

This chapter describes those features of layout shapes that help you lay out and manipulate an entire line of text. Line span, line length and line breaking, text direction, and justification can affect the text of a whole line, regardless of the number or characteristics of the individual style runs making up the line.

Although it is possible to create and draw a layout shape based solely on information presented in the chapter “Layout Shapes” in this book, most applications need to use the information presented here to take advantage of the layout capabilities that QuickDraw GX provides. Read the information in this chapter if you create layout shapes and need to control line characteristics. If you do not create layout shapes, you do not need the information in this chapter.

Before reading this chapter, you should be familiar with the information in the chapters “Introduction to QuickDraw GX Typography,” “Typographic Shapes,” “Typographic Styles,” and “Layout Shapes” in this book. You should also be familiar with the general concepts of QuickDraw GX objects, as described in *Inside Macintosh: QuickDraw GX Objects*.

Some of the information in this chapter concerns layout-related properties of the style object. Most layout-related style properties are discussed in the chapter “Layout Styles” in this book. Those discussed here, the justification-related properties, are presented in this chapter because they are typically manipulated in the context of the line as a whole.

This chapter presents detailed information on text direction and nested direction levels, even though the levels array of the layout shape is introduced earlier in this book, in the chapter “Layout Shapes.” Text direction and nested direction levels can affect the entire line, and therefore the details of how to manipulate them are presented here.

The chapter starts by describing how QuickDraw GX defines baselines, line measurement, text direction, and justification. It then describes how to use QuickDraw GX functions to

- set baselines
- determine line lengths and line spans
- break lines
- manipulate text direction, including nested direction levels in mixed-direction text
- perform full and partial justification with a variety of justification techniques
- change the justification behavior of classes of glyphs
- change the justification behavior of individual glyphs

About Line Control and Line Measurement for Layout Shapes

Text-handling with layout shapes is line-based. The features of layout shapes are designed mainly to allow you to lay out individual lines of text that uses complex formatting. Although your application may work with paragraphs, sections, chapters, and other larger text units, each line of your documents is at some point manipulated as a separate layout shape.

Layout Line Control

This section describes how QuickDraw GX defines and allows you to manipulate the following line-based components in text layout:

- baselines
- line measurement and line breaking
- text direction
- justification

Baselines

A **baseline** is an imaginary line that is used to align glyphs in a line of text. A baseline can coincide with various locations in a glyph, such as the bottom, middle, or top, depending on what type of baseline it is. The baseline represents a stable platform, giving a common point of alignment to glyphs of different shapes and sizes.

Baseline Types

There are several common types of baselines used to lay out text:

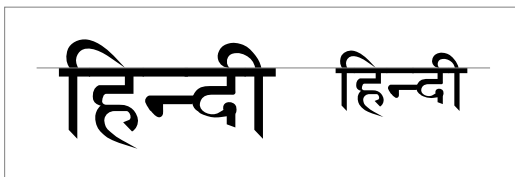
- **Roman baseline.** The baseline used in most Roman scripts, as well as by Arabic and Hebrew. Most of the glyph appears above the Roman baseline, sometimes with portions below it, as with glyphs such as “y” or “j”. The baseline is near the bottom of the entire row of glyphs:



- **Ideographic centered baseline.** A baseline used by Chinese, Japanese, and Korean ideographic scripts; glyphs are centered halfway on the line height:



- **Hanging baseline.** The baseline used by Devanagari and similar scripts. Most of the glyph is below the baseline, sometimes with portions above it, and the baseline is near the top of the glyphs:



- **Math baseline.** A baseline used for setting mathematical expressions, centered on operators such as the minus sign. Such operators usually appear at half the x-height in a font:

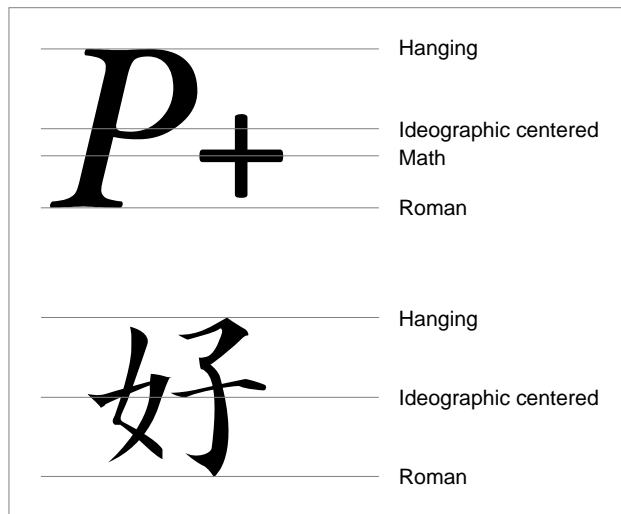
QuickDraw GX supports vertical text, although not by using vertical baselines. All baselines in QuickDraw GX are horizontal, but you can specify text characteristics such that the line is displayed properly when rotated to a vertical position. For more information, see “Baselines for Vertical Text” beginning on page 9-8. Also, see the introductory discussion of vertical text in the chapter “Layout Shapes” in this book.



QuickDraw GX supports a variety of baselines and allows you to mix text of various baselines and sizes on a single line. QuickDraw GX baseline types are defined in the `gxBaselineType` enumeration, described on page 9-58.

Font and Application Control Over Baselines

Each QuickDraw GX-compatible font specifies a baseline type for every glyph in the font. It also specifies the positions of each baseline in the font’s own overall coordinate system. This information is present only for the font as a whole; each glyph in the font shares the same set of baseline positions, although different glyphs in a font may use different baseline types within the set. Figure 9-1 shows where these baselines might be for two glyphs from two different fonts.

Figure 9-1 Baseline positions for two fonts

When you prepare to draw a line of text, you need to identify an overriding baseline (if desired) for each style run, and you also need to identify offsets from the y-coordinate of the layout shape's position for each of the baseline types. QuickDraw GX provides a routine to assist in this process. Figure 9-2 shows examples of a line with six style runs rendered with two different sets of baseline information. Assume for this example that the position of each shape is at the lower-left corner of the first 'A'. The first line shows the last three style runs having their baseline type overridden to the Roman baseline, where the Roman baseline has a delta of zero from the shape's position. The second line shows the baseline type of the first three style runs overridden to the hanging baseline, which has a nonzero delta from the shape's position.

The information overriding the baseline for each style run is contained in the style's run controls, described on page 8-61. The information defining the baseline deltas is contained in the `gxBaselineDeltas` array, described on page 9-59.

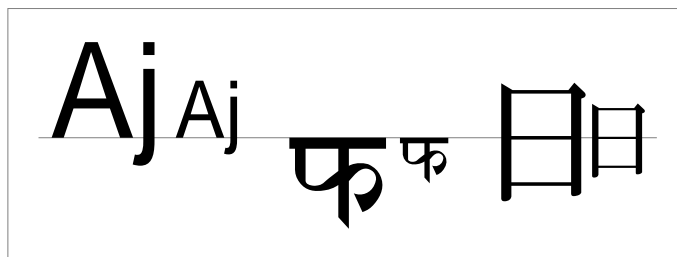
Alignment of Multiple Baselines

This section describes how QuickDraw GX makes use of the baseline deltas to lay out text. If a layout shape comprises glyphs of only one baseline type, the alignment of the glyphs is obvious. But if the text in the shape is of different sizes or uses several different baseline types, it may not be obvious at first glance how best to line up the text:

- One (incorrect) possibility might be to simply align all glyphs with a y-delta of 0. That strategy works for text whose default baseline happens to be $y = 0$, but it does not work for other text. Figure 9-2 shows the misalignment resulting from using a hanging baseline for mixed-size Roman text and a Roman baseline for Devanagari text.

Figure 9-2 How the same glyphs can align to different baselines

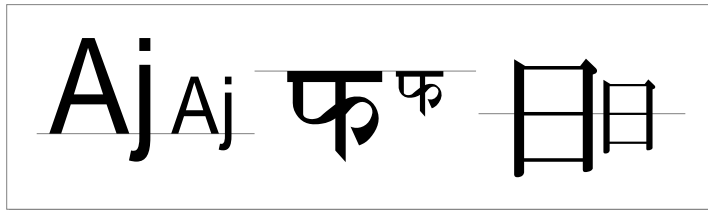
- Another (incorrect) possibility is to line up each style run's default baseline with the primary baseline you have chosen for the layout. Figure 9-3 shows an example in which the dominant style run is the leftmost and the default baseline type for the layout shape is Roman. The text within each style run aligns with its own baseline type: Roman text sits properly on its Roman baseline, Devanagari text hangs from its baseline, and the Chinese text straddles its ideographic centered baseline. Each style run is correct within itself, but the runs do not line up correctly because the baselines should not be aligned.

Figure 9-3 Text with multiple baselines aligned to $y = 0$ 

Layout Line Control

- The best results occur when the default baseline for each style run is aligned with the appropriate choice from the set of baselines as defined in the dominant style run. This is the method that QuickDraw GX is designed to support. Figure 9-4 shows the same text as Figure 9-3, but this time the Devanagari and Chinese text align with the hanging and ideographic centered baseline types defined for the Roman baseline of the Roman text.

Figure 9-4 Preferred alignment for multiple baselines



If you supply QuickDraw GX with the proper information about your text (what the deltas are between the other baseline types and what baseline type each of your style runs uses), it automatically aligns the baselines as shown in Figure 9-4, giving you the best results for multilanguage layout.

Drop capitals

Drop capitals are large uppercase letters that drop below the main line of text for aesthetic reasons. You can create a drop capital by setting the baseline type for individual runs of letters to the hanging baseline type in the run controls and by making sure the array of distances to the various baselines is set up, as shown in Figure 9-16 on page 9-30. ♦

See the section “Setting Baselines” beginning on page 9-27 for more information on baselines and how to control them.

Baselines for Vertical Text

As noted previously, vertical text in a layout shape is just a special case of horizontal text. There is no vertical baseline and no vertical line direction in QuickDraw GX; a vertical line is calculated and laid out as if it were horizontal.

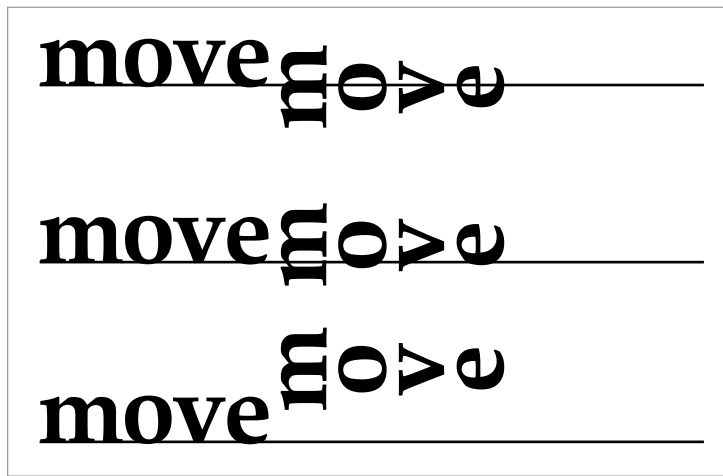
When QuickDraw GX creates text that is to be displayed vertically (meaning the `gxVerticalText` text attribute in its style object is set), it rotates the text’s individual glyphs 90 degrees counterclockwise. It then sets the glyphs on a baseline (which at this point is horizontal). If the glyphs do not have explicitly defined vertical metrics, QuickDraw GX synthesizes a centered vertical baseline and places the glyphs on it. Before drawing the shape, your application is responsible for setting its transform object’s mapping to rotate it 90 degrees (clockwise), thus making it vertical and restoring the glyphs to their proper orientation. You are also responsible for realigning the vertical glyphs, if necessary.

For runs of text that are vertical, baseline alignment takes on a slightly different meaning. Figure 9-5, for example, is a layout shape drawn three times, each consisting

Layout Line Control

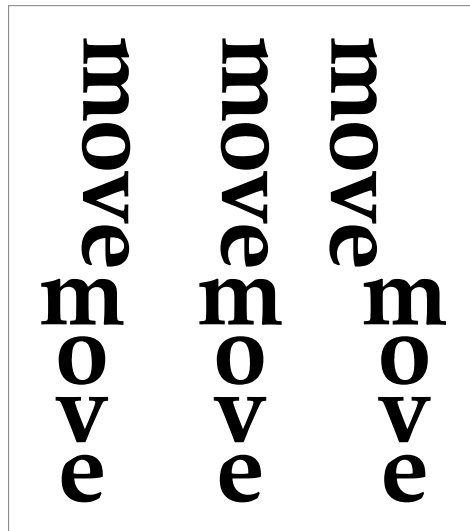
of two runs of text. The second run of text in each shape has the `gxVerticalText` text attribute set. The first line shows the result of setting `baselineType` to `gxRomanBaseline`, the second line shows the result of setting `baselineType` to `gxIdeographicCenterBaseline`, and the third line shows the result of setting `baselineType` to `gxHangingBaseline`. For all three lines, the baseline deltas were derived from the horizontal run of text using the `GXSetStyleBaselineDeltas` call.

Figure 9-5 Creating vertical text in a layout shape



The sample code used to generate Figure 9-5 is Listing 9-2 on page 9-31. If you rotated each shape produced by Listing 9-2 to a vertical position and then redrew it, you would get the text shown in Figure 9-6.

Figure 9-6 Rotating vertical text in a layout shape



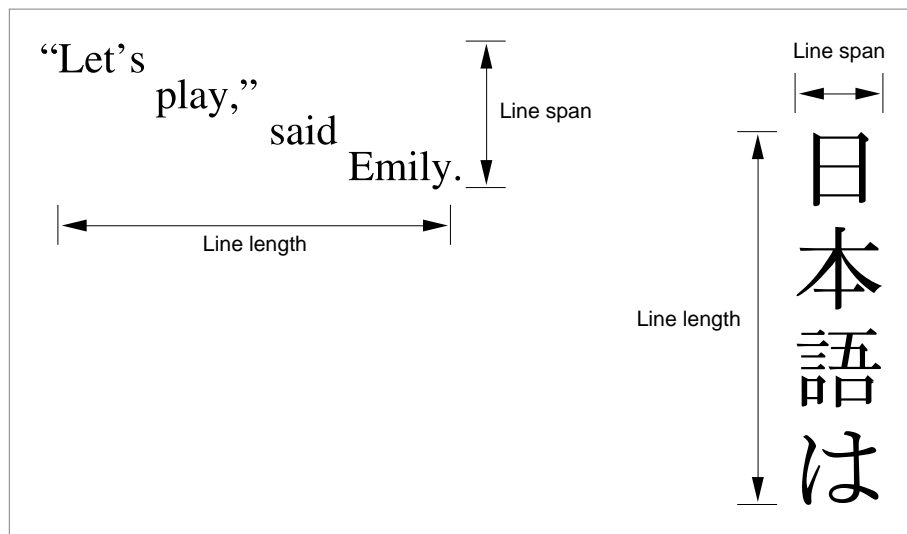
For more information, see “Drawing Vertical Text” beginning on page 9-30 and Listing 9-2 on page 9-31.

Line Measurement

Measuring the dimensions of text lines is a standard process in performing text layout. For QuickDraw GX layout shapes, line measurement can involve complex calculations.

For a horizontal line of text, line length is its width, and line span is its line height. For a vertical line of text, line length is a height measurement, and line span is a width measurement. Figure 9-7 illustrates length and span for both horizontal and vertical lines. Because QuickDraw GX treats vertical text as if it were horizontal, the subsequent examples in this section all assume horizontal lines (length is equivalent to width, and span is equivalent to height).

Figure 9-7 Line length and line span



Line Length

Determining the length of a line of text is fundamental to all layout, and especially for line breaking. You can use the width of the standard bounding rectangle or the typographic bounding rectangle of the layout shape to get two possibly slightly different measures of

Layout Line Control

its overall line length. QuickDraw GX also provides a function that gives you the length of any range of text within a layout shape, up to and including the entire shape.

See the section “Determining Line Lengths” beginning on page 9-32 for more information.

Line Span

Because the typographic capabilities of QuickDraw GX are limited to laying out and drawing single lines of text, any time you work with multiple lines you need to determine how far to separate the lines to avoid overlap, truncation, or excess leading. Whereas in simple text it is relatively easy to calculate line spacing based on a single text-size value, such calculations can be very complex in layout shapes. To space lines properly, you need to account for different text sizes in different style runs and the possibility of cross-stream kerning, cross-stream shifting, and shifted baselines.

You can use the height of the standard bounding rectangle or the typographic bounding rectangle of the layout shape to get two possibly slightly different measures of its line span. QuickDraw GX also provides a function that returns the line span for any layout shape, no matter how complex. QuickDraw GX uses that span when calculating the shape of the caret and the height of highlight areas; you can use it to determine where to place line starts.

QuickDraw GX also provides a function that allows you to set the line span for any layout shape, in case you need to force a specific line spacing or caret height.

See the section “Determining Line Spans” beginning on page 9-33 for more information.

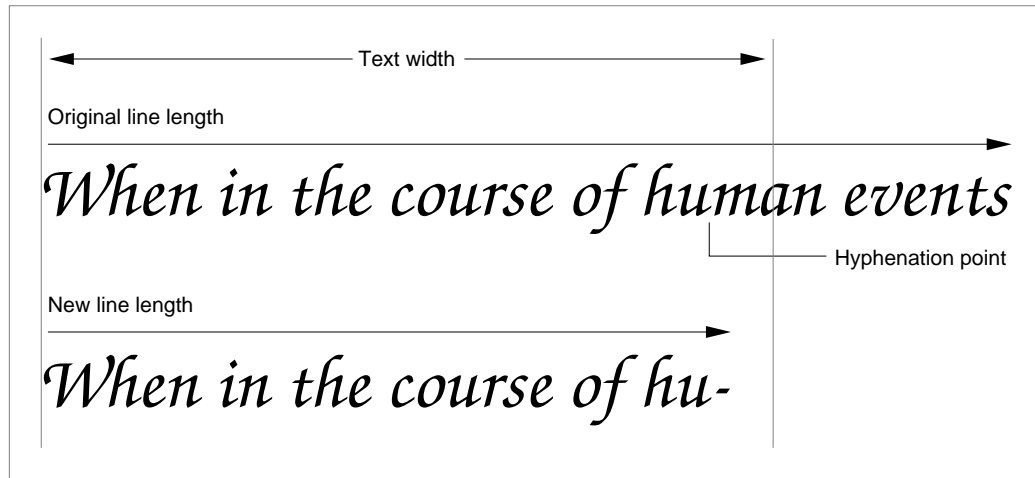
Line Breaking

If you work with text that can wrap to multiple lines, line breaking is a task of fundamental importance. Although QuickDraw GX provides several functions that help you with line breaking, an important point to remember is that *line-breaking decisions are still up to your application*. QuickDraw GX and the fonts it uses have no information about morphology, or the internal structure of words. Your application, perhaps with the help of the Macintosh WorldScript international resources, must decide where to end a line. What QuickDraw GX can do is provide fast ways to determine line length at potential break points, identify break points that are most efficient in terms of line layout, and create new layout shapes for each line.

