +
                                    layoutOptions.width;
   GXDrawLine(&myLine);

   /* create and draw the layout shape (unjustified) */
   myStyle = NewLayoutStyle((char *) "\pTimes Roman", ff(36), 0,
                            nil, nil, 0, nil);
   layout = GXNewLayout(1, &len, (void *) &myString,
                        1, &len, &myStyle,
                        0, nil, nil,
                        &layoutOptions, &myPoint);
   GXDrawShape(layout);

   /* give the shape full justification but default behavior */
   layoutOptions.just = fract1;
   GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
               &layoutOptions, nil);
   GXMoveShape(layout, 0, ff(54));
   GXDrawShape(layout);

   /*
      Now override the default justification behavior to give
      the glyph following edge offset 3—-which in this sample
      is a whitespace glyph--unlimited gap absorption. Its
```

```
      justification priority is higher than non-space glyphs,
      so whitespace characters will absorb all justification.
   */

   /* get an array of all the glyph codes in the line */
   GXGetLayoutGlyphs(layout, glyphcodes, nil, nil, nil,
                     nil, nil, nil);

   /* find the index of the glyph following offset 3 */
   GXGetOffsetGlyphs(layout, 3, true, nil,
                     &firstGlyph, &secondGlyph);

   /* override the justification of the next glyph's glyphcode */
   glyphJustOverride.glyph = glyphcodes[firstGlyph - 1];
   glyphJustOverride.override.growFlags = overrideUnlimited +
                                          unlimitedGapAbsorption;
   glyphJustOverride.override.shrinkFlags = 0;
   GXSetStyleRunGlyphJustOverrides(myStyle, 1,
                                   &glyphJustOverride);

   /* finally, redraw the shape */
   GXMoveShape(layout, 0, ff(54));
   GXDrawShape(layout);
   .
   .
   .
}
```

Figure 9-26 shows the results of executing the code in Listing 9-10.

Figure 9-26 Results of overriding justification behavior of the whitespace glyph



Layout Line Control Reference

This section describes the constants, data types, and functions with which you can manipulate the layout of text lines using layout shapes.

Constants and Data Types

QuickDraw GX uses the constants and structures described in this section to define baselines and to override justification behavior.

Baseline Types

Text of different scripts most naturally aligns with different baselines. The `gxBaselineType` enumeration specifies the preferred baseline to use for text of a given font in a particular style run. It is specified in the `BaselineType` field of the run controls structure of each style run in a layout shape.

The baseline types enumeration provides constants for all defined baseline types. Your application need never fill in this value in the run controls structure; if you instead specify `gxNoOverrideBaseline`, QuickDraw GX determines the proper baseline type for the style run from information in the style's font.

```
enum {
   gxRomanBaseline               = 0,
   gxIdeographicCenterBaseline,
   gxIdeographicLowBaseline,
   gxHangingBaseline,
   gxMathBaseline,
   gxLastBaseline                = 31,
   gxNumberOfBaselineTypes       = gxLastBaseline + 1,
   gxNoOverrideBaseline          = 255
};

typedef unsigned long gxBaselineType;
```

Constant descriptions

`gxRomanBaseline`
: The baseline used by most Roman-script languages.

`gxIdeographicCenterBaseline`
: The most common baseline used by Chinese, Japanese, and Korean ideographic scripts, in which the ideographs are centered halfway through the line height.

`gxIdeographicLowBaseline`
: A baseline used by Chinese, Japanese, and Korean scripts. Similar to `gxIdeographicCenterBaseline`, but with the glyphs lowered. This baseline is most commonly used to align Roman glyphs within ideographic fonts to Roman glyphs in Roman fonts.

`gxHangingBaseline`
: The baseline used by Devanagari and related scripts, in which the bulk of most glyphs is below the baseline. This baseline type is also used for drop capitals in Roman scripts.

`gxMathBaseline`
: The baseline used for setting mathematics. It is centered on symbols such as the minus sign (at half the x-height).

`gxLastBaseline`
: No baseline value may exceed this value. Baseline values between `gxMathBaseline` and `gxLastBaseline` are reserved.

`gxNumberOfBaselineTypes`
: The total number of baseline types (= `gxLastBaseline` + 1).

`gxNoOverrideBaseline`
: Instructs QuickDraw GX to use the standard baseline value from the current font.

For other information about and examples of various types of baselines, see “Baseline Types” on page 9-4.

Baseline Deltas Array

The baseline deltas array (type `gxBaselineDeltas`) is used in the baseline structure, described next. Also, the `GXGetStyleBaselineDeltas` function, described on page 9-66, returns values in a baseline deltas array. Baseline deltas is an array of distances (in points) between the various baseline types and y = 0.

```
typedef Fixed gxBaselineDeltas[gxNumberOfBaselineTypes];
```

Baseline Structure

The baseline structure (type `gxLineBaselineRecord`) controls the positions of baselines with respect to one another in a line of text.

```
typedef struct {
   gxBaselineDeltas  deltas;
} gxLineBaselineRecord;
```

Field descriptions

`deltas`
: The offsets (in points) from y = 0 to every other baseline type for this line. If you are filling in this structure manually, you need to fill in only those values that correspond to the set of baselines present on the line.

You can fill in this array by calling the `GXGetStyleBaselineDeltas` function, described on page 9-66.

Justification Priorities

Glyphs in a font can be assigned justification priorities by the font designer. In general, QuickDraw GX applies justification to glyphs on a line in order of glyph priority, from high to low. Your application can override these priorities to change the order in which glyphs participate in justification. These are the defined justification priorities (lower numbers represent higher priorities):

```
enum {
   gxKashidaPriority             = 0,
   gxWhiteSpacePriority          = 1,
   gxInterCharPriority           = 2,
   gxNullJustificationPriority   = 3,
   gxNumberOfJustificationPriorities
};

typedef unsigned char gxJustificationPriority;
```

Constant descriptions

`gxKashidaPriority`
: The highest priority. Typically used for kashidas (extension bars) in Arabic. Glyphs with this priority are extended or compressed before all other glyphs in the line.

`gxWhiteSpacePriority`
: Typically assigned to whitespace (interword) glyphs. Glyphs with this priority are extended or compressed, usually by the addition or removal of white space, after all glyphs on the line with priority `gxKashidaPriority` have been extended or compressed to the maximum amount permitted.

`gxInterCharPriority`
: Assigned to all glyphs that do not have `gxKashidaPriority` or `gxWhiteSpacePriority`. Glyphs with this priority are extended or compressed, typically by the addition or removal of white space, after all glyphs on the line with priority `gxWhiteSpacePriority` have been extended or compressed to the maximum amount permitted.

`gxNullJustificationPriority`
: Available as a priority for glyphs that you want to participate in justification last of all.

`gxNumberOfJustificationPriorities`
: The number of defined justification priorities. You can use this value for range-checking, size allocation, or loop control.

Note
The justification priorities have names that describe the types of glyphs that typically have those priorities, but you can assign any priority to any glyph. The actual kind of justification that QuickDraw GX applies—for example, kashida or whitespace—is defined for each glyph by the font. The priority specifies only the order in which glyphs participate in justification. u

You can override the justification priority for an individual glyph in a style run by using the glyph justification override structure, described on page 9-64. You can override the behavior of all glyphs of a given justification priority for an entire style run in by using the priority justification override structure, described on page 9-63. In each case, you specify the justification priority in the `growFlags` or `shrinkFlags` field of one or more width delta structures. The width delta structure is described next.

Width Delta Structure

A width delta structure contains all the information needed to override the distribution behavior of a glyph or set of glyphs during justification. It is used in both the priority justification override structure and the glyph justification override structure. In each case the width delta structure can specify both a change in priority and a change in distribution behavior.

```
typedef struct {
   Fixed                   beforeGrowLimit;
   Fixed                   beforeShrinkLimit;
   Fixed                   afterGrowLimit;
   Fixed                   afterShrinkLimit;
   gxJustificationFlags    growFlags;
   gxJustificationFlags    shrinkFlags;
} gxWidthDeltaRecord;
```

Field descriptions

`beforeGrowLimit`
The number of points by which a 1-point glyph can grow on the left side (top side for vertical text). A value of 0.2, for example, means that a 24-point glyph can have by no more than 4.8 points of extra space added on the left or top side.

`beforeShrinkLimit`
The number of points by which a 1-point glyph can shrink on the left or top side. If specified, this value should be negative.

`afterGrowLimit` The number of points by which a 1-point glyph can grow on the right side (bottom for vertical text).

`afterShrinkLimit`
The number of points by which a 1-point glyph can shrink on the right or bottom side. If specified, this value should be negative.

`growFlags` Justification flags, used to control the overriding behavior when justification entails lengthening the line.

`shrinkFlags` Justification flags, used to control the overriding behavior when justification entails shortening the line.

A justification flag in the `growFlags` field controls whether or not the `beforeGrowLimit` and `afterGrowLimit` values are applied to this style run. Likewise, a flag in the `shrinkFlags` field controls whether or not the `beforeShrinkLimit` and `afterShrinkLimit` values are applied. Your application can thus selectively override certain cases (such as the grow case only), while retaining default behavior for other cases. Justification flags are described next.

Justification Flags

The justification flags control which aspects of the normal, font-specified justification behavior of a glyph or set of glyphs are to be overridden. Justification flags make up two fields in the width delta structure, described in the previous section.

```
enum {
   gxOverridePriority            = 0x8000,
   gxOverrideLimits              = 0x4000,
   gxOverrideUnlimited           = 0x2000,
   gxUnlimitedGapAbsorption      = 0x1000,
   gxJustificationPriorityMask   = 0x000F
   gxAllJustificationFlags       = gxOverridePriority|
                                   gxOverrideLimits|
                                   gxOverrideUnlimited|
                                   gxUnlimitedGapAbsorption|
                                   gxJustificationPriorityMask
};

typedef unsigned short gxJustificationFlags;
```

Constant descriptions

`gxOverridePriority`
This bit specifies whether or not QuickDraw GX overrides justification priority. If the bit is set, the justification priority in the `gxJustificationPriorityMask` part of the justification flags is used for the glyphs this width delta structure applies to. If it is cleared, QuickDraw GX uses the default justification priority for those glyphs. If it is cleared, the priority mask bits must also be set to 0. The use of this flag is to determine whether QuickDraw GX should use or override the default priority.

`gxOverrideLimits`
This bit specifies whether or not QuickDraw GX overrides grow and shrink limits. If the bit is set, the grow and shrink limits in

the width delta structure are used for the glyphs this width delta structure applies to. If it is cleared, QuickDraw GX uses the default grow and shrink limits for those glyphs. If it is cleared, the limits values must also be set to 0. The use of this flag is to determine whether QuickDraw GX should use or override the default priority.

`gxOverrideUnlimited`
: This bit specifies whether or not QuickDraw GX applies the `gxUnlimitedGapAbsorption` justification flag. If the bit is set, the state of the `gxUnlimitedGapAbsorption` flag is taken into account. If it is cleared, the `gxUnlimitedGapAbsorption` flag must also be set to 0. The use of this flag is to determine whether QuickDraw GX should use or override the default priority.

`gxUnlimitedGapAbsorption`
: If this flag is set, QuickDraw GX distributes all remaining justification gap to the glyphs this width delta structure applies to, even if that would violate the grow or shrink limits. If this field is not zero, you must also set the `gxOverrideUnlimited` bit.

`gxJustificationPriorityMask`
: Identifies the new justification priority for the glyphs this width delta structure applies to. Only a single valid justification priority value, as defined on page 9-60, is permitted. If this flag is set, the `gxOverrideLimits` bit must also be set.

All bits in the justification flags value not accounted for by the above constants must be set to 0.

You set the `gxOverridePriority`, `gxOverrideLimits`, or `gxOverrideUnlimited` bits to choose which aspects of font-specified justification behavior to override. For example, to change the priority of white space glyphs to be the same as intercharacter priority, set `growFlags` to have the `gxOverridePriority` bit and `gxJustificationPriorityMask` to `gxInterCharPriority`. If you clear the `gxOverridePriority` bit, the value in the `gxJustificationPriorityMask` flag is ignored, and the font-specified justification priority applies. Note that you must also clear the `gxJustificationPriorityMask` bits.

As another example, to change the grow limits of the glyphs to which a width delta structure applies, set the `gxOverrideLimits` bit in the justification flags of the `growFlags` field and specify the new grow limits in the `beforeGrowLimit` and `afterGrowLimit` fields.

Priority Justification Override Structure

A priority justification override structure specifies overriding justification behavior for all of the glyphs of a given style run. It contains an array of width delta structures, one for each justification priority.

```
typedef struct {
   gxWidthDeltaRecord   deltas[gxNumberOfJustificationPriorities];
} gxPriorityJustificationOverride;
```

Field descriptions

`deltas` The array of width delta structures. There is one width delta structure for each priority level, in index order.

The width delta structure is described on page 9-61. This structure is the primary input to the `GXSetStyleRunPriorityJustOverride` function.

Each width delta structure in the priority justification override structure specifies overrides for all glyphs of a given justification priority. Thus, for each priority, you can

- Change the priority: assign all glyphs of one priority to another priority.
- Change the behavior: leave the priority the same, but change the priority behavior of all glyphs of that priority.
- Change both: assign all glyphs of one priority to another priority, and change their justification behavior from the defaults of either priority.

Note that if you change one priority to another and change the default behavior of all glyphs of that priority, the behavior of other glyphs already having that priority is not changed. For example, if you change all glyphs with a priority of `gxInterCharPriority` to `gxWhiteSpacePriority` and give them special behavior, glyphs that already have a priority of `gxWhiteSpacePriority` retain their default behavior. Only `gxInterCharPriority` glyphs are overridden, and so the overriding behavior applies to only those glyphs.

Unlimited gap absorption is a special case in that it applies across an entire line instead of just to a single style run. If both the `gxOverrideUnlimited` bit and the `gxUnlimitedGapAbsorption` flag are set in any width delta structure for the glyph justification override structure of any style run on a line, QuickDraw GX distributes the current justification gap among all instances of that glyph in all style runs on the line

Glyph Justification Override Structure

The glyph justification override structure assigns an overriding justification priority and behavior to a specific glyph in a style run. It contains a glyph code and a width delta structure.

```
typedef struct {
   gxGlyphcode          glyph;
   gxWidthDeltaRecord   override;
} gxGlyphJustificationOverride;
```

Field descriptions

`glyph` The glyph code that specifies the glyph in the font.

`override` The width delta structure to use for that glyph.

The width delta structure is described on page 9-61. An array of glyph justification override structures is the primary input to the `GXSetStyleRunGlyphJustOverrides` function, described on page 9-79.

The width delta structure in the glyph justification override structure specifies overrides for all instances of a single glyph (specified by glyph code). Thus, for that glyph, you can

- change the glyph's justification priority
- change the glyph's justification behavior: leave its priority the same, but make its behavior different from that of other glyphs of the same priority
- change both: give it a new priority, and change its justification behavior from the defaults of either priority

Changing a glyph's priority and giving it non-default behavior has no effect on the behavior of other glyphs of either the old or new priority.

Unlimited gap absorption is a special case in that it applies across an entire line instead of just to a single style run. If both the `gxOverrideUnlimited` bit and the `gxUnlimitedGapAbsorption` flag are set in any width delta structure for the glyph justification override structure of any style run on a line, QuickDraw GX distributes the current justification gap among all instances of that glyph in all style runs on the line.

Functions

This section describes the functions with which you can manipulate characteristics of lines of text in layout shapes. You can use these functions to

- manipulate baselines
- measure line span
- break lines
- override the behaviors of justification priorities
- override the justification behaviors of individual glyphs

Manipulating Baselines

The function described in this section allows you to retrieve the distances among baselines for a given line of text.

GXGetStyleBaselineDeltas

You can use the `GXGetStyleBaselineDeltas` function to retrieve the distances from y = 0 to each of the other baseline types.

```
void GXGetStyleBaselineDeltas(gxStyle baseStyle,
                              gxBaselineType baseType,
                              gxBaselineDeltas returnedDeltas);
```

`baseStyle` A reference to the style object whose baseline positions are to control placement of all glyphs for the line of text.

`baseType` The primary baseline—that is, the baseline to have a y-delta of 0.

`returnedDeltas`
A `gxBaselineDeltas` array. On return, contains the distances from y = 0 to each of the 32 baseline types.

DESCRIPTION

The `GXGetStyleBaselineDeltas` function constructs and returns an array of distances from y = 0 to each of the 32 baseline types. The distances are computed based on the font and text size specified in the style object you pass in the `baseStyle` parameter.

The style object you pass to this function typically represents the dominant style run on the line: the style run whose baselines are used to control the placement of all glyphs on the line. However, you can pass any style object reference; it need not represent any style run actually present on the line.

If you put the results of this function in the layout options structure of the layout shape, QuickDraw GX uses those baseline positions to draw all text on the line.

ERRORS, WARNINGS, AND NOTICES

Errors

```
style_is_nil
parameter_out_of_range
```

SEE ALSO

For an example of the use of this function, see Listing 9-1 on page 9-28.

Baseline types are described in the section “Baseline Types” on page 9-4. The baseline type enumeration is described on page 9-58. The baseline deltas array is described on page 9-59.

The layout options structure is described in the chapter “Layout Shapes” in this book.

Measuring Line Span

The functions described in this section allow you to get and set the span of a text line.

GXGetLayoutSpan

You can use the `GXGetLayoutSpan` function to retrieve the span of the text line for the specified layout shape.

```
void GXGetLayoutSpan(gxShape layout, Fixed *lineAscent,
                     Fixed *lineDescent);
```

`layout` A reference to the layout shape whose line span you need.

`lineAscent` A pointer to a `Fixed` value. On return, the value is the distance, in points, from y = 0 to the highest ascent line in this layout shape. If you pass `nil` for this parameter, no value is returned in it.

`lineDescent` A pointer to a `Fixed` value. On return, the value is the distance, in points, from y = 0 to the lowest descent line in this layout shape. If you pass `nil` for this parameter, no value is returned in it.

DESCRIPTION

The `GXGetLayoutSpan` function returns the line span (height for horizontal lines; width for vertical lines) for the specified layout shape. The line span is the distance, orthogonal to the baseline, from the lowest descent line to the highest ascent line in the shape. In calculating line span, `GXGetLayoutSpan` takes into account all text sizes on the line, as well as any cross-stream shifting, cross-stream kerning, multiple baselines, and glyph substitution that may have occurred.

`GXGetLayoutSpan` returns its results in points, which for QuickDraw GX are exactly 72 per inch. (Standard typographic points are 72.27 per inch.)

You can use the information returned by this function to position lines of text when drawing. QuickDraw GX uses line span to control the heights of carets and highlight areas, and to define hit-testing regions; if you create your own carets or highlighting shapes, you can use the results of this function for that purpose also.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`

SEE ALSO

For an example of the use of this function, see Listing 9-3 on page 9-34.

You can set the line span with the `GXSetLayoutSpan` function, described next.

GXSetLayoutSpan

You can use the `GXSetLayoutSpan` function to assign a span to the text line for the specified layout shape.

```
void GXSetLayoutSpan(gxShape layout, Fixed lineAscent,
                     Fixed lineDescent);
```

`layout` A reference to the layout shape whose line span is to be set.

`lineAscent`
The distance, in points, from y = 0 to the highest ascent permitted in this layout shape.

`lineDescent`
The distance, in points, from y = 0 to the lowest descent line for this layout shape.

DESCRIPTION

The `GXSetLayoutSpan` function allows your application to specify the line span (height for horizontal lines; width for vertical lines) of a layout shape.

QuickDraw GX uses line span to control the heights of carets and highlight areas, and to define hit-testing regions. If you draw successive lines of text with a fixed line spacing, you can use `GXSetLayoutSpan` to make sure that carets, highlights, and hit-testing areas do not overlap from line to line. If you do not use the `GXSetLayoutSpan` function, then all QuickDraw GX functions use a line span that corresponds to the largest ascent and descent present in the line. That span is affected by such features as varying text sizes, cross-stream shifting, cross-stream kerning, multiple baselines, and glyph substitution.

You specify the `lineAscent` and `lineDescent` parameters in points (72 per inch). The `lineAscent` parameter is measured above (to the right of, for vertical text) the line's dominant baseline; `lineDescent` is measured below (to the left of, for vertical text) the same baseline. In Roman text both values are generally positive.

SPECIAL CONSIDERATIONS

If you call `GXGetLayoutSpan` after having altered the line span with `GXSetLayoutSpan`, the result will be the line span that you have set. If you need to recover the line span as originally calculated by QuickDraw GX, make sure you save that information before calling `GXSetLayoutSpan`.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`

SEE ALSO

Functions whose results are affected by the result of this function include `GXGetLayoutCaret`, `GXGetLayoutHighlight`, `GXGetLayoutVisualHighlight`, and `GXHitTestLayout`. All are described in the chapter "Layout Carets, Highlighting, and Hit-Testing" in this book.

To obtain the span of a line, use the `GXGetLayoutSpan` function, described in the previous section.

Breaking Lines

The functions described in this section allow you to locate a line break (`GXGetLayoutBreakOffSet`), determine the exact display width of the broken line (`GXGetLayoutRangeWidth`), and make a new layout shape for it (`GXGetNewLayoutFromRange`).

GXGetLayoutBreakOffset

You can use the `GXGetLayoutBreakOffset` function to determine the point at which to break a line of text.

```
gxByteOffset GXGetLayoutBreakOffset(gxShape layout,
                                    gxByteOffset startOffset,
                                    Fixed lineWidth,
                                    long hyphenationCount,
                                    const gxByteOffset hyphenationPoints[],
                                    boolean *startIsStaked,
                                    gxByteOffset *priorStake,
                                    gxByteOffset *nextStake);
```

`layout` A reference to the layout shape containing the text to be broken.

`startOffset`
The byte offset in the source text of the first character in the line.

`lineWidth` The available display length, in points, for the line of text.

`hyphenationCount`
The number of entries in the `hyphenationPoints` parameter (the size of the hyphenation array). If you pass `nil` for the `hyphenationPoints` parameter, this parameter must be set to 0.

`hyphenationPoints`
An array of hyphenation points. A hyphenation point is an edge offset that your application considers a preferred point for a line break. You may pass `nil` for this parameter.

`startIsStaked`
A pointer to a Boolean value. On return, it indicates whether the edge offset passed in the `startOffset` parameter is a staked position.

`priorStake` A pointer to a `gxByteOffset` value. On return, it specifies the edge offset of the staked position in the source text that precedes the returned offset.

`nextStake` A pointer to a `gxByteOffset` value. On return, it specifies the edge offset of the staked position in the source text that follows the returned offset.

function result The edge offset corresponding to the trailing edge of the last glyph that fits completely into the display line, given the specified starting offset, display length, and preferred hyphenation points. (*Last* means last in input order, not display order.)

DESCRIPTION

The `GXGetLayoutBreakOffset` function returns the approximate edge offset following the last character whose glyph fits completely on a line having the display length specified by `lineWidth`. The offset is only approximate because, if a ligature falls on the line boundary, `GXGetLayoutBreakOffset` does not divide the ligature into component glyphs to get the exact offset.

You can pass a hyphenation array to the function in the `hyphenationPoints` parameter. The array consists of a set of edge offsets, each of which represents a preferred point at which to break the line. The array must be sorted by increasing offset value. If you pass a hyphenation array to this function and at least one of its values falls within the range of the line being broken, the function result is one of the values in the array. If you pass `nil`, the function result is the offset corresponding to the last glyph that physically fits in the line width.

The `startIsStaked`, `priorStake`, and `nextStake` parameters help you define staked offsets in your source text. A **staked offset** is one that forms a natural break in terms of the text-layout processing performed by QuickDraw GX. Staked offsets reflect typographic concerns (for instance, do not break up ligatures, rearranged sequences of glyphs, or kerning sets) rather than adherence to the rules of hyphenation. (Technically, it means that all state machines that control layout processing are in their ground state.)

On return from this function, the `startIsStaked` parameter is set to `true` if the starting offset corresponds to a staked value, and the `priorStake` and `nextStake` parameters give the nearest staked offsets behind and ahead of the function result (whether or not they fall within the limits of the line). A value of `gxNoStake` (–1) in either one of these parameters means that QuickDraw GX has not found an adjacent staked location on that line.

If you need to perform text layout with maximum efficiency, you can use this information to start and end lines at staked offsets. Most applications, however, can pass `nil` for `startIsStaked`, `priorStake`, and `nextStake`.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
count_is_less_than_zero
inconsistent_parameters
```

SEE ALSO

For an example of the use of this function, see Listing 9-3 on page 9-34.

