

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

Layout Styles


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

 Style object		
Pen width	<i>Font</i>	Run controls
Cap	Text face	Kerning adjustments array
Join	Text size	Glyph substitutions array
Dash	Alignment	Run features array
Pattern	Font variations	Priority justification override
Curve error	Encoding	Glyph justification overrides array
Attributes	Text attributes	
Owner count		
<i>Tag list</i>		

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

Layout Styles

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

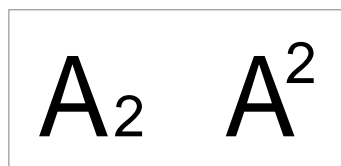
Your application can specify two types of positional shifts that apply equally to all glyphs in a style run. **With-stream shift** adds or removes space before or after each glyph in the run, and can be used for manual kerning or letterspacing. **Cross-stream shift** raises or lowers the entire style 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

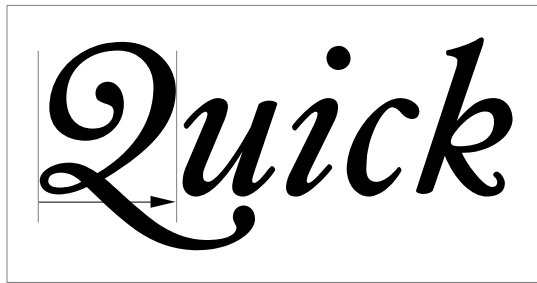
A sample function that makes use of both with-stream and cross-stream shift is shown in Listing 8-2 on page 8-43.

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

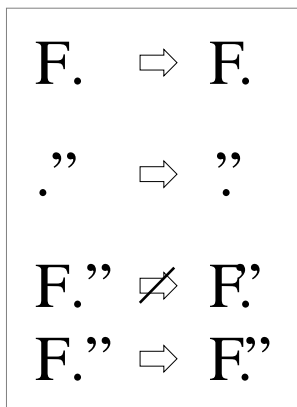
Kerning is an automatic, fine adjustment to normal spacing that QuickDraw GX applies to specific pairs or groups of glyphs, or to glyphs in specific contexts. It is commonly used to increase the overlap between glyphs that “fit together” naturally. Unlike manual shifting, kerning does not apply evenly 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



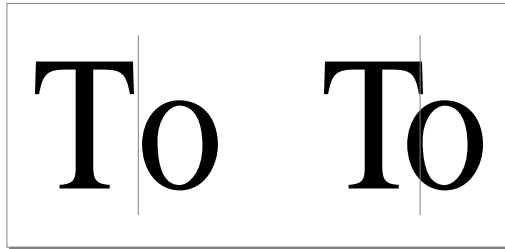
QuickDraw GX uses information in font tables to determine how much to increase or decrease the space between glyphs. In the general case, this amount can depend on more than just the two adjacent glyphs: it can also depend on preceding or following glyphs, or even on glyphs in other parts of the line. 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



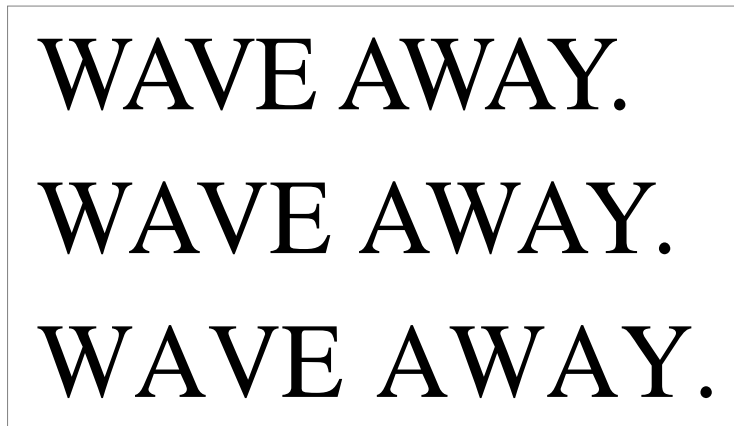
Cross-stream kerning is required for scripts like Taliq (used in Urdu). It can also be used to assist in the creation of automatic fractions. (See page 8-32 for additional discussion of automatic fractions.)

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

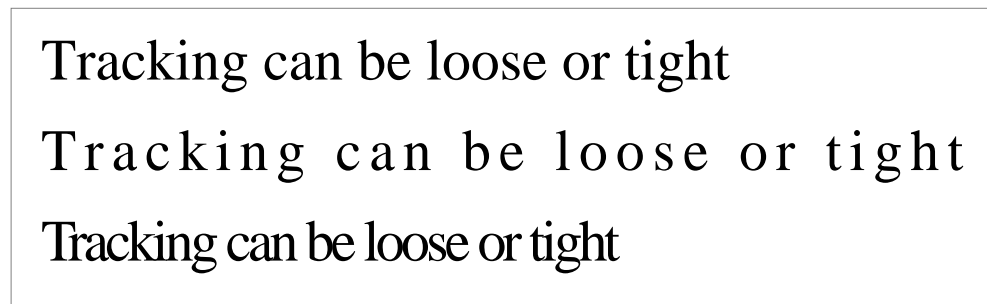
Tracking is another glyph-positioning adjustment you can control. You can expand or contract the spacings of all glyphs in a style run by applying a value to that style run. This value, called the **track setting**, uses information defined by the font to specify whether interglyph spacing is to be tightened 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

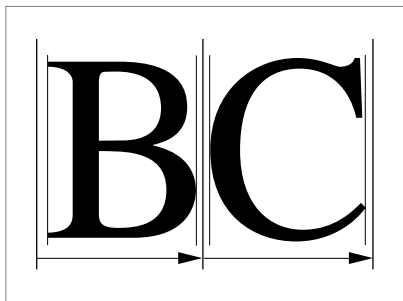


The sample function that generated Figure 8-10 is shown in Listing 8-3 on page 8-45.