Layout Line Control

Figure 9-8 shows the basic factors involved in the line-breaking decision. The **text width** is the area between the margins, within which all displayed text must fit. If the line length of your whole layout shape is less than the text width, there is no need to break the line at all. But if it is greater, you need to break the line so that it fits within the text width and also satisfies the requirements of the language.

Figure 9-8 Factors in line breaking



QuickDraw GX allows you to provide a set of **hyphenation points**, edge offsets in the source text at which it is appropriate to break the line. If you do so, QuickDraw GX uses that information in suggesting a line-break position. If you do not provide hyphenation points, QuickDraw GX calculates a break position at the last whole glyph that fits within the text width.

In returning a suggested break point, QuickDraw GX also notifies you of the closest positions to that break point that are staked. A **stake** is an edge offset in the source text at which point, if a decision is made to break the line there, the break will be “clean” in terms of layout processing—that is, not in the middle of a ligature, or inside a kerning pair, or between a pair of rearranged glyphs.

Staked offsets are separate from and largely independent of hyphenation points. The best hyphenation point may be within a ligature (the word “offload” would break within the “fl” ligature), so it may not be a staked position; a staked position (such as between the “l” and “o” in “offload”) may not be an acceptable hyphenation point. QuickDraw GX provides the information on staked positions to help you be most efficient in laying out text during line breaking; it does not take the place of the morphological information you must have to calculate proper hyphenation points.

See the section “Breaking Lines” beginning on page 9-33 for more information on line-breaking and examples of line-breaking algorithms. See the section “Using Macintosh WorldScript for Line Breaking” beginning on page 9-37 for information on using international resources to help with line-breaking decisions.

Text Direction

Displayed text always has a **direction**, which is the direction in which a person's eyes move when reading successive glyphs. Different languages have different directions, and some individual languages support more than one direction. Text of Roman languages typically has a left-to-right direction (although vertical text is used occasionally); text in Chinese and Japanese may have a vertical direction, although left-to-right text is also common (and right to left is possible). Arabic and Hebrew have a predominantly right-to-left direction, although some text in both languages is written left to right.

Vertical text

Throughout this section and in much of the rest of this chapter, the discussion is in terms of horizontal text lines. Because vertical text in QuickDraw GX can be thought of as horizontal text in which the individual glyphs are rotated 90 degrees, you can apply the discussions here to vertical text by substituting "top-to-bottom" for "left-to-right," "width" for "height," and "height" for "width." ♦

Direction is multileveled; horizontal text of one direction may have embedded text of the opposite direction, and the entire sequence may be embedded within a line of text that has either direction. To help sort out the complications, QuickDraw GX defines two direction-related concepts:

- **Glyph direction**, the most fundamental and smallest-scale directionality. It is applied to individual glyphs.
- **Dominant direction**, a broader direction control imposed on groups of glyphs. (The broadest control on text direction is **line direction**, and you can think of it as the dominant direction for an entire line of text.)

This section describes those concepts and shows you how to control them properly when laying out lines of mixed-direction text.

Glyph Direction

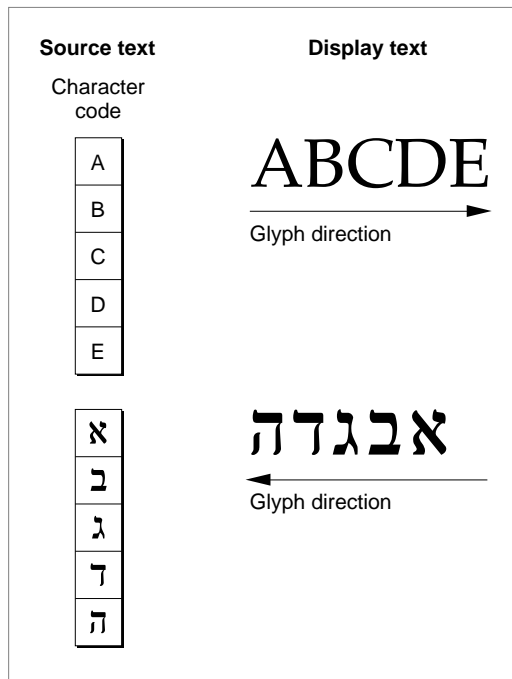
Every glyph in a font has a defined direction that determines how it is positioned in relation to the previous and subsequent (in reading order) glyphs. That direction is implied in the positions of the glyph's leading and trailing edges; for example, if a glyph's leading edge is on its left side, that glyph has a left-to-right direction.

When QuickDraw GX draws a sequence of glyphs, it takes glyph direction into account when calculating the order in which to display them. If a sequence of glyphs all have left-to-right direction, the left-to-right display order in which QuickDraw GX displays them matches the sequential order of their characters in the source text. If the sequence is

Layout Line Control

all right-to-left glyphs, QuickDraw GX displays them in an order that is opposite to the sequential order of their characters in the source text. Figure 9-9 shows examples of both kinds of sequences.

Figure 9-9 How glyph direction affects display order



QuickDraw GX always displays consecutive groups of glyphs that have a common direction in the sequence determined by that direction, regardless of other direction properties that may be imposed on the text. Your application has no control over this process, other than to override it globally for an entire style run. You can impose a strong right-to-left or left-to-right direction on all of the glyphs of a style run by setting a flag value in the run controls structure of the style object. The run controls structure is described in the chapter “Layout Styles” in this book.

Types of Glyph Direction

Because of the complications that arise in languages that support two directions and in single lines with text of different languages and directions, QuickDraw GX recognizes several types of directionality for glyphs. Most glyphs have a **strong type** of direction, meaning that they are always read only in the direction defined for them. Glyphs with strong left-to-right direction include most alphabetic, syllabic, and Han ideographic glyphs for most languages. Glyphs with strong right-to-left direction include Arabic and Hebrew alphabetic glyphs and punctuation.

Layout Line Control

Glyphs of numbers and their associated symbols are considered to have a **weak type** of direction. They typically are read left-to-right but are placed on the line as if they had the direction of the adjacent glyphs. (See the next section, “Dominant Direction” beginning on page 9-15, for a discussion of how glyph direction affects placement of blocks of glyphs on the line.)

Glyphs with weak direction include European and Arabic numbers, European number separators (such as the figure space, the period, and the slash), European number terminators (such as the plus and minus signs, the percent sign, and currency symbols), and the common number separators (the colon and the comma).

Neutral glyphs are glyphs without inherent direction. They generally take on the direction of the surrounding text. Neutral glyphs include block separators (such as the paragraph separator or line separator) and whitespace glyphs.

Normally, your application need not be concerned with the details of the type or class of direction associated with a particular glyph, although QuickDraw GX uses it in determining the reordering necessary to lay out a line of mixed-direction text. If you need to, you can override direction by setting a value in the run controls structure of the style object. Note, however, that the override applies to an entire style run, and that you can override into one of the strong types only. The run controls structure is described in the chapter “Layout Styles” in this book. For more information on direction classes and how they affect text layout, see *The Unicode Standard: Worldwide Character Encoding, Version 1.0*, Volume 1.

Dominant Direction

The **dominant direction** of a line (or any other text run) is the overall, controlling direction within which the individual glyph directions are set. Dominant direction does not reverse glyph direction; it is a higher-level effect. If a Hebrew word is embedded in a line of Roman text, the dominant direction for that line is left to right, but the Hebrew word is still laid out right to left, as expected. Conversely, Roman text embedded in a line of Hebrew, in which the dominant direction is right to left, is still displayed left to right.

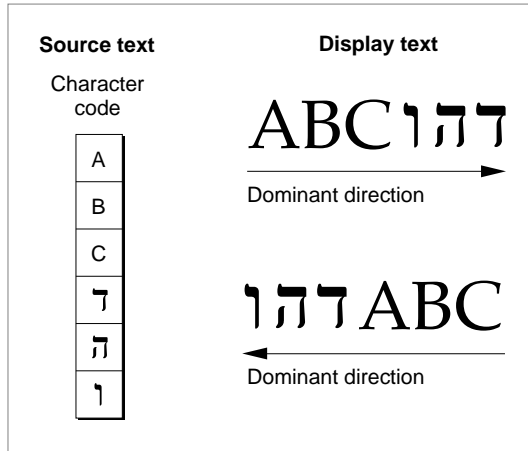
For this reason, dominant direction has significance only in mixed-direction text. If all the text on a line has a uniform glyph direction, such as the text shown in Figure 9-9 on page 9-14, its display order is unchanged no matter what the dominant direction is. Changing the dominant direction has no effect on the display of either line of text in Figure 9-9.

In mixed-direction text, however, changing the dominant direction has a significant effect. Figure 9-10 shows a layout shape whose source text consists of three Roman characters followed by three Hebrew characters. In the upper display line, the dominant direction is specified as left to right, implying that the Hebrew letters are embedded in a line of Roman text. The Hebrew letters are laid out right to left, but the line as a whole (the groups of letters of a given direction) is laid out left to right.

Layout Line Control

The lower display line in Figure 9-10 shows the same text when the dominant direction is specified as right to left, implying that the Roman letters are embedded in a line of Hebrew text. The Roman letters are laid out left to right, but the line as a whole (the groups of letters of a given direction) is laid out right to left.

Figure 9-10 How dominant direction affects display order



The QuickDraw GX text layout model accounts for dominant direction as well as glyph direction, automatically performing any reordering needed for correct display of simple mixed-direction lines of text such as those shown in Figure 9-10. Furthermore, in QuickDraw GX the concept of dominant direction is not limited to line direction. Any section of text in a layout shape can have a dominant direction, and dominant directions exist in a nested hierarchy in which the line direction is simply the lowest nesting level.

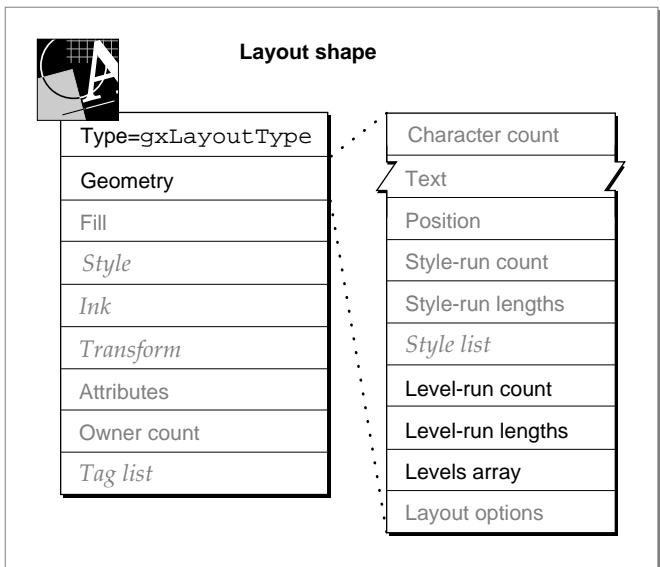
This nesting of dominant directions allows you to preserve very complex multiple-language formatting. You specify the nested hierarchy of direction levels in the levels array of the layout shape. The levels array is introduced in the chapter “Layout Shapes” in this book. The next two sections show how to use the levels array to perform complex multilanguage formatting.

The Levels Array of the Layout Shape Object

Figure 9-11 shows the geometry of the layout shape. Part of the geometry is the levels array, represented by the properties level run count, level lengths array, and levels array.

It is in the levels array that you specify the dominant direction for the line of text in a layout shape. It is also in the levels array that you can specify additional nested levels of direction.

Figure 9-11 The levels array property of the layout shape

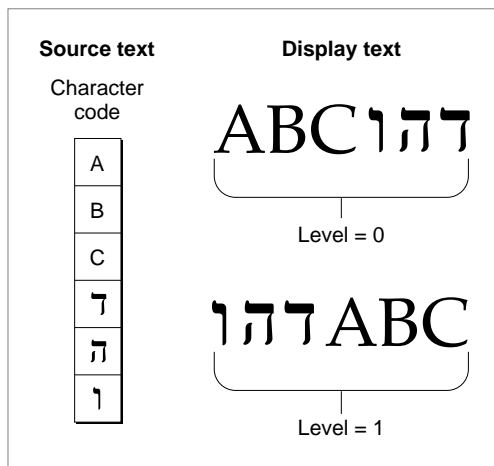


Layout Line Control

Figure 9-12 demonstrates the simplest relationship between nesting level and text direction. It shows the same layout shape as displayed in Figure 9-10 on page 9-16.

- If you specify a single direction-level run with a level of 0 for your entire layout shape (or if your levels array is `nil`, the default), you are specifying a left-to-right dominant direction for the line. The line is reordered and displayed as shown in the upper line of Figure 9-12 (identical to the upper line of Figure 9-10).
- If you specify a single direction-level run with a level of 1 for your entire layout shape, you are specifying a right-to-left dominant direction for the line. The line is reordered and displayed as shown in the lower line of Figure 9-12 (identical to the lower line of Figure 9-10).

Figure 9-12 How nesting level relates to text direction



In fact, if you specify any even number for a level, QuickDraw GX considers text of that level to have a dominant direction of left to right. Likewise, if you specify any odd number for a level, QuickDraw GX considers text of that level to have a dominant direction of right to left.

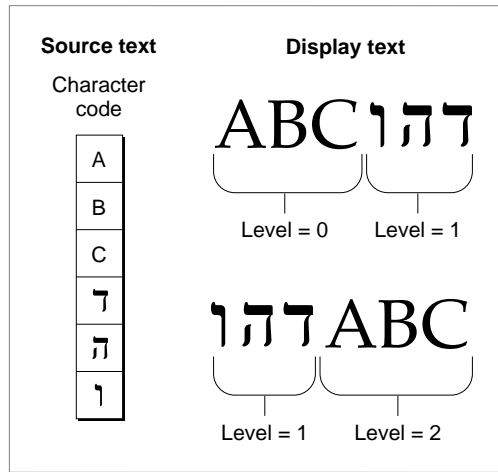
Note that a change in level need not accompany each change in glyph direction. QuickDraw GX automatically reorders glyphs appropriately according to their glyph directions.

Where a line contains more than one level of text, QuickDraw GX looks at the *lowest* level to determine the dominant direction for the line. Figure 9-13 shows alternative ways to specify the direction information shown in Figure 9-12 for the layout shape:

- In the upper line of Figure 9-13, each run of text in a given direction has its own level. The formatting of the display is identical to the upper line of Figure 9-12 because QuickDraw GX notes that the lowest level on the line is an even number and therefore orders the line from left to right.

- In the lower line of Figure 9-13, the runs of text are the same, except the Roman text has a level of 2, not 0. In this case, the lowest level on the line is an odd number, and QuickDraw GX therefore orders the line from right to left. The formatting of the display is identical to the lower line of Figure 9-12.

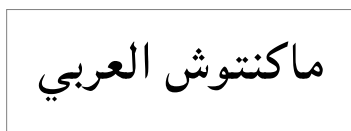
Figure 9-13 Multiple nesting direction levels in one line



The use of levels other than 0 and 1 is unnecessary for this simple example, because you can specify a single level for the entire line. In most situations, even with mixed-direction text, you never need to use more than a single level run (with value 0 or 1) for your layout shape. The next section, however, shows how you can use multiple nested direction levels to prevent incorrect reordering in more complex situations.

Forced Reordering With Nested Direction Levels

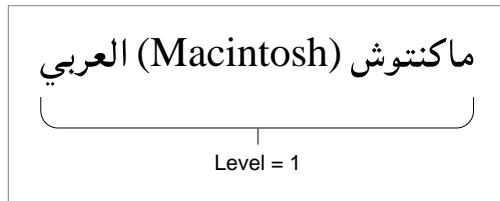
This section shows one example of when the explicit use of nesting levels may be necessary. Suppose, for example, that the following is a layout shape consisting of the Arabic-language equivalent of the phrase “Arabic Macintosh”:



In this case, all glyphs have a right-to-left direction, and specification of nesting levels (even for the dominant direction for the line) is unnecessary.

Layout Line Control

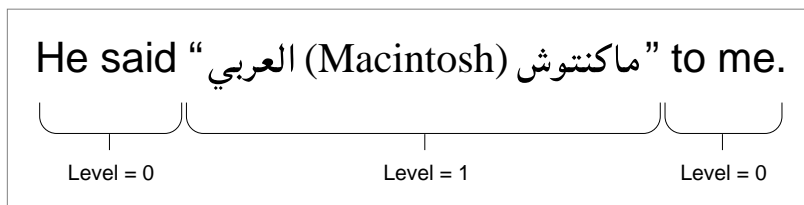
Now, however, assume that the English-language phrase “(Macintosh)” is added between the two Arabic words. To make sure that the line is ordered correctly when drawn, you must specify a right-to-left dominant direction for the line. You can do that by giving the entire layout a level of 1 (odd). The resulting layout is drawn correctly, as follows:



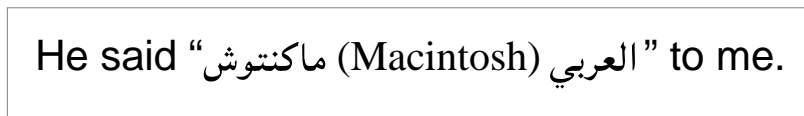
If you had not specified any level for the line, or if you had specified an even value, the two Arabic words would have been reversed—equivalent to saying “Macintosh Arabic” in English. The resulting layout would be drawn incorrectly, like this:



Finally, suppose that this predominantly Arabic phrase is made part of an English-language sentence. To preserve the overall right-to-left ordering of the phrase, you must give it an odd nesting level; to preserve the overall left-to-right ordering of the sentence itself, you must give the English parts a lower, even nesting level. The resulting layout is drawn correctly, as follows:



If you had simply given the entire sentence an even level (or specified no level at all), the Arabic words would once again have been reversed, and the sentence would have been incorrectly displayed, as follows:



Consider an even more complex example. Suppose you subsequently embedded the entire sentence in a line of Arabic text. You would then need to assign a level of 1 to those new (outer) Arabic parts and promote the levels 0 and 1 in the original sentence to levels 2 and 3. This is because QuickDraw GX looks at the lowest level to determine the overall line direction, which in this case would need be right to left (odd nesting level).

Your application can usually determine the intended dominant direction for a line when text is being entered, either from a system setting or from a user selection. However, it may not be able to generate these more complex nesting levels automatically without

user intervention. Therefore, to give users control over the orderings of complex phrases, you may want to allow them to set levels explicitly as they enter text, or to impose levels on the text they have previously created.

For more information and examples, see the section “Manipulating Nested Direction Levels” beginning on page 9-38.