After obtaining a line-break position, you can determine the exact width of the line up to that position by calling the `GXGetLayoutRangeWidth` function, described next. Then you can use the `GXNewLayoutFromRange` function, described on page 9-72, to create a new layout shape from that range of text.

GXGetLayoutRangeWidth

You can use the `GXGetLayoutRangeWidth` function to return the exact display length (width for horizontal text; height for vertical text) of a portion of the text in a layout shape.

```
Fixed GXGetLayoutRangeWidth(gxShape layout,
                            gxByteOffset startOffset,
                            gxByteOffset endOffset,
                            gxShape supplementaryText);
```

`layout` A reference to the layout shape containing the text whose display length is to be determined.

`startOffset` The edge offset in the source text before the first character to include.

`endOffset` The edge offset in the source text after the last character to include.

`supplementaryText`
A reference to a layout shape that contains any text, such as a hyphen, that you want to add to the text being measured by this function. Pass `nil` for this parameter if you do not want to add any text.

function result The exact display length, in points, of the portion of the layout shape between the two offsets you specify.

DESCRIPTION

The `GXGetLayoutRangeWidth` function takes two offsets within a layout shape and returns the exact length, in points (72 per inch), of that portion of the layout shape. The `GXGetLayoutRangeWidth` function can also take into account any supplementary text you may want to add to this part of the layout shape, such as a hyphen or a more complex structure.

For example, “Zuk-ker” is the correct hyphenation of the German word “Zucker.” Thus, you need to delete the “c” and add a “k-”, instead of just adding a hyphen. In this case, you set the `endOffset` parameter so that it doesn’t include the “c”, and you specify a shape containing “k-” in the `supplementaryText` parameter.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
```

SEE ALSO

Before calling `GXGetLayoutRangeWidth`, you can obtain a text range for line breaking with the `GXGetLayoutBreakOffset` function, described on page 9-69. You can create a new layout shape from that range of text by calling the `GXNewLayoutFromRange` function, described next.

GXNewLayoutFromRange

You can use the `GXNewLayoutFromRange` function to create a new layout shape from a range of text within an existing layout shape. Typically, the new shape represents a single line of text whose limits have been determined by a call to the `GXGetLayoutBreakOffset` function.

```
gxShape GXNewLayoutFromRange(gxShape layout,
                             gxByteOffset startOffset,
                             gxByteOffset endOffset,
                             const gxLayoutOptions *layoutOptions,
                             gxShape supplementaryText);
```

`layout` A reference to the layout shape containing the range of text to be used.

`startOffset`
: The edge offset in the source text before the first character to include in the new shape.

`endOffset` The edge offset in the source text after the last character to include in the new shape.

`layoutOptions`
: A pointer to a layout options structure containing the layout options you want to apply to the new layout shape.

`supplementaryText`
: A reference to a layout shape that contains any text, such as a hyphen, that you want to add to the new layout shape. Pass `nil` for this parameter if you do not want to add any text.

function result A reference to the new layout shape.

DESCRIPTION

The `GXNewLayoutFromRange` function takes a range of source text from a layout shape plus any additional text you include, and returns a new layout shape. You usually call the `GXNewLayoutFromRange` function after first calling the `GXGetLayoutBreakOffset` function, and possibly the `GXGetLayoutRangeWidth` function.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
```

SEE ALSO

For an example of the use of this function, see Listing 9-3 on page 9-34.

You can obtain a text position for line breaking by calling the `GXGetLayoutBreakOffset` function, described on page 9-69. After obtaining a line-break position, you can determine the exact width of the line up to that position by calling the `GXGetLayoutRangeWidth` function, described on page 9-71.

Overriding the Behaviors of Justification Priorities

The functions described in this section allow you to the override justification behavior of classes of glyphs, based on justification priority. You can specify either the style object whose behavior is to be changed (`GXGetStyleRunPriorityJustOverride`, `GXSetStyleRunPriorityJustOverride`) or the shape object whose associated style object is to be altered (`GXGetShapeRunPriorityJustOverride`, `GXSetShapeRunPriorityJustOverride`).

GXGetStyleRunPriorityJustOverride

You can use the `GXGetStyleRunPriorityJustOverride` function to retrieve a priority justification override structure from a style object.

```
long GXGetStyleRunPriorityJustOverride(gxStyle source,
                                       gxPriorityJustificationOverride
                                       *priorityJustificationOverride);
```

`source` A reference to the style object whose priority justification override structure you need.

`priorityJustificationOverride`
A pointer to a priority justification override structure. On return, the structure contains the priority justification overrides for the style object. You can pass `nil` for this parameter if you do not need to retrieve the priority justification override structure itself.

function result If the style object has a priority justification override structure, this value is 1. Otherwise, it is 0.

DESCRIPTION

The `GXGetStyleRunPriorityJustOverride` function returns the priority justification override structure from the style object you specify in the `source` parameter.

ERRORS, WARNINGS, AND NOTICES

Errors
`style_is_nil`

SEE ALSO

To set the priority justification overrides for a style run, use the `GXSetStyleRunPriorityJustOverride` function, described next.

To retrieve the priority justification overrides associated with a particular shape, use the `GXGetShapeRunPriorityJustOverride` function, described on page 9-76. To set the priority justification overrides associated with a particular shape, use the `GXSetShapeRunPriorityJustOverride` function, described on page 9-77.

To retrieve the glyph justification overrides for a style run, use the `GXGetStyleRunGlyphJustOverrides` function, described on page 9-78.

GXSetStyleRunPriorityJustOverride

You can use the `GXSetStyleRunPriorityJustOverride` function to assign a priority justification override structure to a style object (or to remove it from the style object).

```
void GXSetStyleRunPriorityJustOverride(gxStyle target,
                                       const gxPriorityJustificationOverride
                                       *priorityJustificationOverride);
```

`target` A reference to the style object whose priority justification override structure you want to modify.

`priorityJustificationOverride`
A pointer to the priority justification override structure to assign.

DESCRIPTION

The `GXSetStyleRunPriorityJustOverride` function copies the specified `gxPriorityJustificationOverride` structure into the priority justification override property of the style object specified in the `target` parameter.

If you pass `nil` for the `priorityJustificationOverride` parameter, the function removes the priority justification override structure (if any) from the style object.

ERRORS, WARNINGS, AND NOTICES

Errors

```
style_is_nil
inconsistent_parameters
```

Notices (debugging version)

```
attributes_already_set
```

SEE ALSO

For examples of the use of this function, see the unnumbered code listing on page 9-52 and Listing 9-9 on page 9-52.

To retrieve the priority justification overrides for a style run, use the `GXGetStyleRunPriorityJustOverride` function, described on page 9-74.

To retrieve the priority justification overrides associated with the style associated with a particular shape, use the `GXGetShapeRunPriorityJustOverride` function, described next. To set the priority justification overrides associated with the style associated with a particular shape, use the `GXSetShapeRunPriorityJustOverride` function, described on page 9-77.

To set the glyph justification overrides for a style run, use the `GXSetStyleRunGlyphJustOverrides` function, described on page 9-79.

GXGetShapeRunPriorityJustOverride

You can use the `GXGetShapeRunPriorityJustOverride` function to retrieve the priority justification override structure from the style object associated with a shape.

```
long GXGetShapeRunPriorityJustOverride(gxShape source,
                                       gxPriorityJustificationOverride
                                       *priorityJustificationOverride);
```

`source` A reference to the shape object whose associated style object contains the priority justification override structure you need.

`priorityJustificationOverride`
A pointer to a priority justification override structure. On return, the structure contains the priority justification overrides for the style object associated with the specified shape. You can pass `nil` for this parameter if you do not need to retrieve the priority justification override structure itself.

function result If the style object associated with the specified shape has a priority justification override structure, this value is 1. Otherwise, it is 0.

DESCRIPTION

The `GXGetShapeRunPriorityJustOverride` function returns the priority justification override structure from the style object associated with the shape you specify in the `source` parameter.

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the `GXNewLayout` function, for example, you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as `GXGetStyleRunPriorityJustOverride`.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`

SEE ALSO

To set the priority justification overrides associated with a particular shape, use the `GXSetShapeRunPriorityJustOverride` function, described next.

To retrieve the priority justification overrides for a style run, use the `GXGetStyleRunPriorityJustOverride` function, described on page 9-74.

To set the priority justification overrides for a style run, use the GXSetStyleRunPriorityJustOverride function, described on page 9-75.

To retrieve the glyph justification overrides associated with a particular shape, use the GXGetShapeRunGlyphJustOverrides function, described on page 9-81.

GXSetShapeRunPriorityJustOverride

You can use the GXSetShapeRunPriorityJustOverride function to assign a priority justification override structure to the style object associated with a shape (or to remove it from the style object).

```
void GXSetShapeRunPriorityJustOverride(gxShape target, const
                                       gxPriorityJustificationOverride
                                       *priorityJustificationOverride);
```

`target` A reference to the shape object whose associated style object contains the priority justification override structure you want to modify.

`priorityJustificationOverride`
A pointer to the priority justification override structure to assign.

DESCRIPTION

The GXSetShapeRunPriorityJustOverride function copies the specified gxPriorityJustificationOverride structure into the priority justification override property of the style object associated with the shape specified in the `target` parameter.

If you pass `nil` for the `priorityJustificationOverride` parameter, the function removes the priority justification override structure (if any) from the style object.

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the GXNewLayout function, for example, you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as GXSetStyleRunPriorityJustOverride.

ERRORS, WARNINGS, AND NOTICES

Errors
`shape_is_nil`

Notices (debugging version)
`attributes_already_set`

SEE ALSO

To retrieve the priority justification overrides associated with a particular shape, use the `GXGetShapeRunPriorityJustOverride` function, described on page 9-76.

To retrieve the priority justification overrides for a style run, use the `GXGetStyleRunPriorityJustOverride` function, described on page 9-74. To set the priority justification overrides for a style run, use the `GXSetStyleRunPriorityJustOverride` function, described on page 9-75.

To set the glyph justification overrides associated with a particular shape, use the `GXSetShapeRunGlyphJustOverrides` function, described on page 9-82.

Overriding the Justification Behaviors of Individual Glyphs

The functions described in this section allow you to the override the justification behavior of individual glyphs in a style run. You can specify either the style object associated with the glyphs to be changed (`GXGetStyleRunGlyphJustOverrides`, `GXSetStyleRunGlyphJustOverrides`) or the shape object whose style object is associated with the glyphs (`GXGetShapeRunGlyphJustOverrides`, `GXSetShapeRunGlyphJustOverrides`).

GXGetStyleRunGlyphJustOverrides

You can use the `GXGetStyleRunGlyphJustOverrides` function to retrieve an array of glyph justification override structures from a style object.

```
long GXGetStyleRunGlyphJustOverrides(gxStyle source,
                                     gxGlyphJustificationOverride
                                     glyphJustificationOverrides[]);
```

`source` A reference to the style object whose array of glyph justification overrides you need.

`glyphJustificationOverrides`
An array of glyph justification override structures. On return, the array contains the glyph justification overrides for the style object. If you pass `nil` for this parameter, nothing is returned in this array. However, the function result is still the correct number of glyph justification override structures in the style.

function result The number of glyph justification override structures in the style object.

DESCRIPTION

The `GXGetStyleRunGlyphJustOverrides` function returns the number of glyph justification override structures currently contained in the style object you specify in the `source` parameter.

To get the overrides themselves, you need to allocate an array to pass in the `glyphJustificationOverrides` parameter when calling this function. To get the right size for the array, you can first call the function with a value of `nil` for the `glyphJustificationOverrides` parameter. Then use the function result to allocate an array of the proper size and call `GXGetStyleRunGlyphJustOverrides` a second time, this time passing the array.

SPECIAL CONSIDERATIONS

Because QuickDraw GX can reorder the elements in a style object's glyph-justification overrides array, the order of elements returned in the `glyphJustificationOverrides` parameter to this function may differ from the order in which they were originally assigned to the style.

ERRORS, WARNINGS, AND NOTICES

Errors
`style_is_nil`

SEE ALSO

To set the glyph justification overrides for a style run, use the `GXSetStyleRunGlyphJustOverrides` function, described next.

To retrieve the glyph justification overrides associated with a particular shape, use the `GXGetShapeRunGlyphJustOverrides` function, described on page 9-81. To set the glyph justification overrides associated with a particular shape, use the `GXSetShapeRunGlyphJustOverrides` function, described on page 9-82.

To retrieve the priority justification overrides for a style run, use the `GXGetStyleRunPriorityJustOverride` function, described on page 9-74.

GXSetStyleRunGlyphJustOverrides

You can use the `GXSetStyleRunGlyphJustOverrides` function to assign an array of glyph justification override structures to a style object, or to remove all glyph justification overrides from it.

```
void GXSetStyleRunGlyphJustOverrides(gxStyle target, long count,
                                     const gxGlyphJustificationOverride
                                     glyphJustificationOverrides[]);
```

`target` A reference to the style object whose glyph justification overrides you want to modify.

`count` The number of glyph justification override structures you want to assign.

`glyphJustificationOverrides`
: The array of glyph justification override structures to be assigned to the style object.

DESCRIPTION

The `GXSetStyleRunGlyphJustOverrides` function copies an array of `gxGlyphJustificationOverride` structures into the glyph justification overrides property of the specified style object.

If you pass `nil` for the `glyphJustificationOverrides` parameter and 0 for the `count` parameter, the function removes all glyph justification override structures from the style object.

If `count` is 0 and `glyphJustificationOverrides` is non-`nil`, or if `count` is nonzero and `glyphJustificationOverrides` is `nil`, `GXSetStyleRunGlyphJustOverrides` posts an `inconsistent_parameters` error.

ERRORS, WARNINGS, AND NOTICES

Errors

```
style_is_nil
parameter_out_of_range
count_is_less_than_zero
inconsistent_parameters
```

Notices (debugging version)

```
attributes_already_set
```

SEE ALSO

For an example of the use of this function, see Listing 9-9 on page 9-52.

To retrieve the glyph justification overrides for a style run, use the `GXGetStyleRunGlyphJustOverrides` function, described on page 9-78.

To retrieve the glyph justification overrides associated with the style associated with a particular shape, use the `GXGetShapeRunGlyphJustOverrides` function, described next. To set the glyph justification overrides associated with the style associated with a particular shape, use the `GXSetShapeRunGlyphJustOverrides` function, described on page 9-82.

To set the priority justification overrides for a style run, use the `GXSetStyleRunPriorityJustOverride` function, described on page 9-75.

GXGetShapeRunGlyphJustOverrides

You can use the GXGetShapeRunGlyphJustOverrides function to retrieve an array of glyph justification override structures from the style object referenced by a shape.

```
long GXGetShapeRunGlyphJustOverrides(gxShape source,
                                     gxGlyphJustificationOverride
                                     glyphJustificationOverrides[]);
```

`source` A reference to the shape object whose associated style object contains the glyph justification overrides you need.

`glyphJustificationOverrides`
An array of glyph justification override structures. On return, the array contains the glyph justification overrides for the style object associated with the specified shape. If you pass `nil` for this parameter, nothing is returned in this array. However, the function result is still the correct number of glyph justification override structures in the style associated with the shape.

function result The number of glyph justification override structures in the style object associated with the specified shape.

DESCRIPTION

The GXGetShapeRunGlyphJustOverrides function returns the number of glyph justification override structures currently contained in the style object of the shape you specify in the `source` parameter.

To get the overrides themselves, you need to allocate an array to pass in the `glyphJustificationOverrides` parameter when calling this function. To get the right size for the array, you can first call the function with a value of `nil` for the `glyphJustificationOverrides` parameter. Then use the function result to allocate an array of the proper size and call GXGetShapeRunGlyphJustOverrides a second time, this time passing the array.

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the GXNewLayout function, for example, you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as GXGetStyleRunGlyphJustOverrides.

SPECIAL CONSIDERATIONS

Because QuickDraw GX can reorder the elements in a style object's glyph-justification overrides array, the order of elements returned in the `glyphJustificationOverrides` parameter to this function may differ from the order in which they were originally assigned to the style.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`

SEE ALSO

To set the glyph justification overrides associated with a particular shape, use the `GXSetShapeRunGlyphJustOverrides` function, described next.

To retrieve the glyph justification overrides for a style run, use the `GXGetStyleRunGlyphJustOverrides` function, described on page 9-78.
To set the glyph justification overrides for a style run, use the `GXSetStyleRunGlyphJustOverrides` function, described on page 9-79.

To retrieve the priority justification overrides associated with a particular shape, use the `GXGetShapeRunPriorityJustOverride` function, described on page 9-76.

GXSetShapeRunGlyphJustOverrides

You can use the `GXSetShapeRunGlyphJustOverrides` function to assign an array of glyph justification override structures to the style object associated with a shape, or to remove all glyph justification overrides from it.

```
void GXSetShapeRunGlyphJustOverrides(gxShape target, long count,
                                     const gxGlyphJustificationOverride
                                     glyphJustificationOverrides[]);
```

`target` A reference to the shape object whose style object contains the glyph justification overrides you want to modify.

`count` The number of glyph justification override structures you want to assign.

`glyphJustificationOverrides`
The array of glyph justification override structures to be assigned to the style object associated with the specified shape.

DESCRIPTION

The `GXSetShapeRunGlyphJustOverrides` function copies an array of glyph justification override structures into the glyph justification overrides property of the style object associated with the shape you specify in the `target` parameter.

If you pass `nil` for the `glyphJustificationOverrides` parameter and 0 for the `count` parameter, the function removes all glyph justification override structures from the style object.

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the `GXNewLayout` function, for example, you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as `GXSetStyleRunGlyphJustOverrides`.

If `count` is 0 and `glyphJustificationOverrides` is non-`nil`, or if `count` is nonzero and `glyphJustificationOverrides` is `nil`, `GXSetShapeRunGlyphJustOverrides` posts an `inconsistent_parameters` error.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
count_is_less_than_zero
inconsistent_parameters
```

Notices (debugging version)

```
attributes_already_set
```

SEE ALSO

To retrieve the glyph justification overrides associated with a particular shape, use the `GXGetShapeRunGlyphJustOverrides` function, described on page 9-81.

To retrieve the glyph justification overrides for a style run, use the `GXGetStyleRunGlyphJustOverrides` function, described on page 9-78.
To set the glyph justification overrides for a style run, use the `GXSetStyleRunGlyphJustOverrides` function, described on page 9-79.

To set the priority justification overrides associated with the style associated with a particular shape, use the `GXSetShapeRunPriorityJustOverride` function, described on page 9-77.

Summary of Layout Line Control

Constants and Data Types

Baseline Types

```
enum {
   gxRomanBaseline               = 0,
   gxIdeographicCenterBaseline,
   gxIdeographicLowBaseline,
   gxHangingBaseline,
   gxMathBaseline,
   gxLastBaseline                = 31,
   gxNumberOfBaselineTypes       = gxLastBaseline + 1,
   gxNoOverrideBaseline          = 255
};

typedef unsigned long gxBaselineType;
```

Baseline Deltas Array

```
typedef Fixed gxBaselineDeltas[gxNumberOfBaselineTypes];
```

Baseline Structure

```
typedef struct {
   gxBaselineDeltas  deltas;
} gxLineBaselineRecord;
```

Justification Priorities

```
enum {
   gxKashidaPriority             = 0,
   gxWhiteSpacePriority          = 1,
   gxInterCharPriority           = 2,
   gxNullJustificationPriority   = 3,
   gxNumberOfJustificationPriorities
};

typedef unsigned char gxJustificationPriority;
```

Width Delta Structure

```
typedef struct {
   Fixed                  beforeGrowLimit;
   Fixed                  beforeShrinkLimit;
   Fixed                  afterGrowLimit;
   Fixed                  afterShrinkLimit;
   gxJustificationFlags   growFlags;
   gxJustificationFlags   shrinkFlags;
} gxWidthDeltaRecord;
```

Justification Flags

```
enum {
   gxOverridePriority           = 0x8000,
   gxOverrideLimits             = 0x4000,
   gxOverrideUnlimited          = 0x2000,
   gxUnlimitedGapAbsorption     = 0x1000,
   gxJustificationPriorityMask  = 0x000F
   gxAllJustificationFlags      = gxOverridePriority|
                                  gxOverrideLimits|
                                  gxOverrideUnlimited|
                                  gxUnlimitedGapAbsorption|
                                  gxJustificationPriorityMask
};

typedef unsigned short gxJustificationFlags;
```

Priority Justification Override Structure

```
typedef struct {
   gxWidthDeltaRecord   deltas[gxNumberOfJustificationPriorities];
} gxPriorityJustificationOverride;
```

Glyph Justification Override Structure

```
typedef struct {
   gxGlyphcode          glyph;
   gxWidthDeltaRecord   override;
} gxGlyphJustificationOverride;
```

Functions

Manipulating Baselines

```
void GXGetStyleBaselineDeltas
                            (gxStyle baseStyle, gxBaselineType baseType,
                             gxBaselineDeltas returnedDeltas);
```

Measuring Line Span

```
void GXGetLayoutSpan        (gxShape layout, Fixed *lineAscent,
                             Fixed *lineDescent);
void GXSetLayoutSpan        (gxShape layout, Fixed lineAscent,
                             Fixed lineDescent);
```

Breaking Lines

```
gxByteOffset GXGetLayoutBreakOffset
                            (gxShape layout, gxByteOffset startOffset,
                             Fixed lineWidth, long hyphenationCount,
                             const gxByteOffset hyphenationPoints[],
                             boolean *startIsStaked,
                             gxByteOffset *priorStake,
                             gxByteOffset *nextStake);
Fixed GXGetLayoutRangeWidth (gxShape layout, gxByteOffset startOffset,
                             gxByteOffset endOffset,
                             gxShape supplementaryText);
gxShape GXNewLayoutFromRange(gxShape layout, gxByteOffset startOffset,
                             gxByteOffset endOffset,
                             const gxLayoutOptions *layoutOptions,
                             gxShape supplementaryText);
```

Overriding the Behaviors of Justification Priorities

```
long GXGetStyleRunPriorityJustOverride
                            (gxStyle source,
                             gxPriorityJustificationOverride
                             *priorityJustificationOverride);
void GXSetStyleRunPriorityJustOverride
                            (gxStyle target,
                             const gxPriorityJustificationOverride
                             *priorityJustificationOverride);
long GXGetShapeRunPriorityJustOverride
                            (gxShape source,
                             gxPriorityJustificationOverride
                             *priorityJustificationOverride);
```