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.

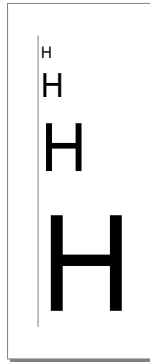
Figure 8-11 Advance widths, including side bearings to allow for interglyph spacing



Layout Styles

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

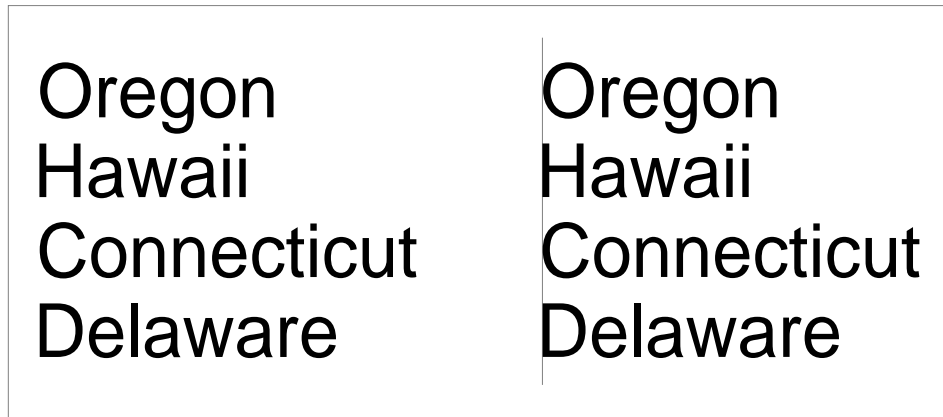


The second problem is that due to optical effects, curved lines do not appear to line up properly with straight lines. To make them appear to line up, some compensation must occur. On baselines, for example, curved letters such as “C” or “S” are generally designed to extend slightly below the baseline, 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

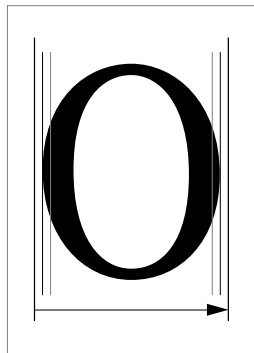


This same effect should happen at the edges of lines. On the left side of Figure 8-14, the “O” in “Oregon” and the “C” in “Connecticut” appear to be indented compared to the “H” and “D” glyphs. However, as shown by the vertical line on the right, the outlines of the four glyphs are exactly aligned. The 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.

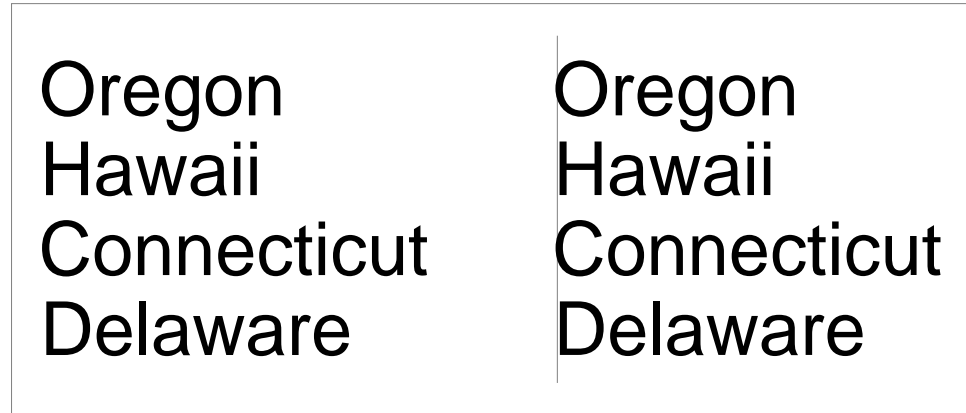
In Figure 8-15, the black arrow spans the distance from the origin to the advance width of the glyph, defining the width of the glyph plus side bearings. The spaces between the solid lines represent the side bearings, and the dashed lines represent the **optical edges** of the glyph. Note that on each 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

Layout Styles

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

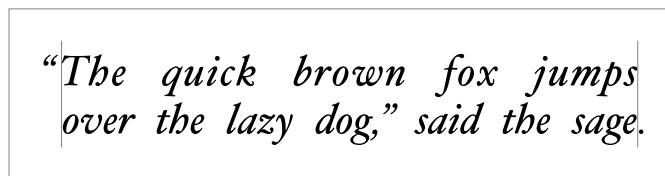


QuickDraw GX applies optical alignment by default. Your application can suppress it by setting the flag `gxNoOpticalAlignment` in the run controls structure. The `gxNoOpticalAlignment` flag is described on page 8-61.

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.