Justification

Justification is the process of typographically fitting a line of text to a given width (or height, in the case of vertical text). In QuickDraw GX, some of the information that controls justification behavior is contained in the layout shape itself, whereas other information is contained in the style objects associated with the layout shape.

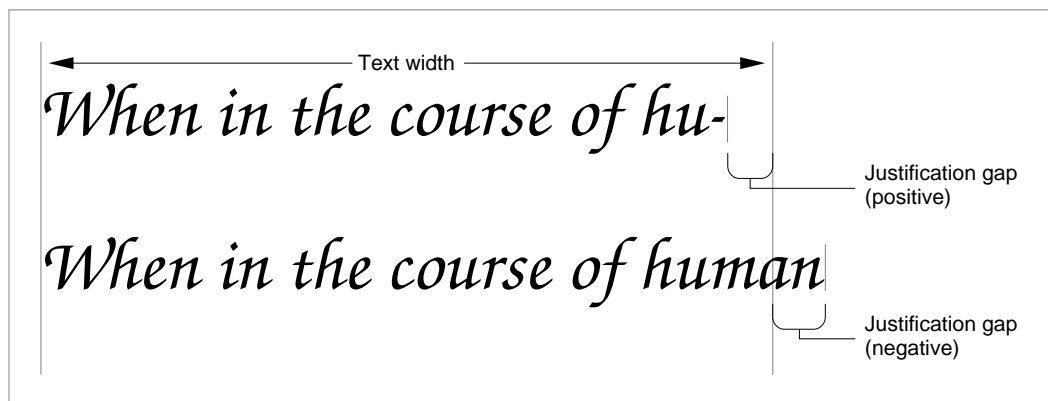
This section presents the QuickDraw GX justification model, notes where the information that controls it is stored, and discusses how to override aspects of it to produce special justification effects.

The Justification Model

The QuickDraw GX justification model is very powerful and completely multilingual. It supports the assignment of additional space to different classes of glyphs at different priority levels, and includes facilities for handling complex kashida-like justification such as is used in Arabic or script Roman. This section describes how the model works.

The **justification gap** is the difference in the length of a line of text before and after justification. For full justification, it is the difference between the text width and the length of the unjustified line (without considering hanging punctuation and optical effects, described in the chapter “Layout Styles” in this book). If the line must be stretched to fit the allotted space, the justification gap is positive and must be distributed among the glyphs of the line as extra width. If the line must be shrunk to fit, the justification gap is negative and that space must therefore be removed from the glyphs of the line. (See Figure 9-14.) QuickDraw GX can handle both positive and negative justification gaps and can even use different behaviors in the positive and negative cases.

Figure 9-14 Justification gap



Layout Line Control

Remember that, as noted in the chapter “Layout Shapes” in this book, justification is a continuous value that is specified in the `just` field of the layout options structure. Figure 9-14 illustrates the justification gap for a line that is to be fully justified (`just = 1.0`); the justification gap is the entire difference between line length and text width. If the line were to be only partially justified, the justification gap would be proportionally smaller. For example, if the line in Figure 9-14 were to be 50 percent justified (`just = 0.5`), the justification gap would be half the amount shown in the figure. QuickDraw GX supports partial or full justification, of both positive and negative justification gap, in all situations.

Justification Priority, Grow Limits, and Shrink Limits

Justification as performed by QuickDraw GX is a multistage process. QuickDraw GX does not have to, for example, assign intercharacter and interword white space in a fixed proportion when stretching a line. Instead, QuickDraw GX recalculates glyph positions in several passes, based on **justification priority**. Glyphs with higher priority are processed earlier. If, after processing all glyphs of a given priority, more justification gap remains, QuickDraw GX then processes glyphs of the next lower priority, and so on, until all the needed justification gap is taken up. The defined justification priorities are listed in Table 9-1.

Table 9-1 Justification priorities

Constant	Value	Explanation
<code>gxKashidaPriority</code>	0	The highest priority. Glyphs with this priority are adjusted first.
<code>gxWhiteSpacePriority</code>	1	Glyphs with this priority are adjusted after all glyphs with priority <code>gxKashidaPriority</code> .
<code>gxInterCharPriority</code>	2	Glyphs with this priority are adjusted after all glyphs with higher priority.
<code>gxNullJustificationPriority</code>	3	Glyphs with this priority have the lowest justification priority and are adjusted last.

Note

The justification priorities specify only the order in which glyphs participate in justification. The names of the priorities suggest the kinds of effects that are typical at each level, but the actual justification technique that QuickDraw GX applies—for example, addition of kashida extenders or white space—is defined for each glyph by the font. ♦

As an example of the steps involved in justification, in Roman fonts the whitespace glyphs typically have a higher justification priority than other glyphs. During justification, QuickDraw GX first assigns extra (interword) space to those glyphs alone, until

either the entire justification gap is accounted for or a specified **grow limit** for the justification of whitespace glyphs is reached. At that point, if more gap remains, QuickDraw GX then starts assigning extra (intercharacter) space to other glyphs. For this reason, intercharacter spacing need not occur as often as it does in proportional justification models (or even at all, if your application wishes).

Tables in the glyphs' font determine the assignment of justification priorities to glyphs and the specification of limits to the amount of width that can be added for each priority. Each glyph of a given priority can be stretched by a given amount on its right side and by a possibly different amount on its left side, after which processing passes to the next lower priority of glyph. (Note that "stretching" in this case can mean addition of white space, addition of connecting glyphs, such as kashidas, actual stretching of the glyph form itself, or other methods of taking up the space.) Likewise, each glyph of a given priority has **shrink limits**, used when the justification gap is negative. Shrink limits can be different on the left and right sides of a glyph and different from the grow limits.

When it assigns a certain amount of space to a glyph (or subtracts space), QuickDraw GX looks at the glyph's left and right grow limits (or shrink limits) and adds the space (or subtracts it) on each side, until those limits are reached.

Sometimes a justification gap remains even after QuickDraw GX has processed all priorities of glyphs on the line and has stretched each by the maximum amount. In this case, QuickDraw GX goes back to the highest priority glyphs on the line and distributes the remaining gap among them.

A font can specify that certain glyphs have **unlimited gap absorption**, meaning that, at the point during justification at which they are processed, all remaining justification gap is assigned to them.

The font also specifies, for each glyph, the actual technique of justification that must be applied to it. For example, most glyphs in most Roman fonts grow by the addition of white space to both sides of the glyph. Certain Arabic glyphs grow by the addition of kashidas (extender bars) to their right sides only. Other glyphs in some fonts grow by changing their shapes—for example, a hyphen might grow or shrink when the line it is part of is justified. For examples of each of these kinds of justification, see the section "Displaying Partial Justification" beginning on page 9-46.

Justification priorities are described further on page 9-60.

Postcompensation Action

Once QuickDraw GX has completed its processing and calculated the extra space to add to all glyphs, it may either draw the line as it is, or it may perform postcompensation action on it. **Postcompensation action** consists of addition, substitution, or modification of glyphs as needed to complete the justification. In many cases, postcompensation action is not needed; the existing glyphs are drawn in their revised positions, surrounded by whitespace. However, there are several cases in which it is either necessary or useful:

- **Addition of kashidas.** Justification in Arabic involves adding extension bars to the right sides of certain glyphs. The justification calculations create the proper amount of space to hold those extension bars; postcompensation action inserts the glyphs for them. A similar postcompensation process adds connectors between glyphs for justified Roman text using cursive fonts.

Layout Line Control

- **Changing glyph shape.** Certain glyphs in some fonts contribute to the justification of a line by actually deforming, rather than by having white space added on either side. In that case, postcompensation action consists of deforming the glyph to fill its new space. Deforming may be either by simple **glyph stretching**, which uses a text face mechanism, or by **glyph ductility**, which uses a font variation mechanism. Text faces are described in the chapter “Typographic Styles” in this book; font variations are described in the chapter “Font Objects” in this book.
- **Ligature decomposition.** Depending on the amount of white space surrounding a ligature, postcompensation action may replace that ligature with its component glyphs, after which QuickDraw GX recalculates the positions of all glyphs on the line.
- **Substitution of wider glyphs.** Some fonts have wider versions of certain glyphs, such as hyphens, that postcompensation action can substitute for the original glyphs.

Overriding Justification Behavior

If you want to provide custom justification behavior, your application can override any of the font-specified priorities and limits in a given style run, either for an individual glyph or for a whole priority of glyphs. Additionally, you can specify that a given glyph or a given priority of glyph is to have unlimited gap absorption; in other words, when that glyph or priority is processed during justification, all remaining justification gap must be assigned to it, regardless of its specified shrink or grow limits.


Your application cannot in general override the kind of justification that takes place for a given glyph—addition of white space, addition of a kashida, or stretching, or ductility. You can, however, prevent postcompensation action by setting the `gxNoSpecialJustification` flag in the run controls structure of the style run to which the text belongs. You can also override the threshold at which ligature decomposition starts by placing an appropriate value in the `decompositionAdjustmentFactor` field of the run controls structure of the style run to which the text belongs. The run controls structure is described in the chapter “Layout Styles” in this book.

Justification Properties of the Shape Object and Style Object

Some of the information that controls justification behavior in QuickDraw GX is contained in the layout shape itself, whereas other information is contained in the style objects associated with that layout shape.

The `just` field of the layout options structure in the geometry of a layout shape contains a `fract` value between 0 and 1.0 that defines whether and to what extent a layout shape is to be justified. A value of 0 means no justification; a value of 1 means full justification; values in between specify varying degrees of justification. The layout options structure is described in the chapter “Layout Shapes” in this book.

Figure 9-15 shows the justification-related properties of the style object. Because each style run in a layout shape has its own style object, these properties, unlike the `just` value in the layout options structure, need not affect the entire layout shape.

Figure 9-15 Justification-related properties of the style object


Style object		
Pen width	Font	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		
Tag list		

There are three principal layout-specific properties of the style object that control justification behavior. Two of them are described in this chapter; both function as overrides to font-specified default behavior:

- **Priority justification override structure.** Each entry in this structure overrides the behavior of all glyphs of a given justification priority. If this property is set to `nil`, justification for each priority for this style run defaults to the font-specified behavior. See the next section, “Priority Justification Override,” for more information on this structure.
- **Glyph justification overrides array.** This array overrides the justification behavior of one or more individual glyphs. If this property is set to `nil`, justification for all glyphs in the style run defaults to the font-specified behavior. See the section “Glyph Justification Overrides” on page 9-26 for more information on this array.

Justification overrides basically have two effects: (1) changing the limits to which a given glyph or class of glyphs can be stretched (or shrunk) during justification, and (2) changing the justification priority of the glyph or class of glyphs. Overrides do not change the kind of justification that is applied—white space, kashida, glyph stretching, and so on.

One other layout-specific property of the style object, the run controls structure, contains one field and one flag that affect justification behavior. The field is the `decompositionAdjustmentFactor` field, and the flag is the `gxNoSpecialJustification` flag. Both affect postcompensation action, described on page 9-23. The run controls structure itself is described in the chapter “Layout Styles” in this book.

Priority Justification Override

The priority justification override structure specifies overriding justification behavior for specific classes of glyphs in a given style run. The structure is organized by priority; for each justification priority, it specifies the overriding behavior, if any. That behavior can include changes to the grow or shrink limits for that priority and even a change in priority value for that priority. The structure is a simple array of width delta structures, one for each defined justification priority.

```
typedef struct {
    gxWidthDeltaRecord    deltas[gxNumberOfJustificationPriorities];
} gxPriorityJustificationOverride;
```

Each width delta structure specifies, for both the grow and shrink cases, limits to the amount of space that can be added (or removed) from both the right and left sides of each of the glyphs of the given justification priority. This is its format:

```
typedef struct {
    Fixed                beforeGrowLimit;
    Fixed                beforeShrinkLimit;
    Fixed                afterGrowLimit;
    Fixed                afterShrinkLimit;
    gxJustificationFlags growFlags;
    gxJustificationFlags shrinkFlags;
} gxWidthDeltaRecord;
```

The `growFlags` and `shrinkFlags` fields control whether or not to apply the limits defined in the rest of the structure and whether or not to change other justification behavior, such as the priority itself. The flags also control whether or not unlimited gap absorption (see page 9-24) should be applied to the priority of glyphs specified in the structure. The fields of the width delta structure are described in more detail in the section “Width Delta Structure” beginning on page 9-61.

The priority justification override structure is described in more detail in the section “Priority Justification Override Structure” beginning on page 9-63.

Glyph Justification Overrides

The glyph justification overrides array specifies overriding justification behavior for individual glyphs in a given style run. It consists of an array of glyph justification override structures, one for each glyph whose behavior is to be overridden.

The glyph justification override structure assigns an overriding justification priority and behavior to a specific glyph in a style run. It contains a glyph code and a width delta structure.

```
typedef struct {
    gxGlyphcode          glyph;
    gxWidthDeltaRecord    override;
} gxGlyphJustificationOverride;
```


The width delta structure in this case specifies overrides and flags for all instances of a single glyph (specified by glyph code). The fields of the width delta structure are described in more detail in the section “Width Delta Structure” beginning on page 9-61.

The glyph justification override structure is described in more detail in the section “Glyph Justification Override Structure” beginning on page 9-64.

Using Line Control and Line Measurement With Layout Shapes

This section shows how to use QuickDraw GX functions and structures to measure and control the layout of text lines. In particular, it shows you how to

- set multiple baselines
- determine line lengths
- determine line spans
- break lines
- manipulate nested direction levels
- display continuous justification
- change the behavior of justification priorities
- change the justification behavior of glyphs

Setting Baselines

The section “Baselines” beginning on page 9-4 describes how QuickDraw GX uses font information to lay out multiple-baseline text. To take advantage of the capabilities of QuickDraw GX and thus get the best alignment when drawing a line of text that uses multiple baselines, you can use the following procedure:

1. Decide which baseline type to align the text of each style run to. For example, you may specify a Roman baseline for a Roman style run and a hanging baseline for a Devanagari style run on the same line. Set each baseline type in the `BaselineType` field of the run controls structure of the style object for that style run. (Actually, you need make no explicit assignments if you use the default baseline for the text of each style run.)
2. Determine the distances among baselines to be used for the entire line. Call the `GXGetStyleBaselineDeltas` function to get the distances (based on the font and text size of a style object, which may represent one of the style runs on the line). Store the baseline deltas information in the `baselineRec` field of the layout options structure.

When you draw the layout shape, QuickDraw GX aligns all baselines in all style runs according to the information in the baseline deltas structure.

Layout Line Control

Listing 9-1 is a partial listing of a function that aligns the hanging baselines of two different sizes of text to create drop capitals, for the string “Drop Caps”. The function creates two different style objects, one for the drop capitals and one for the lowercase letters. The function draws the line of text twice: once with the capitals at 70 points and once with capitals at 110 points (the body text is at 40 points in both cases).

Listing 9-1 makes use of application-defined functions `NewLayoutStyle`, `InitializeRunControls`, `InitializeLayoutOptions`, and `GXSetStyleTextSize` to initialize some of the objects and structures used in the listing. The length of the text string is `len`; the layout is drawn at the point `myPoint`. The function calls the QuickDraw GX function `GXGetStyleBaselineDeltas` to get the information needed for aligning the baselines.

Listing 9-1 Aligning baselines to create drop capitals

```
void BaselineAlignment(WindowPtr sampleWindow)
{
    /* define and initialize variables */
    char                *myString = "Drop Caps";
    gxLayoutOptions      layoutOptions;
    gxLineBaselineRecord lineBaselineRecord;
    gxRunControls        runControls;
    gxShape              layout;
    short               runLengths[4];
    gxStyle              dropCapsStyle, regularStyle, styleArray[4];
    .
    .
    .
    /* set the size of the text and each of the style runs */
    runLengths[0] = runLengths[2] = 1;
    runLengths[1] = 4;
    runLengths[3] = 3;

    /* style for lowercase letters is 40-pt, no run controls */
    regularStyle = NewLayoutStyle((char *) "\pTimes Roman",
                                ff(40), 0, nil, nil, 0, nil);

    /* set up hanging-baseline run controls for drop-cap style */
    InitializeRunControls(&runControls);
    runControls.baselineType = gxHangingBaseline;
```

Layout Line Control

```

/* style for drop caps is 70-pt, with run controls */
dropCapsStyle = NewLayoutStyle((char *) "\pTimes Roman",
                                ff(70), 0, &runControls, nil, 0, nil);

/*
   Set up layout options structure for the layout shape; get
   baseline distances from Roman, based on lowercase style.
*/
InitializeLayoutOptions(&layoutOptions);
GXGetStyleBaselineDeltas(regularStyle, gxRomanBaseline,
                        lineBaselineRecord.deltas);
layoutOptions.baselineRec = &lineBaselineRecord;

/* assign styles to each style run */
styleArray[0] = styleArray[2] = dropCapsStyle;
styleArray[1] = styleArray[3] = regularStyle;

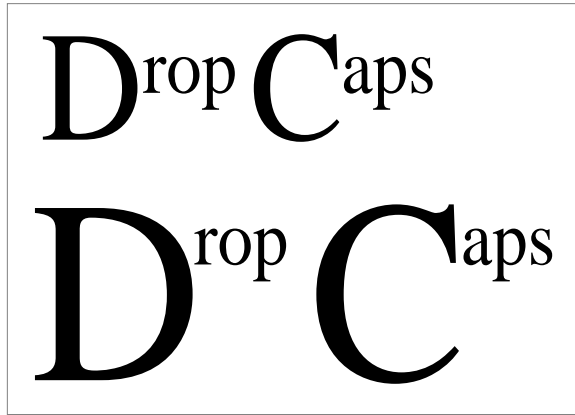
/* create and draw a layout with the above text and styles */
layout = GXNewLayout(1, &len, (void *) &myString,
                    4, runLengths, styleArray,
                    0, nil, nil,
                    &layoutOptions, &myPoint);
GXDrawShape(layout);

/* now modify the size of the capitals, but nothing else */
GXSetStyleTextSize(dropCapsStyle, ff(110));

/*move the shape and draw again; the drop caps still line up */
GXMoveShape(layout, 0, ff(100));
GXDrawShape(layout);
.
.
.
}

```

Figure 9-16 shows the results of executing the function in Listing 9-1.

Figure 9-16 Drop capitals created by aligning baselines

The `GXGetStyleBaselineDeltas` function is described on page 9-66.

Drawing Vertical Text

Because there are no vertical baselines and no vertical line direction in QuickDraw GX, you create a vertical line by using the layout shape's transform object to rotate a horizontal line before drawing it.