```
void GXSetShapeRunPriorityJustOverride
                            (gxShape target,
                             const gxPriorityJustificationOverride
                             *priorityJustificationOverride);
```

Overriding the Justification Behaviors of Individual Glyphs

```
long GXGetStyleRunGlyphJustOverrides
                            (gxStyle source,
                             gxGlyphJustificationOverride
                             glyphJustificationOverrides[]);
void GXSetStyleRunGlyphJustOverrides
                            (gxStyle target, long count,
                             const gxGlyphJustificationOverride
                             glyphJustificationOverrides[]);
long GXGetShapeRunGlyphJustOverrides
                            (gxShape source,
                             gxGlyphJustificationOverride
                             glyphJustificationOverrides[]);
void GXSetShapeRunGlyphJustOverrides
                            (gxShape target, long count,
                             const gxGlyphJustificationOverride
                             glyphJustificationOverrides[]);
```

CHAPTER 10

Layout Carets, Highlighting, and Hit-Testing

Contents

This chapter describes the relationship between character-code position in a layout shape's source text and glyph position in its corresponding display text. Your application uses this information to draw carets, to highlight ranges of text, and to hit-test (to convert from display-text location to source-text location).

Read the information in this chapter if you create layout shapes containing text that the user can select or edit. You do not need the information here if you create only static (unalterable) lines of text. Also, if your application creates only simple text that is better represented by a text shape or glyph shape, you do not need the information in this chapter.

Before reading this chapter, you should be familiar with the information in the chapters "Introduction to QuickDraw GX Typography," "Typographic Shapes," "Typographic Styles," and "Layout Shapes" in this book. You should also be familiar with the general concepts of QuickDraw GX objects, as described in *Inside Macintosh: QuickDraw GX Objects*.

The chapter starts by describing how QuickDraw GX defines locations in source text and in display text of any typographic shape. The chapter next explores how the definition of text locations affects caret handling, highlighting, and hit-testing for layout shapes. It then describes how to use QuickDraw GX functions to

- draw perpendicular or angled carets in single-direction or mixed-direction text
- highlight single-direction or mixed-direction text in two different ways
- hit-test any text in a layout shape
- analyze glyphs for direction and for relationship to source-text position

About Carets, Highlighting, and Hit-Testing for Layout Shapes

Editing text from a layout shape can be a complicated process. Preparing a line of text for display can involve reversing text direction, rearranging glyphs, substituting glyphs, creating ligatures, and moving or modifying glyphs for purposes of justification, kerning, typestyle changes, and so on. Editing that line involves repeating these operations for the edited text. However, QuickDraw GX helps you in this effort by eliminating your need to track the details of conversion from source text to display text and back.

This section describes the relationship between character codes in the source text of a layout shape and glyphs in its display text. It also explains how QuickDraw GX exploits that relationship to help you with caret handling, highlighting, and hit-testing.

Positioning in Source Text and Display Text

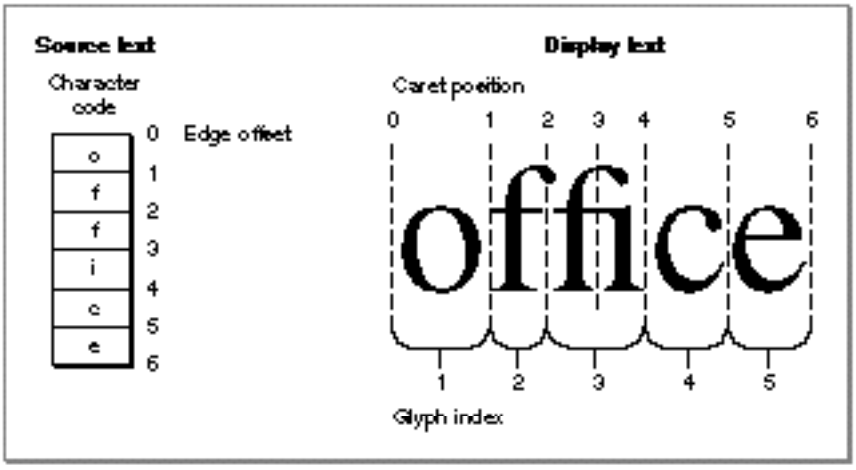
By convention, QuickDraw GX stores text in a typographic shape as a sequence of character codes, which are display-independent numeric representations of the fundamental characters that make up the text. Furthermore, these character codes are stored in input order, that is, the order in which the characters are entered from a keyboard.

When it draws text, QuickDraw GX composes glyphs from those characters and draws the glyphs in display order. Display order is uniformly left to right (or top to bottom for vertical text) and may be very different from input order.

On the screen, the sequence of glyphs in a line of text is specified by **glyph index,** a simple left-to-right (or top-to-bottom) ordering that starts with 1 for the leftmost glyph. In Figure 10-1, for example, a line of text composed of six characters in memory (on the left) is rendered onscreen with five glyphs (on the right).

In memory, the positions of character codes in the source text of a shape could also be represented by index, but they are more commonly represented by edge offset. **Edge offset** is the byte offset from the beginning of a shape's source text to a point *between* character codes in the sequence. Edge offset is zero-based. Figure 10-1 shows an edge offset of 0 marks a point just before the first byte in the source text; an offset of 1 marks a point between the first and second bytes, and so on.

Figure 10-1 Positioning conventions for source text and display text



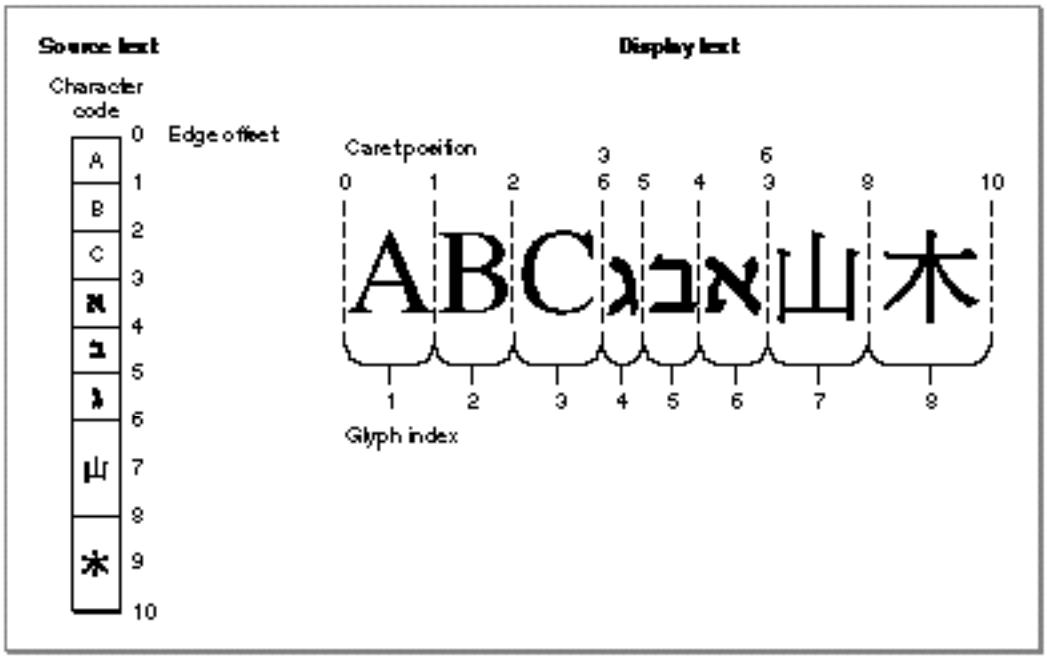
Edge offset is slightly different from the typical notion of byte offset into a buffer, in that a given edge offset is not associated with a unique byte value. An edge offset could refer to either the byte preceding it or the byte following it. This concept is useful because edge offset relates directly to **caret position,** a location on screen that is typically between glyphs. (A **caret** is a vertical or slanted bar, appearing at a caret position in the display text, that marks the point at which text is to be inserted or deleted.) Each caret position relates directly to an edge offset; you can see in Figure 10-1 that the caret positions in the display text are numbered according to their corresponding edge offsets in the source text. A caret at any of the caret positions in Figure 10-1 means an insertion point in the source text at the corresponding edge offset.

The text in Figure 10-1 is fairly simple; it is single-direction text, and all character codes are 1 byte long. One complication is that the "f" and "i" characters are combined upon display into the "fi" ligature. Because the ligature is a single glyph, there is no longer a

one-to-one correspondence between characters and glyphs. Also, there is one possible caret position (at edge offset 3) that is within a glyph rather than at its edge. For the purposes of drawing carets, highlighting, and hit-testing, QuickDraw GX permits you either to allow or disallow caret positions within complex glyphs such as ligatures.

Figure 10-2 shows a slightly more complex example of a line of text from a layout shape. In this layout, three characters of Roman text are followed by three Hebrew characters, in turn followed by two Chinese characters. Note that, because Hebrew text is read from right to left, the order of the Hebrew glyphs on the screen is the reverse of the order of their character codes in the source text. (The line direction, or dominant text direction, of this layout shape is left to right. If it were right to left, the order of the displayed characters would have been somewhat different. See the chapter "Layout Line Control" in this book for a complete discussion of how line direction and different levels of direction runs can cause complications in line layout.)

Figure 10-2 Edge offsets and glyph indexes in mixed-direction text



The reversal of the Hebrew text causes some complications in the relationship between caret position and edge offset. Note in particular what happens at edge offset 3, the boundary (in the source text) between the Roman "C" and the Hebrew "✖." In memory, it is a single point that could allow for inserting a Roman character after the "C" or a Hebrew character before the "✖." On screen, however, that insertion point splits into two caret positions: one to the right of (that is, after) the "C," and one to the right of (that is, before) the "✖."

Within the run of Hebrew text, caret positions increase leftward, as would be expected for right-to-left text. And at the boundary between the Hebrew text and the Chinese text (here written horizontally and left to right), the additional change in direction means that edge offset 6 also converts to two caret positions onscreen: one to the left of (that is, after) the "," and one to the left of (that is, before) the "."

In general, each direction boundary in the source text of a layout shape maps to two different caret positions in the display text, and each direction boundary in the display text maps to two different edge offsets in the source text. This indeterminacy causes complications in caret drawing, highlighting, and hit-testing, as discussed further in subsequent sections of this chapter. However, note that QuickDraw GX handles most of the complications for you.

Figure 10-2 illustrates a second important complication. Note that the Chinese character codes in the source text are each 2 bytes long. Because edge offsets are byte offsets, each subsequent location between Chinese characters has an offset value that is 2 higher than its predecessor. In Figure 10-2, edge offsets (and caret positions) 7 and 9 are invalid—each refers to a point inside a single character.

IMPORTANT

If you are going to support 2-byte characters in your text handling, remember that successive valid edge offsets in the buffer that holds your source text may differ by either 1 or 2. Even in 2-byte languages such as Chinese, Japanese, and Korean, some character codes are only 1 byte long. You cannot assume a single storage size for characters. QuickDraw GX provides functions, such as the `GXGetOffsetGlyphs` function on page 10-56, that help you determine the sizes of character codes. s

Caret Handling

A **caret** is a symbol that indicates where onscreen the next text-editing operation will take place. The caret is commonly represented by a blinking vertical bar (|) in horizontal text, a horizontal bar in vertical text, and a slanted bar in italic or otherwise slanted text.

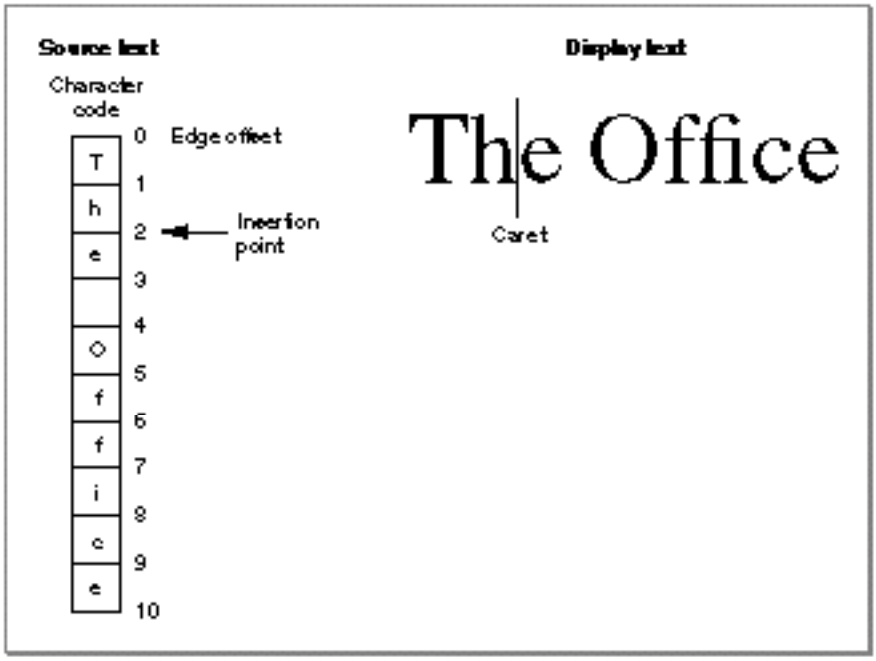
Note

In Macintosh text processing, the caret is not the same as the cursor. The **cursor** is a small icon (often an arrow or I-beam shape) that "shadows" the movement of the mouse or other pointing device. The cursor is controlled by the system, although your application can change its appearance; the caret is controlled entirely by your application. When the user clicks or holds the mouse button down while the cursor is positioned within a line of text, your application sets the caret position or extends the highlight to the cursor position. Otherwise, these positions are independent of each other. u

The caret is the onscreen representation of the insertion point. The **insertion point** is that point in the source text where character codes will be added or deleted when the next editing operation occurs. An insertion point is specified by a single edge offset; the caret

occupies the onscreen caret position that corresponds to that edge offset. Figure 10-3 shows the basic relationship between insertion point, edge offset, and caret position for single-direction text.

Figure 10-3 Insertion point and caret



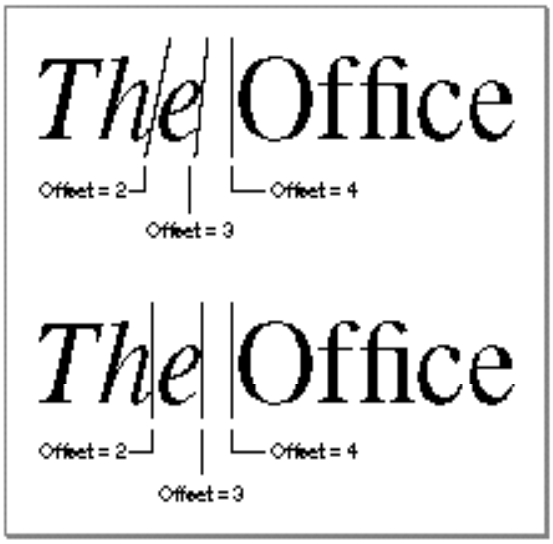
Straight and Angled Carets

When the caret appears in italic text or text that has an intrinsic angle (for instance, ITC Zapf Chancery®), you may wish, for the user's convenience, to display an angled (slanted) caret rather than a straight one. QuickDraw GX supports this capability by using data present in fonts that identifies the intrinsic font angle. Your application can disable this feature, if desired, on a per-run basis.

You can specify that a caret can be straight or angled, meaning that it is either perpendicular to the baseline or at an angle to the baseline that equals the slant of the glyphs between which the caret is drawn. If you choose an angled caret, QuickDraw GX calculates the proper angle; at boundaries of text with different slant, QuickDraw GX calculates an average angle for the caret.

In Figure 10-4, for example, the word "The" is italic, whereas the subsequent characters are not. The angled caret is parallel to the slant of the italic text at the caret position whose edge offset is 2; it has only half the slant at the caret position whose edge offset is 3; and it is vertical at caret positions with edge offsets of 4 and greater.

Figure 10-4 Angled and straight carets in single-direction text



Note also from Figure 10-4 that if you specify a straight (vertical) caret, it remains perpendicular to the baseline even in italic text.

Drawing an angled cursor

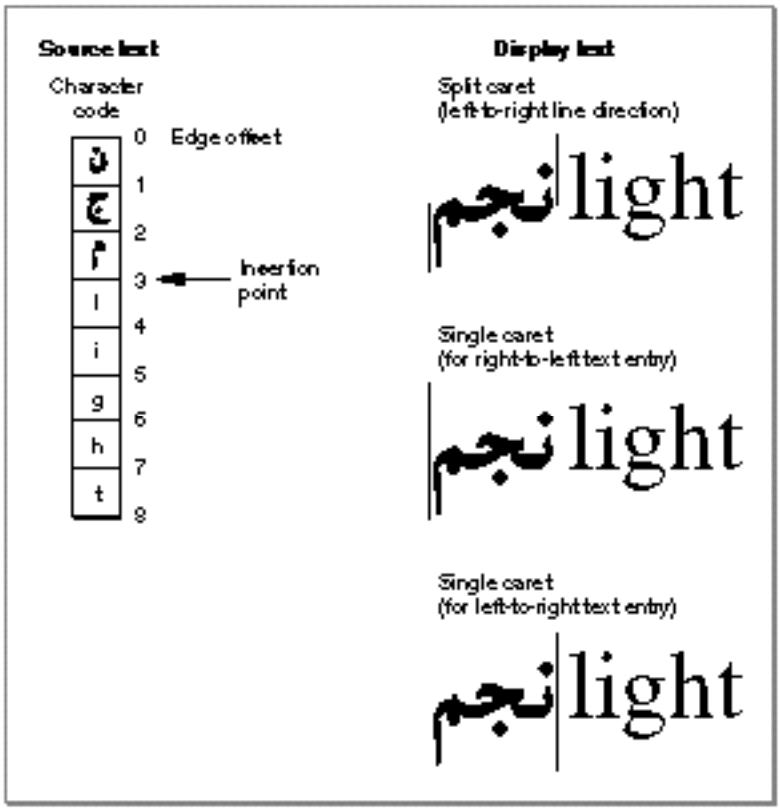
You may also want to slant the cursor when it passes over areas of slanted text. QuickDraw GX provides a function to help you do that efficiently. See “Drawing the Cursor at the Correct Angle Within a Given Area” beginning on page 10-22. u

Split and Single Carets

At direction boundaries in mixed-direction text, a single insertion point in memory can require two caret positions onscreen, one for text entry in each direction. QuickDraw GX supports two different methods for handling that situation: a split caret or a single caret. Figure 10-5 shows examples of both caret types for an insertion point at the boundary between Arabic and Roman text.

A **split caret,** or dual caret, is the preferred caret shape provided by QuickDraw GX for use with mixed-direction text. A split caret consists of a high caret and a low caret, each measuring half the line’s height. The two separate half-carets appear only when the insertion point is at the boundary between two direction runs in a line of text. The high (dominant) caret is displayed at the caret position for insertion of text whose direction corresponds to the line direction (the dominant direction for the whole line of text). The low caret is displayed at the caret position for insertion of text whose direction is counter to the line direction. When the caret position is unambiguous (not on a direction boundary), the primary and secondary carets are at the same position, so the user sees one caret.

Figure 10-5 Split caret and single carets at a direction boundary in mixed-direction text



In Figure 10-5 (top), a split caret appears when the insertion point is at the direction boundary at edge offset 3. Because the line direction for this example is left to right, the high caret appears to the left of the space character before the word “light”, to allow insertion of Roman characters before the space character. The low caret appears at the far left of the line, to allow insertion of Arabic characters after the “ ”.

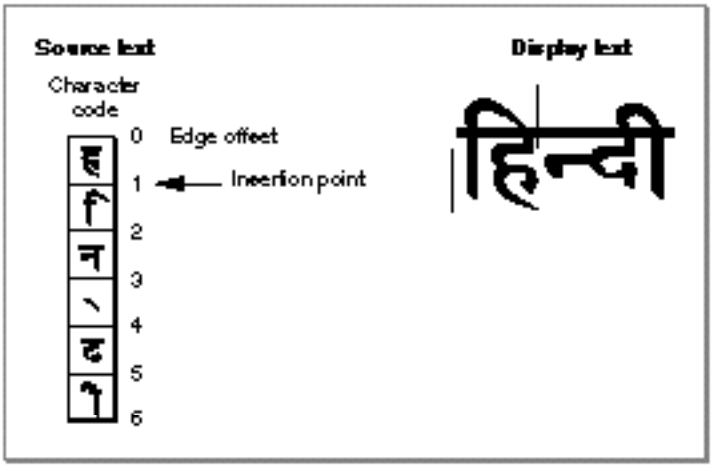
A **single caret** can also be used in mixed-direction text. It is either a **left-to-right caret** or a **right-to-left caret,** which refers not to any difference in appearance but to difference in location. When the caret position is unambiguous (not on a direction boundary), it is a standard text-insertion caret. At a direction boundary, it is also a single caret that appears at the place where the next text insertion or deletion will occur, given the application’s specification of text direction.

In Figure 10-5 (middle and bottom), a single caret appears at either of two positions when the insertion point is at the direction boundary at edge offset 3. If the current input text direction is left to right (corresponding to Roman text entry), the caret appears to the left of the ”l” character before the word “light”. If the current direction is right-to-left (corresponding to Arabic text entry), the caret appears at the far left of the line. Note that switching input directions repeatedly without entering any characters causes the caret to jump between the two caret positions. Note that it is your responsibility to monitor the user’s choice of keyboard or language and set the caret type appropriately.