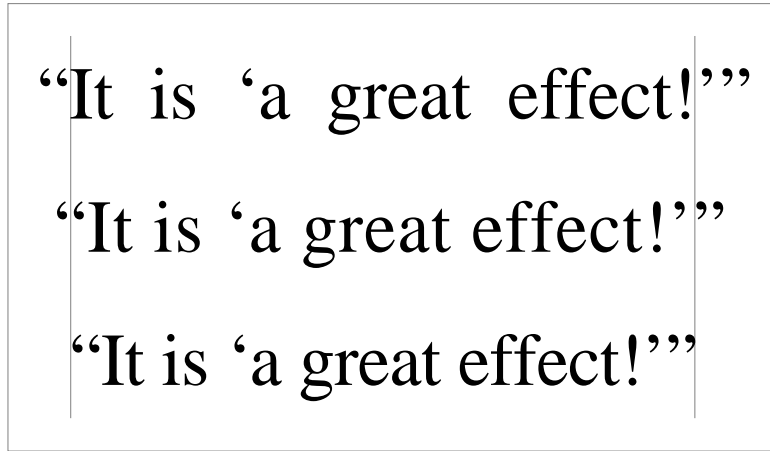
Figure 8-17 Automatic hanging punctuation



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



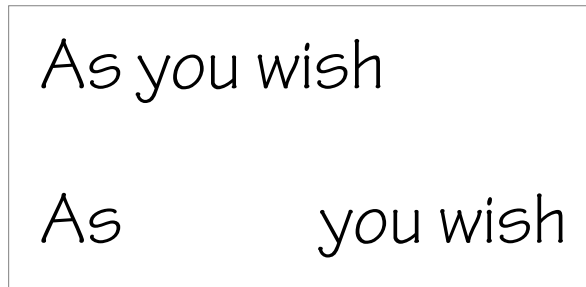
Imposed Width

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.

Layout Styles

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 $(2x + b)$. 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.

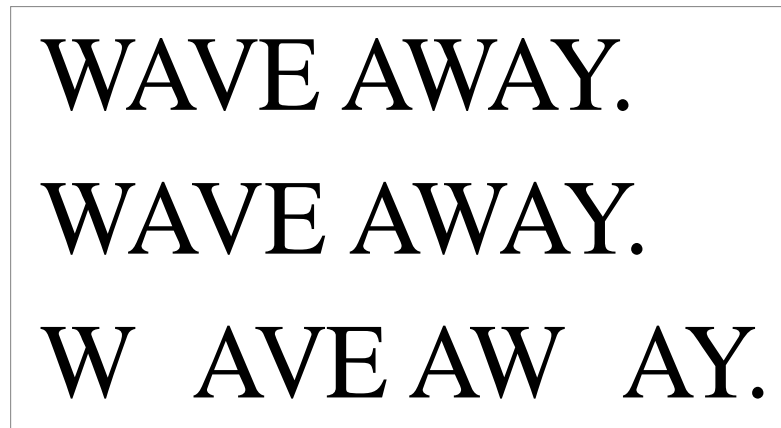
With a scale factor of 0, you would get

$$x' = x + (0x + b)$$

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



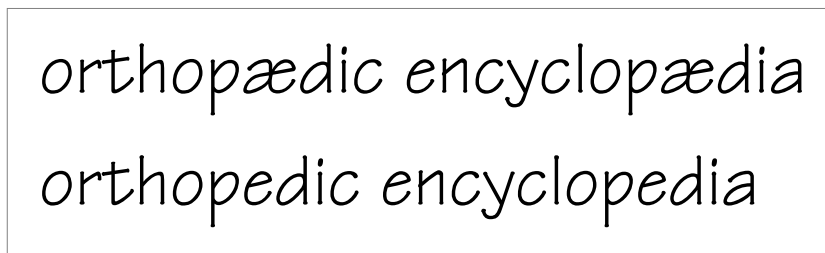
The sample function that generated Figure 8-21 is shown in Listing 8-7 on page 8-50.

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
<code>allTypographicFeaturesType</code>	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.
<code>ligaturesType</code>	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.
<code>cursiveConnectionType</code>	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
<code>letterCaseType</code>	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.
<code>verticalSubstitutionType</code>	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.
<code>linguisticRearrangementType</code>	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.
<code>smartSwashType</code>	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.
<code>diacriticsType</code>	Controls whether diacritical marks are shown or hidden. Table 8-9 on page 8-31 lists the feature selectors related to this feature type.
<code>verticalPositionType</code>	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.
<code>fractionsType</code>	Controls automatic substitution or formation of fractions. Table 8-11 on page 8-33 lists the feature selectors related to this feature type.
<code>overlappingCharactersType</code>	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.
<code>characterShapeType</code>	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.
<code>numberSpacingType</code>	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
<code>numberCaseType</code>	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.
<code>styleOptionsType</code>	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.
<code>typographicExtrasType</code>	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.
<code>mathematicalExtrasType</code>	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.
<code>ornamentSetsType</code>	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.
<code>characterAlternativesType</code>	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.
<code>designComplexityType</code>	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.

Layout Styles

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
<code>allTypeFeaturesOnSelector</code>	Tells QuickDraw GX to use the font features specified in this style run’s run-features array and the defaults specified by the font.
<code>allTypeFeaturesOffSelector</code>	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

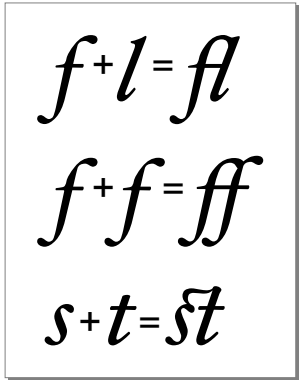
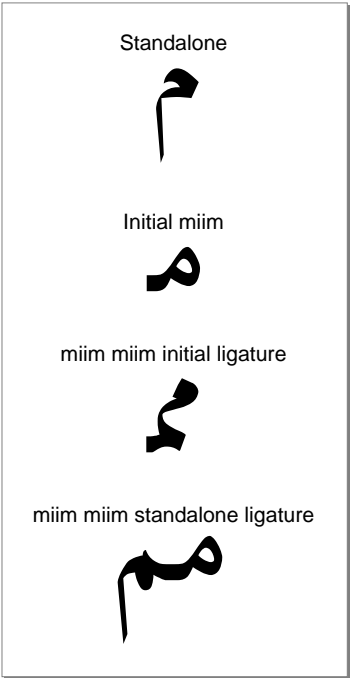