To draw a layout shape as a line of vertical text, follow these steps:

1. Set the `gxVerticalText` text attribute for all style runs of the layout shape. Setting this text attribute has the effect of rotating each individual glyph by 90 degrees counterclockwise.
2. Lay out and measure the line as if it were horizontal. Caret positions, hit-testing, and measurements of line span and line length will be meaningful if you consider the line as horizontal, with rotated glyphs.
3. Call the `GXRotateShape` function or `GXRotateTransform` function to rotate the line 90 degrees clockwise when it is drawn. Because, for layout shapes, the `gxMapTransformShape` attribute is set, calling `GXRotateShape` does not affect the geometry of the layout itself; it changes only the mapping of the layout shape's transform object.
4. Draw the shape.

Listing 9-2 is a sample program that creates a layout shape several times, drawing it with and without the `gxVerticalText` text attribute set and using several baseline types.

Listing 9-2 Creating and drawing vertical text

```

char *myString = "movemove"
InitializeRunControls(&controls);
controls.baselineType = gxRomanBaseline;

myStyles[0] = NewLayoutStyle((char *) "\pHoefler Text", ff(48),
0,
    nil, nil, 0, nil);
myStyles[1] = NewLayoutStyle((char *) "\pHoefler Text", ff(48),
    gxVerticalText, &controls, nil, 0, nil);
GXGetStyleBaselineDeltas(myStyles[0], gxRomanBaseline,
    lbr.deltas);
options.baselineRec = &lbr;
myLens[0] = myLens[1] = 4;

layout = GXNewLayout(
    1, &len, (void *) &myString,
    2, myLens, myStyles,
    0, nil, nil,
    &options, &myPoint);
GXDrawShape(layout);

controls.baselineType = gxIdeographicCenterBaseline;
GXSetStyleRunControls(myStyles[1], &controls);
GXMoveShape(layout, 0, ff(80));
GXDrawShape(layout);

controls.baselineType = gxHangingBaseline;
GXSetStyleRunControls(myStyles[1], &controls);
GXMoveShape(layout, 0, ff(80));
GXDrawShape(layout);

```

For the results of Listing 9-2, see Figure 9-5 on page 9-9.

Layout Line Control

Some Asian languages use rotated Roman glyphs in the vertical text lines, as shown in Figure 9-17. To create such an effect, follow the above steps but put the Roman text in a separate style run and do not set the `gxVerticalText` text attribute for that style run. When you rotate and draw the shape, the Roman glyphs will then be rotated 90 degrees clockwise, as desired.

Figure 9-17 Rotated Roman glyphs in vertical text



Determining Line Lengths

You can determine the length of a line in a layout shape in at least three ways:

- Call the `GXGetShapeBounds` function to determine the standard bounding rectangle of the layout shape. This rectangle exactly encloses just the “inked” parts (the black pixels) of the displayed glyphs. The width of the rectangle is the length of the line, including any hanging punctuation and accounting for shifts due to optical alignment.
- Call the `GXGetShapeTypographicBounds` function to determine the typographic bounding rectangle of the layout shape. This rectangle spans the layout shape from the lowest descent line to the highest ascent line, regardless of whether any glyphs extend to those lines. The width of the rectangle extends from the origin of the first glyph through the advance width of the last glyph.
- Call the `GXGetLayoutRangeWidth` function, passing it a range that is the entire layout shape. The width it returns is equivalent to the width of the typographic bounds: it extends from the origin of the first glyph through the advance width of the last glyph.

The `GXGetLayoutRangeWidth` function is described on page 9-71. The `GXGetShapeBounds` function is described in the “Geometric Operations” chapter in *Inside Macintosh: QuickDraw GX Graphics*. The `GXGetShapeTypographicBounds` function is described in the chapter “Typographic Shapes” in this book.

Determining Line Spans

For drawing carets and highlighting areas, and for hit-testing, QuickDraw GX automatically calculates the proper line span (distance from the lowest descender to the highest ascender on the line) for a layout shape. Line span affects the height of caret shapes and highlight areas that QuickDraw GX calculates and returns to you, through functions such as `GXGetLayoutCaret` and `GXGetLayoutHighlight`. It also affects the area sensitive to hits for purposes of hit-testing.

QuickDraw GX provides the `GXGetLayoutSpan` function so that you can determine how far apart to space the text lines your application draws. (Line span plus any leading, or extra gap, that you add equals the line-to-line distance in multiline text.) This function returns the correct span for any line that is a layout shape. QuickDraw GX calculates a line height for the given line according to all the information it has, including all text sizes, manual and automatic position shifts, multiple baselines, and so on.

You can also use the functions `GXGetShapeBounds` and `GXGetShapeTypographicBounds` to determine line span. The height of the rectangle returned by `GXGetShapeTypographicBounds` is equivalent to the values returned by `GXGetLayoutSpan`; the value returned by `GXGetShapeBounds` may be smaller, since the bounding rectangle encloses only the black pixels of the displayed glyphs.

If your application wishes to set the line span manually and thus affect these results, use the `GXSetLayoutSpan`. Note that if you alter the line span with `GXSetLayoutSpan`, `GXGetLayoutSpan` returns the line span that you have set. If you need to recover the line span as originally calculated by QuickDraw GX, you can call `GXSetLayoutSpan` with a line span of 0.

For an example of the use of the `GXGetLayoutSpan` function for positioning line starts, see Listing 9-3 on page 9-34.

The `GXGetLayoutSpan` function is described on page 9-67. The `GXSetLayoutSpan` function is described on page 9-68. The `GXGetShapeBounds` function is described in the chapter “Geometric Operations” in *Inside Macintosh: QuickDraw GX Graphics*. The `GXGetShapeTypographicBounds` function is described in the chapter “Typographic Shapes” in this book.

Breaking Lines

QuickDraw GX text formatting is line-based, not paragraph-based. To lay out multiple lines of text, you need to determine for each line where to break the text, and then create a layout shape for that line.

The QuickDraw GX functions most used for line breaking are `GXGetLayoutBreakOffset`, `GXGetLayoutRangeWidth`, and `GXNewLayoutFromRange`. The `GXGetLayoutBreakOffset` function tells you at which point you can break a line if you want it to fit within a given text length. The `GXGetLayoutRangeWidth` function gives you the length of a range of text within a layout shape. `GXNewLayoutFromRange` function creates a new layout shape from a range of text within an existing layout shape.

Layout Line Control

The typical procedure to follow for each line to be laid out is this:

1. Call `GXGetLayoutBreakOffset`, starting with the offset in the source text that starts the line, to see how much will fit between the margins you specify.
2. Use the results of `GXGetLayoutBreakOffset` to determine the exact point in the source text at which to break the line. You can use the results of the function directly, or you can do additional forward or backward processing in the text to determine the most meaningful break point. See, for example “Using Macintosh WorldScript for Line Breaking” on page 9-37.
3. Optionally, check on your results by calling `GXGetLayoutRangeWidth` to compare the display width of the new range of text with you margins. You can account for extra characters such as hyphens when you make this calculation.
4. Call `GXNewLayoutFromRange` to create a new layout shape that represents the line. The new shape can have extra display glyphs, such as hyphens, that are not part of the source text.

Listing 9-3 is a library function (`NewStyledParagraph`) that creates a set of individual layout shapes, each representing a single line, from a larger layout shape that represents an entire paragraph. It calls `GXGetLayoutBreakOffset` to get initial line breaks and then does some simple processing on the results to determine the actual breaks. Next it calls `GXNewLayoutFromRange` to create a new layout shape for each line. When drawing the lines, it calls `GXGetLayoutSpan` to determine how far apart to separate the lines. (Leading between the lines is specified by the `extraLineGap` variable.)

Listing 9-3 uses several other library-defined functions for moving back and forth in the source text, such as `GetTextPiecePtr`, `GetPreviousOffset`, and `GetNextOffset`. It also constructs a library-defined data structure called a `ParagraphRecord` that holds information about the paragraph, such as the number of lines and a reference to each layout shape making up the paragraph. Some variables used here, such as `lineStartsCount`, are global to this function.

Listing 9-3 Breaking a Roman layout shape into individual lines of a paragraph

```
ParagraphRecordHandle NewStyledParagraph(
    long textRunCount,
    const void *text[],
    const short textRunLengths[],
    long styleRunCount,
    const gxStyle styles[],
    const short styleRunLengths[],
    long levelRunCount,
    const short levels[],
    const short levelRunLengths[],
    long totalByteCount,
    const gxLayoutOptions *layoutOptions,
    Fixed lineHeight,
    const gxPoint *firstOrigin)
```


Layout Line Control

```

{
    /* define & initialize variables */
    boolean                startIsStaked;
    gxByteOffset           lineStarts[lineStartsCount],
                          newLineStart, nextStake,
                          nls2, priorStake, thisLineStart;

    char                   *pChar, *pSav;
    Fixed                  currLineDelta, lineAscent, lineDescent;
    gxLayoutOptions        specialOptions;
    ParagraphRecordHandle  paraHandle;
    gxShape                 bigLayout, thisLine;
    short                  i, level = 0, nextLineIndex;

    /* set up for left-aligned, unjustified text */
    specialOptions = *layoutOptions;
    specialOptions.just = specialOptions.flush = 0;
    specialOptions.width = 0;

    /* first, create a "big" layout for the whole paragraph */
    bigLayout = GXNewLayout(textRunCount, textRunLengths, text,
                          styleRunCount, styleRunLengths, styles,
                          levelRunCount, levelRunLengths, levels,
                          &specialOptions, firstOrigin);

    /*
       Next, compute all the line breaks for the paragraph
       and store their offsets in a temporary array.
    */
    thisLineStart = 0;
    nextLineIndex = 0;
    while (thisLineStart < totalByteCount &&
          nextLineIndex < lineStartsCount - 1)
    {
        lineStarts[nextLineIndex++] = thisLineStart;

        /*
           There is no hyphenation array, so the break is marked at
           the last glyph that fits on the line, regardless of
           its position in a word.
        */
        newLineStart = GXGetLayoutBreakOffset(bigLayout,
                                              thisLineStart,
                                              layoutOptions->width,
                                              0, nil, &startIsStaked,
                                              &priorStake, &nextStake);
        if (newLineStart == totalByteCount) break;
    }
}

```

Layout Line Control

```

/*
    Backtrack to first prior space before end of line,
    to rebreak line at word boundary. Your application
    should substitute a more sophisticated line breaking
    algorithm here.
*/
nls2 = newLineStart;
pSav = pChar = GetTextPiecePtr(text, textRunLengths,
                                GetPreviousOffset(bigLayout, newLineStart));
while (nls2 >= lineStarts[nextLineIndex-1] && *pChar != ' ')
    pChar = GetTextPiecePtr(text, textRunLengths,
                            nls2 = GetPreviousOffset(bigLayout, nls2));

/* if we've backed all the way up to the beginning of the
   line, use the line break originally returned from
   GXGetLayoutBreakOffset. Otherwise, take the new offset.
*/
if (nls2 <= lineStarts[nextLineIndex-1])
    thisLineStart = newLineStart;
else
{
    if (pSav != pChar) nls2 = GetNextOffset(bigLayout, nls2);
    thisLineStart = nls2;
}
}

/* put the last line-start entry beyond the end of the text */
lineStarts[nextLineIndex] = (short) totalByteCount;

/* now allocate space for the ParagraphRecord */
paraHandle = (ParagraphRecordHandle) NewHandle(
    (Size) (sizeof(ParagraphRecord) +
            nextLineIndex * sizeof(gxShape)));
(*paraHandle)->nLayouts = nextLineIndex; /* No. of lines */

/*
    Now create layout shapes for each of the lines
    and put them in the ParagraphRecord.
*/
specialOptions = *layoutOptions;
currLineDelta = 0;

```

Layout Line Control

```

for (i = 0; i < nextLineIndex; i++)
{
    /* don't justify the last line */
    if (i == nextLineIndex - 1) specialOptions.just = 0;

    /* create the layout for this line */
    thisLine = GXNewLayoutFromRange(bigLayout, lineStarts[i],
                                    lineStarts[i+1], &specialOptions, nil);

    /* move the line downward by the proper amount */
    if (currLineDelta) GXMoveShape(thisLine, 0, currLineDelta);

    /* add this line to the paragraph record */
    (*paraHandle)->layouts[i] = thisLine;

    /* calculate amount by which to move next line down */
    if (lineHeight) currLineDelta += lineHeight;
    else
    {
        GXGetLayoutSpan(thisLine, &lineAscent, &lineDescent);
        currLineDelta += lineAscent + lineDescent + extraLineGap;
    }
}
(*paraHandle)->totalHeight = currLineDelta;

GXDisposeShape(bigLayout); /* get rid of the big layout */
return paraHandle;         /* use the paragraph record */
}

```

The `GXGetLayoutBreakOffset` function is described on page 9-69. The `GXGetLayoutRangeWidth` function is described on page 9-71. The `GXNewLayoutFromRange` function is described on page 9-72.

Using Macintosh WorldScript for Line Breaking

QuickDraw GX and the fonts it uses have no information on how to break words according to the rules of any language. Your application must make the decision on where to end a line meaningfully.

However, Macintosh system software includes a group of managers, extensions, and resources known collectively as **WorldScript**. WorldScript was created to allow multi-language text processing and includes the Script Manager, Text Utilities, Text Services Manager, international resources, and other components. The typographic capabilities of QuickDraw GX replace much of the functionality of WorldScript. Nevertheless, you may still find parts of WorldScript useful to help you with line breaking.

Layout Line Control

The key WorldScript components for line breaking are the Text Utilities `FindWordBreaks` procedure and the string-manipulation resource (type 'itl2'), one of the WorldScript international resources. Every script system supplied with Macintosh computers includes a string-manipulation resource; that resource contains line-break information, specific to that script system, that the `FindWordBreaks` procedure uses when breaking lines.

To use WorldScript along with QuickDraw GX for line breaking, you can use a procedure like the following:

1. Use `GXGetLayoutBreakOffset` to determine the last glyph of the layout shape that fits into the available width.
2. Pass the edge offset equivalent to the trailing edge of that glyph to the `FindWordBreaks` procedure, passing also the script code that defines the script system to which the text in that style run belongs.
3. In response, `FindWordBreaks` returns the edge offsets of the nearest word boundaries before and after the offset you pass in, according to information in the given script's string-manipulation resource. Normally, you would use the "before" offset to define the end of your line, although the "after" offset is also possible, given the justification model's handling of the shrink case.

Another possible approach is to analyze the last style run in your line before calling `GXGetLayoutBreakOffset`. You could use `FindWordBreaks` repeatedly to build a hyphenation array and then pass that array to `GXGetLayoutBreakOffset`; in that case, `GXGetLayoutBreakOffset` will return an acceptable break point in the language of that text.

IMPORTANT

Script codes for WorldScript are different from the script codes used in the encoding property of QuickDraw GX style objects. For example, in WorldScript, 0 is Roman script and 1 is Japanese; for QuickDraw GX, 0 is no script, 1 is Roman, and 2 is Japanese. Do not use any routines from WorldScript that presuppose a GrafPort because QuickDraw GX does not directly use GrafPorts. ▲

The `FindWordBreaks` procedure is described in the chapter "Text Utilities" in *Inside Macintosh: Text*. The string-manipulation resource is described in the appendix "International Resources" in *Inside Macintosh: Text*.

Manipulating Nested Direction Levels

To force a dominant direction on a line of text or on a phrase within a line, you can assign it a direction level. You define a run for it and assign that run a level in the levels array of the layout shape geometry.

- For lines with uniform left-to-right text, or left-to-right text with isolated embedded phrases of right-to-left text, you can ignore the levels array completely. Alternatively, you can define one run for the entire layout and assign it any even value (0 is most efficient).

Layout Line Control

- For lines with uniform right-to-left text, or right-to-left text with isolated embedded phrases of left-to-right text, you can define one run for the entire layout and assign it any odd value (1 is most efficient).
- For lines with complex (mixed-direction) phrases of one dominant direction embedded in a line of opposite dominant direction, you need to define a direction run and level for each phrase whose dominant direction must be preserved. An even level causes the dominant direction for that phrase to be left to right; an odd level causes the dominant direction to be right to left. The dominant direction of the line as a whole is defined by the lowest level (odd or even) on the line.

QuickDraw GX permits up to 15 nested direction levels, although a line that uses more than 2 or 3 is quite rare.

Listing 9-4 is a partial listing showing a simple, static example that defines a levels array and assigns levels to it to preserve the proper ordering of a line of mixed-direction text. The line combines Arabic and English and contains five phrases. The line is drawn at the location posn.

Listing 9-4 Defining nested direction levels for a line of text

```

gxShape      layout;
gxStyle      timesStyle, helveticaStyle, baghdadStyle;
gxStyle      textStyles[5];
char         *textRuns[5];
short        textLengths[5], totalLength;
static short  levelRunLengths[3], levels[3];

/* define the 5 separate text strings for the line */
char *text1 = "He said ";
/*
    The following is "Macintosh" in Arabic:
    meem, alif, kaf, noon, tah, wau, shin
*/
static char text2[] =
    {0xE5, 0xC7, 0xE3, 0xE6, 0xCA, 0xE8, 0xD4, 0};
char *text3 = " (Macintosh) ";
/*
    The following is "Arabic" in Arabic:
    alif, lam, ein, reh, beh, yeh
*/
static char text4[] = {0xC7, 0xE4, 0xD9, 0xD1, 0xC8, 0xEA, 0};
char *text5 = "" to me.";
.
.
.
```

Layout Line Control

```

/* Initialize the text runs to pass into GXNewLayout */
textRuns[0] = text1;
textRuns[1] = text2;
textRuns[2] = text3;
textRuns[3] = text4;
textRuns[4] = text5;

textLengths[0] = strlen (text1);
textLengths[1] = strlen (text2);
textLengths[2] = strlen (text3);
textLengths[3] = strlen (text4);
textLengths[4] = strlen (text5);

/*
   Initialize the direction levels arrays: here's where the
   lengths of the nested direction levels are defined.
   The first direction run is the first phrase; the second
   direction run is the second 3 phrases; and the third
   run is the last phrase.
*/
levelRunLengths[0] = textLengths[0];
levelRunLengths[1] = textLengths[1] + textLengths[2] +
                    textLengths[3];
levelRunLengths[2] = textLengths[4];

totalLength = levelRunLengths[0] + levelRunLengths[1] +
              levelRunLengths[2];

/*
   Define the levels for each of the direction runs. The
   desired dominant direction is left to right; and so and the
   first and last runs have a level of 0; the central phrase
   must remain right to left, so it has a level of 1.
*/
levels[0] = 0;
levels[1] = 1;
levels[2] = 0;

/* define some styles; first is 36 pt. Helvetica */
helveticaStyle = NewLayoutStyle((char *) "\pHelvetica",
                                ff(36), 0, nil,
                                nil, 0, nil);

```

Layout Line Control

```

/* second is 36 pt. Baghdad Plain (Arabic) */
baghdadStyle = NewLayoutStyle((char *) "\pBaghdad Plain",
                               ff(36), 0, nil,
                               nil, 0, nil);

/* third is 36 pt. Times */
timesStyle = NewLayoutStyle((char *) "\pTimes Roman",
                             ff(36), 0, nil,
                             nil, 0, nil);

/* assign the styles to the style runs */
textStyles[0] = helveticaStyle;
textStyles[1] = baghdadStyle;
textStyles[2] = timesStyle;
textStyles[3] = baghdadStyle;
textStyles[4] = helveticaStyle;

/* now build and draw the layout */
layout = GXNewLayout(5, textLengths, (void *)textRuns,
                    5, textLengths, textStyles,
                    3, levelRunLengths, levels,
                    nil, &posn);
GXDrawShape (layout);

.
.
.