Split carets and linguistic rearrangement

Split carets can arise in one other case: linguistic (Indic-style) rearrangement. In this case, all the text is uniformly left to right, but because the visual order of certain glyphs on the line is different from the input order of the character codes in the source text, the returned caret is split. The high caret is located at the caret position with the numerically higher edge offset of the two for that character; see Figure 10-6. u

Figure 10-6 Split caret with linguistically rearranged glyphs



Carets and vertical text

Vertical text is never reordered nor linguistically rearranged; its display order (top to bottom) is always the same as its input order. Therefore, text direction is always (the equivalent of) left to right, and split carets cannot occur. u

Caret Position and Split Ligatures

For defining caret positions, you can treat ligatures as single, indivisible glyphs or you can split them into subareas representing each of their component characters. If you want ligatures to be treated as single glyphs, you set the `gxNoLigatureSplits` flag in the run controls structure of the style run containing the ligatures. If you clear the `gxNoLigatureSplits` flag, QuickDraw GX allows caret positions within the ligature, as defined by the font. By default, the flag is cleared.

Figure 10-7 shows two examples of carets drawn at all valid caret positions in the word “office”. In the upper example, ligature splits are permitted; in the lower example, they are not.

Figure 10-7 Caret positions with and without ligature splits

For an example of the effect of setting and clearing the `gxNoLigatureSplits` flag, see "Positioning the Caret Within Ligatures" beginning on page 10-24. Run controls are explained in the chapter "Layout Styles" in this book.

Note
Whether or not you permit ligature splits, editing should still occur one character at a time, not one glyph at a time. Thus, if the caret were positioned to the right of an "fi" ligature, a single backspace should not delete the whole ligature; it should delete only the "i", leaving an "f" glyph in place of the ligature. u

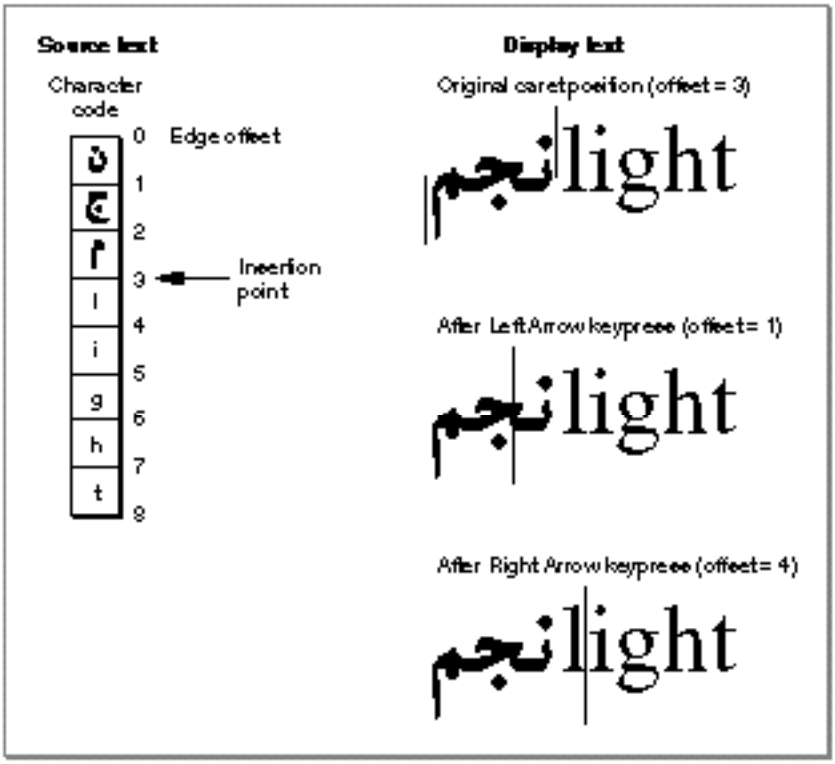
Arrow Keys and Caret Movement

When the user presses the Right Arrow key, the caret should move one character to the right in horizontal text. When the user presses the Left Arrow key, the caret should move one character to the left. If your layout shape has ligatures, mixed 1-byte and 2-byte characters, mixed-direction text, or text that has undergone linguistic rearrangement, determining the edge offset of the next caret position is not always obvious.

Figure 10-8 shows a line of mixed Roman and Arabic text in which the dominant direction is left to right. Suppose that the current insertion point is at edge offset 3 in the source text. That offset represents a direction boundary; the high caret marks the point for inserting Roman text, and the low caret marks the point for inserting Arabic text. If the user presses the Left Arrow key, the high caret moves one space to the left. That puts the caret across the direction boundary, and corresponds to moving several characters backward in the (Arabic) source text, which puts the new edge offset at 1. For split carets, the high caret should follow the arrows.

If instead the user presses the Right Arrow key, the high caret moves one space to the right. That action puts the caret across the direction boundary and corresponds to moving one character forward in the source text. The new edge offset is therefore 4.

Figure 10-8 Moving the caret with Left and Right Arrow keys



If the display text contains ligatures, caret movement in response to the pressing of an arrow key can depend on whether caret positions within ligatures are valid. See “Caret Position and Split Ligatures” on page 10-10.

QuickDraw GX provides functions that allow you to determine the proper edge offset of the new insertion point when the user moves the caret position with the arrow keys. See “Positioning the Caret in Response to Arrow Keypresses” beginning on page 10-22.

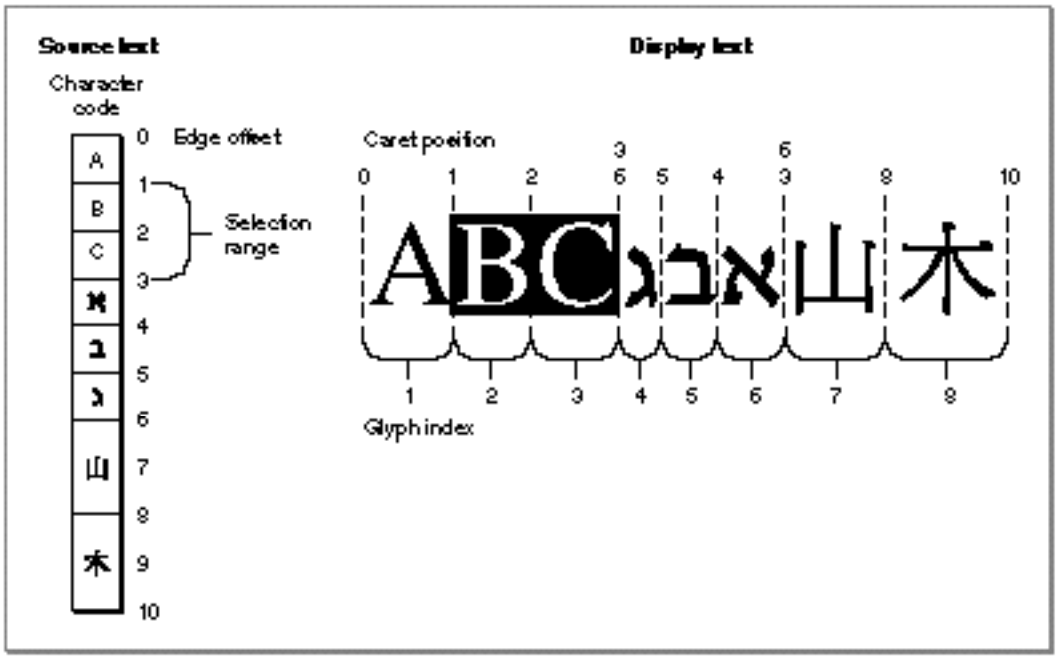
Highlighting

Highlighting is the display of portions of a line of text in inverse video or with a colored background. Just as a caret is the onscreen representation of an insertion point in source text, highlighting is the onscreen representation of a selection range in source text.

A **selection range** is the contiguous sequence of characters in memory that the next editing operation (deletion or replacement) will affect. The onscreen glyphs corresponding to those characters are commonly highlighted. The characters in a selection range are always contiguous in memory, but their glyphs are not necessarily contiguous on screen.

Like insertion points and carets, selection ranges and highlighting are described by edge offsets and caret positions. Figure 10-9 shows the simplest example of highlighting. The text represents a layout shape with mixed Roman, Hebrew, and Chinese text in which the dominant direction is left to right. A selection range from edge offsets 1 to 3 yields a simple, contiguous highlighting rectangle. The rectangle encloses two glyphs that correspond exactly to the two characters in the selection range.

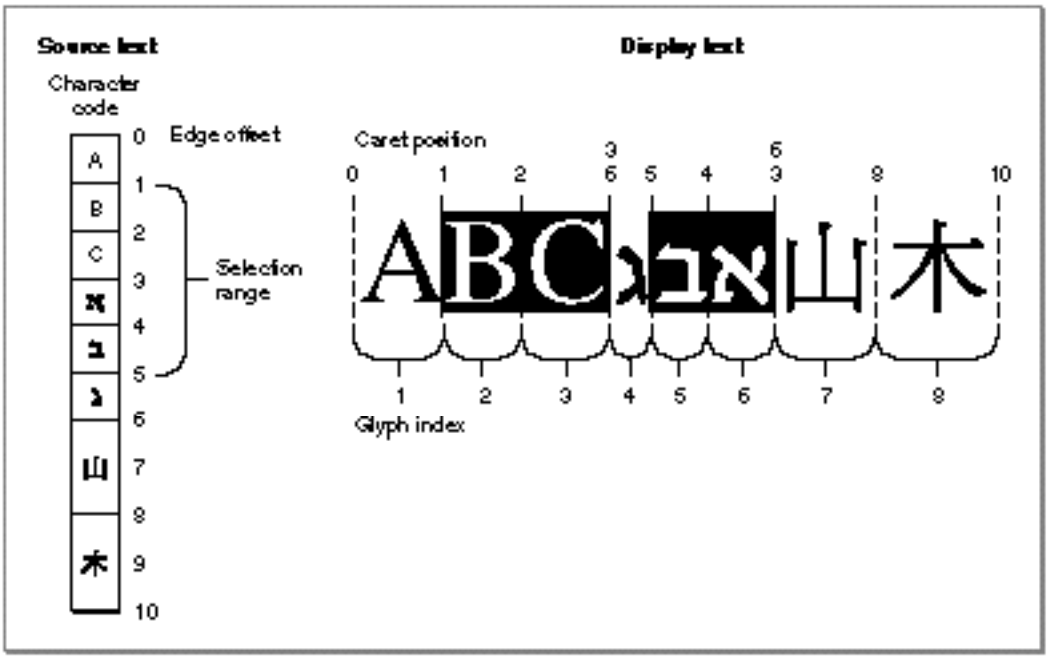
Figure 10-9 Highlighting in single-direction text



Visually Discontiguous and Contiguous Highlighting

Complications arise in highlighting when a selection range crosses direction boundaries. In Figure 10-10, which represents the same layout shape as Figure 10-9, the selection range extends from offset 1 to offset 5. The equivalent highlighted area on the screen consists of two separate rectangles. Note that the highlighted glyphs still correspond exactly to the characters in the selection range. In more complicated layout shapes with many direction boundaries and complex levels of direction runs, a single selection range can become many discontiguous highlighted areas on the screen.

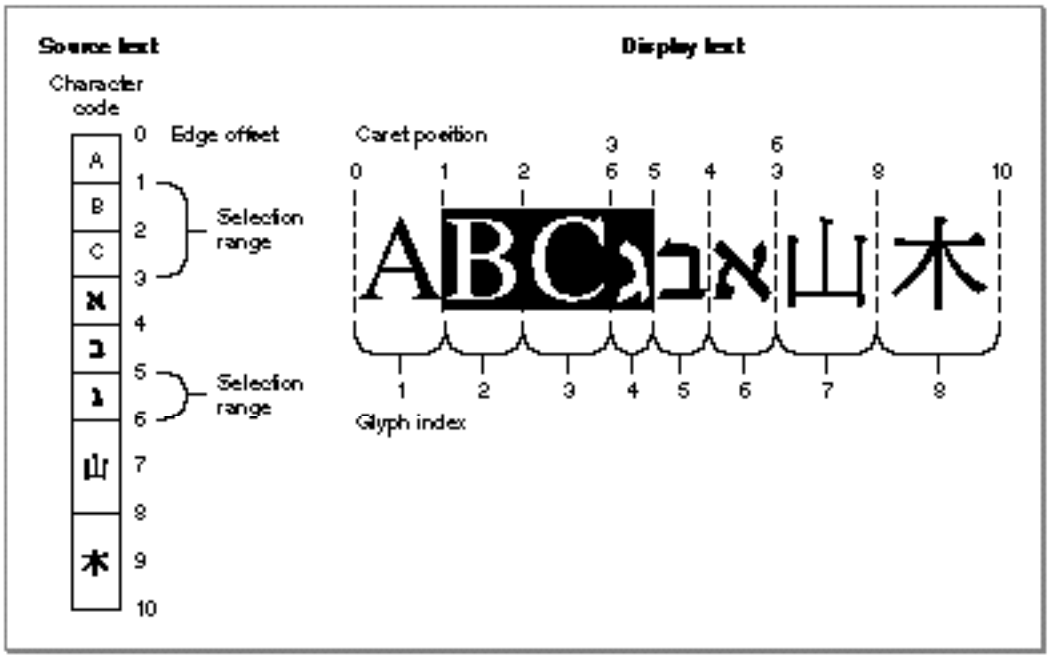
Figure 10-10 Discontiguous visual highlighting in mixed-direction text



The kind of highlighting shown in Figure 10-9 and Figure 10-10 is the most typical, and QuickDraw GX takes care of calculating the contiguous or discontiguous highlighting shapes that represent any selection range that you specify. However, for display simplicity, you can also specify that a single visually contiguous highlighting shape be used between two offsets, even if it crosses direction boundaries. Figure 10-11 shows an example of this visually contiguous highlighting, using the same layout shape and the same edge offsets as used by the selection range in Figure 10-10.

Note that the selection range represented by the highlighting in Figure 10-11 is *not* equivalent to the selection range represented in Figure 10-10, even though the same edge offsets apply in each case. (For contiguous highlighting, leading-edge information,

Figure 10-11 Contiguous visual highlighting in mixed-direction text



as discussed below under "Hit-Testing," is also required.) In general, when contiguous highlighting crosses direction boundaries in text, the selection range is discontiguous and does *not* correspond exactly to the characters between the two edge offsets represented.

Caret Angle and Tiled Highlighting

Just as QuickDraw GX supports angled carets within text that has slanted glyphs, it also allows you to slant the edges of highlighted areas in the same way. If you specify angled carets, the edges of highlighted areas will also be angled, where appropriate. In that case, your highlighting areas may be parallelograms or even trapezoids, instead of rectangles. See, for example, Figure 10-19 on page 10-27.

It is usually important to make all possible highlighted areas in a line of text unique. Every character's highlighted area should be disjoint from all other characters' areas, with the union of all characters' highlighted areas being exactly equal to the entire line's highlight area. In that case, there are no gaps or overlaps between adjacent highlight areas, and there is never any uncertainty in the interpretation of a hit-test for any point within the area of the line. Highlighting that meets these criteria is called **tiled highlighting.**

If you specify straight highlighting (by specifying straight carets), the highlighted areas calculated by QuickDraw GX are always tiled. If you specify angled highlighting, the highlighted areas calculated by QuickDraw GX are almost always tiled; only in cases of extreme slant with superscripts or subscripts is tiling not maintained.

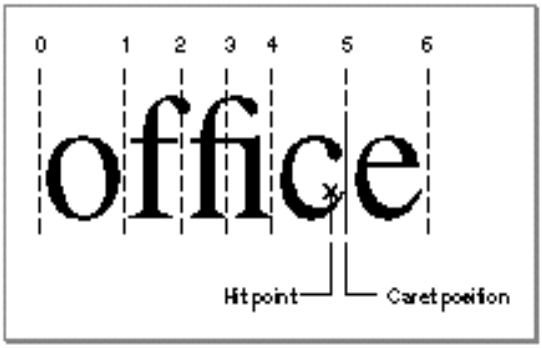
Hit-Testing

Hit-testing is the process of converting a location within a line of display text into an edge offset in the source text of that line. Its purpose is to allow you to convert user actions on displayed glyphs into editing operations on the characters of a typographic shape.

An important concept for hit-testing is that of leading and trailing edges. A glyph's **leading edge** is the edge of a glyph that is encountered first when reading text of that glyph's language. Its **trailing edge** is the edge encountered last. For glyphs of left-to-right text, the leading edge is the left edge; for glyphs of right-to-left text, the leading edge is the right edge.

Figure 10-12 shows the basics of hit-testing. The hit point (×) may represent, for example, the location of a mouse click. In response, your application needs to find the correct edge offset for subsequent text insertion and then draw a caret at the proper caret position.

Figure 10-12 Hit point and caret position in hit-testing



When used for hit-testing layout shapes, QuickDraw GX tells you which edge offset corresponds to the hit point and whether the hit was on the leading edge or the trailing edge of the hit glyph. In Figure 10-12, the hit point is within the area of the glyph "c" and closer to its trailing edge than its leading edge. The logical place to draw the caret is thus between glyphs "c" and "e." In this case, QuickDraw GX returns an edge offset of 5 and a leading-edge value of `false`. You can then pass that edge offset back to another QuickDraw GX function to obtain the correct caret to draw. QuickDraw GX returns a polygon shape of the right size, angle, and position (caret position 5).

If the hit point had been within glyph "e," near its leading edge, the returned edge offset would still have been 5 (because the proper caret position would still be 5), but the leading-edge value would have been `true`.

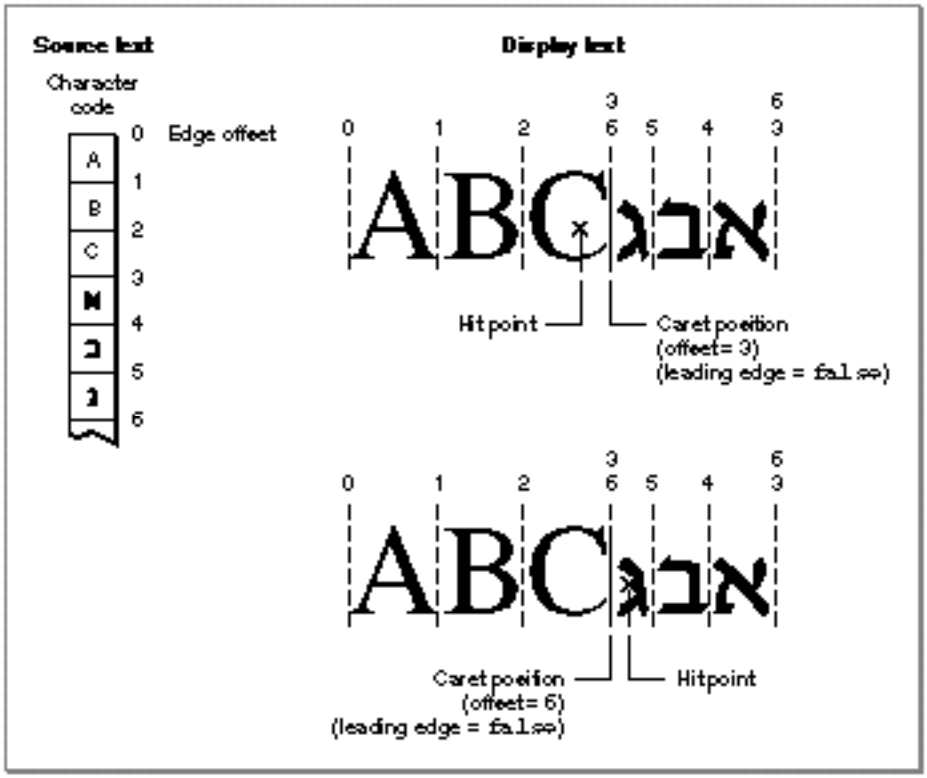
In single-direction text, the leading-edge value is not needed, but at direction boundaries in mixed-direction text both the offset and the leading-edge value are needed to know what kind of text insertion is expected—and where—in the source text. For a given

offset, if the leading edge value is `true`, the text to be inserted has the direction of the source-text character *following* the edge offset. If the leading edge value is `false`, the text to be inserted has the direction of the source-text character *preceding* the edge offset.

For example, Figure 10-13 shows mixed Roman and Hebrew text from the same layout shape as shown in Figure 10-11 on page 10-15. Neighboring hit points on either side of a direction boundary give two different results:

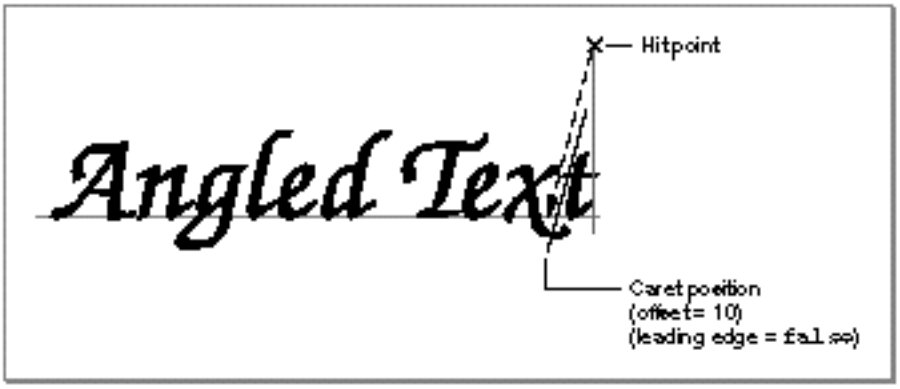
- The first hit point yields an edge offset of 3, which could mean insertion of either Roman text after the "c" or Hebrew text before the "ג". The leading-edge designation of `false` means that Roman text is to be entered at offset 3.
- The second hit point is close to the first but yields an offset of 6. That offset could mean insertion of either Hebrew text after the "ג" or left to right text before the character starting at offset 6. The leading-edge designation of `false` means that in this case Hebrew text is to be entered at offset 6.

Figure 10-13 Hit-testing in mixed-direction text



For angled text, the hit point is projected to the baseline, parallel to the caret angle, to determine the correct edge offset. As shown in Figure 10-14, the caret projection to the baseline can yield a completely different caret position than a vertical projection would.

Figure 10-14 Projecting the hit point to the baseline



QuickDraw GX returns other information besides the offset and leading-edge value when you use it for hit-testing layout shapes. You can use the information to tell how close the hit point was to either edge of the hit glyph, what the edge offset of either side of the hit glyph is, whether or not the hit point was actually within the area of the text line, and what area around the hit point would yield the same edge offset. See “Performing Hit-Testing” beginning on page 10-28 for more information and examples.

Using Carets, Highlighting, and Hit-Testing With Layout Shapes

This section describes how to use QuickDraw GX functions to draw carets, to highlight, to hit-test, and to analyze glyphs for their direction and for their relationship to characters in a layout shape’s source text.

Drawing Carets

QuickDraw GX provides functions that let you locate and draw a caret correctly in the display text of any layout shape. This section shows you how to draw single and split carets in simple and complex text, and how to change the caret position when the user presses the Right or Left Arrow key.

Getting the Caret Shape

The `GXGetLayoutCaret` function takes an edge offset in a layout shape and returns a shape that represents a text caret. Determining the caret's shape means determining its position on the screen, its angle in italic or otherwise slanted text, and its height (to correspond to the text size). In accordance with QuickDraw GX typographic conventions, the caret is a single line when positioned between two characters of like directionality, and possibly a split caret (a pair of short lines at different places) at the boundaries between text of opposing direction.

The caret returned by `GXGetLayoutCaret` has all those properties automatically specified. Because the caret is a shape object, it has associated style, ink, and transform objects. The style and ink have the same default properties as those for any polygon shape. You can specify details of a caret's appearance, such as its thickness, color, and transfer mode, by modifying its style or ink objects. The caret shape's transform object is always identical to the layout shape's transform, so that the caret matches any moved, scaled, or rotated text.

The `GXGetLayoutCaret` function generates a split caret at direction boundaries if you specify `gxSplitCaretType` for the `caretType` parameter. When italic text occurs in a line containing mixed directions and the caret is split, the top and bottom portions of the caret may have different slants.

Listing 10-1 is a partial listing of a test function that creates and draws a layout shape twice, with the text specified in the string `myString` (of byte length `len`). The function then determines the caret shape for each edge offset in the layout shape's source text and draws the caret. The first time, it specifies `gxHighlightAverageAngle` in `GXGetLayoutCaret` to ensure that if the text is angled, the caret will be angled also; the second time, it specifies `gxNoCaretAngle` to make the caret vertical regardless of the text slant. (Specifying `gxSplitCaretType` ensures that two carets are drawn for caret positions at direction boundaries in mixed-direction text.)

Listing 10-1 Drawing angled and straight carets at all caret positions

```
   char            *myString = "Angled Text";
   gxStyle         myStyle;
   gxPoint         myPoint;
   gxShape         caret, highlight, layout;
   short           i, len = 0;
   gxStyle         myStyle;
.
.
.
   len = strlen(myString);
   myStyle = NewLayoutStyle((char *)
                        "\pZapf Chancery Medium Italic", ff(50),
                        0, nil, nil, 0, nil);
```