Figure 8-24 A ligature in Arabic text



Layout Styles

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”

Independent	Final	Medial	Initial
ه	ـه	هـ	هـ

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
<code>requiredLigaturesOnSelector</code> <code>requiredLigaturesOffSelector</code>	Allows or prevents the use of ligatures that the font designates as required by the language (such as certain Arabic ligatures).
<code>commonLigaturesOnSelector</code> <code>commonLigaturesOffSelector</code>	Allows or prevents the use of ligatures that the font designates as “common,” or normally used (such as the “fi” ligature in Roman text).
<code>rareLigaturesOnSelector</code> <code>rareLigaturesOffSelector</code>	Allows or prevents the use of ligatures that the font designates as “rare” (such as “ct” or “ss” ligatures).
<code>logosOnSelector</code> <code>logosOffSelector</code>	Allows or prevents the use of ligatures that the font designates as logotypes (typically used for trademarks or other special display text).
<code>rebusPicturesOnSelector</code> <code>rebusPicturesOffSelector</code>	Allows or prevents the use of rebuses (pictures that represent words or syllables).
<code>diphthongLigaturesOnSelector</code> <code>diphthongLigaturesOffSelector</code>	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

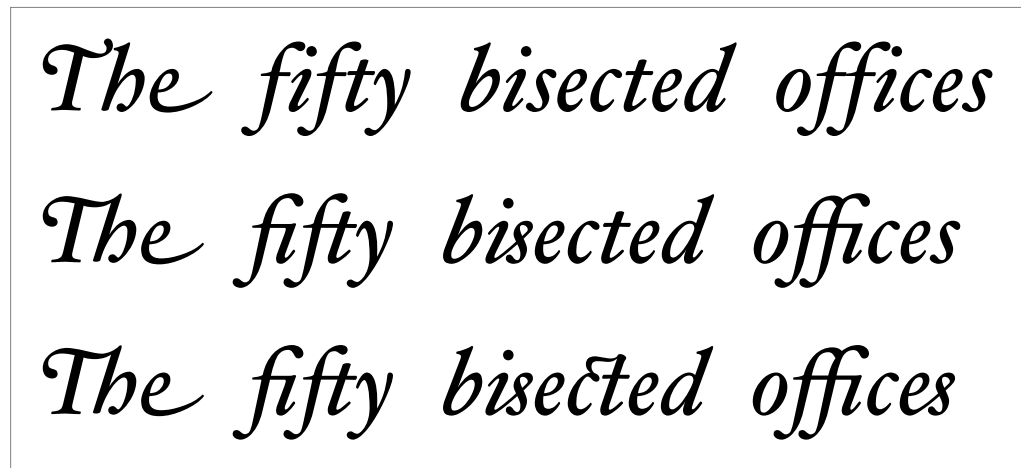
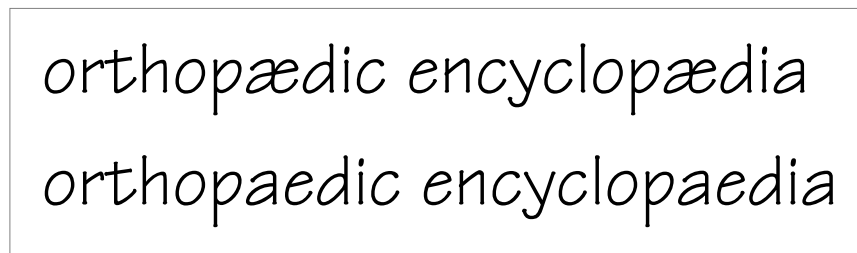


Figure 8-27 shows the results of selection (upper) and deselection (lower) of diphthong ligatures.

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
<code>unconnectedSelector</code>	Disables cursive connection.
<code>partiallyConnectedSelector</code>	Specifies noncontextual cursive connection.
<code>cursiveSelector</code>	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

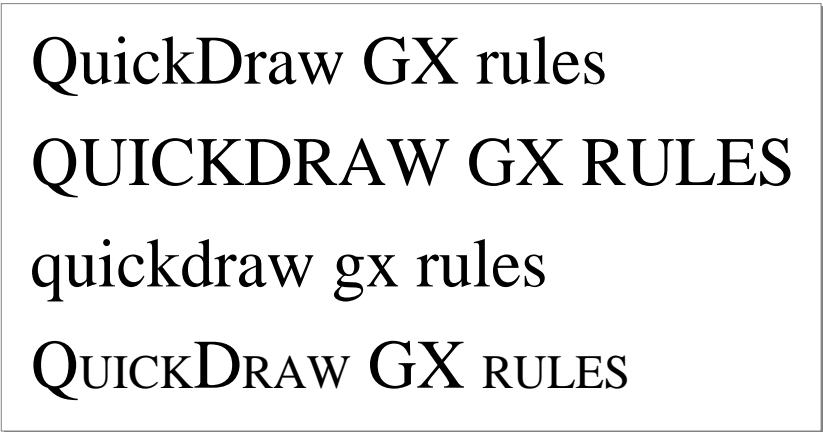
In fonts for languages in which case is significant, QuickDraw GX allows you to specify certain automatic case changes. If the font supports the letter case feature type, you can select features that specify case changes such as those shown in Table 8-5.