```

Figure 9-18 shows the results of executing the code in Listing 9-4. Compare this figure and the levels defined in Listing 9-4 with the phrases and levels shown in the section “Forced Reordering With Nested Direction Levels” beginning on page 9-19.

Figure 9-18 A text line with nested direction levels

He said “ماكنتوش (Macintosh) العربي” to me.

The levels array is described in the chapter “Layout Shapes” in this book.

Overriding the Glyph Direction in a Style Run

Listing 9-5 shows an example of overriding the glyph direction in a style run, to make the individual glyphs in a sequence appear in reverse order. This changing of glyph direction is useful only for special effects; it is entirely different from imposing a dominant direction on a text run, which you do by assigning a direction level to it.

The function in Listing 9-5 creates a layout shape named `layout`; the shape uses a style object named `myStyle`. The function uses the library function `InitializeRunControls` to initialize a run controls structure, and the library function `NewLayoutStyle` to create and initialize a style object. The function initially draws the shape at the point `myPoint`. The length of the string composing the layout shape is `len`.

Listing 9-5 Overriding the glyph direction in a style run

```
{
char          *myString = "QuickDraw GX";
gxRunControls runControls;
.
.
.
InitializeRunControls(&runControls);

/* create the style object and the layout shape */
myStyle = NewLayoutStyle((char *) "\pTimes Roman", ff(24),
                        0, nil, nil, 0, nil);

layout = GXNewLayout(1, &len, (void *) &myString,
                    1, &len, &myStyle,
                    0, nil, nil,
                    nil, &myPoint);
GXDrawShape(layout);

/* override the glyph direction in the run controls flags */
runControls.flags = gxImposeRightToLeft;
GXSetStyleRunControls(myStyle, &runControls);
GXMoveShape(layout, 0, ff(48));
GXDrawShape(layout);
.
.
.
}
```


Figure 9-19 shows the results of executing the code in Listing 9-5. Note that in this case the dominant direction for the line is still left-to-right, but the individual glyph directions are now all right to left.

Figure 9-19 Results of overriding glyph direction



Justifying Lines by Stretching and Shrinking

Listing 9-6 shows a simple example of justifying lines of different lengths to a single text width. The example first draws three unjustified lines. It then defines the text width to be equal to the width of the middle-length line, and specifies full justification for all lines. QuickDraw GX stretches or compresses the lines to fit.

The function in Listing 9-6 uses a single layout shape (`layout`) and a single style object (`myStyle`) for all three lines. It uses the layout options structure `layoutOptions`. It draws the first line at the location `myPoint`. It uses the function `GXGetShapeTypographicBounds` to determine the (unjustified) width of the middle text line.

Listing 9-6 A simple justification example

```
{
    char        *myString3 = "A shorter line of text";
    char        *myString  = "A medium-length line of text";
    char        *myString2 = "A slightly lengthier line of text";
    gxLine      myLine;
    gxPoint     advance, myPoint;
    short       len;
    gxRectangle bounds;
    .
    .
    .
    /* draw left margin */
    myLine.first.x = myLine.last.x = myPoint.x;
    myLine.first.y = ff(0);
    myLine.last.y = ff(600);
    GXDrawLine(&myLine);
}
```

Layout Line Control

```

/* set up and draw the short line, unjustified */
myStyle = NewLayoutStyle((char *) "\pSkia Regular", ff(24),
                        0, nil, nil, 0, nil);
layoutOptions.just = 0;
len = strlen(myString3);
layout = GXNewLayout(1, &len, (void *) &myString3,
                    1, &len, &myStyle,
                    0, nil, nil,
                    &layoutOptions, &myPoint);
GXDrawShape(layout);

/* draw middle line, unjustified */
len = strlen(myString);
GXSetLayout(layout, 1, &len, (void *) &myString, 0,
            nil, nil, 0, nil, nil, nil, nil);
GXMoveShape(layout, 0, ff(36));
GXDrawShape(layout);

/* draw right margin at normal width of middle line */
GXGetShapeTypographicBounds(layout, &bounds);
myLine.first.x = myLine.last.x =
    myPoint.x + bounds.right - bounds.left;
GXDrawLine(&myLine);

/* draw third line, unjustified */
len = strlen(myString2);
GXSetLayout(layout, 1, &len, (void *) &myString2, 0,
            nil, nil, 0, nil, nil, nil, nil);
GXMoveShape(layout, 0, ff(36));
GXDrawShape(layout);

/* set width and justification of layout options structure */
layoutOptions.just = fract1;
layoutOptions.width = bounds.right - bounds.left;

/* draw all three lines again, fully justified this time */
len = strlen(myString3);
GXSetLayout(layout, 1, &len, (void *) &myString3, 0,
            nil, nil, 0, nil, nil, &layoutOptions, nil);
GXMoveShape(layout, 0, ff(60));
GXDrawShape(layout);

```

Layout Line Control

```

len = strlen(myString);
GXSetLayout(layout, 1, &len, (void *) &myString, 0,
            nil, nil, 0, nil, nil, &layoutOptions, nil);
GXMoveShape(layout, 0, ff(36));
GXDrawShape(layout);

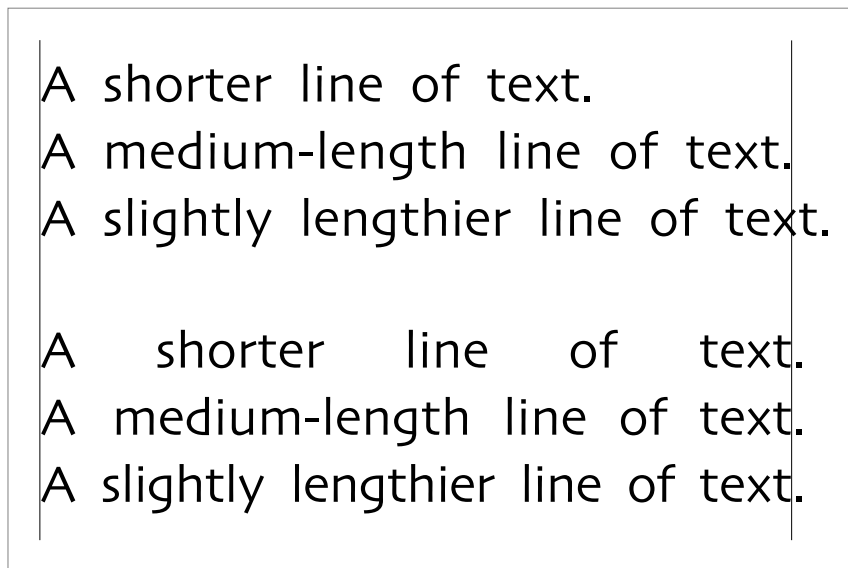
len = strlen(myString2);
GXSetLayout(layout, 1, &len, (void *) &myString2, 0,
            nil, nil, 0, nil, nil, &layoutOptions, nil);
GXMoveShape(layout, 0, ff(36));
GXDrawShape(layout);

.
.
.
}

```

Figure 9-20 shows the results of executing the code in Listing 9-6. The upper three lines are not justified; the lower three lines are fully justified, to a width that matches the unjustified length of the second line. In the lower three lines, the third line is compressed, and the first line is stretched, to accommodate the justification gap. The justified text in the lower three lines does not include the periods because a period is considered a hanging glyph in this font.

Figure 9-20 Unjustified (upper) and justified (lower) lines of different lengths



Displaying Partial Justification

Justification in QuickDraw GX is continuous, rather than just “on” or “off.” You specify the amount of justification in the `just` field of the layout options structure in the layout shape geometry. (Justification amounts between 0 and 100 percent are sometimes called *ragged justification*.)

This section illustrates several different kinds of justification. It shows how they function differently and how the appearance changes as justification increases from none to full. In each case the application controls only the amount of justification that is applied. Font settings control the kind of justification that is applied, and this justification happens automatically.

Justification With White Space

Listing 9-7 is a partial listing of a sample function that illustrates how continuous justification works in Roman text. It draws the same string of left-aligned text five times: once unjustified, once 25 percent justified, once 50 percent justified, once 75 percent justified and once fully justified. In this case, justification is achieved through addition of both intercharacter and interword white space.

Listing 9-7 uses the library function `NewLayoutStyle` to create and initialize a style object. It creates a layout shape named `layout` and a style object named `myStyle`. It draws the first line at the location `myPoint`. The text string in the shape has the length `len`.

Listing 9-7 Displaying partial justification using white space

```
{
    /* define and initialize variables */
    char          *myString = "A line of text";
    gxLine        myLine;
    .
    .
    InitializeLayoutOptions(&layoutOptions);
    layoutOptions.width = ff(500);

    /* draw two vertical lines to mark the margins */
    myLine.first.x = myLine.last.x = myPoint.x;
    myLine.first.y = 0;
    myLine.last.y = ff(1000);
    GXDrawLine(&myLine);
    myLine.first.x = myLine.last.x = myPoint.x +
                                   layoutOptions.width;
    GXDrawLine(&myLine);
}
```

Layout Line Control

```

/* create and draw the layout shape (unjustified) */
myStyle = NewLayoutStyle((char *) "\pTimes Roman", ff(36), 0,
                        nil, nil, 0, nil);

layout = GXNewLayout(1, &len, (void *) &myString,
                    1, &len, &myStyle,
                    0, nil, nil,
                    nil, &myPoint);
GXDrawShape(layout);

/* give the shape 25% justification and redraw */
layoutOptions.just = fract1 / 4;
GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
            &layoutOptions, nil);
GXMoveShape(layout, 0, ff(54));
GXDrawShape(layout);

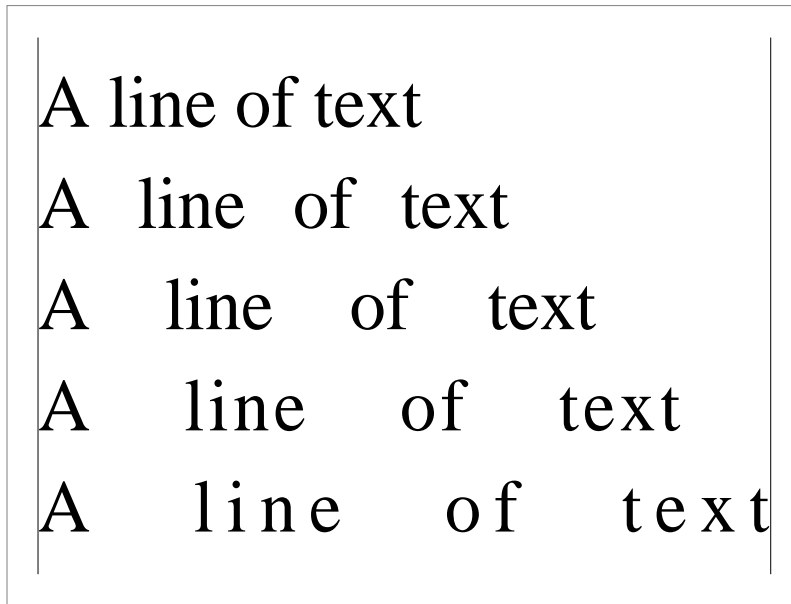
/* give the shape 50% justification and redraw */
layoutOptions.just = fract1 / 2;
GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
            &layoutOptions, nil);
GXMoveShape(layout, 0, ff(54));
GXDrawShape(layout);

/* give the shape 75% justification and redraw */
layoutOptions.just = 3 * (fract1 / 4);
GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
            &layoutOptions, nil);
GXMoveShape(layout, 0, ff(54));
GXDrawShape(layout);

/* give the shape 100% (full) justification and redraw */
layoutOptions.just = fract1;
GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
            &layoutOptions, nil);
GXMoveShape(layout, 0, ff(54));
GXDrawShape(layout);
.
.
.
}

```

Figure 9-21 shows the results of executing the code in Listing 9-7.

Figure 9-21 Five degrees of justification with white space

Justification With Kashidas

Listing 9-8 shows a few fragments of a sample function that illustrates how continuous justification works in Arabic text. The function draws the same string of right-aligned text five times: once unjustified, once 25 percent justified, once 50 percent justified, once 75 percent justified and once fully justified. In this case, justification is achieved through addition of kashidas to certain glyphs.

The code in Listing 9-8 is essentially identical to that in Listing 9-7 on page 9-46, with the exception of the fragments listed here. No different coding is necessary to cause Arabic justification than is used to cause justification with white space.

Listing 9-8 Displaying partial justification with kashidas

```
static char arabicString[8] = {'\xE5', '\xC7', '\xE3', '\xE6',
                              '\xCA', '\xE8', '\xD4', 0};

void ExtenderBars(WindowPtr sampleWindow)
{
    /* define and initialize variables */
    char          *myString = (char *) &arabicString[0];
    gxLine        myLine;
    .
    .
    .
}
```

Layout Line Control

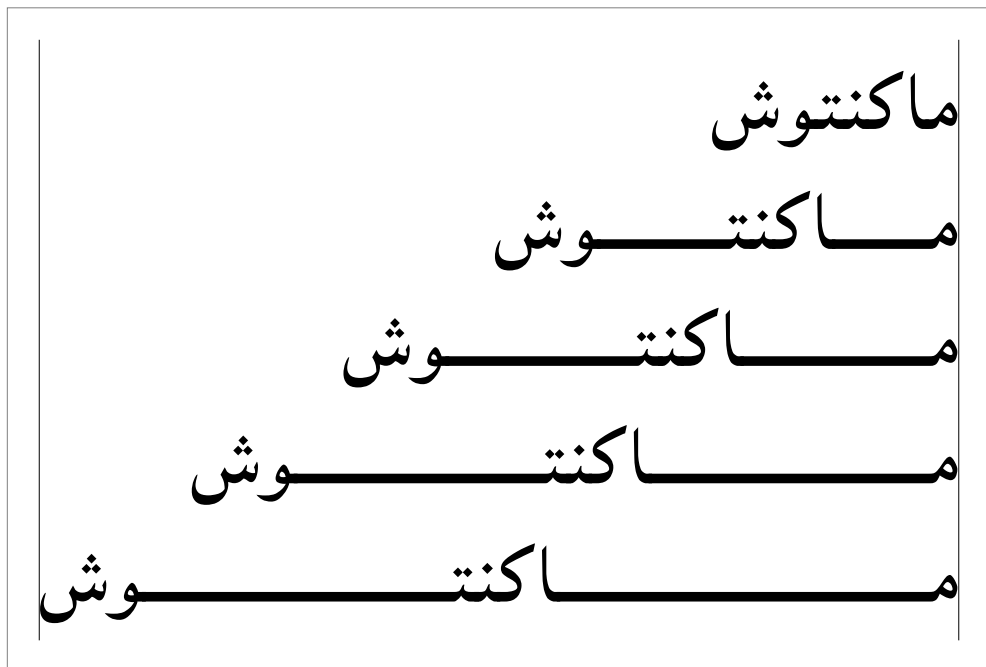
```

layoutOptions.flush = fract1; /* right-aligned */
.
.
.
/* create the style for the layout shape */
myStyle = NewLayoutStyle((char *) "\pDiwan", ff(36),
                        noMetricsGridText, nil, nil, 0, nil);
.
.
.
/* create the shape and draw it at various justifications */
.
.
.
}

```

Figure 9-22 shows the results of executing the function in Listing 9-8.

Figure 9-22 Five degrees of justification with kashidas

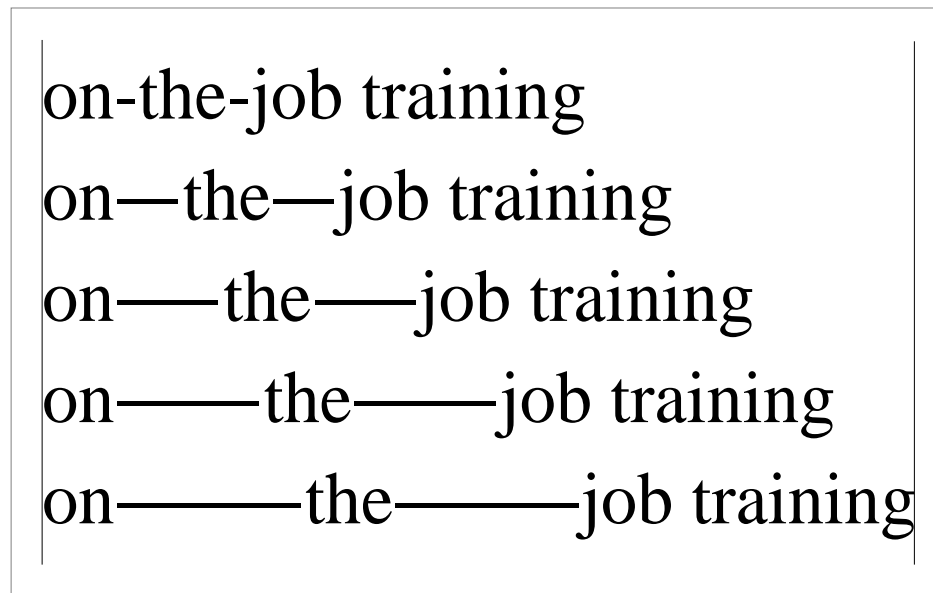


Justification With Glyph Deformation

Figure 9-23 shows the results of executing a sample program similar to the others in this section. The figure shows a line of text drawn with successively increasing justification values. In this case, all justification is by intercharacter space; however, one glyph (the hyphen) acts differently from the other glyphs. Instead of any glyphs gaining white space on either side, the hyphen gradually stretches to take up all of the justification gap.

The font in this example uses glyph stretching, which employs a text face mechanism to widen the glyph. Other fonts may use glyph ductility, which is a font-variation mechanism, to achieve similar results.