```
layout = GXNewLayout( 1, &len, (void *) &myString,
                      1, &len, &myStyle,
                      0, nil, nil,
                      nil, &myPoint);
GXDrawShape(layout);

for (i = 0; i <= len; i++)
{
   caret = GXGetLayoutCaret(layout, i,
                         gxHighlightAverageAngle,
                         gxSplitCaretType, nil);
   GXDrawShape(caret);
   GXDisposeShape(caret);
}
GXMoveShape(layout, 0, ff(75));
GXDrawShape(layout);

for (i = 0; i <= len; i++)
{
   caret = GXGetLayoutCaret(layout, i,
                            gxNoCaretAngle,
                            gxSplitCaretType, nil);
   GXDrawShape(caret);
   GXDisposeShape(caret);
}
```

The result is two lines of display text with carets drawn at each caret position, as shown in Figure 10-15.

Figure 10-15 Drawing all possible caret positions in a layout shape's text

Listing 10-2 is a code fragment that demonstrates how to specify different caret types for a given offset in a layout shape. The code first creates a new layout shape with two style runs (one in Roman text and one in Arabic text; the text runs are specified in `textPtrs`, their lengths are specified in `runLengths`, their style objects in `styleArray`). It then draws the layout three times (starting at `myPoint`), each time drawing a caret at the screen position corresponding to edge offset 4 in the source text—the boundary between the Roman and Arabic text.

Listing 10-2 Drawing three different types of caret at one edge offset

```
/* create and draw a layout shape with two style runs */
layout = GXNewLayout(2, runLengths, (void *)textPtrs,
                     2, runLengths, styleArray,
                     0, nil, nil,
                     nil, &myPoint);
GXDrawShape(layout);

/* first draw a split caret at offset 4 */
caret = GXGetLayoutCaret(layout, 4, gxHighlightStraight,
                         gxSplitCaretType, nil);
GXDrawShape(caret);

/* move the layout shape downward and redraw */
GXMoveShape(layout, 0, ff(72));
GXDrawShape(layout);

/* next draw a left-to-right caret at offset 4 */
GXGetLayoutCaret(layout, 4, gxHighlightStraight,
                gxLeftRightKeyboardCaret, caret);
GXDrawShape(caret);

/* move the layout shape downward and redraw */
GXMoveShape(layout, 0, ff(72));
GXDrawShape(layout);

/* finally draw a right-to-left caret at offset 4 */
GXGetLayoutCaret(layout, 4, gxHighlightStraight,
                gxRightLeftKeyboardCaret, caret);
GXDrawShape(caret);
```

Figure 10-16 shows the results of executing this code. The layout is drawn with a split caret, then with a left-to-right caret, and finally with a right-to-left caret. Your application chooses which caret to draw during program execution; if you choose a single caret, you should choose the one that corresponds to the current direction (left to right or right to left) for text input by the user.

Figure 10-16 Drawing different caret types at a single edge offset

The `GXGetLayoutCaret` function is described on page 10-44. The `gxCaretType` enumeration is described on page 10-41. The `GXNewLayout` function is described in the chapter "Layout Shapes" in this book.

Drawing the Cursor at the Correct Angle Within a Given Area

In addition to controlling the shape, location, and style of the caret, you may also want to modify the cursor, as mouse movements cause it to pass over different parts of your text. For example, you may want to slant the standard Macintosh "I-beam" cursor when it passes over italic text.

You could do this by making repeated calls to `GXGetLayoutCaret` as the cursor moves, and using the resulting shape to calculate the proper angle for the cursor. The `GXGetCaretAngleArea` function returns the angle for the caret, and therefore for the cursor, corresponding to a given point in the display text. The function also returns the tracking area for that angle—the contiguous area in which the angle stays the same.

The `GXGetLayoutCaret` and `GXGetCaretAngleArea` functions are described on page 10-44 and page 10-46, respectively.

Positioning the Caret in Response to Arrow Keypresses

When the user presses the Right Arrow key, the caret should move one position to the right; when the user presses the Left Arrow key, the caret should move one position to the left. You can use the `GXGetRightVisualOffset` and `GXGetLeftVisualOffset` functions to determine where to place the caret when either the Right or Left Arrow key is pressed.

These functions take an edge offset in the source text and return an offset that you can pass to `GXGetLayoutCaret` (page 10-44) to get a caret at the next position. The `GXGetRightVisualOffset` function returns the edge offset corresponding to the next

caret position to the right (or down); `GXGetLeftVisualOffset` returns an edge offset corresponding to the next caret position to the left (or up). If you are using a split caret with your text, `GXGetRightVisualOffset` and `GXGetLeftVisualOffset` return a edge offset corresponding to the position next to the dominant (high) caret.

Listing 10-3 is a portion of a key-down event handler for a layout-editing library. The listing shows the response of the handler to arrow keypresses. In the left-arrow case, the handler calls `GXGetLeftVisualOffset` to get the offset for the next caret position, defines the selection range as that edge offset, and then highlights (draws the caret).

The function in Listing 10-3 uses the library-defined function and `ShowHighlight` to redraw the highlighting or caret, and `NewSelection` to update the library's layout data structures.

Listing 10-3 A key-down handler using `GXGetRightVisualOffset` and `GXGetLeftVisualOffset`

```
void LayoutEditKey(LayoutEditHandle handle, char key)
.
.
.
   switch (key)
   {
      case leftArrow:
      case rightArrow:
      {
         .
         .
         .

         if (key == leftArrow)
            newCaret = GXGetLeftVisualOffset(layout->layout,
                     layout->selectionRanges.ranges[0].minOffset);
         else
            newCaret = GXGetRightVisualOffset( layout->layout,
                     layout->selectionRanges.ranges[0].maxOffset);

         NewSelection(layout, newCaret, newCaret);
         ShowHighlight(layout);
         break;
      }

      case backSpace:
      {
         .
         .
         .
```

The `GXGetRightVisualOffset` and `GXGetLeftVisualOffset` functions are described on page 10-47 and page 10-48, respectively.

Positioning the Caret Within Ligatures

You can specify that caret positions within a ligature are not to be permitted, meaning that caret-drawing, highlighting, and hit-testing must consider that ligatures are indivisible glyphs. You do this by setting the `gxNoLigatureSplits` flag in the run controls structure of the style run containing the ligatures.

Listing 10-4 draws the same line of text twice, first without specifying any value for `gxNoLigatureSplits`, and then with the flag set. It first creates the layout shape at the location `myPoint`. The style object of the shape uses the run controls structure `runControls`.

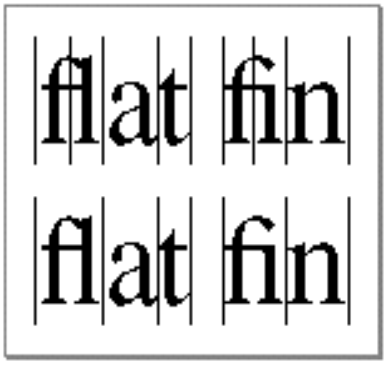
Listing 10-4 Preventing ligature splits for caret positioning

```
void LigatureSplits(WindowPtr sampleWindow)
   {
   char           *myString = "flat fin";
   short          i, len = 0;
   gxPoint        myPoint;
   gxRunControls  runControls;
   gxShape        caret, layout;
   gxStyle        myStyle;
   .
   .
   .
   myStyle = NewLayoutStyle((char *) "\pTimes Roman", ff(60),
                            0, nil, nil, 0, nil);

   layout = GXNewLayout(1, &len, (void *) &myString,
                        1, &len, &myStyle,
                        0, nil, nil,
                        nil, &myPoint);
   GXDrawShape(layout);

   for (i = 0; i < (len + 1); i++)
   {
      caret = GXGetLayoutCaret(layout, i,
                               gxHighlightAverageAngle,
                               gxSplitCaretType, nil);
      GXDrawShape(caret);
      GXDisposeShape(caret);
   }
```

```
runControls.flags = gxNoLigatureSplits;
GXSetStyleRunControls(myStyle, &runControls);
GXMoveShape(layout, 0, ff(75));
GXDrawShape(layout);

for (i = 0; i < (len + 1); i++)
{
   caret = GXGetLayoutCaret(layout, i,
                            gxHighlightAverageAngle,
                            gxSplitCaretType, nil);
   GXDrawShape(caret);
   GXDisposeShape(caret);
}
.
.
.
```

Figure 10-17 shows the results of executing the code in Listing 10-4. In the upper line, carets are drawn in the middle of the “fl” and “fi” ligatures; in the lower line, there are no valid caret positions within the ligatures.

Figure 10-17 All caret positions drawn with (upper) and without (lower) ligature splits



Drawing Highlighting

Highlighting single-direction text in a layout shape is straightforward. You call the `GXGetLayoutHighlight` function, passing it the two edge offsets between which you want the highlight to appear; the function returns a rectangular or trapezoidal shape that corresponds in size and location to the highlighted area. The with-stream edges of the highlight shape (the right and left edges, in horizontal text) are perpendicular or slanted, as appropriate, depending on whether the text is slanted and whether you have specified angled highlighting.

When you draw the highlight, you most typically use the highlight color specified by the user and the `gxHighlightMode` transfer mode.

The following code fragment creates and draws a layout shape with five style runs, one of which has a right-to-left text direction. It then highlights the area between edge offsets 4 and 13, all of which is within left-to-right text. In the call to `GXGetLayoutHighlight`, it specifies `gxHighlightAverageAngle` to slant the edge of the highlight that abuts slanted text. This code uses the library functions `SetShapeCommonTransfer` and `SetShapeCommonColor` to set up the ink object for the highlight shape.

```
   layout = GXNewLayout( 5, textLengths, (void *) textRuns,
                            5, textLengths, textStyles,
                            0, nil, nil,
                            &layoutOptions, &posn);
   GXDrawShape (layout);

/* make a highlight shape from offset 4 to offset 13 */
   highlight = GXGetLayoutHighlight (layout, 4, 13,
                                     gxHighlightAverageAngle, nil);
   SetShapeCommonTransfer (highlight, gxHighlightMode);
   SetShapeCommonColor (highlight, gxWhite);
   GXDrawShape(highlight);
```

Figure 10-18 shows the results of the highlighting.

Figure 10-18 Contiguous highlighting from offsets 4 to 13 in single-direction text

Highlighting Discontiguously in Mixed-Direction Text

In mixed-direction text, highlighting is just as straightforward as in single-direction text, although it can be more complex in appearance, and there are two possible highlights you may want to generate. The `GXGetLayoutHighlight` function returns a highlighted area that corresponds exactly to the selection range defined by the two edge offsets. The highlighting can be visually discontiguous when the selection range crosses direction boundaries.

The following code fragment highlights the selection range between edge offsets 4 and 19—crossing a direction boundary—in the same layout shape created in the previous

fragment. Like the previous example, this code calls `GXGetLayoutHighlight` and specifies `gxHighlightAverageAngle`.

```
/* make a highlight shape from offset 4 to offset 19 */
   highlight = GXGetLayoutHighlight (layout, 4, 19,
                                     gxHighlightAverageAngle, nil);
   SetShapeCommonTransfer (highlight, gxHighlightMode);
   SetShapeCommonColor (highlight, gxWhite);
   GXDrawShape(highlight);
```

Figure 10-19 shows the results of the highlighting. The selection range appears as two separate highlighted trapezoids.

Figure 10-19 Discontiguous highlighting from offsets 4 to 19 in mixed-direction text

Highlighting Contiguously in Mixed-Direction Text

The highlight shape returned by `GXGetLayoutHighlight` corresponds exactly to the selection range defined by the two offsets passed to it. However, the discontiguous nature of the highlighting in mixed-direction text may confuse some users, especially when the highlighting is dynamic—that is, when the highlighting is continually drawn and redrawn as the user moves the cursor through the text while holding down the mouse button.

You can obtain a simpler shape for highlighting by calling the `GXGetLayoutVisualHighlight` function. The `GXGetLayoutVisualHighlight` function always returns a single, visually contiguous highlighted area representing the visual range between the caret positions and leading-edge states of the two specified offsets, even when the text has mixed directions. Because the highlighted area is visually contiguous, the selection range it defines does not always correspond exactly to the range between the two offsets in the source text. If the user performs an editing operation (such as deleting, cutting, or pasting) on a part of the text that is highlighted continuously, you must be sure to replace the characters in the source text that represent the highlighted glyphs exactly, and not simply all the characters between the offsets used to generate that highlight.

The following code fragment again highlights the area between edge offsets 4 and 19 in the same layout shape created in the previous fragments. Unlike the previous examples, however, this code calls `GXGetLayoutVisualHighlight` instead of

GXGetLayoutHighlight. (Also, unlike the code that calls GXGetLayoutHighlight, this code must specify whether to highlight from the leading edge or trailing edge of the glyph corresponding to each offset.)

```
/* make contiguous highlighting from offset 4 to offset 19 */
   highlight = GXGetLayoutVisualHighlight (layout,
                                    4, true, 19, false,
                                    gxHighlightAverageAngle, nil);
   SetShapeCommonTransfer (highlight, gxHighlightMode);
   SetShapeCommonColor (highlight, gxWhite);
   GXDrawShape(highlight);
```

Figure 10-20 shows the results of the executing this code. Note that, even though the offsets passed to the highlighting functions are the same, some glyphs that are not highlighted in Figure 10-19 are highlighted Figure 10-20. Conversely, some glyphs that are highlighted in Figure 10-19 are not highlighted Figure 10-20.

Figure 10-20 Contiguous highlighting from offsets 4 to 19 in mixed-direction text

Providing Dynamic Highlighting

Dynamic highlighting is the process of continually drawing and redrawing the highlighted area as the user moves the cursor through the text while holding down the mouse button. Dynamic highlighting works in conjunction with hit-testing, described in the next section.

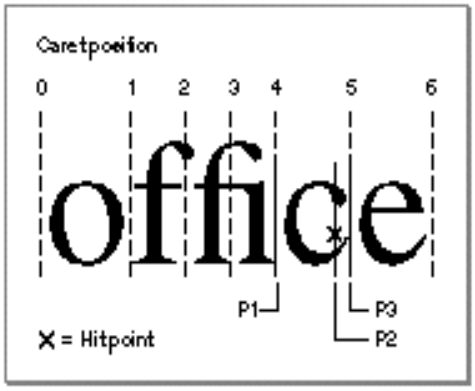
If your application uses GXHitTestLayout for hit-testing, your hit-test function can perform dynamic highlighting by making repeated calls to GXHitTestLayout (and GXGetLayoutHighlight or GXGetLayoutVisualHighlight) as long as the mouse button is held down. Listing 10-5 on page 10-31 shows an example of dynamic highlighting. (That example does not show the call to GXGetLayoutHighlight; that call occurs in the application-defined function NewSelectionAndHighlight.)

Performing Hit-Testing

You can use the GXHitTestLayout function to determine the source-text offset corresponding to a hit anywhere in the display text of a layout shape. The function returns (a) the edge offset corresponding to the closest edge of the glyph beneath the hit point and (b) the edge offset of the other edge of that same glyph.

For example, in Figure 10-21, the source text for the word “office” has edge offsets 0 through 6, and the display text consists of five glyphs (including the “fi” ligature). Above the glyphs, the numbers denote the edge offsets equivalent to the glyph edges.

Figure 10-21 `GXHitTestLayout` example



As you read the following sections, assume a hit occurs at the spot marked “×”, above point P2, near the trailing edge of the “c.” The next section discusses how `GXHitTestLayout` interprets the significance of the hit.

Layout Hit Info Structure

The `GXHitTestLayout` function returns information in a layout hit info structure, is described on page 10-43. That structure gives all the relevant information about the hit.

In Figure 10-21, the offset closest to the hit is 5, so in this case the value of the `hitSideOffset` field in the `layoutHitInfo` structure is 5. This value specifies the edge offset corresponding to the caret position between the glyphs “c” and “e.” The value of the `nonHitSideOffset` field is 4, which specifies the other side of the “c” glyph. If the hit had occurred on the left side of the “e” glyph, the value of `hitSideOffset` would still have been 5, but the value of `nonHitSideOffset` would have been 6.

In this example, the value of the `leadingEdge` field in the `layoutHitInfo` structure is `false`, because the hit occurred nearest the trailing edge of the glyph. If the hit had occurred on the left side of the glyph, or if the glyph had right-to-left directionality and the hit was on the right side, the `leadingEdge` field would have been `true`.

The `layoutHitInfo` structure contains two with-stream distances as well. In this example, the value of the `firstPartialDist` field is the left-side partial distance: the horizontal distance from the point P1 to the point P2 in this case. The value of `lastPartialDist` is the right-side partial distance: the horizontal distance from the point marked P2 to the point marked P3 in this case.

The `layoutHitInfo` structure also contains, in the `inLoose` field, an indication of whether or not the hit was actually within the area of the layout shape. In Figure 10-21, the value of the `inLoose` field is `true`, because the value of the hit point specifies that the hit occurred within the layout's area. If the hit point had been at the same horizontal position but above or below the line of text, all the returned information would have been the same, except that the value of `inLoose` would have been `false`. The `GXHitTestLayout` function projects the hit point to the baseline of the text so that your application can highlight the text dynamically, even if the mouse cursor strays out of the area of the layout shape while the mouse button is held down. (In multiline text, if the cursor strays far enough out of the layout shape's area, you would instead extend the dynamic highlight into the layout shapes that represent the other lines the cursor is passing over.)

If the hit had occurred in the middle of the "fi" ligature, the values of the two offsets returned in the `layoutHitInfo` structure would depend on whether the ligature is treated as a single glyph or as two partial glyphs; see "Caret Position and Split Ligatures" on page 10-10. If you had specified that ligatures are to be treated as single glyphs, and if the hit had occurred in the "fi" ligature in Figure 10-21, `GXHitTestLayout` would return either 2 or 4 as the value of `hitSideOffset`, depending on whether the hit point was closer to the "f" portion or the "i" portion of the ligature. If you had specified that ligatures are to be split, and the hit were near the center of the ligature, the value of `hitSideOffset` would be 3.

Mouse Tracking Area

The `GXHitTestLayout` function returns a modification of a shape that you pass it. The shape defines the mouse tracking area for the returned edge offset (`hitSideOffset`). The mouse tracking area is that area in the display text of the layout shape for which hits will yield the same edge offset. For repositioning the caret or for providing dynamic highlighting, for example, you can use the mouse tracking area to minimize the need for calls to `GXHitTestLayout`. In other words, you need to call `GXHitTestLayout` to get a new edge offset only when the mouse moves outside of this tracking area. You do not need to pass a shape if you don't wish to do mouse tracking in this way.

Sample Hit-Test Function

Listing 10-5 is a hit-test function from a layout-shape editing library. It demonstrates the use of `GXHitTestLayout` to convert mouse clicks to offsets for the purpose of drawing carets and highlighting. It also performs dynamic highlighting while the user holds down the mouse button.

This function refers to the layout shape with a pointer (`layout`), and uses the library-defined functions `GetShapeViewPort` (to get the layout's view port), `NewSelectionAndHighlight` (to update the selection range and highlight shape), and `DisposeShapeAt` (to dispose of shapes).

Listing 10-5 Using the `GXHitTestLayout` function

```
void LayoutEditClick(LayoutEditHandle handle, gxPoint hitDown)

/* lock the layout edit handle, initialize variables */
{
   LayoutEditPtr     layout = LockEditHandle(handle);
   gxPoint           lastPoint = hitDown;
   SelectionOffset   firstHitOffset, lastHitOffset;
   gxLayoutHitInfo   hitInfo;
   boolean           oldIsCaret, newIsCaret = true;
   gxShape           diffHighlight = nil, oldHighlight = nil;
   gxViewPort layoutViewPort = GetShapeViewPort(layout->layout);

   /* get the offset for the hit point */
   GXHitTestLayout(layout->layout, &hitDown,
                  gxHighlightAverageAngle, &hitInfo, nil);
   firstHitOffset = lastHitOffset =
                        (SelectionOffset) hitInfo.hitSideOffset;

   /* erase the old highlight (stored in layout edit structure) */
   GXDrawShape(layout->highlight);

   /* make the selection a caret at firstHitOffset */
   NewSelectionAndHighlight(layout, firstHitOffset,
                           firstHitOffset);
   GXDrawShape(layout->highlight);

   /*
      Recompute the selection and the highlight as long as the
      mouse button is still down.
   */
   while (Button())
   {
      GXGetViewPortMouse(layoutViewPort, &hitDown);

      /* continue if the mouse hasn't moved */
      if (hitDown.x == lastPoint.x &&
                              hitDown.y == lastPoint.y) continue;

      lastPoint = hitDown;
      GXHitTestLayout(layout->layout, &hitDown,
                        gxHighlightAverageAngle, &hitInfo, nil);
```

```
      /* continue if the selection hasn't changed */
      if (hitInfo.hitSideOffset == lastHitOffset) continue;