Table 8-5 Feature selectors for the `letterCaseType` feature type

Constant	Explanation
<code>upperAndLowerCaseSelector</code>	Specifies no case conversion.
<code>allCapsSelector</code>	Specifies conversion of all letters to uppercase. (This feature is noncontextual.)
<code>allLowerCaseSelector</code>	Specifies conversion of all letters to lowercase. (This feature is noncontextual.)
<code>smallCapsSelector</code>	Specifies conversion of all lowercase letters to small caps. (This feature is noncontextual.)
<code>initialCapsSelector</code>	Specifies conversion of all lowercase letters at the beginnings of words to uppercase. (This feature is contextual.)
<code>initialCapsAndSmallCapsSelector</code>	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. ♦

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

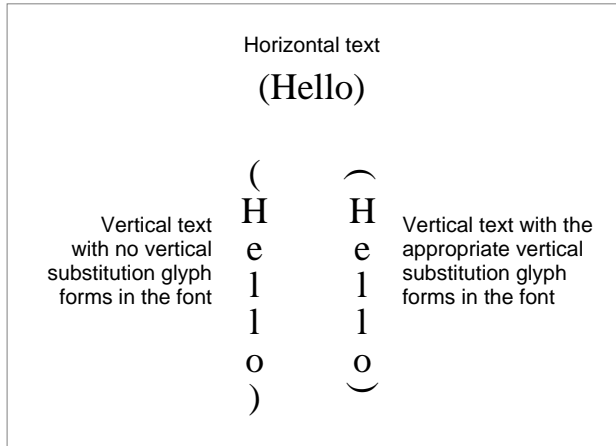
`substituteVerticalFormsOnSelector`
`substituteVerticalFormsOffSelector`

Explanation

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

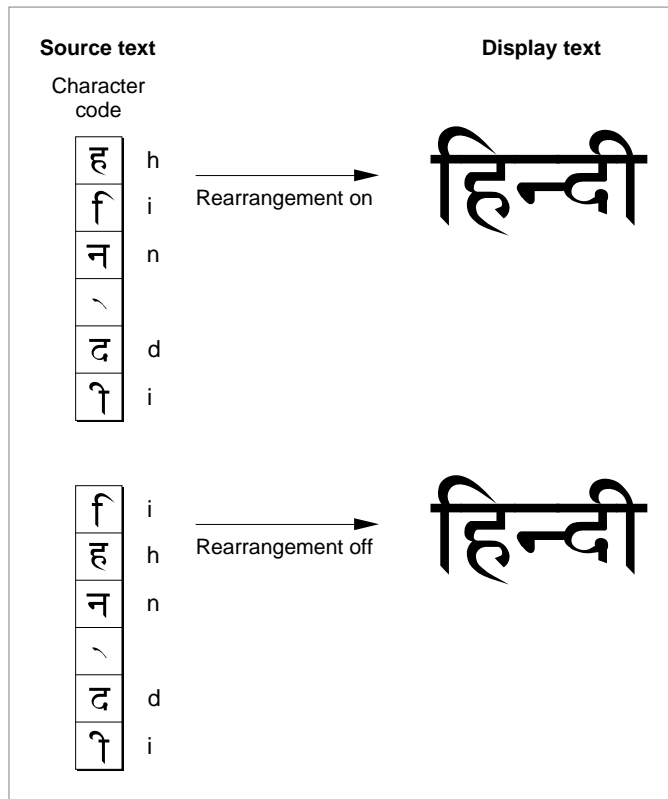
Linguistic (Indic-style) rearrangement is a standard feature of Devanagari and other South Asian scripts. However, users may not always want it to occur, preferring instead to enter characters in an “already reversed” order. If a font supports the rearrangement feature type, you can either allow the 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
<code>linguisticRearrangementOnSelector</code>	Allows or prevents the automatic rearrangement of certain glyphs as required by language rules.
<code>linguisticRearrangementOffSelector</code>	

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.

Layout Styles

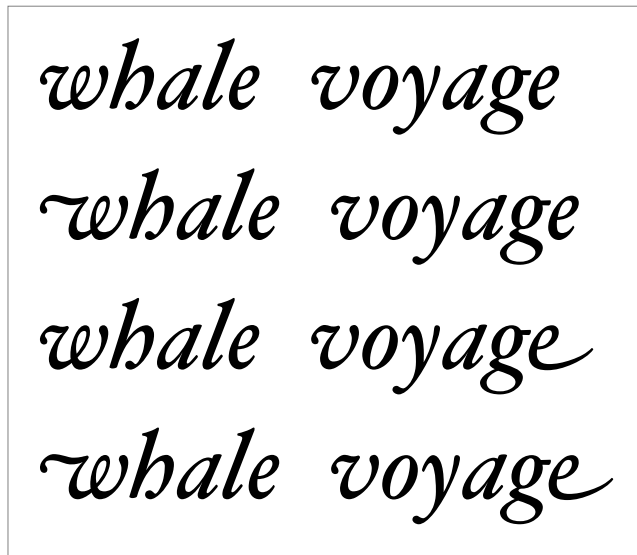
Collections of swash forms called **smart swashes** can be designated by the font designer and put in swash tables. Smart swashes are contextual and swashes are not. If the font supports the smart swashes feature type, you can select features that allow you to specify sets of swashes, such as shown in Table 8-8.