Figure 9-23 Glyph stretching during increasing justification



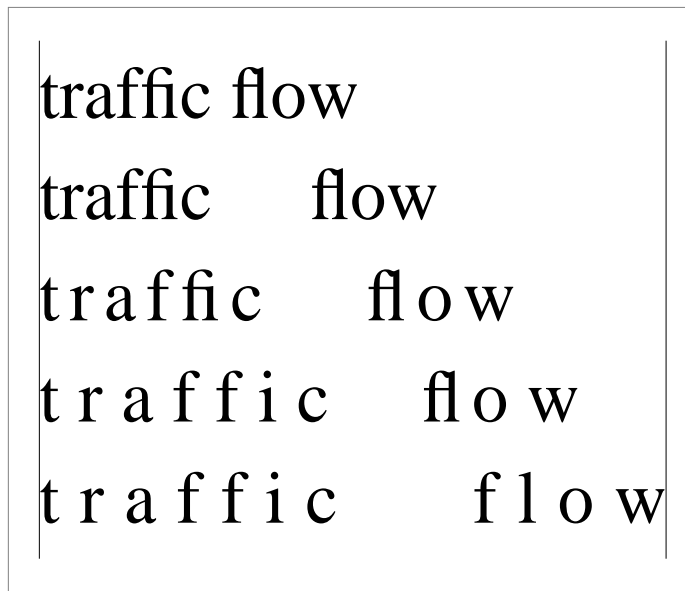
The application does not specifically request or control this behavior; it is controlled by the particular font used in this sample. However, because glyph stretching and glyph ductility are part of postcompensation action, you can prevent them from happening by setting the `gxNoSpecialJustification` flag in the run controls structure of the style run to which the text belongs. Postcompensation action is described on page 9-23; the run controls structure is described in the chapter “Layout Styles” in this book.

Justification and Ligature Decomposition

Figure 9-24 shows the results of executing another sample program that draws a line of text with successively increasing justification values. In this case, justification is by both interword and intercharacter space. But in addition, once the added white space reaches a certain threshold (equivalent to somewhere between 50 percent and 75 percent justifica-

tion in this example), the “fi” and “fl” ligatures decompose into their component glyphs, which then spread apart to take up the justification gap. Note that the “fi” ligature decomposes before the “fl” ligature in this font.

Figure 9-24 Ligature decomposition during increasing justification



The application does not specifically request this behavior; the font governs ligature decomposition during justification. However, you can control it in two ways. First, because ligature decomposition is part of postcompensation action, you can prevent it from happening by setting the `gxNoSpecialJustification` flag in the run controls structure of the style run to which the text belongs. Second, you can modify the threshold at which it occurs by placing a value in the `decompositionAdjustmentFactor` field of the run controls structure. For more information on the `decompositionAdjustmentFactor` field, see the chapter “Layout Styles” in this book.

Postcompensation action is described on page 9-23. The run controls structure is described in the chapter “Layout Styles” in this book.

The `just` field of the layout options structure is described in the chapter “Layout Shapes” in this book.

Changing the Behavior of Justification Priorities

A priority justification override structure contains the optional overrides for all priority levels that might occur within a single style run. The `deltas` array contains the width delta structures for each priority level. This is the primary input to the `GXSetStyleRunPriorityJustOverride` function.

Layout Line Control

For example, assume the user wants to override the normal (font-specified) justification behavior so that intercharacter spacing has the same priority as interword spacing when extra space must be distributed in the line. The following code fragment would produce the desired effect. Assume that there is a priority justification override structure called `overrides`, that the style to be affected is called `targetStyle`, and that the grow limits for glyphs with whitespace priority are `whiteSpaceBeforeLimit` and `whiteSpaceAfterLimit`.

```
myFlags = gxOverrideLimits | gxOverridePriority |
                                gxWhiteSpacePriority;
overrides.deltas[gxInterCharPriority].growFlags = myFlags;
overrides.deltas[gxInterCharPriority].beforeGrowLimit =
                                whiteSpaceBeforeLimit;
overrides.deltas[gxInterCharPriority].afterGrowLimit =
                                whiteSpaceAfterLimit;
GXSetStyleRunPriorityJustOverride(targetStyle, &overrides);
```

As another example, Listing 9-9 is a partial listing of a sample function that demonstrates how to override justification priorities. It draws four strings, three of which are fully justified, as four separate layout shapes. (All four strings have nearly the same number of characters.) Two of the shapes have the default (font-specified) justification behavior. A third layout shape has a justification priority override that forces all justification adjustments to be made to interword spaces. A fourth has a justification priority override that forces all justification adjustments to be made to intercharacter spaces.

Listing 9-9 uses the run-controls structure `controls`, the layout-options structure `layoutOptions`, and the priority-justification override structures `allToSpace` and `allToChar`. It draws the text lines starting at the location `posn`.

Listing 9-9 Overriding justification priorities

```
.
.
.
gxShape      layout1, layout2, layout3, layout4;
gxStyle      style1, style2, style3, style4;
gxPriorityJustificationOverride allToSpace, allToChar;
char *text1 =
    "Unjustified text with no extra space applied to the line.";
char *text2 =
    "Fully justified with interword and intercharacter space.";
```

Layout Line Control

```

char *text3 =
    "Fully justified with the use of interword space alone.";
char *text4 =
    "Fully justified with use of intercharacter space alone.";
.
.
.
/*
    Set up the "all to space" override. Set unlimited gap
    absorption for the grow flag for white space priority, so
    that white space characters will absorb all justification
    gap.
*/
allToSpace.deltas[whiteSpacePriority].growFlags |=
    (overrideUnlimited | unlimitedGapAbsorption);

/*
    Set up the "all to intercharacter" override. Set unlimited
    gap absorption for the grow flag for intercharacter
    priority, and then change the priority itself to kashida
    (the highest priority). This forces all justification
    gap to be distributed among intercharacter space, and not to
    white space.
*/
allToChar.deltas[interCharPriority].growFlags |=
    (overrideUnlimited | unlimitedGapAbsorption |
     overridePriority | kashidaPriority);

/* set up the positioning for the lines of text */
layoutOptions.width = ff(500);    /* extra width */
layoutOptions.just = fract1;      /* fully justified */

/* build a style object and a layout shape for each line */
len = strlen(text1);
style1 = NewLayoutStyle((char *) "\pTimes Roman", ff(14),
                        0, nil, nil, 0, nil);
layout1 = GXNewLayout(1, &len, (void *) &text1,
                      1, &len, &style1,
                      0, nil, nil,
                      nil, &posn);

```

Layout Line Control

```

posn.y += ff(30);
len = strlen(text2);
style2 = NewLayoutStyle((char *) "\pTimes Roman", ff(14),
                        0, nil, nil, 0, nil);
GXSetStyleRunControls(style2, &controls);
layout2 = GXNewLayout(1, &len, (void *) &text2,
                    1, &len, &style2,
                    0, nil, nil,
                    &layoutOptions, &posn);

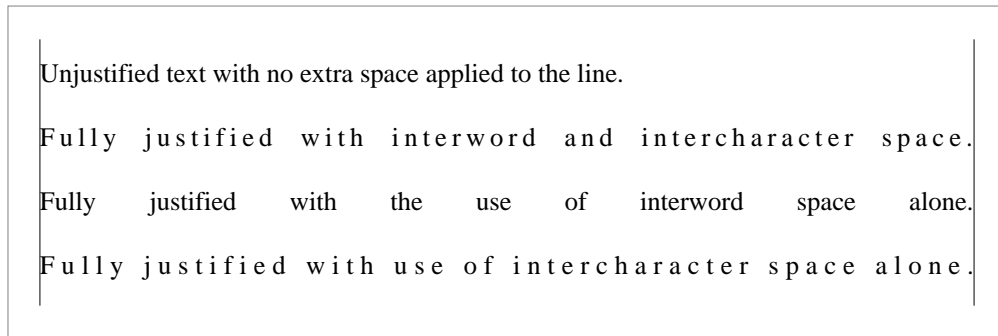
posn.y += ff(30);
len = strlen(text3);
style3 = NewLayoutStyle((char *) "\pTimes Roman", ff(14),
                        0, nil, nil, 0, nil);
GXSetStyleRunPriorityJustOverride(style3, &allToSpace);
layout3 = GXNewLayout(1, &len, (void *) &text3,
                    1, &len, &style3,
                    0, nil, nil,
                    &layoutOptions, &posn);

posn.y += ff(30);
style4 = NewLayoutStyle((char *) "\pTimes Roman", ff(14),
                        0, nil, nil, 0, nil);
GXSetStyleRunPriorityJustOverride(style4, &allToChar);
layout4 = GXNewLayout(1, &len, (void *) &text4,
                    1, &len, &style4,
                    0, nil, nil,
                    &layoutOptions, &posn);

GXDrawShape (layout1);
GXDrawShape (layout2);
GXDrawShape (layout3);
GXDrawShape (layout4);
.
.
.

```

Figure 9-25 shows the results of executing the code in Listing 9-9.

Figure 9-25 Results of justification priority overrides on intercharacter and interword spacing

The `GXSetStyleRunPriorityJustOverride` function is described on page 9-75.

To retrieve the priority justification overrides for a style run, use the `GXGetStyleRunPriorityJustOverride` function, described on page 9-74. To retrieve the priority justification overrides for the style object associated with a particular shape, use the `GXGetShapeRunPriorityJustOverride` function, described on page 9-76. To set the priority justification overrides of the style object associated with a particular shape, use the `GXSetShapeRunPriorityJustOverride` function, described on page 9-77.

Changing Justification Behavior of Individual Glyphs

You can use one or more glyph justification override structures to assign an overriding justification priority and behavior to one or more specific glyphs in a style run. This section shows a single example, one involving the whitespace glyph.

Listing 9-10 is a partial listing of a sample function that illustrates how to use a glyph justification override to alter whitespace behavior. The function draws the same string of text three times: once unjustified, and twice fully justified. The first justified line shows the default (font-specified) justification behavior. In the second justified line, the justification of the whitespace glyph is overridden to force it to take up all justification gap.

The function in Listing 9-10 on page 9-56 creates a layout shape named `layout` and a style object named `myStyle`. It uses the layout-options structure `layoutOptions`. It draws the first line at the location `myPoint`. The text string in the shape has the length `len`.

Listing 9-10 Overriding justification behavior of the whitespace glyph

```

void UnlimitedGapAbsorption(WindowPtr sampleWindow)
{
    char                *myString = "all to space";
    gxGlyphcode         glyphcodes[12];
    gxGlyphJustificationOverride glyphJustOverride;
    gxLine              myLine;
    unsigned short      firstGlyph, secondGlyph;
    .
    .
    .
    layoutOptions.width = ff(320);
    layoutOptions.just = 0;

    /* draw two vertical lines to mark the margins */
    myLine.first.x = myLine.last.x = myPoint.x;
    myLine.first.y = 0;
    myLine.last.y = ff(500);
    GXDrawLine(&myLine);

    myLine.first.x = myLine.last.x = myPoint.x +
                                layoutOptions.width;
    GXDrawLine(&myLine);

    /* create and draw the layout shape (unjustified) */
    myStyle = NewLayoutStyle((char *) "\pTimes Roman", ff(36), 0,
                            nil, nil, 0, nil);
    layout = GXNewLayout(1, &len, (void *) &myString,
                        1, &len, &myStyle,
                        0, nil, nil,
                        &layoutOptions, &myPoint);
    GXDrawShape(layout);

    /* give the shape full justification but default behavior */
    layoutOptions.just = fract1;
    GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
                &layoutOptions, nil);
    GXMoveShape(layout, 0, ff(54));
    GXDrawShape(layout);

    /*
       Now override the default justification behavior to give
       the glyph following edge offset 3--which in this sample
       is a whitespace glyph--unlimited gap absorption. Its

```

Layout Line Control

```

        justification priority is higher than non-space glyphs,
        so whitespace characters will absorb all justification.
    */

    /* get an array of all the glyph codes in the line */
    GXGetLayoutGlyphs(layout, glyphcodes, nil, nil, nil,
                      nil, nil, nil);

    /* find the index of the glyph following offset 3 */
    GXGetOffsetGlyphs(layout, 3, true, nil,
                      &firstGlyph, &secondGlyph);

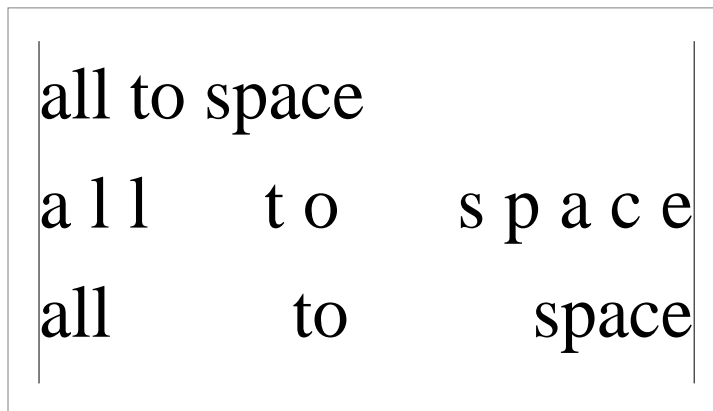
    /* override the justification of the next glyph's glyphcode */
    glyphJustOverride.glyph = glyphcodes[firstGlyph - 1];
    glyphJustOverride.override.growFlags = overrideUnlimited +
                                         unlimitedGapAbsorption;
    glyphJustOverride.override.shrinkFlags = 0;
    GXSetStyleRunGlyphJustOverrides(myStyle, 1,
                                    &glyphJustOverride);

    /* finally, redraw the shape */
    GXMoveShape(layout, 0, ff(54));
    GXDrawShape(layout);
    .
    .
    .
}

```

Figure 9-26 shows the results of executing the code in Listing 9-10.

Figure 9-26 Results of overriding justification behavior of the whitespace glyph



Layout Line Control Reference

This section describes the constants, data types, and functions with which you can manipulate the layout of text lines using layout shapes.

Constants and Data Types

QuickDraw GX uses the constants and structures described in this section to define baselines and to override justification behavior.

Baseline Types

Text of different scripts most naturally aligns with different baselines. The `gxBaselineType` enumeration specifies the preferred baseline to use for text of a given font in a particular style run. It is specified in the `BaselineType` field of the run controls structure of each style run in a layout shape.

The baseline types enumeration provides constants for all defined baseline types. Your application need never fill in this value in the run controls structure; if you instead specify `gxNoOverrideBaseline`, QuickDraw GX determines the proper baseline type for the style run from information in the style's font.

```
enum {
    gxRomanBaseline                = 0,
    gxIdeographicCenterBaseline,
    gxIdeographicLowBaseline,
    gxHangingBaseline,
    gxMathBaseline,
    gxLastBaseline                 = 31,
    gxNumberOfBaselineTypes       = gxLastBaseline + 1,
    gxNoOverrideBaseline          = 255
};
```

```
typedef unsigned long gxBaselineType;
```

Constant descriptions

`gxRomanBaseline`

The baseline used by most Roman-script languages.

`gxIdeographicCenterBaseline`

The most common baseline used by Chinese, Japanese, and Korean ideographic scripts, in which the ideographs are centered halfway through the line height.

Layout Line Control

`gxIdeographicLowBaseline`

A baseline used by Chinese, Japanese, and Korean scripts. Similar to `gxIdeographicCenterBaseline`, but with the glyphs lowered. This baseline is most commonly used to align Roman glyphs within ideographic fonts to Roman glyphs in Roman fonts.

`gxHangingBaseline`

The baseline used by Devanagari and related scripts, in which the bulk of most glyphs is below the baseline. This baseline type is also used for drop capitals in Roman scripts.

`gxMathBaseline`

The baseline used for setting mathematics. It is centered on symbols such as the minus sign (at half the x-height).

`gxLastBaseline`

No baseline value may exceed this value. Baseline values between `gxMathBaseline` and `gxLastBaseline` are reserved.

`gxNumberOfBaselineTypes`

The total number of baseline types (= `gxLastBaseline` + 1).

`gxNoOverrideBaseline`

Instructs QuickDraw GX to use the standard baseline value from the current font.

For other information about and examples of various types of baselines, see “Baseline Types” on page 9-4.

Baseline Deltas Array

The baseline deltas array (type `gxBaselineDeltas`) is used in the baseline structure, described next. Also, the `GXGetStyleBaselineDeltas` function, described on page 9-66, returns values in a baseline deltas array. Baseline deltas is an array of distances (in points) between the various baseline types and $y = 0$.

```
typedef Fixed gxBaselineDeltas[gxNumberOfBaselineTypes];
```

Baseline Structure

The baseline structure (type `gxLineBaselineRecord`) controls the positions of baselines with respect to one another in a line of text.

```
typedef struct {
    gxBaselineDeltas deltas;
} gxLineBaselineRecord;
```

Field descriptions

deltas The offsets (in points) from $y = 0$ to every other baseline type for this line. If you are filling in this structure manually, you need to fill in only those values that correspond to the set of baselines present on the line.

You can fill in this array by calling the `GXGetStyleBaselineDeltas` function, described on page 9-66.

Justification Priorities

Glyphs in a font can be assigned justification priorities by the font designer. In general, QuickDraw GX applies justification to glyphs on a line in order of glyph priority, from high to low. Your application can override these priorities to change the order in which glyphs participate in justification. These are the defined justification priorities (lower numbers represent higher priorities):

```
enum {
    gxKashidaPriority           = 0,
    gxWhiteSpacePriority        = 1,
    gxInterCharPriority         = 2,
    gxNullJustificationPriority = 3,
    gxNumberOfJustificationPriorities
};

typedef unsigned char gxJustificationPriority;
```

Constant descriptions

`gxKashidaPriority`

The highest priority. Typically used for kashidas (extension bars) in Arabic. Glyphs with this priority are extended or compressed before all other glyphs in the line.

`gxWhiteSpacePriority`

Typically assigned to whitespace (interword) glyphs. Glyphs with this priority are extended or compressed, usually by the addition or removal of white space, after all glyphs on the line with priority `gxKashidaPriority` have been extended or compressed to the maximum amount permitted.

`gxInterCharPriority`

Assigned to all glyphs that do not have `gxKashidaPriority` or `gxWhiteSpacePriority`. Glyphs with this priority are extended or compressed, typically by the addition or removal of white space, after all glyphs on the line with priority `gxWhiteSpacePriority` have been extended or compressed to the maximum amount permitted.

`gxNullJustificationPriority`

Available as a priority for glyphs that you want to participate in justification last of all.

`gxNumberOfJustificationPriorities`

The number of defined justification priorities. You can use this value for range-checking, size allocation, or loop control.

Note

The justification priorities have names that describe the types of glyphs that typically have those priorities, but you can assign any priority to any glyph. The actual kind of justification that QuickDraw GX applies—for example, kashida or whitespace—is defined for each glyph by the font. The priority specifies only the order in which glyphs participate in justification. ♦

You can override the justification priority for an individual glyph in a style run by using the glyph justification override structure, described on page 9-64. You can override the behavior of all glyphs of a given justification priority for an entire style run in by using the priority justification override structure, described on page 9-63. In each case, you specify the justification priority in the `growFlags` or `shrinkFlags` field of one or more width delta structures. The width delta structure is described next.