      /* save the old highlight and calculate the new one */
      oldIsCaret = newIsCaret;
      lastHitOffset = (SelectionOffset) hitInfo.hitSideOffset;
      newIsCaret = (lastHitOffset == firstHitOffset);

      if (oldIsCaret != newIsCaret)
      /*
         If it has changed from a caret to a highlight
         or vice versa, redraw the entire highlight or caret.
      */
      {
         GXDrawShape(layout->highlight);           /* erase old */
         NewSelectionAndHighlight(layout, firstHitOffset,
                                  lastHitOffset);
         GXDrawShape(layout->highlight);           /* draw new */
      }
      else
      /*
         Otherwise, to reduce flicker, draw only the difference
         between the new and the old highlight.
      */
      {
         oldHighlight = GXCopyToShape(oldHighlight,
                                      layout->highlight);
         NewSelectionAndHighlight(layout, firstHitOffset,
                                  lastHitOffset);
         diffHighlight = GXCopyToShape(diffHighlight,
                                      layout->highlight);
         GXExcludeShape(diffHighlight, oldHighlight);
         GXDrawShape(diffHighlight);         /* draw difference */
      }
   }
   DisposeShapeAt(&diffHighlight);
   DisposeShapeAt(&oldHighlight);
}
```

The layout hit info structure is described on page 10-43.

Analyzing Glyphs

You can use the functions described in this chapter to analyze several aspects of the relationship between a glyph and its characters. You can determine the directionality of a glyph, you can determine the edge offsets corresponding to both edges of a ligature, you can convert a glyph index to an edge offset, and you can convert an edge offset to a glyph index.

Determining the Direction of a Glyph

To determine the direction of a glyph in a complex layout shape that has mixed-direction text and possibly several levels of direction runs, you can call the `GXHitTestLayout` function for the left side of the glyph and test the value of the `leadingEdge` flag in the `gxLayoutHitInfo` structure returned by the function. If the value of the flag is `true`, the glyph is left to right; otherwise it is right to left. (Vertical text is always considered to be left to right by QuickDraw GX.)

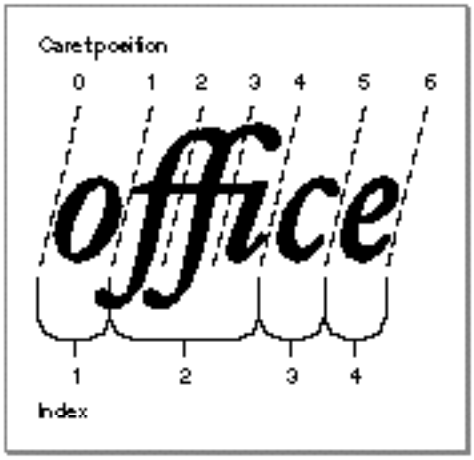
You can use the `GXGetLayoutGlyphs` function to obtain the location of any glyph in the layout shape, and pass that information to `GXHitTestLayout` to get the flag value.

Determining the Offsets for Each Edge of a Ligature

You can use the `GXGetCompoundCharacterLimits` function to find the bounding offsets equivalent to the leading and trailing edges of a character. The function is especially useful for determining the bounding offsets of a ligature, which may represent several characters.

For example, the text of a layout shape containing the word “office” is displayed in Figure 10-22. The source text has edge offsets 0 through 6, the equivalent caret positions for which are shown in the display text. If the “f,” “f,” and “i” characters are represented by a single “ffi” ligature, then the display text itself consists of only four glyphs, with glyph indices as shown.

Figure 10-22 Caret positions and glyph indexes for one display version of the word “office”



You pass the GXGetCompoundCharacterLimits function a trial offset: an edge offset bordering on or interior to the glyph of interest. The GXGetCompoundCharacterLimits function returns a minimum offset and a maximum offset representing the bounding edges of that glyph. If the trial offset is between glyphs in the display text, GXGetCompoundCharacterLimits notifies you of that condition and considers both bounding glyphs as a single glyph for the purposes of computing minimum and maximum offset.

For example, if you call GXGetCompoundCharacterLimits for the text in the layout shape shown in Figure 10-22, passing it in turn each possible offset, the function returns the following results:

Trial offset	Minimum offset	Maximum offset	On boundary?
0	0	1	true
1	0	4	true
2	1	4	false
3	1	4	false
4	1	5	true
5	4	6	true
6	5	6	true

Finding the Equivalent Glyphs to an Offset in the Source Text

You can directly convert an edge offset into an identification of an individual glyph or glyphs in the display text of a layout shape. This conversion is useful, for instance, if you want to substitute one glyph for another or to perform a graphics operation on a glyph at a particular offset.

The function shown in Listing 10-6 locates and draws a box around the glyph whose trailing edge corresponds to edge offset 6 in the source text of a layout shape. It uses the GXGetOffsetGlyphs function for that purpose. Depending on information in the run-features array of the style object associated with the layout shape—which in this case controls which ligatures may be formed—that one offset may correspond to different glyphs. Listing 10-6 draws the layout shape and highlights the glyph at offset 6 three times, one for each of three different ligature settings.

The listing first creates the layout shape at the location myPoint, and uses the library-defined function NewLayoutStyle to create a style object. The length of the string is len.

Listing 10-6 Converting an edge offset to a glyph index

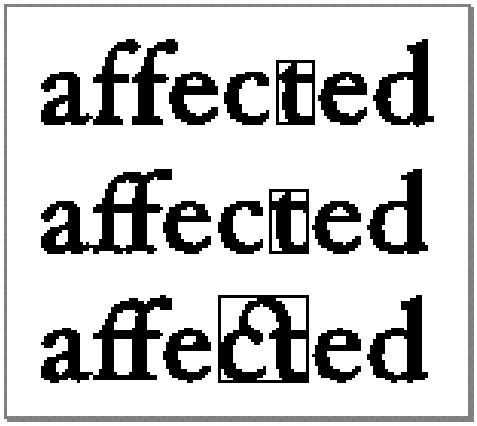
```
char                 *myString = "affected";
gxLayoutOffsetState  offsetState;
gxRectangle          boundingBoxes[20];
gxRunFeature         runFeature[3];
```

```
gxShape              layout;
short                len;
gxStyle              myStyle;
unsigned short       firstGlyph, secondGlyph;
.
.
.
   /* set up the style object and run features array */
   myStyle = NewLayoutStyle((char *) "\pHoefler Text", ff(36), 0,
                            nil, nil, 0, nil);
   runFeature[0].featureType = ligaturesType;
   runFeature[0].featureSelector = requiredLigaturesOffSelector;
   runFeature[1].featureType = ligaturesType;
   runFeature[1].featureSelector = commonLigaturesOffSelector;
   runFeature[2].featureType = ligaturesType;
   runFeature[2].featureSelector = rareLigaturesOffSelector;
   GXSetStyleRunFeatures(myStyle, 3, runFeature);

/* create and draw the layout with no ligatures */
layout = GXNewLayout(1, &len, (void *) &myString,
                     1, &len, &myStyle,
                     0, nil, nil,
                     nil, &myPoint);
GXDrawShape(layout);

/*
   Draw a frame around the glyph whose trailing edge corresponds
   to edge offset 6. In this case, it’s glyph 6 (index = 6).
*/
GXGetOffsetGlyphs(layout, 6, false, &offsetState,
                  &firstGlyph, &secondGlyph);
GXGetGlyphMetrics(layout, nil, boundingBoxes, nil);
GXDrawRectangle(boundingBoxes + firstGlyph - 1,
                  gxClosedFrameFill);
GXDisposeShape(layout);

/* reinitialize the style; this time allow normal ligatures */
runFeature[0].featureSelector = requiredLigaturesOnSelector;
runFeature[1].featureSelector = commonLigaturesOnSelector;
GXSetStyleRunFeatures(myStyle, 3, runFeature);
myPoint.y += ff(50);
```

```
/* re-create and redraw the layout a second time */
layout = GXNewLayout(1, &len, (void *) &myString,
                     1, &len, &myStyle,
                     0, nil, nil,
                     nil, &myPoint);
GXDrawShape(layout);

/*
   Draw a frame again at the same offset and trailing edge;
   now it encloses a ligature whose glyph index is 5.
*/
GXGetOffsetGlyphs(layout, 6, false, &offsetState,
                  &firstGlyph, &secondGlyph);
GXGetGlyphMetrics(layout, nil, boundingBoxes, nil);
GXDrawRectangle(boundingBoxes + firstGlyph - 1,
                gxClosedFrameFill);
GXDisposeShape(layout);

/* Re-initialize the style; include all the optional ligatures */
runFeature[2].featureSelector = rareLigaturesOnSelector;
GXSetStyleRunFeatures(myStyle, 3, runFeature);
myPoint.y += ff(50);

/* re-create and redraw the layout a third time */
layout = GXNewLayout(1, &len, (void *) &myString,
                     1, &len, &myStyle,
                     0, nil, nil,
                     nil, &myPoint);
GXDrawShape(layout);

/*
   Draw the frame once more at the same offset and trailing edge;
   now it encloses the ligature whose glyph index is 4.
*/
GXGetOffsetGlyphs(layout, 6, false, &offsetState,
                  &firstGlyph, &secondGlyph);
GXGetGlyphMetrics(layout, nil, boundingBoxes, nil);
GXDrawRectangle(boundingBoxes + firstGlyph - 1,
                gxClosedFrameFill);
.
.
.
```

Figure 10-23 shows the results of executing the function in Listing 10-6. The same edge offset results in a glyph with a different glyph index being framed in each case.

Figure 10-23 Using `GXGetOffsetGlyphs` to locate glyphs corresponding to known offsets



The function in Listing 10-6 uses the glyph index and a call to `GXGetGlyphMetrics` for the purpose of getting the rectangle shape to draw around the proper glyph in each case. If, after obtaining the glyph index, you need the actual glyph code that identifies the glyph within its font, you can call the `GXGetLayoutGlyphs` function. The `GXGetLayoutGlyphs` function is described in the chapter "Layout Shapes" in this book. The `GXGetGlyphMetrics` function is described in the chapter "Typographic Shapes" in this book.

Finding the Equivalent Offset to a Glyph in the Display Text

You can directly convert a glyph index (plus leading edge information) into an edge offset in the source text of a layout shape. You may need this conversion, for instance, to obtain the character code for a particular glyph, or to substitute one character for another in a layout shape's source text.

The function shown in Listing 10-7 locates the character associated with glyph 17 in a layout shape's display text and then highlights that character's glyph. First it calls the `GXGetGlyphOffset` function to get the offset for the leading edge of glyph 17 and then calls `GXGetLayoutHighlight` to highlight the region from that offset to the next. Like the function in Listing 10-6 on page 10-34, this function sets values in the run features array of the layout shape's style object to control which ligatures may be formed. Listing 10-7 draws the layout shape and highlights glyph 17 three times, once for each of three different ligature settings.

This listing draws the initial layout shape at the location `myPoint`, and uses the library-defined functions `NewLayoutStyle` (to create a style object), `SetShapeCommonTransfer` (to assign a special transfer mode for fast highlighting).

Listing 10-7 Converting a glyph index to an edge offset

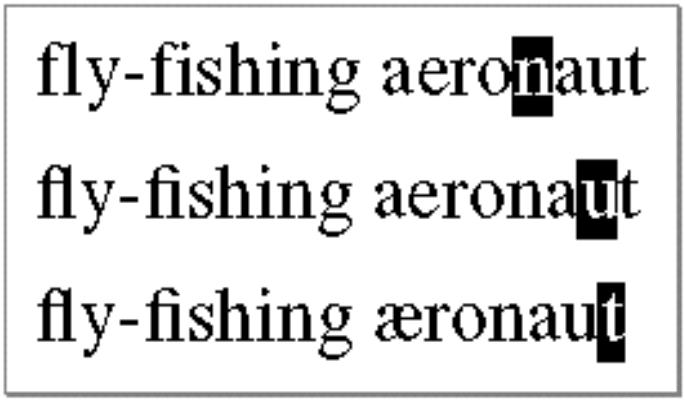
```
boolean        leadingEdge, wasRealCharacter;
char           *myString = "fly-fishing aeronaut";
gxByteOffset   offset;
gxRunFeature   runFeature[3];
gxShape        highlight, layout;
short          len = 0;
gxStyle        myStyle;
.
.
.
len = strlen(myString);

   /* set up the style object and run features array */
myStyle = NewLayoutStyle((char *) "\pTimes Roman", ff(36),
                          0, nil, nil, 0, nil);
   runFeature[0].featureType = ligaturesType;
   runFeature[0].featureSelector = requiredLigaturesOffSelector;
   runFeature[1].featureType = ligaturesType;
   runFeature[1].featureSelector = commonLigaturesOffSelector;
   runFeature[2].featureType = ligaturesType;
   runFeature[2].featureSelector = diphthongLigaturesOffSelector;
GXSetStyleRunFeatures(myStyle, 3, runFeature);

/* create and draw the layout, with no ligatures */
layout = GXNewLayout(1, &len, (void *) &myString,
                     1, &len, &myStyle,
                     0, nil, nil,
                     nil, &myPoint);
GXDrawShape(layout);

/*
   Get the offset for left edge of glyph 17 ( = “n”),
   then highlight from that offset to the next offset.
   (This simple example assumes a 1-byte character code.)
*/
GXGetGlyphOffset(layout, 17, true, &offset,
                  &leadingEdge, &wasRealCharacter);
highlight = GXGetLayoutHighlight(layout, offset, offset + 1,
                                 gxHighlightAverageAngle, nil);
SetShapeCommonTransfer(highlight, gxHighlightMode);
GXDrawShape(highlight);
GXDisposeShape(layout);
GXDisposeShape(highlight);
```

```
/* reinitialize the style; this time allow normal ligatures */
runFeature[0].featureSelector = requiredLigaturesOnSelector;
runFeature[1].featureSelector = commonLigaturesOnSelector;
GXSetStyleRunFeatures(myStyle, 3, runFeature);
myPoint.y += ff(50);

/* re-create and redraw the layout a second time */
layout = GXNewLayout(1, &len, (void *) &myString,
                     1, &len, &myStyle,
                     0, nil, nil,
                     nil, &myPoint);
GXDrawShape(layout);

/* locate and highlight glyph 17 again; this time it's "u" */
GXGetGlyphOffset(layout, 17, true, &offset,
                  &leadingEdge, &wasRealCharacter);
highlight = GXGetLayoutHighlight(layout, offset, offset + 1,
                                 gxHighlightAverageAngle, nil);
SetShapeCommonTransfer(highlight, gxHighlightMode);
GXDrawShape(highlight);
GXDisposeShape(layout);
GXDisposeShape(highlight);

/* reinitialize the style; allow all optional ligatures */
runFeature[2].featureSelector = diphthongLigaturesOnSelector;
GXSetStyleRunFeatures(myStyle, 3, runFeature);
myPoint.y += ff(50);

/* re-create and redraw the layout a third time */
layout = GXNewLayout(1, &len, (void *) &myString,
                     1, &len, &myStyle,
                     0, nil, nil,
                     nil, &myPoint);
GXDrawShape(layout);

/* locate and highlight glyph 17 once more; this time it's "t" */
GXGetGlyphOffset(layout, 17, true, &offset,
                  &leadingEdge, &wasRealCharacter);
highlight = GXGetLayoutHighlight(layout, offset, offset + 1,
                                 gxHighlightAverageAngle, nil);
SetShapeCommonTransfer(highlight, gxHighlightMode);
GXDrawShape(highlight);
```

```
GXDisposeShape(layout);
GXDisposeShape(highlight);
.
.
.
```

Figure 10-24 shows the results of executing the function Listing 10-7. Note that in each case, glyph 17 corresponds to a different offset in the source text.

Figure 10-24 Using `GXGetGlyphOffset` to locate a glyph's character



Layout Carets, Highlighting, and Hit-Testing Reference

This section provides reference information to the constants, data structures, and functions that allow you to

- draw carets and highlighting properly in the display text of layout shapes
- hit-test the display text of layout shapes
- convert between edge offsets in the source text and glyph indexes in the display text of layout shapes

Constants and Data Types

This section describes the `gxHighlightType`, `gxCaretType`, and `gxLayoutOffsetState` enumerations, and the `gxLayoutHitInfo` structure.

Highlighting Type

The `gxHighlightType` enumeration controls the shape that QuickDraw GX returns for carets and highlighting in a layout shape.

```
enum {
   gxHighlightStraight     = 0,
   gxHighlightAverageAngle = 1
};
typedef unsigned long gxHighlightType;
```

Constant descriptions

`gxHighlightStraight`
: The highlighting is perpendicular to the baseline.

`gxHighlightAverageAngle`
: The highlighting is slanted at the angle specified by the font for slanted highlights, or at the average of the two angles at the boundary between text of different slants.

Note

The run control flag `gxNoCaretAngle` can override the effects of the highlight type selected. However, for basic layout highlighting, selecting a highlight type using the `gxHighlightType` data type is sufficient. For more information about run control flags, see the chapter "Layout Styles" in this book. u

Caret Type

The `gxCaretType` enumeration controls the kind of caret that QuickDraw GX returns via the `GXGetLayoutCaret` function (page 10-44).

```
enum {
   gxSplitCaretType           = 0,
   gxLeftRightKeyboardCaret   = 1,
   gxRightLeftKeyboardCaret   = 2
   };
typedef unsigned long gxCaretType;
```

Constant descriptions

`gxSplitCaretType`
: The preferred caret to use at all times. If all the text is unidirectional, the caret always appears as one piece. It appears as two partial carets at direction boundaries. When split, the high caret appears at the caret location for insertion of text in the dominant direction; the low caret appears at the caret location for insertion of text in the opposite direction.

`gxLeftRightKeyboardCaret`
: A single caret that appears at the proper location for left-to-right text entry.

`gxRightLeftKeyboardCaret`
: A single caret that appears at the proper location for right-to-left text entry.

In general, you should always use a split caret.The other two types of carets should be used only when providing a choice of directionality for mixed-direction text. Note that, if you select a single caret type, you must always synchronize it with the direction (left to right or right to left) associated with the user's current text-entry direction.

Layout Offset State

The layout offset state is one of the values returned by the `GXGetOffsetGlyphs` function (page 10-56). It gives an indication of where the specified edge offset lies in relation to adjacent characters in the source text.

```
enum {
   gxOffset8_8             = 0,
   gxOffset8_16            = 1,
   gxOffset16_8            = 2,
   gxOffset16_16           = 3,
   gxOffsetInvalid         = 4,
   gxOffsetInsideLigature  = 0x8000
   };
typedef unsigned short gxLayoutOffsetState;
```

Constant descriptions

`gxOffset8_8` The specified offset corresponds to the edge of a glyph and is the boundary between two 8-bit characters.

`gxOffset8_16` The specified offset corresponds to the edge of a glyph and is the boundary between an 8-bit character code and a (following) 16-bit character code.

`gxOffset16_8` The specified offset corresponds to the edge of a glyph and is the boundary between a 16-bit character code and a (following) 8-bit character code.

`gxOffset16_16` The specified offset corresponds to the edge of a glyph and is the boundary between two 16-bit character codes.

`gxOffsetInvalid`
: The specified offset is interior to a 16-bit character code.

`gxOffsetInsideLigature`
: The specified offset corresponds to the interior of a ligature glyph. This value can be returned in addition to another `gxLayoutOffsetState` value.

At offsets marking the beginning and end of the source text for the line, at which there is only one bounding character, the layout offset state has the value `gxOffset8_8` or `gxOffset16_16`, depending on the size of the bounding character code.

Layout Hit Info Structure

The layout hit info structure (type `gxLayoutHitInfo`) contains the information that is returned by the `GXHitTestLayout` function (page 10-54).

```
typedef struct {
   Fixed          firstPartialDist;
   Fixed          lastPartialDist;
   gxByteOffset   hitSideOffset;
   gxByteOffset   nonHitSideOffset;
   boolean        leadingEdge;
   boolean        inLoose;
} gxLayoutHitInfo;
```

Field descriptions

`firstPartialDist`
: The (with-stream) distance from the left or top edge of the hit glyph to the actual hit point.

`lastPartialDist`
: The (with-stream) distance from the right or bottom edge of the hit glyph to the actual hit point.

`hitSideOffset`
: The edge offset corresponding to one edge of the hit ligature (the edge that is closer to the hit point). For example, you do not get the offset from the other edge of the ligature. However, see also the discussion of the `gxNoLigatureSplits` flag (below).

`nonHitSideOffset`
: The edge offset corresponding to the other edge of the hit glyph (the edge that is farther from the hit point). However, see also the discussion of the `gxNoLigatureSplits` flag (below).

`leadingEdge`
: A Boolean value specifying whether the hit occurred closer to the leading edge of the hit glyph (`true`), or closer to its trailing edge (`false`).

`inLoose`
: A Boolean value specifying whether the hit occurred within the highlighted area. This value is `true` if the hit occurred within the highlighted area of the shape as a whole. In general, hits outside of this area would not be considered hits within the text of the layout shape. Nevertheless, even when the value of `inLoose` is `false`, the layout hit info reflects the correct projection of the hit point to the baseline in the layout shape. You can therefore use `GXHitTestLayout` to perform dynamic highlighting, even when the mouse drifts outside the line of text being highlighted.

There is an interaction between the `hitSideOffset` and `nonHitSideOffset` fields and the state of the `gxNoLigatureSplits` run controls flag (described in this book in the chapter "Layout Styles") associated with the style run in which the hit occurred. If the hit occurred on a glyph that is a ligature (encompassing more than one character in the source text), the `gxNoLigatureSplits` flag controls whether just the two outermost offsets of the ligature are returned (flag is set) or whether the appropriate intermediate offsets are generated (the default case, where the flag is clear).

Functions

This section describes the QuickDraw GX functions that allow you to

- manipulate carets
- highlight text
- perform hit-testing
- convert between characters and glyphs

Manipulating Carets in a Layout Shape

The functions described in this section allow you to obtain a caret shape, determine the area within which a given caret shape is valid, and move the caret position properly when an arrow key is pressed.

GXGetLayoutCaret

You can use the `GXGetLayoutCaret` function to obtain a shape that describes the caret for a given edge offset in the source text of a layout shape.

```
gxShape GXGetLayoutCaret(gxShape layout, gxByteOffset offset,
                         gxHighlightType highlightType,
                         gxCaretType caretType, gxShape caret);
```

`layout` A reference to the layout shape whose caret you need to draw.

`offset` The edge offset defining the insertion point in the source text.

`highlightType` The angle of caret to use (perpendicular or oblique), of type `gxHighlightType`.

`caretType` The type of caret to use (single or split), of type `gxCaretType`.

`caret` A reference to a shape object. You may supply an existing caret shape here for `GXGetLayoutCaret` to reuse; if you pass `nil` for this parameter, `GXGetLayoutCaret` allocates a new shape to return in its function result.

function result The shape describing the caret for the insertion point specified by the `offset` parameter. If you pass an existing shape in the `caret` parameter, `GXGetLayoutCaret` modifies the shape as necessary and returns it; otherwise, `GXGetLayoutCaret` returns a new shape.

DESCRIPTION

The `GXGetLayoutCaret` function returns a shape that you can use to draw a caret with correct form and locations in the display text of a layout shape, given the edge offset value that you pass to the function. In simple horizontal single-direction text, `GXGetLayoutCaret` returns a caret shape that is a vertical bar between two glyphs. In italic text, the caret is angled (if you specify the oblique highlight type). At direction boundaries in mixed-direction text (and in text that has undergone linguistic rearrangement), the caret is split into two separate halves (if you specify a split caret in the `caretType` parameter). If the input edge offset corresponds to an interior point in a ligature, the resulting caret is located on the edge of the ligature if the `gxNoLigatureSplits` flag (in the run controls structure of the style run that includes the offset) is set; otherwise, the caret is located at a point within the ligature.

If you pass `nil` for the `caret` parameter, `GXGetLayoutCaret` creates a new caret shape and returns it as the function result. You can also pass an existing caret shape to save QuickDraw GX the overhead of disposing of one shape and creating another. If you pass an existing shape, `GXGetLayoutCaret` does not change the shape's fill or any of its style or ink values. If `GXGetLayoutCaret` creates a new shape, however, it sets the shape fill to frame fill. If the layout shape has a transform, the caret shape this function returns has the same transform.

The `highlightType` parameter controls the angle of the with-stream edges of the highlighting shape (the left and right edges for horizontal text). If the highlight type is `gxHighlightStraight`, the highlighting shape has edges that are perpendicular to the baseline, and the highlighting is always tiled (contiguous and nonambiguous across boundaries of text with different slant). If the highlight type is `gxHighlightAverageAngle`, the angle of the edge of the highlighting area is the average of the slants of the two glyphs on either side of the edge. In this case also, highlighting is usually tiled; highlighting is not tiled only in cases of extreme slant coupled with superscripts or subscripts. (In such a case a triangular highlighting area may result.)

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
```

SEE ALSO

For a description of how carets appear in a layout shape, see "Drawing Highlighting" on page 10-25. Caret types are described on page 10-41. Highlighting types are described on page 10-41.

For examples of the use of the `GXGetLayoutCaret` function, see page 10-19.

The `gxNoLigatureSplits` flag is described with the run controls structure in the chapter "Layout Styles" in this book.

GXGetCaretAngleArea

You can use the `GXGetCaretAngleArea` function to get (a) the angle needed to draw a cursor in italic text as well as (b) the area in the display text within which that angle is valid.

```
gxShape GXGetCaretAngleArea(gxShape layout,
                            const gxPoint *hitPoint,
                            gxHighlightType highlightType,
                            gxShape caretArea,
                            short *returnedRise,
                            short *returnedRun);
```

`layout` A reference to the layout shape whose caret-angle information you need.

`hitPoint` A pointer to a point structure that contains the location for which you need the caret angle. The location is in local (view port) coordinates.

`highlightType`
The kind of caret (perpendicular or oblique) that would be drawn in this text, of type `gxHighlightType`.