Table 8-8 Feature selectors for the `smartSwashType` feature type

Constant	Explanation
<code>wordInitialSwashesOnSelector</code> <code>wordInitialSwashesOffSelector</code>	Allows or prevents the substitution of swash variants that begin words.
<code>wordFinalSwashesOnSelector</code> <code>wordFinalSwashesOffSelector</code>	Allows or prevents the substitution of swash variants that end words.
<code>lineInitialSwashesOnSelector</code> <code>lineInitialSwashesOffSelector</code>	Allows or prevents the substitution of swash variants that begin lines.
<code>lineFinalSwashesOnSelector</code> <code>lineFinalSwashesOffSelector</code>	Allows or prevents the substitution of swash variants that end lines.
<code>nonFinalSwashesOnSelector</code> <code>nonFinalSwashesOffSelector</code>	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. ♦

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
<code>showDiacriticsSelector</code>	Specifies that QuickDraw GX is to form accent ligatures on the glyphs they apply to.
<code>hideDiacriticsSelector</code>	Specifies that QuickDraw GX is not to form any accent ligatures.
<code>decomposeDiacriticsSelector</code>	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)

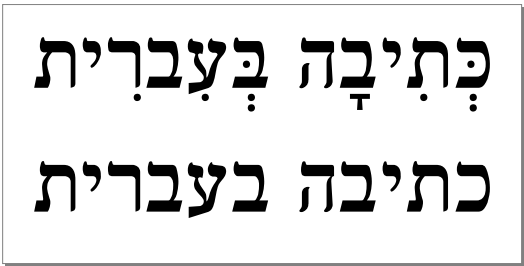
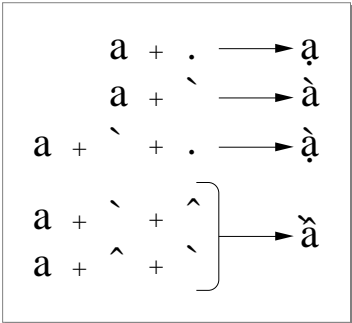


Figure 8-34 shows an example of text drawn with and without its accents.

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
<code>normalPositionSelector</code>	Specifies use of normally positioned glyph set.
<code>superiorsSelector</code>	Specifies use of superiors: glyph variants that are positioned above the baseline, used typically for superscripts.
<code>inferiorsSelector</code>	Specifies use of inferiors: glyph variants that are positioned below the baseline, used typically for subscripts.
<code>ordinalsSelector</code>	Specifies contextual substitution of glyphs that replace ordinal designations attached to numerals (such as “1 st ” substituting for “1st”).

Fractions

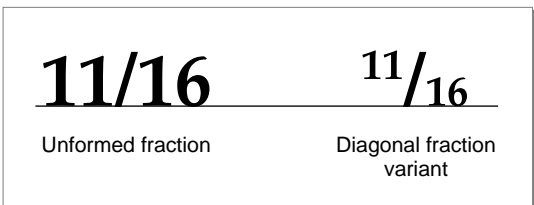
There are several ways to generate fractions with QuickDraw GX. For a font that supports the fractions feature type, you may be able to select between two different types of automatic fraction generation, as shown in Table 8-11.

Table 8-11 Feature selectors for the `fractionsType` feature type

Constant	Explanation
<code>noFractionsSelector</code>	Specifies no substitution or construction of fractions.
<code>verticalFractionsSelector</code>	Specifies replacement of slash-separated numeric sequences with predrawn fraction glyphs, if present in the font.
<code>diagonalFractionsSelector</code>	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. ♦

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”.

Layout Styles

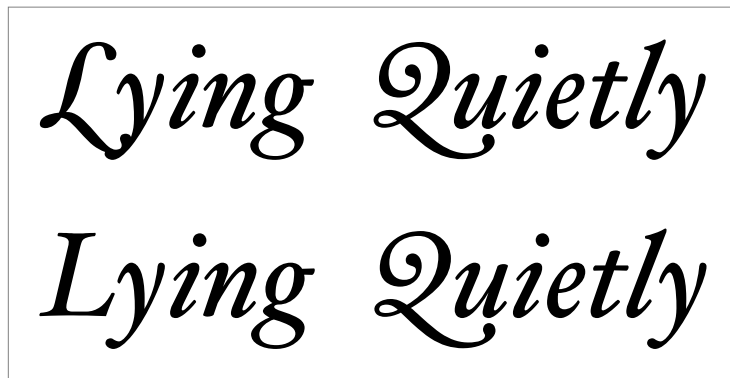
For fonts that support the glyph overlap feature type, you can specify that no glyph may overlap the outline of the following glyph. If it does, a non-overlapping form of the glyph is substituted. Table 8-12 lists the selectors for this feature.

Table 8-12 Feature selectors for the `overlappingCharactersType` feature type

Constant	Explanation
<code>preventOverlapOnSelector</code>	Prevents or allows the collision of an extended part of one glyph with an adjacent glyph.
<code>preventOverlapOffSelector</code>	

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. ♦
Table 8-13 lists the selectors for this feature.

Table 8-13 Feature selectors for the `characterShapeType` feature type

Constant	Explanation
<code>traditionalCharactersSelector</code>	Specifies the use of traditional characters.
<code>simplifiedCharactersSelector</code>	Specifies the use of simplified characters.