Width Delta Structure

A width delta structure contains all the information needed to override the distribution behavior of a glyph or set of glyphs during justification. It is used in both the priority justification override structure and the glyph justification override structure. In each case the width delta structure can specify both a change in priority and a change in distribution behavior.

```
typedef struct {
    Fixed          beforeGrowLimit;
    Fixed          beforeShrinkLimit;
    Fixed          afterGrowLimit;
    Fixed          afterShrinkLimit;
    gxJustificationFlags growFlags;
    gxJustificationFlags shrinkFlags;
} gxWidthDeltaRecord;
```

Field descriptions**beforeGrowLimit**

The number of points by which a 1-point glyph can grow on the left side (top side for vertical text). A value of 0.2, for example, means that a 24-point glyph can have by no more than 4.8 points of extra space added on the left or top side.

beforeShrinkLimit

The number of points by which a 1-point glyph can shrink on the left or top side. If specified, this value should be negative.

afterGrowLimit The number of points by which a 1-point glyph can grow on the right side (bottom for vertical text).

afterShrinkLimit

The number of points by which a 1-point glyph can shrink on the right or bottom side. If specified, this value should be negative.

Layout Line Control

<code>growFlags</code>	Justification flags, used to control the overriding behavior when justification entails lengthening the line.
<code>shrinkFlags</code>	Justification flags, used to control the overriding behavior when justification entails shortening the line.

A justification flag in the `growFlags` field controls whether or not the `beforeGrowLimit` and `afterGrowLimit` values are applied to this style run. Likewise, a flag in the `shrinkFlags` field controls whether or not the `beforeShrinkLimit` and `afterShrinkLimit` values are applied. Your application can thus selectively override certain cases (such as the grow case only), while retaining default behavior for other cases. Justification flags are described next.

Justification Flags

The justification flags control which aspects of the normal, font-specified justification behavior of a glyph or set of glyphs are to be overridden. Justification flags make up two fields in the width delta structure, described in the previous section.

```
enum {
    gxOverridePriority          = 0x8000,
    gxOverrideLimits           = 0x4000,
    gxOverrideUnlimited         = 0x2000,
    gxUnlimitedGapAbsorption    = 0x1000,
    gxJustificationPriorityMask = 0x000F
    gxAllJustificationFlags    = gxOverridePriority |
                                gxOverrideLimits |
                                gxOverrideUnlimited |
                                gxUnlimitedGapAbsorption |
                                gxJustificationPriorityMask
};
```

```
typedef unsigned short gxJustificationFlags;
```

Constant descriptions

`gxOverridePriority`

This bit specifies whether or not QuickDraw GX overrides justification priority. If the bit is set, the justification priority in the `gxJustificationPriorityMask` part of the justification flags is used for the glyphs this width delta structure applies to. If it is cleared, QuickDraw GX uses the default justification priority for those glyphs. If it is cleared, the priority mask bits must also be set to 0. The use of this flag is to determine whether QuickDraw GX should use or override the default priority.

`gxOverrideLimits`

This bit specifies whether or not QuickDraw GX overrides grow and shrink limits. If the bit is set, the grow and shrink limits in

the width delta structure are used for the glyphs this width delta structure applies to. If it is cleared, QuickDraw GX uses the default grow and shrink limits for those glyphs. If it is cleared, the limits values must also be set to 0. The use of this flag is to determine whether QuickDraw GX should use or override the default priority.

`gxOverrideUnlimited`

This bit specifies whether or not QuickDraw GX applies the `gxUnlimitedGapAbsorption` justification flag. If the bit is set, the state of the `gxUnlimitedGapAbsorption` flag is taken into account. If it is cleared, the `gxUnlimitedGapAbsorption` flag must also be set to 0. The use of this flag is to determine whether QuickDraw GX should use or override the default priority.

`gxUnlimitedGapAbsorption`

If this flag is set, QuickDraw GX distributes all remaining justification gap to the glyphs this width delta structure applies to, even if that would violate the grow or shrink limits. If this field is not zero, you must also set the `gxOverrideUnlimited` bit.

`gxJustificationPriorityMask`

Identifies the new justification priority for the glyphs this width delta structure applies to. Only a single valid justification priority value, as defined on page 9-60, is permitted. If this flag is set, the `gxOverrideLimits` bit must also be set.

All bits in the justification flags value not accounted for by the above constants must be set to 0.

You set the `gxOverridePriority`, `gxOverrideLimits`, or `gxOverrideUnlimited` bits to choose which aspects of font-specified justification behavior to override. For example, to change the priority of white space glyphs to be the same as intercharacter priority, set `growFlags` to have the `gxOverridePriority` bit and `gxJustificationPriorityMask` to `gxInterCharPriority`. If you clear the `gxOverridePriority` bit, the value in the `gxJustificationPriorityMask` flag is ignored, and the font-specified justification priority applies. Note that you must also clear the `gxJustificationPriorityMask` bits.

As another example, to change the grow limits of the glyphs to which a width delta structure applies, set the `gxOverrideLimits` bit in the justification flags of the `growFlags` field and specify the new grow limits in the `beforeGrowLimit` and `afterGrowLimit` fields.

Priority Justification Override Structure

A priority justification override structure specifies overriding justification behavior for all of the glyphs of a given style run. It contains an array of width delta structures, one for each justification priority.

```
typedef struct {
    gxWidthDeltaRecord    deltas[gxNumberOfJustificationPriorities];
} gxPriorityJustificationOverride;
```

Layout Line Control

Field descriptions

deltas The array of width delta structures. There is one width delta structure for each priority level, in index order.

The width delta structure is described on page 9-61. This structure is the primary input to the `GXSetStyleRunPriorityJustOverride` function.

Each width delta structure in the priority justification override structure specifies overrides for all glyphs of a given justification priority. Thus, for each priority, you can

- Change the priority: assign all glyphs of one priority to another priority.
- Change the behavior: leave the priority the same, but change the priority behavior of all glyphs of that priority.
- Change both: assign all glyphs of one priority to another priority, and change their justification behavior from the defaults of either priority.

Note that if you change one priority to another and change the default behavior of all glyphs of that priority, the behavior of other glyphs already having that priority is not changed. For example, if you change all glyphs with a priority of `gxInterCharPriority` to `gxWhiteSpacePriority` and give them special behavior, glyphs that already have a priority of `gxWhiteSpacePriority` retain their default behavior. Only `gxInterCharPriority` glyphs are overridden, and so the overriding behavior applies to only those glyphs.

Unlimited gap absorption is a special case in that it applies across an entire line instead of just to a single style run. If both the `gxOverrideUnlimited` bit and the `gxUnlimitedGapAbsorption` flag are set in any width delta structure for the glyph justification override structure of any style run on a line, QuickDraw GX distributes the current justification gap among all instances of that glyph in all style runs on the line

Glyph Justification Override Structure

The glyph justification override structure assigns an overriding justification priority and behavior to a specific glyph in a style run. It contains a glyph code and a width delta structure.

```
typedef struct {
    gxGlyphcode      glyph;
    gxWidthDeltaRecord  override;
} gxGlyphJustificationOverride;
```

Field descriptions

glyph The glyph code that specifies the glyph in the font.

override The width delta structure to use for that glyph.

The width delta structure is described on page 9-61. An array of glyph justification override structures is the primary input to the `GXSetStyleRunGlyphJustOverrides` function, described on page 9-79.

The width delta structure in the glyph justification override structure specifies overrides for all instances of a single glyph (specified by glyph code). Thus, for that glyph, you can

- change the glyph's justification priority
- change the glyph's justification behavior: leave its priority the same, but make its behavior different from that of other glyphs of the same priority
- change both: give it a new priority, and change its justification behavior from the defaults of either priority

Changing a glyph's priority and giving it non-default behavior has no effect on the behavior of other glyphs of either the old or new priority.

Unlimited gap absorption is a special case in that it applies across an entire line instead of just to a single style run. If both the `gxOverrideUnlimited` bit and the `gxUnlimitedGapAbsorption` flag are set in any width delta structure for the glyph justification override structure of any style run on a line, QuickDraw GX distributes the current justification gap among all instances of that glyph in all style runs on the line.

Functions

This section describes the functions with which you can manipulate characteristics of lines of text in layout shapes. You can use these functions to

- manipulate baselines
- measure line span
- break lines
- override the behaviors of justification priorities
- override the justification behaviors of individual glyphs

Manipulating Baselines

The function described in this section allows you to retrieve the distances among baselines for a given line of text.

GXGetStyleBaselineDeltas

You can use the `GXGetStyleBaselineDeltas` function to retrieve the distances from $y = 0$ to each of the other baseline types.

```
void GXGetStyleBaselineDeltas(gxStyle baseStyle,
                             gxBaselineType baseType,
                             gxBaselineDeltas returnedDeltas);
```

baseStyle A reference to the style object whose baseline positions are to control placement of all glyphs for the line of text.

baseType The primary baseline—that is, the baseline to have a y -delta of 0.

returnedDeltas A `gxBaselineDeltas` array. On return, contains the distances from $y = 0$ to each of the 32 baseline types.

DESCRIPTION

The `GXGetStyleBaselineDeltas` function constructs and returns an array of distances from $y = 0$ to each of the 32 baseline types. The distances are computed based on the font and text size specified in the style object you pass in the `baseStyle` parameter.

The style object you pass to this function typically represents the dominant style run on the line: the style run whose baselines are used to control the placement of all glyphs on the line. However, you can pass any style object reference; it need not represent any style run actually present on the line.

If you put the results of this function in the layout options structure of the layout shape, QuickDraw GX uses those baseline positions to draw all text on the line.

ERRORS, WARNINGS, AND NOTICES

Errors

`style_is_nil`
`parameter_out_of_range`

SEE ALSO

For an example of the use of this function, see Listing 9-1 on page 9-28.

Baseline types are described in the section “Baseline Types” on page 9-4. The baseline type enumeration is described on page 9-58. The baseline deltas array is described on page 9-59.

The layout options structure is described in the chapter “Layout Shapes” in this book.

Measuring Line Span

The functions described in this section allow you to get and set the span of a text line.

GXGetLayoutSpan

You can use the `GXGetLayoutSpan` function to retrieve the span of the text line for the specified layout shape.

```
void GXGetLayoutSpan(gxShape layout, Fixed *lineAscent,
                    Fixed *lineDescent);
```

`layout` A reference to the layout shape whose line span you need.

`lineAscent` A pointer to a `Fixed` value. On return, the value is the distance, in points, from `y = 0` to the highest ascent line in this layout shape. If you pass `nil` for this parameter, no value is returned in it.

`lineDescent` A pointer to a `Fixed` value. On return, the value is the distance, in points, from `y = 0` to the lowest descent line in this layout shape. If you pass `nil` for this parameter, no value is returned in it.

DESCRIPTION

The `GXGetLayoutSpan` function returns the line span (height for horizontal lines; width for vertical lines) for the specified layout shape. The line span is the distance, orthogonal to the baseline, from the lowest descent line to the highest ascent line in the shape. In calculating line span, `GXGetLayoutSpan` takes into account all text sizes on the line, as well as any cross-stream shifting, cross-stream kerning, multiple baselines, and glyph substitution that may have occurred.

`GXGetLayoutSpan` returns its results in points, which for QuickDraw GX are exactly 72 per inch. (Standard typographic points are 72.27 per inch.)

You can use the information returned by this function to position lines of text when drawing. QuickDraw GX uses line span to control the heights of carets and highlight areas, and to define hit-testing regions; if you create your own carets or highlighting shapes, you can use the results of this function for that purpose also.

ERRORS, WARNINGS, AND NOTICES

Errors`shape_is_nil`

SEE ALSO

For an example of the use of this function, see Listing 9-3 on page 9-34.

You can set the line span with the `GXSetLayoutSpan` function, described next.

GXSetLayoutSpan

You can use the `GXSetLayoutSpan` function to assign a span to the text line for the specified layout shape.

```
void GXSetLayoutSpan(gxShape layout, Fixed lineAscent,
                    Fixed lineDescent);
```

`layout` A reference to the layout shape whose line span is to be set.

`lineAscent` The distance, in points, from `y = 0` to the highest ascent permitted in this layout shape.

`lineDescent` The distance, in points, from `y = 0` to the lowest descent line for this layout shape.

DESCRIPTION

The `GXSetLayoutSpan` function allows your application to specify the line span (height for horizontal lines; width for vertical lines) of a layout shape.

QuickDraw GX uses line span to control the heights of carets and highlight areas, and to define hit-testing regions. If you draw successive lines of text with a fixed line spacing, you can use `GXSetLayoutSpan` to make sure that carets, highlights, and hit-testing areas do not overlap from line to line. If you do not use the `GXSetLayoutSpan` function, then all QuickDraw GX functions use a line span that corresponds to the largest ascent and descent present in the line. That span is affected by such features as varying text sizes, cross-stream shifting, cross-stream kerning, multiple baselines, and glyph substitution.

You specify the `lineAscent` and `lineDescent` parameters in points (72 per inch). The `lineAscent` parameter is measured above (to the right of, for vertical text) the line's dominant baseline; `lineDescent` is measured below (to the left of, for vertical text) the same baseline. In Roman text both values are generally positive.

SPECIAL CONSIDERATIONS

If you call `GXGetLayoutSpan` after having altered the line span with `GXSetLayoutSpan`, the result will be the line span that you have set. If you need to recover the line span as originally calculated by QuickDraw GX, make sure you save that information before calling `GXSetLayoutSpan`.

ERRORS, WARNINGS, AND NOTICES**Errors**

`shape_is_nil`

SEE ALSO

Functions whose results are affected by the result of this function include `GXGetLayoutCaret`, `GXGetLayoutHighlight`, `GXGetLayoutVisualHighlight`, and `GXHitTestLayout`. All are described in the chapter “Layout Carets, Highlighting, and Hit-Testing” in this book.

To obtain the span of a line, use the `GXGetLayoutSpan` function, described in the previous section.

Breaking Lines

The functions described in this section allow you to locate a line break (`GXGetLayoutBreakOffset`), determine the exact display width of the broken line (`GXGetLayoutRangeWidth`), and make a new layout shape for it (`GXGetNewLayoutFromRange`).

GXGetLayoutBreakOffset

You can use the `GXGetLayoutBreakOffset` function to determine the point at which to break a line of text.

```
gxByteOffset GXGetLayoutBreakOffset(gxShape layout,
                                     gxByteOffset startOffset,
                                     Fixed lineWidth,
                                     long hyphenationCount,
                                     const gxByteOffset hyphenationPoints[],
                                     boolean *startIsStaked,
                                     gxByteOffset *priorStake,
                                     gxByteOffset *nextStake);
```

`layout` A reference to the layout shape containing the text to be broken.

`startOffset` The byte offset in the source text of the first character in the line.

Layout Line Control

<code>lineWidth</code>	The available display length, in points, for the line of text.
<code>hyphenationCount</code>	The number of entries in the <code>hyphenationPoints</code> parameter (the size of the hyphenation array). If you pass <code>nil</code> for the <code>hyphenationPoints</code> parameter, this parameter must be set to 0.
<code>hyphenationPoints</code>	An array of hyphenation points. A hyphenation point is an edge offset that your application considers a preferred point for a line break. You may pass <code>nil</code> for this parameter.
<code>startIsStaked</code>	A pointer to a Boolean value. On return, it indicates whether the edge offset passed in the <code>startOffset</code> parameter is a staked position.
<code>priorStake</code>	A pointer to a <code>gxByteOffset</code> value. On return, it specifies the edge offset of the staked position in the source text that precedes the returned offset.
<code>nextStake</code>	A pointer to a <code>gxByteOffset</code> value. On return, it specifies the edge offset of the staked position in the source text that follows the returned offset.
<i>function result</i>	The edge offset corresponding to the trailing edge of the last glyph that fits completely into the display line, given the specified starting offset, display length, and preferred hyphenation points. (<i>Last</i> means last in input order, not display order.)

DESCRIPTION

The `GXGetLayoutBreakOffset` function returns the approximate edge offset following the last character whose glyph fits completely on a line having the display length specified by `lineWidth`. The offset is only approximate because, if a ligature falls on the line boundary, `GXGetLayoutBreakOffset` does not divide the ligature into component glyphs to get the exact offset.

You can pass a hyphenation array to the function in the `hyphenationPoints` parameter. The array consists of a set of edge offsets, each of which represents a preferred point at which to break the line. The array must be sorted by increasing offset value. If you pass a hyphenation array to this function and at least one of its values falls within the range of the line being broken, the function result is one of the values in the array. If you pass `nil`, the function result is the offset corresponding to the last glyph that physically fits in the line width.

The `startIsStaked`, `priorStake`, and `nextStake` parameters help you define staked offsets in your source text. A **staked offset** is one that forms a natural break in terms of the text-layout processing performed by QuickDraw GX. Staked offsets reflect typographic concerns (for instance, do not break up ligatures, rearranged sequences of glyphs, or kerning sets) rather than adherence to the rules of hyphenation. (Technically, it means that all state machines that control layout processing are in their ground state.)

On return from this function, the `startIsStaked` parameter is set to `true` if the starting offset corresponds to a staked value, and the `priorStake` and `nextStake` parameters give the nearest staked offsets behind and ahead of the function result (whether or not they fall within the limits of the line). A value of `gxNoStake` (–1) in either one of these parameters means that QuickDraw GX has not found an adjacent staked location on that line.

If you need to perform text layout with maximum efficiency, you can use this information to start and end lines at staked offsets. Most applications, however, can pass `nil` for `startIsStaked`, `priorStake`, and `nextStake`.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`parameter_out_of_range`
`count_is_less_than_zero`
`inconsistent_parameters`

SEE ALSO

For an example of the use of this function, see Listing 9-3 on page 9-34.

After obtaining a line-break position, you can determine the exact width of the line up to that position by calling the `GXGetLayoutRangeWidth` function, described next. Then you can use the `GXNewLayoutFromRange` function, described on page 9-72, to create a new layout shape from that range of text.

GXGetLayoutRangeWidth

You can use the `GXGetLayoutRangeWidth` function to return the exact display length (width for horizontal text; height for vertical text) of a portion of the text in a layout shape.

```
Fixed GXGetLayoutRangeWidth(gxShape layout,
                             gxByteOffset startOffset,
                             gxByteOffset endOffset,
                             gxShape supplementaryText);
```

`layout` A reference to the layout shape containing the text whose display length is to be determined.

`startOffset` The edge offset in the source text before the first character to include.

`endOffset` The edge offset in the source text after the last character to include.

Layout Line Control

`supplementaryText`

A reference to a layout shape that contains any text, such as a hyphen, that you want to add to the text being measured by this function. Pass `nil` for this parameter if you do not want to add any text.

function result The exact display length, in points, of the portion of the layout shape between the two offsets you specify.

DESCRIPTION

The `GXGetLayoutRangeWidth` function takes two offsets within a layout shape and returns the exact length, in points (72 per inch), of that portion of the layout shape. The `GXGetLayoutRangeWidth` function can also take into account any supplementary text you may want to add to this part of the layout shape, such as a hyphen or a more complex structure.