`caretArea` A reference to a shape object. You may supply an existing caret-area shape here for `GXGetCaretAngleArea` to reuse; if you pass `nil` for this param-eter, `GXGetCaretAngleArea` allocates a new shape to return in its function result.

`returnedRise`
A pointer to a `short` value. On return, it contains the vertical component of the caret angle.

`returnedRun`
A pointer to a `short` value. On return, it contains the horizontal component of the caret angle.

function result The shape describing the caret area for the point specified by the `hitPoint` parameter. The caret area is the area of the display text within which the caret or cursor can move and retain the same angle. If you pass an existing shape in the `caretArea` parameter, `GXGetCaretAngleArea` modifies the shape as necessary and returns it; otherwise, `GXGetCaretAngleArea` returns a new shape.

DESCRIPTION

The `GXGetCaretAngleArea` function helps you draw the cursor (the small icon, commonly an "I-beam" shape or arrow, that moves with mouse movements) at the proper angle for any slanted or italic text. The function provides two kinds of information. First, it returns the angle of the caret (and therefore the cursor) for caret positions in the vicinity of the hit point. Second, it returns a shape describing the area within which the cursor can move and retain the given angle.

The shape returned by `GXGetCaretAngleArea` can be used as a cursor-tracking area. As long as the cursor does not move outside of it, the shape of the cursor need not change. Thus, you can minimize the calls you need to make to `GXGetCaretAngleArea` or `GXGetLayoutCaret` as the user moves the cursor across the text.

If you pass `nil` for the `caretArea` parameter, `GXGetCaretAngleArea` creates a new caret-area shape and returns it as the function result. You can also pass an existing caret-area shape to save QuickDraw GX the overhead of disposing of one shape and creating another. If the layout shape has a transform, the caret-area shape this function returns has the same transform.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
```

SEE ALSO

Highlight types are described on page 10-41. The `GXGetLayoutCaret` function is described in the previous section.

GXGetRightVisualOffset

You can use the `GXGetRightVisualOffset` function to determine the edge offset corresponding to the next caret position to the right (or downward, for vertical text) in a layout shape.

```
gxByteOffset GXGetRightVisualOffset(gxShape layout,
                                    gxByteOffset currentOffset);
```

`layout` A reference to the layout shape whose right visual offset you need.

`currentOffset`
The edge offset in the source text corresponding to the current caret position in the display text.

function result The edge offset in the source text corresponding to the next rightward (or downward) caret position in the display text.

DESCRIPTION

The `GXGetRightVisualOffset` function determines the next caret position to the right (or down) in the display text of a layout shape. This is where the caret would move if the user pressed the Right or Down Arrow key. The function takes into account the text direction (left to right or right to left). It also considers whether the text has undergone linguistic rearrangement. If the caret passes through a ligature, the resulting offset depends on whether the `gxNoLigatureSplits` flag is set. If so, the caret position passes entirely across the ligature; if not, the caret position can be interior to the ligature.

If a split caret moves across a direction boundary, this function describes the movement of the high (dominant) caret only.

SPECIAL CONSIDERATIONS

This function may not always work correctly for right-to-left carets (type `gxRightLeftKeyboardCaret`) at direction boundaries in mixed-direction text.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
```

SEE ALSO

For an example of how to use this function, see Listing 10-3 on page 10-23.

Caret types are described on page 10-41.

The `gxNoLigatureSplits` flag is described with the run controls structure in the chapter “Layout Styles” in this book.

The complementary function `GXGetLeftVisualOffset` is described next.

GXGetLeftVisualOffset

You can use the `GXGetLeftVisualOffset` function to determine the edge offset corresponding to the next caret position to the left (or upward, for vertical text) in a layout shape.

```
gxByteOffset GXGetLeftVisualOffset(gxShape layout,
                                   gxByteOffset currentOffset);
```

`layout` A reference to the layout shape whose left visual offset you need.

`currentOffset`
The edge offset in the source text corresponding to the current caret position in the display text.

function result The edge offset in the source text corresponding to the next leftward (or upward) caret position in the display text.

DESCRIPTION

The `GXGetLeftVisualOffset` function determines the next caret position to the left (or upward) in the display text of a layout shape. This is where the caret would move to if the user pressed the Left or Up Arrow key. The function takes into account the text direction (left to right or right to left). It also considers whether the text has undergone linguistic rearrangement. If the caret passes through a ligature, the resulting offset depends on whether or not the `gxNoLigatureSplits` flag is set. If so, the caret position passes entirely across the ligature; if not, the caret position can be interior to the ligature.

If a split caret moves across a direction boundary, this function describes the movement of the high (dominant) caret only.

SPECIAL CONSIDERATIONS

This function may not always work correctly for right-to-left carets (type `gxRightLeftKeyboardCaret`) at direction boundaries in mixed-direction text.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
```

SEE ALSO

For an example of how to use this function, see Listing 10-3 on page 10-23.

Caret types are described on page 10-41.

The `gxNoLigatureSplits` flag is described with the run controls structure in the chapter “Layout Styles” in this book.

The complementary function `GXGetRightVisualOffset` is described in the previous section.

Highlighting in a Layout Shape

The functions described in this section allow you to highlight text and, for mixed-direction text, allow you to choose whether or not to make your highlights visually contiguous.

GXGetLayoutHighlight

You can use the `GXGetLayoutHighlight` function to obtain a shape to use to draw the highlighting corresponding to a selection range in a specified layout shape. If the selection range includes text of different directions, the highlighting shape may be discontiguous.

```
gxShape GXGetLayoutHighlight(gxShape layout,
                             gxByteOffset startOffset,
                             gxByteOffset endOffset,
                             gxHighlightType highlightType,
                             gxShape highlight);
```

`layout` A reference to the layout shape you want to highlight.

`startOffset`
The edge offset in the source text that marks the start of the selection range. If this value is the same as `endOffset`, `GXGetLayoutHighlight` generates a caret.

`endOffset` The edge offset marking the end of the selection range. If this value is the same as `startOffset`, `GXGetLayoutHighlight` generates a caret. You can specify `gxSelectToEnd` for this parameter to select to the end of the text in the layout shape.

`highlightType`
The type of highlight to use (perpendicular or oblique), of type `gxHighlightType`.

`highlight` A reference to a shape object. You may supply an existing highlight shape here for `GXGetLayoutHighlight` to reuse; if you pass `nil` for this parameter, `GXGetLayoutHighlight` allocates a new shape to return in its function result.

function result The shape describing the highlight for the selection range specified by the `startOffset` and `endOffset` parameters. If you pass an existing shape in the `highlight` parameter, `GXGetLayoutHighlight` modifies the shape as necessary and returns it; otherwise, `GXGetLayoutHighlight` returns a new shape.

DESCRIPTION

The `GXGetLayoutHighlight` function calculates a shape corresponding to the area covered by the glyphs corresponding to all the characters between the two edge offsets `startOffset` and `endOffset`. For single-direction text this is usually a simple rectangular or trapezoidal area, but for mixed-direction text the area that is returned may be quite complex.

If you pass `nil` for the `highlight` parameter, `GXGetLayoutHighlight` creates a new highlight shape and returns it as the function result. You can also pass an existing highlight shape to save the overhead of disposing of one shape and creating another.

If either of the offsets corresponds to a point interior to a ligature, the appearance of the highlight depends on the state of the `gxNoLigatureSplits` flag in the run controls structure of the style run containing the offset. If the flag is set, the highlight extends across the entire ligature; if it is clear, only the portion of the ligature corresponding to the included characters is highlighted.

The `highlightType` parameter controls the angle of the with-stream edges of the highlighting shape (the left and right edges for horizontal text). If the highlight type is `gxHighlightStraight`, the highlighting shape has edges that are perpendicular to the baseline, and the highlighting is always tiled (contiguous and nonambiguous across boundaries of text with different slant). If the highlight type is `gxHighlightAverageAngle`, the angle of the edge of the highlighting area is the average of the slants of the two glyphs on either side of the edge. In this case also, highlighting is usually tiled. Highlighting is not tiled only in cases of extreme slant coupled with superscripts or subscripts. (In such a case a triangular highlighting area may result.)

If the layout shape has a transform, the highlight shape this function returns has the same transform.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
```

SEE ALSO

For an example of how to use this function, see page 10-26 and page 10-27.

For a description of how highlighting applies to a layout shape, see “Drawing Highlighting” on page 10-25. Highlight types are described on page 10-41.

The `gxNoLigatureSplits` flag is described with the run controls structure in the chapter “Layout Styles” in this book.

GXGetLayoutVisualHighlight

You can use the `GXGetLayoutVisualHighlight` function to obtain a shape that describes a single, contiguous highlighting band corresponding to two edge offsets in a specified layout shape. The shape is a solid connection between the beginning and end glyphs—even across direction boundaries in the text or if it results in a visual highlight that covers discontiguous text in the backing store.

```
gxShape GXGetLayoutVisualHighlight(gxShape layout,
                                   gxByteOffset startOffset,
                                   long startLeadingEdge,
                                   gxByteOffset endOffset,
                                   long endLeadingEdge,
                                   gxHighlightType highlightType,
                                   gxShape highlight);
```

`layout` A reference to the layout shape you want to highlight.

`startOffset`
The edge offset in the source text that marks the start of the selection range. If this value is the same as `endOffset`, `GXGetLayoutVisualHighlight` generates a caret.

`startLeadingEdge`
If this value is `true`, the highlight starts at the leading edge of the glyph whose leading edge bounds the caret position corresponding to the `startOffset` parameter. Otherwise, the highlight starts at the trailing edge of the glyph whose trailing edge bounds the caret position corresponding to the `startOffset` parameter.

`endOffset` The edge offset in the source text that marks the end of the selection range. If this value is the same as `startOffset`, `GXGetLayoutVisualHighlight` generates a caret. You can specify `gxSelectToEnd` for this parameter to select to the end of the text in the layout shape.

`endLeadingEdge`
If this value is `true`, the highlight ends at the leading edge of the glyph whose leading edge bounds the caret position corresponding to the `endOffset` parameter. Otherwise, the highlight ends at the trailing edge of the glyph whose trailing edge bounds the caret position corresponding to the `endOffset` parameter.

`highlightType`
The type of highlight to use (perpendicular or oblique), of type `gxHighlightType`.

`highlight` A reference to a shape object. You may supply an existing highlight shape here for `GXGetLayoutVisualHighlight` to reuse; if you pass `nil` for this parameter, `GXGetLayoutVisualHighlight` allocates a new shape to return in its function result.

function result The shape describing the highlight for the selection range specified by the `startOffset` and `endOffset` parameters, and given the `startLeadingEdge` and `endLeadingEdge` constraints. If you pass an existing shape in the `highlight` parameter, `GXGetLayoutVisualHighlight` modifies the shape as necessary and returns it; otherwise, `GXGetLayoutVisualHighlight` returns a new shape.

DESCRIPTION

The `GXGetLayoutVisualHighlight` function calculates a shape that is a single, contiguous highlighting rectangle (or trapezoid) between the glyphs corresponding to the edge offsets `startOffset` and `endOffset` and the leading-edge states specified in `startLeadingEdge` and `endLeadingEdge`.

The shape returned by `GXGetLayoutVisualHighlight` function is similar to that returned by `GXGetLayoutHighlight`, except that the result of `GXGetLayoutVisualHighlight` always represents the continuous visual range between the two specified offsets and edges; it never consists of discontiguous visual segments, even in mixed-direction text. (For single-direction text, calling `GXGetLayoutVisualHighlight` yields the same results as calling `GXGetLayoutHighlight` for the same start and end offsets.)

This function is most useful for mixed-direction text, where only a contiguous highlight is desired. For dynamic highlighting as the user moves the cursor through the display text, contiguous highlighting may be less confusing.

If you pass `nil` for the `highlight` parameter, `GXGetLayoutVisualHighlight` creates a new highlight shape and returns it as the function result. You can also pass an existing highlight shape to save QuickDraw GX the overhead of disposing of one shape and creating another.

If either of the offsets corresponds to a point interior to a ligature, the appearance of the highlight depends on the state of the `gxNoLigatureSplits` flag in the run controls structure of the style run containing the offset. If the flag is set, the highlight extends across the entire ligature; if it is clear, only the portion of the ligature corresponding to the included characters is highlighted.

The `highlightType` parameter controls the angle of the with-stream edges of the highlighting shape (the left and right edges for horizontal text). If the highlight type is `gxHighlightStraight`, the highlighting shape has edges that are perpendicular to the baseline, and the highlighting is always tiled (contiguous and nonambiguous across boundaries of text with different slant). If the highlight type is `gxHighlightAverageAngle`, the angle of the edge of the highlighting area is the average of the slants of the two glyphs on either side of the edge. In this case also, highlighting is usually tiled; highlighting is not tiled only in cases of extreme slant coupled with superscripts or subscripts. (In such a case a triangular highlighting area may result.)

If the layout shape has a transform, the highlight shape this function returns has the same transform.

SPECIAL CONSIDERATIONS

The highlight shape returned by this function, and therefore the selection range it implies, do *not* in general correspond to the continuous set of characters between the two edge offsets used as input to the function. Where contiguous highlighting crosses direction boundaries, the resulting selection range is discontiguous and can be quite complex, possibly involving characters beyond the range of edge offsets used as input. For this reason, use of this function is not recommended.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
```

SEE ALSO

For an example of how to use this function, see page 10-28.

For a description of how highlighting applies to a layout shape, see "Drawing Highlighting" on page 10-25. Highlight types are described on page 10-41.

The `gxNoLigatureSplits` flag is described with the run controls structure in the chapter "Layout Styles" in this book.

Hit-Testing in a Layout Shape

The function described in this section allows you to hit-test the text of any layout shape, no matter how complex. For hit-testing of shapes other than layout shapes, you can use the `GXHitTestShape` function, described in the shape objects chapter of *Inside Macintosh: QuickDraw GX Objects.*

GXHitTestLayout

You can use the `GXHitTestLayout` function to convert a view port location (representing, for example, the position of a mouse-down event) into an edge offset in the source text of a layout shape.

```
gxByteOffset GXHitTestLayout(gxShape layout,
                             const gxPoint *hitDown,
                             gxHighlightType highlightType,
                             gxLayoutHitInfo *hitInfo,
                             gxShape hitTrackingArea);
```

`layout` A reference to the layout shape to hit-test.

`hitDown` A pointer to a point structure that contains the location to test. The location is in local (view port) coordinates.

`highlightType`
: The kind of highlight used (perpendicular or oblique), of type `highlightType`.

`hitInfo`
: A pointer to a layout hit info structure. On return, the structure contains the results of the hit-test. If you pass `nil` for this parameter, the hit info structure is not filled out but the function result is still valid.

`hitTrackingArea`
: A reference to a shape object. On return, the shape specifies the mouse tracking area that corresponds to the specified hit point. You may supply an existing shape here for `GXHitTestLayout` to reuse. Pass `nil` for this parameter if you do not want `GXHitTestLayout` to return the hit-tracking area.

function result
: The edge offset corresponding to the location where the hit occurred. This value always equals the value of the `hitSideOffset` field of `gxLayoutHitInfo` structure returned in the `hitInfo` parameter.

DESCRIPTION

The `GXHitTestLayout` function determines which part of which glyph in the display text of a layout shape is closest to a particular location. It then returns in the `hitInfo` parameter the equivalent source-text edge offset as the function result, and leading-edge information.

You must specify the test location in the local coordinates of the view port in which the text is displayed. For hit-testing of mouse-down events, you must convert mouse coordinates (such as might be returned in an event record) to view port coordinates (with a function such as `GXQDGlobalToGXLocal` or `GXGetViewPortMouse`) before you call `GXHitTestLayout`.

The `GXHitTestLayout` function also returns a mouse tracking area, which specifies the area in the display text within which the resulting edge offset is valid. For subsequent hits anywhere within that area, there may be no need to call `GXHitTestLayout` again, because the resulting edge offset will be the same. The `hitTrackingArea` parameter is modified if it is not `nil`; if it is `nil`, it is not generated.

The hit info structure (returned in the `hitInfo` parameter) provides more than just leading-edge information. It also returns the exact position of the hit point within the hit glyph, two edge offsets corresponding to the two edges of the hit glyph, and an indication of whether the hit point is actually outside the boundaries of the layout shape (such as above or below, for horizontal text).

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_is_nil
parameter_out_of_range
```

SEE ALSO

For an example of how to use this function, see Listing 10-5 on page 10-31.

Coordinate systems and view ports are described in the chapter "View-Related Objects" in *Inside Macintosh: QuickDraw GX Objects*. Functions such as `GXQDGlobalToGXLocal` and `GXGetViewPortMouse`, which convert from Macintosh coordinates to QuickDraw GX coordinates, are described in the chapter "The QuickDraw GX and the Macintosh Environment" in *Inside Macintosh: QuickDraw GX Environment and Utilities*.

The `GXHitTestShape` function, used for hit-testing shapes other than layout shapes, is described in the chapter "Shape Objects" in *Inside Macintosh: QuickDraw GX Objects*. Typographic information returned by `GXHitTestShape` is described in the chapter "Typographic Shapes" in this book.

Highlighting types are described on page 10-41. The layout hit info structure is described on page 10-43.

Converting Between Glyphs and Characters in a Layout Shape

Functions described in this section allow you to convert glyphs to character codes and back. The `GXGetOffsetGlyphs` and `GXGetGlyphOffset` functions map back and forth between edge offsets in the source text and glyph indices in the display text of a layout shape. The `GXGetCompoundCharacterLimits` function maps the edges of a ligature glyph to offsets in the source text.

GXGetOffsetGlyphs

You can use the `GXGetOffsetGlyphs` function to determine which glyphs border the caret positions corresponding to a particular edge offset in the source text.

```
void GXGetOffsetGlyphs(gxShape layout, gxByteOffset trial,
                       long leadingEdge,
                       gxLayoutOffsetState *offsetState,
                       unsigned short *firstGlyph,
                       unsigned short *secondGlyph);
```

`layout` A reference to the layout shape whose glyphs you want to identify.

`trial` An edge offset in the source text. The glyph or glyphs bounding the caret positions corresponding to that offset are the glyphs to be returned.

`leadingEdge`
Specify `true` if you want the function to return in the `firstGlyph` parameter the glyph corresponding to the character code *following* the edge offset specified in the `trial` parameter; specify `false` if you want the glyph corresponding to the character code *preceding* the edge offset.

`offsetState`
: A pointer to a layout offset state value. On return, it specifies the characteristics (such as the sizes of the bounding character codes and whether the offset corresponds to the interior of a glyph) of the edge offset specified in the `trial` parameter.

`firstGlyph`
: A pointer to a `short` value. On return, it contains the glyph index of the glyph corresponding to the specified values in the `trial` and `leadingEdge` parameters.

`secondGlyph`
: A pointer to a `short` value. On return, it contains the glyph index of the other glyph corresponding to the offset specified in the `trial` parameter. In other words, it contains the glyph index that would have been returned in the `firstGlyph` parameter if the `leadingEdge` parameter had had the opposite value.

DESCRIPTION

The `GXGetOffsetGlyphs` function determines which glyph in the display text of a layout shape corresponds to the specified edge offset and leading-edge state in the source text. The function returns its result as a glyph index in the `firstGlyph` parameter. (A glyph index is the 1-based position of a glyph in the left-to-right, or top-to-bottom, display order of a layout shape's display text.)

The function also returns, in the `secondGlyph` parameter, the glyph index of the other glyph that bounds the caret position corresponding to the specified edge offset. Typically, that is the bounding glyph whose leading-edge state is opposite to that of the glyph returned in the `firstGlyph` parameter.

The `GXGetOffsetGlyphs` function also returns information on the nature of the boundary at the specified edge offset, in the form of a layout offset state value. It tells you the sizes of the bounding character codes, and whether the offset is interior to a ligature or whether it is invalid (interior to a single character code).

In the case of an edge offset that corresponds to the interior of a compound glyph such as a ligature, both `firstGlyph` and `secondGlyph` contain the same value: the index of the glyph containing the offset. In the case of an invalid edge offset (between the two bytes of a 16-bit character code), `firstGlyph` and `secondGlyph` both contain the index of the glyph corresponding to that character.

You can call this function to perform a geometric operation on the glyph corresponding to a known edge offset in the source text. You can get a copy of the glyph array that makes up the display text of a layout shape by calling the `GXGetLayoutGlyphs` function.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`parameter_out_of_range`

SEE ALSO

For an example of how to use this function, see Listing 10-6 on page 10-34.

Layout offset state values are described on page 10-42.

The complementary function `GXGetGlyphOffset` is described next.

Glyph indexes are described in the section "Positioning in Source Text and Display Text" beginning on page 10-3. The `GXGetLayoutGlyphs` function is described in the chapter "Layout Shapes" in this book.

GXGetGlyphOffset

You can use the `GXGetGlyphOffset` function to find the edge offset in the source text corresponding to a particular edge of a particular glyph in a layout shape.

```
void GXGetGlyphOffset(gxShape layout, long trial,
                      long onLeftTop, gxByteOffset *offset,
                      boolean *leadingEdge,
                      boolean *wasRealCharacter);
```

`layout` A reference to the layout shape containing the glyph whose corresponding edge offset you need.

`trial` A (1-based) glyph index specifying the position of the glyph in the display text.

`onLeftTop` A Boolean value indicating whether the trial parameter specifies the edge offset corresponding to the left (or top, for vertical text) edge of the glyph (`true`) or the edge offset corresponding to the right (or bottom, for vertical text) edge of the glyph (`false`).

`offset` A pointer to a caret-offset value. On return, it contains the edge offset in the source text corresponding to the glyph and edge specified in the `trial` and `onLeftTop` parameters.

`leadingEdge`
A pointer to a Boolean value. On return, the value is `true` if the specified edge of the specified glyph is the leading edge; otherwise the value is `false`.

`wasRealCharacter`
A pointer to a Boolean value. On return, the value is `true` if the specified glyph corresponds to one or more character codes in the source text; otherwise the value is `false`.

DESCRIPTION

The `GXGetGlyphOffset` function determines which edge offset in the source text of a layout shape corresponds to the specified edge of the specified glyph in the display text. You specify the input glyph by index. (A glyph index is the 1-based position of a glyph in the left-to-right, or top-to-bottom, display order of a layout shape's display text.) The function returns its result in the `offset` parameter.

The function also returns, in the `leadingEdge` parameter, information indicating whether the specified edge of the specified glyph is its leading edge. It also returns, in the `wasRealCharacter` parameter, whether or not the specified glyph corresponds to an actual character code in the source text. For example, no character codes correspond to kashida glyphs. (If the value of `wasRealCharacter` is `false`, `GXGetGlyphOffset` returns an offset corresponding to the right side of the glyph to the left of the kashida glyph.)

You can call this function to perform an operation on the character code corresponding to a known glyph in the display text. You can get a copy of the glyph array that makes up the display text of a layout shape by calling the `GXGetLayoutGlyphs` function.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
```

SEE ALSO

For an example of how to use this function, see Listing 10-7 on page 10-38.

The complementary function `GXGetOffsetGlyphs` is described in the previous section.

Glyph indexes are described in the section “Positioning in Source Text and Display Text” beginning on page 10-3. The `GXGetLayoutGlyphs` function is described in the chapter “Layout Shapes” in this book.

GXGetCompoundCharacterLimits

You can use the `GXGetCompoundCharacterLimits` function to find the edge offsets that correspond to the leading and trailing edges of a particular glyph (itself specified by edge offset).