Number Width

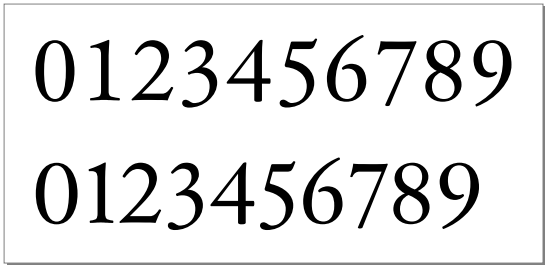
Many fonts support both proportional-width and fixed-width numerals, as shown in Figure 8-38. In proportional-width numerals the “1” is narrower than the “0”, whereas in fixed-width numerals they (and all the other numerals) have identical widths. Fixed-width numerals are also called *columnating* because 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
<code>monospacedNumbersSelector</code>	Specifies the use of fixed-width (columnating) numerals.
<code>proportionalNumbersSelector</code>	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

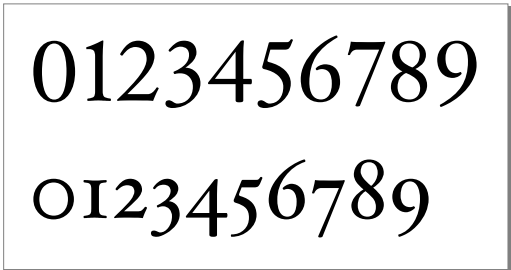
Some fonts support both lowercase (also called traditional or old-style) numerals, in which some glyphs extend below the baseline, and uppercase (also called *lining*) numerals, in which no glyphs extend below the baseline. For fonts that support the number case feature type, you can select either kind of numeral. Table 8-15 lists the selectors for this feature.

Table 8-15 Feature selectors for the `numberCaseType` feature type

Constant	Explanation
<code>lowerCaseNumbersSelector</code>	Specifies the use of lowercase (old-style) numerals.
<code>upperCaseNumbersSelector</code>	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
<code>noStyleOptionsSelector</code>	Specifies the use of the standard glyph set.
<code>displayTextSelector</code>	Specifies the use of a glyph set that is designed for best display at large sizes (over 24 point).
<code>engravedTextSelector</code>	Specifies the use of a glyph set that has contrasting strokes parallel to the main stroke, giving an engraved effect.
<code>illuminatedCapsSelector</code>	Specifies the use of a glyph set with complex decoration surrounding the glyphs of capital letters.
<code>titlingCapsSelector</code>	Specifies the use of a glyph set in which capital letters have a special form for display in titles.
<code>tallCapsSelector</code>	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
<code>hyphensToEmDashOnSelector</code> <code>hyphensToEmDashOffSelector</code>	Allows or prevents the automatic replacement of two adjacent hyphens with an em dash.
<code>hyphenToEnDashOnSelector</code> <code>hyphenToEnDashOffSelector</code>	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.
<code>unslashedZeroOnSelector</code> <code>unslashedZeroOffSelector</code>	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
<code>formInterrobangOnSelector</code> <code>formInterrobangOffSelector</code>	Allows or prevents the automatic replacement of the sequence “?!” or “!?” with the font’s interrobang glyph.
<code>smartQuotesOnSelector</code> <code>smartQuotesOffSelector</code>	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
<code>hyphenToMinusOnSelector</code> <code>hyphenToMinusOffSelector</code>	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 (–).
<code>asteriskToMultiplyOnSelector</code> <code>asteriskToMultiplyOffSelector</code>	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 (×).
<code>slashToDivideOnSelector</code> <code>slashToDivideOffSelector</code>	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 (÷).
<code>inequalityLigaturesOnSelector</code> <code>inequalityLigaturesOffSelector</code>	Allows or prevents the automatic replacement of sequences such as “>=” and “<=” with equivalent ligatures “≥” and “≤”.
<code>exponentsOnSelector</code> <code>exponentsOffSelector</code>	Allows or prevents the automatic replacement of the sequence <i>exponentiation glyph</i> —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. ♦

Ornament Sets

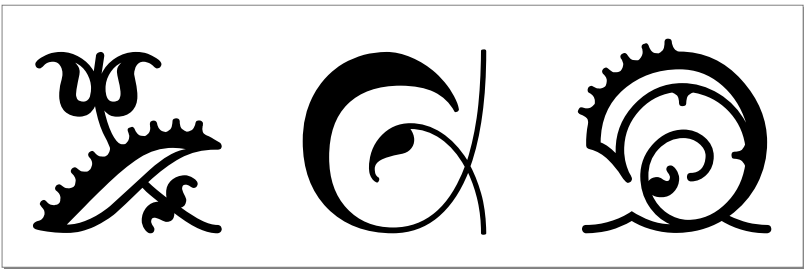
Fonts may include ornamental, nonalphabetic glyph sets used for various purposes. With a font that supports the ornament set feature type, you may be able to select among those glyph sets, using selectors such as those shown in Table 8-19.

Table 8-19 Feature selectors for the `ornamentSetsType` feature type

Constant	Explanation
<code>noOrnamentsSelector</code>	Specifies the use of no ornamental glyph sets.
<code>dingbatsSelector</code>	Specifies the use of dingbats: arrows, stars, bullets, and so on.
<code>piCharactersSelector</code>	Specifies the use of pi characters: related nonalphabetic symbols, such as musical notation glyphs.
<code>fleuronsSelector</code>	Specifies the use of fleurons: ornaments such as flowers, vines, and leaves.
<code>decorativeBordersSelector</code>	Specifies the use of decorative borders: glyphs used in interlocking patterns to form text borders.
<code>internationalSymbolsSelector</code>	Specifies the use of international symbols, such as the barred circle representing “no”.
<code>mathSymbolsSelector</code>	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.