```
void GXGetCompoundCharacterLimits(gxShape layout,
                                  gxByteOffset trial,
                                  gxByteOffset *minOffset,
                                  gxByteOffset *maxOffset,
                                  boolean *onBoundary);
```

`layout` A reference to the layout shape containing the glyph you want to analyze.

`trial` An edge offset in the source text. The glyph corresponding to that offset is the glyph to be analyzed. If the offset is between glyphs, both glyphs bounding that offset are taken into account: the glyph to be analyzed for `minOffset` is the glyph corresponding to the character preceding the `trial` offset, and the glyph to be analyzed for `maxOffset` is the glyph

corresponding to the character following the `trial` offset. If the offset is at the beginning (or end) of the source text, the glyph corresponding to the first (or last) character in the text is the glyph to be analyzed.

`minOffset` A pointer to an edge-offset value. On return, it specifies the edge offset in the source text corresponding to the leading edge of the glyph corresponding to the character specified by the `trial` parameter.

`maxOffset` A pointer to a caret-offset value. On return, it specifies the edge offset in the source text corresponding to the trailing edge of the glyph corresponding to the character specified by the `trial` parameter.

`onBoundary` A pointer to a Boolean value. On return, it is `true` if the edge offset specified in the `trial` parameter corresponds to a caret position between glyphs in the display text. It is `false` if the offset corresponds to a caret position in the interior of a compound glyph (such as a ligature).

DESCRIPTION

Given an edge offset in the source text of a layout shape, the `GXGetCompoundCharacterLimits` function first determines which glyph in the display text corresponds to that offset, and then returns the edge offsets corresponding to both edges of the glyph. For a compound character such as a ligature, `GXGetCompoundCharacterLimits` thus lets you determine which characters make up the ligature.

The function returns the offsets in two parameters. The value of `minOffset` is the edge offset that marks the leading edge of the glyph that includes (or precedes, if the value of `onBoundary` is `true`) the `trial` offset. The value of `maxOffset` is the edge offset that marks the trailing edge of the glyph that includes (or follows, if the value of `onBoundary` is `true`) the `trial` offset.

If the `trial` offset precedes the first character or follows the last character in the source text, `GXGetCompoundCharacterLimits` returns a value of `true` for `onBoundary`, and returns edge offsets bounding the glyph corresponding to the first or last character, respectively.

You can use `GXGetCompoundCharacterLimits` to determine whether a particular edge offset corresponds to the interior of a ligature and, if it does, what the bounding offsets of the characters making up that ligature are.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
```

SEE ALSO

For information about related functions, see the descriptions of `GXGetOffsetGlyphs` on page 10-56and `GXGetGlyphOffset` on page 10-58.

Summary of Layout Carets, Highlighting, and Hit-Testing

Constants and Data Types

Highlighting Type

```
enum {
   gxHighlightStraight     = 0,
   gxHighlightAverageAngle = 1
};
typedef unsigned long gxHighlightType;
```

Caret Type

```
enum {
   gxSplitCaretType        = 0,
   gxLeftRightKeyboardCaret = 1,
   gxRightLeftKeyboardCaret = 2
  };
typedef unsigned long gxCaretType;
```

Layout Offset State

```
enum {
   gxOffset8_8             = 0,
   gxOffset8_16            = 1,
   gxOffset16_8            = 2,
   gxOffset16_16           = 3,
   gxOffsetInvalid         = 4,
   gxOffsetInsideLigature  = 0x8000
   };
typedef unsigned short gxLayoutOffsetState;
```

Layout Hit Inf Structure

```
typedef struct {
   Fixed      firstPartialDist;
   Fixed      lastPartialDist;
   gxByteOffset hitSideOffset;
   gxByteOffset nonHitSideOffset;
   boolean    leadingEdge;
   boolean    inLoose;
} gxLayoutHitInfo;
```

Functions

Manipulating Carets in a Layout Shape

```
gxShape GXGetLayoutCaret     (gxShape layout, gxByteOffset offset,
                              gxHighlightType highlightType,
                              gxCaretType caretType, gxShape caret) ;
gxShape GXGetCaretAngleArea (gxShape layout, const gxPoint *hitPoint,
                              gxHighlightType highlightType,
                              gxShape caretArea, short *returnedRise,
                              short *returnedRun);
gxByteOffset GXGetRightVisualOffset
                             (gxShape layout, gxByteOffset currentOffset);
gxByteOffset GXGetLeftVisualOffset
                             (gxShape layout, gxByteOffset currentOffset);
```

Highlighting in a Layout Shape

```
gxShape GXGetLayoutHighlight(gxShape layout, gxByteOffset startOffset,
                              gxByteOffset endOffset,
                              gxHighlightType highlightType,
                              gxShape highlight);
gxShape GXGetLayoutVisualHighlight
                             (gxShape layout, gxByteOffset startOffset,
                              long startLeadingEdge, gxByteOffset endOffset,
                              long endLeadingEdge,
                              gxHighlightType highlightType,
                              gxShape highlight);
```

Hit-Testing in a Layout Shape

```
gxByteOffset GXHitTestLayout(gxShape layout, const gxPoint *hitDown,
                              gxHighlightType highlightType,
                              gxLayoutHitInfo *hitInfo,
                              gxShape hitTrackingArea);
```

Converting Between Glyphs and Characters in a Layout Shape

```
void GXGetOffsetGlyphs       (gxShape layout, gxByteOffset trial,
                              long leadingEdge,
                              gxLayoutOffsetState *offsetState,
                              unsigned short *firstGlyph,
                              unsigned short *secondGlyph);
void GXGetGlyphOffset        (gxShape layout, long trial,
                              long onLeftTop, gxByteOffset *offset,
                              boolean *leadingEdge,
                              boolean *wasRealCharacter);
void GXGetCompoundCharacterLimits
                             (gxShape layout, gxByteOffset trial,
                              gxByteOffset *minOffset,
                              gxByteOffset *maxOffset, boolean *onBoundary);
```

Glossary

absolute position A specific position, given in coordinates, for the origin of each character or glyph in the glyph shape. Compare **relative position.**

advance bits array An array that determines whether the points in the positions array are absolute or relative. The advance bits array contains 1 bit for every character or glyph in the shape**.**

advance height The distance from the top of a glyph to the bottom of the glyph, including the top-side bearing and bottom-side bearing.

advance width The full horizontal width of a glyph as measured from its origin to the origin of the next glyph on the line, including the side bearings on both sides.

alignment The process of placing text in relation to one or both margins.

alphabetic writing system The glyphs that symbolize discrete phonemic elements in a language. Compare **syllabic writing system** and **ideographic writing system.**

angled caret A caret whose angle in relation to the baseline of the display text is equivalent to the slant of the glyphs making up the text. Compare **straight caret.**

ascent line An imaginary horizontal line that corresponds approximately to the tops of the uppercase letters in the font. Uppercase letters are chosen because, among the regularly used glyphs in a font, these are generally the tallest.

automatic form substitution The process of automatically substituting one or more glyphs for one or more other glyphs.

baseline An imaginary line used to align glyphs in a line of text.

baseline delta An array of distances (in points) between the various baseline types and y = 0. See **baseline type.**

baseline type The classification of baseline used with a particular kind of text. See, for example, **Roman baseline.**

bottom-side bearing The white space between the bottom of the glyph and the visible ending of the glyph.

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

byte offset The numbering of character codes in source text. Compare **edge offset.**

caret A vertical or slanted blinking bar, appearing at a caret position in the display text, that marks the point at which text is to be inserted or deleted. Compare **split caret.**

caret angle The angle of a caret or the edges of a highlight. The caret angle can be perpendicular to the baseline or parallel to the angle of the style run's text.

caret position A location on screen, typically between glyphs, that relates directly to a caret offset in the source text.

caret type A designation of the behavior of the caret at direction boundaries in text. See **split caret, left-to-right caret, right-to-left caret.**

character A symbol standing for a sound, syllable, or notion used in writing; one of the simple elements of a written language, for example, the lowercase letter "a" or the number "1". Compare **character code, glyph.**

character code A numerical representation of a character. Each writing system or language has one or more character encodings, tables that relate character codes to the characters they represent. Most character codes have either a 1-byte or 2-byte storage size.

character encoding An internal conversion table for interpreting a specific character set.

contextual form An alternate form of a glyph whose use depends on the glyph's placement in a word.

contiguous highlighting Highlighting that consists of a single, contiguous shape across direction boundaries, even when it does not exactly match the selection range it corresponds to. Compare **discontiguous highlighting.**

counter The oval in glyphs such as "p" or "d".

cross-stream kerning The automatic movement of glyphs perpendicular to the line orientation of the text. Compare **with-stream kerning.**

cross-stream shift A type of positional shift that applies equally to all glyphs in a style run by raising or lowering the entire style run (or shifts it sideways if it's vertical text). Compare **with-stream shift.**

cursor A small icon, often an arrow or an I-beam shape, that moves with the mouse or other pointing device. Compare **caret.**

descent line An imaginary horizontal line that usually corresponds with the bottoms of the descenders in a font. The descent line is the same distance from the baseline for all glyphs in the font, whether or not they have descenders.

direction See **dominant direction, glyph direction, line direction, text direction.**

direction boundary A point between offsets in memory or glyphs in a display, at which the direction of stored or displayed text changes.

direction level A hierarchical ranking of dominant direction in a line. Direction levels can be nested so that complex mixed-direction formatting is preserved.

direction-level run A sequence of contiguous glyphs that share the same text direction.

direction override A means of overriding the directional behavior of glyphs, on a style-run basis, for special effects.

discontiguous highlighting Highlighting that exactly matches the selection range it corresponds to. It may consist of discontiguous areas when the selection range crosses direction boundaries. Compare **contiguous highlighting.**

display order The left-to-right order in which QuickDraw GX displays glyphs. Display order determines the glyph index of each glyph in a line and may differ from the input order of the text. See **glyph index**; compare **input order** and **source text.**

display text The visual representation of the text of a typographic shape. Display text consists of a sequence of glyphs, arranged in display order. Compare **source text.**

dominant direction The direction in which successive groups of glyphs are read. Dominant direction is independent of glyph direction. See also **glyph direction, line direction.**

drop capital A large uppercase letter that drops below the main line of text for aesthetic reasons.

dual caret See **split caret.**

dynamic highlighting The process of continually drawing and redrawing the highlighted area as the user moves the cursor through the text while holding down the mouse button.

edge offset A byte offset into the source text of a layout shape that specifies a position *between* byte values. Edge offsets in source text are related to caret positions in display text. Compare **caret position** and **byte offset.**

face layers A structure that describes part of a text face. Several face layers are combined to form the visual composite of a glyph.

feature selectors A means of defining particular font features in a feature type. See also **feature type.**

feature type A group of font features in a style object that are applied to each style run based on font defaults. See also **feature selectors.**

flat font list A list that QuickDraw GX creates when you flatten a shape that contains fonts. This list specifies which fonts were used in a shape, which glyphs were used in a font, or both.

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, stroke thickness, or the use of serifs.

font attributes A group of flags that modify the behavior or identity of a font.

font descriptors The identifiable characteristics of each font object within a family, such as weight, width, italic slant, and optical point size.

font embedding The technique of storing a font object's binary data in a document so that the text in the document always displays the correct font.

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

font features The set of typographic and layout capabilities that create a specific appearance for a layout shape.

font instance A setting identified by the font's designer that matches specific values along the available variation axes and gives those values a name.

font name 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 An object type that hides the complexity of font data from your application.

font variation An algorithmic way to produce a range of typestyles along a particular variation axis.

font variation suite A complete listing of every axis supported in a font in the order specified by the font. Each axis is given a value in the listing.

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.**

glyph code A number that specifies a particular glyph in a font. Fonts map character codes to glyph codes, which in turn specify individual glyphs.

glyph direction The direction in which successive glyphs are read. Compare **dominant direction.**

glyph ductility The ability to stretch the actual form of a glyph during justification.

glyph index The order of a glyph in a line of display text. The leftmost glyph in a line of text has a glyph index of 1; each succeeding glyph to the right has an index one greater than the previous glyph. Compare **glyph code, edge offset.**

glyph justification overrides array An array that alters the standard justification behavior of one or more individual glyphs.

glyph origin The point that QuickDraw GX uses to position a glyph when drawing.

glyph shape A typographic shape that allows you to vary the position, font, rotation, and scale of each glyph in a line of text. Compare **layout shape** and **text shape**. See **typographic shapes.**

glyph substitutions array An array of glyph codes in a style object that defines which glyph to substitute for another in a specific style run.

grow limit The maximum amount by which glyphs of a given priority can be extended during justification, before processing passes to glyphs of lower priority. Compare **shrink limit.**

hanging baseline The baseline used by Devanagari and similar scripts, where most of the glyph is below the baseline.

hanging glyphs A set of glyphs, usually punctuation, that typically extend beyond the left and right margins of the text area and whose widths are not counted when line length is measured.

highlighting The display of text in inverse video or with a colored background. Highlighting in display text corresponds to a selection range in source text.

highlight type The angular character of carets and edges of highlighting areas. Highlighting and carets are either straight or angled; see **angled caret, straight caret.**

hit-testing The process of converting a location within a line of display text into a caret offset in the source text of that line.

hyphenation point An entry in an array of edge offsets in the source text at which it is appropriate to break a line of display text.

ideographic centered baseline The baseline used by Chinese, Japanese, and Korean ideographic scripts, in which glyphs are centered halfway on the line height.

ideographic writing system The glyphs that symbolize component meanings of words in a language. Compare **syllabic writing system** and **alphabetic writing system.**

imposed width A run control feature that forces a specific width onto the glyphs of a style run, regardless of its text content or other style properties.

index See **glyph index.**

input order The order in which characters are written or entered from a keyboard. The input order of a line of text can differ from its display order. Compare **display order.**

insertion point The point in the source text at which text is to be inserted or deleted. An insertion point is specified by a single caret position. Compare **caret**; see also **caret position.**

justification The process of typographically expanding or compressing a line of text to fit a text width.

justification gap The difference in the length of a line before and after justification.

justification priority The priority order in which classes of glyphs are processed during justification.

kashida An extension-bar glyph that is added to certain Arabic glyphs during justification.

kerning An adjustment to the normal spacing that occurs between two or more specifically named glyphs, known as the *kerning pair.*

kerning adjustments array An array in the style object that overrides the normal kerning for individual pairs of glyphs by specifying a point-size factor and scaling factor.

kerning pair Two specifically named glyphs that are kerned together by a set amount. See also **kerning.**

language The written and spoken methods of combining words to create meaning used by a particular group of people.

layer flag An element of a face layer that describes the characteristics of one layer of a text face. Layer flags are used primarily to determine the underlining capabilities of the text face.

layout shape A typographic shape that allows you to vary a layout shape in typographic aspects. Compare **glyph shape** and **text shape**. See **typographic shapes.**

leading edge The edge of a glyph that is encountered first when reading text of that glyph's language. For glyphs of left-to-right text, the leading edge is the left edge; for glyphs of right-to-left text, the leading edge is the right edge.

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

left-to-right caret A type of caret that, at direction boundaries, appears at the proper caret position for inserting left-to-right text. Compare **right-to-left caret, split caret.**

ligature Two or more glyphs connected to form a single new glyph.

ligature decomposition The replacement of ligatures with the glyphs for their component characters during justification.

ligature splitting The process of separating a ligature into its component glyphs.

line breaking The process of determining the proper location at which to truncate a line of text so that it fits within a given text width.

line direction The overall direction in which a line of text is read. Line direction is the lowest nested level of dominant direction on a line.

line length The distance, in points, from the origin of the first glyph on a line through the advance width of the last glyph.

line span The distance, in points, from the lowest descender on a line to the highest ascender.

margins The left, right, top, and bottom sides of the text area.

math baseline The baseline used for setting mathematical expressions; it is centered on operators such as the minus sign.

mixed-direction text The combination of text with both left-to-right and right-to-left directions within a single line of text.

neutral type A glyph directionality in which the glyph direction is always that of the surrounding glyphs. Compare **strong type, weak type.**

point size The size of a font's glyphs as measured from the baseline of one line of text to the baseline of the next line of single-spaced text. In the United States, point size is measured in typographic points.

point size factor A specific point size that you force onto a style run to create custom kerning. See also **scaling factor** and **kerning adjustments array.**

positions array An array that contains positions for the origin of each character or glyph in the shape. These positions, stored as points, can be relative to the advance width of the previous character or glyph, or they can be absolute positions in coordinates.

postcompensation action The extra processing, such as addition of kashidas and ligature decomposition, that occurs after glyphs have been repositioned during justification.

priority justification override array An array that alters the standard justification behavior for all glyphs of a given justification priority.

QuickDraw GX Font Feature Registry An official document maintained by Apple Computer, Inc., in which feature types and feature selectors are defined and named.

relative position A position for the origin of each character or glyph in the glyph shape given in coordinates relative to the preceding character or glyph. Compare **absolute position.**

right-side bearing The white space on the right side of the glyph; this value may or may not be equal to the value of the left-side bearing.

right-to-left caret A type of caret that, at direction boundaries, appears at the proper caret position for inserting right-to-left text. Compare **left-to-right caret, split caret.**

Roman baseline The baseline used in most Roman scripts and in Arabic and Hebrew.

run A sequence of glyphs that are contiguous in memory and share a set of common attributes.

run controls structure An array that is a property of every style object but is used only by layout shapes. This structure controls various features associated with text in a style run.

run features See **font features.**

scaling factor A specific scale that you force onto a style run to create custom kerning. See also **point size factor** and **kerning adjustments array.**

script A method for depicting words visually.

selection range The contiguous sequence of characters in the source text that mark where the next editing operation is to occur. The glyphs corresponding to those characters are commonly highlighted on screen.

serif The fine lines stemming from and at an angle to the upper and lower ends of the main strokes of a letter—for example, the little "feet" on the bottom of the vertical strokes in the upper-case letter "M" in Times Roman typeface.

shape attributes A group of flags that modify the behavior of a shape object.

shrink limit The maximum amount by which glyphs of a given priority may be compressed during justification, before processing passes to glyphs of lower priority. Compare **grow limit.**

smart swash A variation of an existing glyph (often ornamental) that is contextual. Compare **swash**.

source text A stored sequence of character codes that represents a line of text. Characters in source text are stored in input order. Compare **display order, display text;** see also **input order.**

split caret A type of caret that, at the boundary between text of opposite directions, divides into two parts: a high caret and a low caret, each measuring half the line's height. The two separate half-carets merge into one in unidirectional text. Compare **left-to-right caret, right-to-left caret.**

stake An edge offset in the source text that marks the point at which a line break would be most efficient in terms of layout processing.

storage order See **input order**, **display order, source text.**

storage reference A specification of the storage type used to store a font. See **storage type.**

storage type The method used to store a font in a font object. See **storage reference.**

straight caret A caret that is perpendicular to the baseline of the display text, regardless of the angle of the glyphs making up the text. Compare **angled caret.**

strong type A glyph directionality that is always left to right or right to left. Compare **weak type, neutral type.**

style run A sequence of memory backing store contiguous glyphs that share the same style.

swash A variation of an existing glyph (often ornamental) that is noncontextual. Compare **smart swash**.

syllabic writing system The glyphs that symbolize syllables in a language. Compare **alphabetic writing system** and **ideographic writing system.**

tangents array An array that determines the scaling and orientation of the characters or glyphs in the shape. It contains one entry for each character or glyph in the shape.

text A set of specific symbols that, when displayed in a meaningful order, conveys information.

text area The space on the display device within which the text should fit.

text attributes The set of flags that allow you to specify how QuickDraw GX alters glyph outlines or chooses the proper metrics for horizontal or vertical text.

text direction The direction in which reading proceeds. Roman text has a left-to-right direction; Hebrew and Arabic have a (predominantly) right-to-left direction; Chinese and Japanese can have a vertical direction.

text face An algorithmic way for your application to produce typestyles.

text run A complete unit of text, made up of character codes or glyph codes.

text shape A typographic shape object containing a string of text associated with a single style object. Compare **glyph shape** and **layout shape.** See **typographic shapes.**

text width The area between the margins; it is the length available for displaying a line of text.

tiled highlighting A highlighting mechanism whereby the highlighted area corresponding to every character in a line of text is unique, without gaps or overlaps.

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

tracking Kerning between all glyphs in the shape, not just the kerning pairs already defined by the font. You can increase or decrease interglyph spacing by using a track number. See **kerning.**

track setting A value that specifies the relative tightness or looseness of interglyph spacing.

trailing edge The edge of a glyph that is encountered last when reading text of that glyph's language. For glyphs of left-to-right text, the trailing edge is the right edge; for glyphs of right-to-left text, the trailing edge is the left edge.

typestyle A variant version of glyphs in the same font family. Typical typestyles available on the Macintosh computer include bold, italic, underline, outline, shadow, condensed, and extended.

typographic bounding rectangle The smallest rectangle that encloses the full span of the glyphs from the ascent line to the descent line.

typographic point A unit of measurement describing the size of glyphs in a font. There are 72.27 typographic points per inch, as opposed to 72 points per inch in QuickDraw GX.

typographic shapes The QuickDraw GX shapes that display text: text shapes, glyph shapes, and layout shapes.

unidirectional text A sequence of text that has a single direction. Compare **mixed-direction text.**

unlimited gap absorption The assignment of all justification gap to an individual glyph or priority of glyphs, regardless of the specified grow or shrink limits for that glyph or glyphs.

variation axis A range included in a font by the font designer that allows a font to produce different typestyles.

weak type A glyph directionality that depends on context to determine whether it is left to right or right to left. Compare **strong type, neutral type.**

with-stream kerning The automatic movement of glyphs parallel to the line orientation of the text. Compare **cross-stream kerning.**

with-stream shift A positional shift that applies equally to all glyphs in a style run by adding or removing space before or after each glyph in the run. Compare **cross-stream shift.**

WorldScript A group of Macintosh system software managers, extensions, and resources that facilitate multilanguage text processing.

x-height The position where the top of the lowercase “x” in the font lies; this measurement usually marks the height of the body of all lowercase glyphs, excluding ascenders and descenders, in the font.

Index

Numerals

A

B

C

D

E

F

GA–GXA

GXB

GXC

GXD

GXE, GXF

GXG

GXH

GXI

GXJ, GXK

GXL

GXM

GXN, GXO

GXP, GXQ

GXR

GXS

GXT

GXU–GXZ

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W, X, Y, Z

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter IINTX printer. Final page negatives were output directly from text files on an Optrotech SPrint 220 imagesetter. Line art was created using Adobe™ Illustrator. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino and display type is Helvetica. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

WRITERS
Linda Kyrnitszke, Tom Maremaa, Diane Patterson, David Bice

DEVELOPMENTAL EDITORS
Anne Szabla, Antonio Padial

ILLUSTRATORS
Sandee Karr, Lisa Hymel

PRODUCTION EDITOR
Rex Wolf

PROJECT MANAGER
Trish Eastman

LEAD WRITER
David Bice

LEAD EDITOR
Laurel Rezeau

ART DIRECTORS
Barbara Smyth, Ruth Anderson

COVER DESIGNER
Barbara Smyth

Special thanks to Dave Opstad, Mike Reed

Acknowledgments to Peter “Luke” Alexander, Alex Beaman, Richard Becker, Cary Clark, Matt Deatherage, Robert Dierkes, Eric Mader, Buck Melton, Ian Ritchie, Chris Yerga