Layout Styles

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
<code>noAlternatesSelector</code>	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
<code>designLevel1Selector</code>	Specifies the basic glyph set.
<code>designLevel2Selector</code>	Specifies an alternate glyph set.
<code>designLevel3Selector</code>	Specifies an alternate glyph set.
<code>designLevel4Selector</code>	Specifies an alternate glyph set.
<code>designLevel5Selector</code>	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-

Layout Styles

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);
    }
}
```

Layout Styles

```

    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);

```

Layout Styles

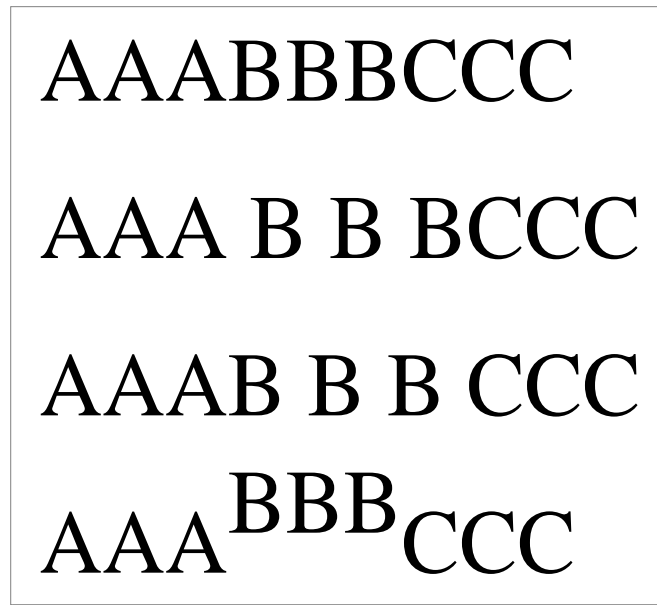
```

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`

Layout Styles

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 = frctl1;

    /* 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);
}
```

Layout Styles

```

    /* 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

Layout Styles

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);
    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;
```

Layout Styles

```

/* 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.

Layout Styles

```

struct gxRunControls {
    gxRunControlFlags    flags;
    Fixed                beforeWithStreamShift;
    Fixed                afterWithStreamShift;
    Fixed                crossStreamShift;
    Fixed                imposedWidth;
    Fixed                track;
    fract                hangingInhibitFactor;
    fract                kerningInhibitFactor;
    Fixed                decompositionAdjustmentFactor;
    gxBaselineType       baselineType;
} ;

```

Field descriptions

<code>flags</code>	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 <code>gxRunControlFlags</code> value.
<code>beforeWithStreamShift</code>	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.
<code>afterWithStreamShift</code>	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.
<code>crossStreamShift</code>	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.
<code>imposedWidth</code>	The width (in points, 72 per inch) to be imposed on each glyph in this style run, if the <code>gxImposeWidth</code> flag is set in the <code>flags</code> 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.

Layout Styles

<code>track</code>	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 <code>gxNoTracking</code> 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.
<code>hangingInhibitFactor</code>	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.
<code> KerningInhibitFactor</code>	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.
<code>decompositionAdjustmentFactor</code>	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.
<code>baselineType</code>	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 <code>gxNoOverrideBaseline</code> . 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

<code>gxNoDirectionOverride</code>	Instructs QuickDraw GX to use the normal direction of the text in the style run.
<code>gxImposeLeftToRight</code>	Instructs QuickDraw GX to force the text to be treated as left-to-right.
<code>gxImposeRightToLeft</code>	Instructs QuickDraw GX to force the text to be treated as right to left.
<code>gxImposeArabic</code>	Instructs QuickDraw GX to force the text to be treated as Arabic letters. Numbers interacting with <code>gxImposeArabic</code> 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 <i>The Unicode Standard: Worldwide Character Encoding, Version 1.0</i> , 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.

Layout Styles

`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. ♦

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`

Layout Styles

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,

Layout Styles

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);
```

Layout Styles

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[]);
```

Layout Styles

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[]);
```

<code>target</code>	A reference to the style object whose kerning adjustments array you are assigning.
<code>count</code>	The number of kerning adjustment structures to assign; the number of elements in the kerning adjustments array.
<code>kerningAdjustments</code>	The array of kerning adjustment structures to assign to the style object. If you specify <code>nil</code> for this parameter and 0 for the <code>count</code> 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`.

Layout Styles

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.

Layout Styles

`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[]);
```

<code>target</code>	A reference to the style object whose glyph substitutions array you are assigning.
<code>count</code>	The number of glyph substitution structures to assign; the number of elements in the glyph substitutions array.
<code>glyphSubstitutions</code>	The array of glyph substitution structures to assign to the style object referenced in the <code>target</code> parameter. If you specify <code>nil</code> for this parameter and 0 for the <code>count</code> 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);
```

Layout Styles

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.

Layout Styles

`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[]);
```

<code>target</code>	A reference to the style object whose run-features array you are assigning.
<code>count</code>	The number of run-feature structures to assign; the number of elements in the run-features array.
<code>runFeatures</code>	The array of run-feature structures to assign to the style object referenced in the <code>target</code> parameter. If you specify <code>nil</code> for this parameter and 0 for the <code>count</code> 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`.

Layout Styles

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[]);
```

<code>target</code>	A reference to the shape object whose associated style object is to be assigned the run-features array.
<code>count</code>	The number of run-feature structures to assign; the number of elements in the run-features array.
<code>runFeatures</code>	The array of run-feature structures to assign to the style object associated with the shape referenced in the <code>target</code> parameter. If you specify <code>nil</code> for this parameter and 0 for the <code>count</code> 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.

Layout Styles

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,
```

Layout Styles

```

    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[]);

```