For example, “Zuk-ker” is the correct hyphenation of the German word “Zucker.” Thus, you need to delete the “c” and add a “k-”, instead of just adding a hyphen. In this case, you set the `endOffset` parameter so that it doesn’t include the “c”, and you specify a shape containing “k-” in the `supplementaryText` parameter.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`parameter_out_of_range`

SEE ALSO

Before calling `GXGetLayoutRangeWidth`, you can obtain a text range for line breaking with the `GXGetLayoutBreakOffset` function, described on page 9-69. You can create a new layout shape from that range of text by calling the `GXNewLayoutFromRange` function, described next.

GXNewLayoutFromRange

You can use the `GXNewLayoutFromRange` function to create a new layout shape from a range of text within an existing layout shape. Typically, the new shape represents a single line of text whose limits have been determined by a call to the `GXGetLayoutBreakOffset` function.

```
gxShape GXNewLayoutFromRange(gxShape layout,
                             gxByteOffset startOffset,
                             gxByteOffset endOffset,
                             const gxLayoutOptions *layoutOptions,
                             gxShape supplementaryText);
```

`layout` A reference to the layout shape containing the range of text to be used.

Layout Line Control

`startOffset`

The edge offset in the source text before the first character to include in the new shape.

`endOffset`

The edge offset in the source text after the last character to include in the new shape.

`layoutOptions`

A pointer to a layout options structure containing the layout options you want to apply to the new layout shape.

`supplementaryText`

A reference to a layout shape that contains any text, such as a hyphen, that you want to add to the new layout shape. Pass `nil` for this parameter if you do not want to add any text.

function result A reference to the new layout shape.

DESCRIPTION

The `GXNewLayoutFromRange` function takes a range of source text from a layout shape plus any additional text you include, and returns a new layout shape. You usually call the `GXNewLayoutFromRange` function after first calling the `GXGetLayoutBreakOffset` function, and possibly the `GXGetLayoutRangeWidth` function.

ERRORS, WARNINGS, AND NOTICES

Errors`shape_is_nil``parameter_out_of_range`

SEE ALSO

For an example of the use of this function, see Listing 9-3 on page 9-34.

You can obtain a text position for line breaking by calling the `GXGetLayoutBreakOffset` function, described on page 9-69. After obtaining a line-break position, you can determine the exact width of the line up to that position by calling the `GXGetLayoutRangeWidth` function, described on page 9-71.

Overriding the Behaviors of Justification Priorities

The functions described in this section allow you to override justification behavior of classes of glyphs, based on justification priority. You can specify either the style object whose behavior is to be changed (`GXGetStyleRunPriorityJustOverride`, `GXSetStyleRunPriorityJustOverride`) or the shape object whose associated style object is to be altered (`GXGetShapeRunPriorityJustOverride`, `GXSetShapeRunPriorityJustOverride`).

GXGetStyleRunPriorityJustOverride

You can use the `GXGetStyleRunPriorityJustOverride` function to retrieve a priority justification override structure from a style object.

```
long GXGetStyleRunPriorityJustOverride(gxStyle source,
                                       gxPriorityJustificationOverride
                                       *priorityJustificationOverride);
```

source A reference to the style object whose priority justification override structure you need.

priorityJustificationOverride
 A pointer to a priority justification override structure. On return, the structure contains the priority justification overrides for the style object. You can pass `nil` for this parameter if you do not need to retrieve the priority justification override structure itself.

function result If the style object has a priority justification override structure, this value is 1. Otherwise, it is 0.

DESCRIPTION

The `GXGetStyleRunPriorityJustOverride` function returns the priority justification override structure from the style object you specify in the `source` parameter.

ERRORS, WARNINGS, AND NOTICES

Errors

`style_is_nil`

SEE ALSO

To set the priority justification overrides for a style run, use the `GXSetStyleRunPriorityJustOverride` function, described next.

To retrieve the priority justification overrides associated with a particular shape, use the `GXGetShapeRunPriorityJustOverride` function, described on page 9-76.

To set the priority justification overrides associated with a particular shape, use the `GXSetShapeRunPriorityJustOverride` function, described on page 9-77.

To retrieve the glyph justification overrides for a style run, use the `GXGetStyleRunGlyphJustOverrides` function, described on page 9-78.

GXSetStyleRunPriorityJustOverride

You can use the `GXSetStyleRunPriorityJustOverride` function to assign a priority justification override structure to a style object (or to remove it from the style object).

```
void GXSetStyleRunPriorityJustOverride(gxStyle target,
                                       const gxPriorityJustificationOverride
                                       *priorityJustificationOverride);
```

target A reference to the style object whose priority justification override structure you want to modify.

priorityJustificationOverride
A pointer to the priority justification override structure to assign.

DESCRIPTION

The `GXSetStyleRunPriorityJustOverride` function copies the specified `gxPriorityJustificationOverride` structure into the priority justification override property of the style object specified in the `target` parameter.

If you pass `nil` for the `priorityJustificationOverride` parameter, the function removes the priority justification override structure (if any) from the style object.

ERRORS, WARNINGS, AND NOTICES

Errors

`style_is_nil`
`inconsistent_parameters`

Notices (debugging version)

`attributes_already_set`

SEE ALSO

For examples of the use of this function, see the unnumbered code listing on page 9-52 and Listing 9-9 on page 9-52.

To retrieve the priority justification overrides for a style run, use the `GXGetStyleRunPriorityJustOverride` function, described on page 9-74.

To retrieve the priority justification overrides associated with the style associated with a particular shape, use the `GXGetShapeRunPriorityJustOverride` function, described next. To set the priority justification overrides associated with the style associated with a particular shape, use the `GXSetShapeRunPriorityJustOverride` function, described on page 9-77.

To set the glyph justification overrides for a style run, use the `GXSetStyleRunGlyphJustOverrides` function, described on page 9-79.

GXGetShapeRunPriorityJustOverride

You can use the `GXGetShapeRunPriorityJustOverride` function to retrieve the priority justification override structure from the style object associated with a shape.

```
long GXGetShapeRunPriorityJustOverride(gxShape source,
                                       gxPriorityJustificationOverride
                                       *priorityJustificationOverride);
```

source A reference to the shape object whose associated style object contains the priority justification override structure you need.

priorityJustificationOverride
 A pointer to a priority justification override structure. On return, the structure contains the priority justification overrides for the style object associated with the specified shape. You can pass `nil` for this parameter if you do not need to retrieve the priority justification override structure itself.

function result If the style object associated with the specified shape has a priority justification override structure, this value is 1. Otherwise, it is 0.

DESCRIPTION

The `GXGetShapeRunPriorityJustOverride` function returns the priority justification override structure from the style object associated with the shape you specify in the `source` parameter.

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the `GXNewLayout` function, for example, you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as `GXGetStyleRunPriorityJustOverride`.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`

SEE ALSO

To set the priority justification overrides associated with a particular shape, use the `GXSetShapeRunPriorityJustOverride` function, described next.

To retrieve the priority justification overrides for a style run, use the `GXGetStyleRunPriorityJustOverride` function, described on page 9-74.

To set the priority justification overrides for a style run, use the `GXSetStyleRunPriorityJustOverride` function, described on page 9-75.

To retrieve the glyph justification overrides associated with a particular shape, use the `GXGetShapeRunGlyphJustOverrides` function, described on page 9-81.

GXSetShapeRunPriorityJustOverride

You can use the `GXSetShapeRunPriorityJustOverride` function to assign a priority justification override structure to the style object associated with a shape (or to remove it from the style object).

```
void GXSetShapeRunPriorityJustOverride(gxShape target, const
                                      gxPriorityJustificationOverride
                                      *priorityJustificationOverride);
```

target A reference to the shape object whose associated style object contains the priority justification override structure you want to modify.

priorityJustificationOverride
 A pointer to the priority justification override structure to assign.

DESCRIPTION

The `GXSetShapeRunPriorityJustOverride` function copies the specified `gxPriorityJustificationOverride` structure into the priority justification override property of the style object associated with the shape specified in the `target` parameter.

If you pass `nil` for the `priorityJustificationOverride` parameter, the function removes the priority justification override structure (if any) from the style object.

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the `GXNewLayout` function, for example, you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as `GXSetStyleRunPriorityJustOverride`.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`

Notices (debugging version)

`attributes_already_set`

SEE ALSO

To retrieve the priority justification overrides associated with a particular shape, use the `GXGetShapeRunPriorityJustOverride` function, described on page 9-76.

To retrieve the priority justification overrides for a style run, use the `GXGetStyleRunPriorityJustOverride` function, described on page 9-74.

To set the priority justification overrides for a style run, use the `GXSetStyleRunPriorityJustOverride` function, described on page 9-75.

To set the glyph justification overrides associated with a particular shape, use the `GXSetShapeRunGlyphJustOverrides` function, described on page 9-82.

Overriding the Justification Behaviors of Individual Glyphs

The functions described in this section allow you to override the justification behavior of individual glyphs in a style run. You can specify either the style object associated with the glyphs to be changed (`GXGetStyleRunGlyphJustOverrides`, `GXSetStyleRunGlyphJustOverrides`) or the shape object whose style object is associated with the glyphs (`GXGetShapeRunGlyphJustOverrides`, `GXSetShapeRunGlyphJustOverrides`).

GXGetStyleRunGlyphJustOverrides

You can use the `GXGetStyleRunGlyphJustOverrides` function to retrieve an array of glyph justification override structures from a style object.

```
long GXGetStyleRunGlyphJustOverrides(gxStyle source,
                                     gxGlyphJustificationOverride
                                     glyphJustificationOverrides[]);
```

source A reference to the style object whose array of glyph justification overrides you need.

glyphJustificationOverrides An array of glyph justification override structures. On return, the array contains the glyph justification overrides for the style object. If you pass `nil` for this parameter, nothing is returned in this array. However, the function result is still the correct number of glyph justification override structures in the style.

function result The number of glyph justification override structures in the style object.

DESCRIPTION

The `GXGetStyleRunGlyphJustOverrides` function returns the number of glyph justification override structures currently contained in the style object you specify in the `source` parameter.

To get the overrides themselves, you need to allocate an array to pass in the `glyphJustificationOverrides` parameter when calling this function. To get the right size for the array, you can first call the function with a value of `nil` for the `glyphJustificationOverrides` parameter. Then use the function result to allocate an array of the proper size and call `GXGetStyleRunGlyphJustOverrides` a second time, this time passing the array.

SPECIAL CONSIDERATIONS

Because QuickDraw GX can reorder the elements in a style object's glyph-justification overrides array, the order of elements returned in the `glyphJustificationOverrides` parameter to this function may differ from the order in which they were originally assigned to the style.

ERRORS, WARNINGS, AND NOTICES

Errors

`style_is_nil`

SEE ALSO

To set the glyph justification overrides for a style run, use the `GXSetStyleRunGlyphJustOverrides` function, described next.

To retrieve the glyph justification overrides associated with a particular shape, use the `GXGetShapeRunGlyphJustOverrides` function, described on page 9-81.

To set the glyph justification overrides associated with a particular shape, use the `GXSetShapeRunGlyphJustOverrides` function, described on page 9-82.

To retrieve the priority justification overrides for a style run, use the `GXGetStyleRunPriorityJustOverride` function, described on page 9-74.

GXSetStyleRunGlyphJustOverrides

You can use the `GXSetStyleRunGlyphJustOverrides` function to assign an array of glyph justification override structures to a style object, or to remove all glyph justification overrides from it.

```
void GXSetStyleRunGlyphJustOverrides(gxStyle target, long count,
                                     const gxGlyphJustificationOverride
                                     glyphJustificationOverrides[]);
```

target A reference to the style object whose glyph justification overrides you want to modify.

count The number of glyph justification override structures you want to assign.

Layout Line Control

`glyphJustificationOverrides`

The array of glyph justification override structures to be assigned to the style object.

DESCRIPTION

The `GXSetStyleRunGlyphJustOverrides` function copies an array of `gxGlyphJustificationOverride` structures into the glyph justification overrides property of the specified style object.

If you pass `nil` for the `glyphJustificationOverrides` parameter and 0 for the count parameter, the function removes all glyph justification override structures from the style object.

If count is 0 and `glyphJustificationOverrides` is non-`nil`, or if count is nonzero and `glyphJustificationOverrides` is `nil`, `GXSetStyleRunGlyphJustOverrides` posts an `inconsistent_parameters` error.

ERRORS, WARNINGS, AND NOTICES

Errors

`style_is_nil`
`parameter_out_of_range`
`count_is_less_than_zero`
`inconsistent_parameters`

Notices (debugging version)

`attributes_already_set`

SEE ALSO

For an example of the use of this function, see Listing 9-9 on page 9-52.

To retrieve the glyph justification overrides for a style run, use the `GXGetStyleRunGlyphJustOverrides` function, described on page 9-78.

To retrieve the glyph justification overrides associated with the style associated with a particular shape, use the `GXGetShapeRunGlyphJustOverrides` function, described next. To set the glyph justification overrides associated with the style associated with a particular shape, use the `GXSetShapeRunGlyphJustOverrides` function, described on page 9-82.

To set the priority justification overrides for a style run, use the `GXSetStyleRunPriorityJustOverride` function, described on page 9-75.

GXGetShapeRunGlyphJustOverrides

You can use the `GXGetShapeRunGlyphJustOverrides` function to retrieve an array of glyph justification override structures from the style object referenced by a shape.

```
long GXGetShapeRunGlyphJustOverrides(gxShape source,
                                     gxGlyphJustificationOverride
                                     glyphJustificationOverrides[]);
```

source A reference to the shape object whose associated style object contains the glyph justification overrides you need.

glyphJustificationOverrides

An array of glyph justification override structures. On return, the array contains the glyph justification overrides for the style object associated with the specified shape. If you pass `nil` for this parameter, nothing is returned in this array. However, the function result is still the correct number of glyph justification override structures in the style associated with the shape.

function result The number of glyph justification override structures in the style object associated with the specified shape.

DESCRIPTION

The `GXGetShapeRunGlyphJustOverrides` function returns the number of glyph justification override structures currently contained in the style object of the shape you specify in the `source` parameter.

To get the overrides themselves, you need to allocate an array to pass in the `glyphJustificationOverrides` parameter when calling this function. To get the right size for the array, you can first call the function with a value of `nil` for the `glyphJustificationOverrides` parameter. Then use the function result to allocate an array of the proper size and call `GXGetShapeRunGlyphJustOverrides` a second time, this time passing the array.

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the `GXNewLayout` function, for example, you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as `GXGetStyleRunGlyphJustOverrides`.

SPECIAL CONSIDERATIONS

Because QuickDraw GX can reorder the elements in a style object's glyph-justification overrides array, the order of elements returned in the `glyphJustificationOverrides` parameter to this function may differ from the order in which they were originally assigned to the style.

ERRORS, WARNINGS, AND NOTICES**Errors**

`shape_is_nil`

SEE ALSO

To set the glyph justification overrides associated with a particular shape, use the `GXSetShapeRunGlyphJustOverrides` function, described next.

To retrieve the glyph justification overrides for a style run, use the `GXGetStyleRunGlyphJustOverrides` function, described on page 9-78.

To set the glyph justification overrides for a style run, use the `GXSetStyleRunGlyphJustOverrides` function, described on page 9-79.

To retrieve the priority justification overrides associated with a particular shape, use the `GXGetShapeRunPriorityJustOverride` function, described on page 9-76.

GXSetShapeRunGlyphJustOverrides

You can use the `GXSetShapeRunGlyphJustOverrides` function to assign an array of glyph justification override structures to the style object associated with a shape, or to remove all glyph justification overrides from it.

```
void GXSetShapeRunGlyphJustOverrides(gxShape target, long count,
                                     const gxGlyphJustificationOverride
                                     glyphJustificationOverrides[]);
```

<code>target</code>	A reference to the shape object whose style object contains the glyph justification overrides you want to modify.
<code>count</code>	The number of glyph justification override structures you want to assign.
<code>glyphJustificationOverrides</code>	The array of glyph justification override structures to be assigned to the style object associated with the specified shape.

DESCRIPTION

The `GXSetShapeRunGlyphJustOverrides` function copies an array of glyph justification override structures into the glyph justification overrides property of the style object associated with the shape you specify in the `target` parameter.

If you pass `nil` for the `glyphJustificationOverrides` parameter and 0 for the `count` parameter, the function removes all glyph justification override structures from the style object.

This function acts only on the single style object that is referenced in the style property of the shape object. It does not access any style object in the style list, which is part of the geometry of a layout shape. (If, when calling the `GXNewLayout` function, for example, you pass `nil` for the `styles` parameter, no style list is created and the new layout shape's style object is referenced through the style property.) If your layout shape uses more than one style object, and therefore uses a style list in its geometry, you need to access those style objects directly with a function such as `GXSetStyleRunGlyphJustOverrides`.

If `count` is 0 and `glyphJustificationOverrides` is non-`nil`, or if `count` is nonzero and `glyphJustificationOverrides` is `nil`, `GXSetShapeRunGlyphJustOverrides` posts an `inconsistent_parameters` error.

ERRORS, WARNINGS, AND NOTICES**Errors**

`shape_is_nil`
`parameter_out_of_range`
`count_is_less_than_zero`
`inconsistent_parameters`

Notices (debugging version)

`attributes_already_set`

SEE ALSO

To retrieve the glyph justification overrides associated with a particular shape, use the `GXGetShapeRunGlyphJustOverrides` function, described on page 9-81.

To retrieve the glyph justification overrides for a style run, use the `GXGetStyleRunGlyphJustOverrides` function, described on page 9-78.

To set the glyph justification overrides for a style run, use the `GXSetStyleRunGlyphJustOverrides` function, described on page 9-79.

To set the priority justification overrides associated with the style associated with a particular shape, use the `GXSetShapeRunPriorityJustOverride` function, described on page 9-77.

Summary of Layout Line Control

Constants and Data Types

Baseline Types

```
enum {
    gxRomanBaseline           = 0,
    gxIdeographicCenterBaseline,
    gxIdeographicLowBaseline,
    gxHangingBaseline,
    gxMathBaseline,
    gxLastBaseline           = 31,
    gxNumberOfBaselineTypes  = gxLastBaseline + 1,
    gxNoOverrideBaseline     = 255
};
```

```
typedef unsigned long gxBaselineType;
```

Baseline Deltas Array

```
typedef Fixed gxBaselineDeltas[gxNumberOfBaselineTypes];
```

Baseline Structure

```
typedef struct {
    gxBaselineDeltas  deltas;
} gxLineBaselineRecord;
```

Justification Priorities

```
enum {
    gxKashidaPriority          = 0,
    gxWhiteSpacePriority       = 1,
    gxInterCharPriority        = 2,
    gxNullJustificationPriority = 3,
    gxNumberOfJustificationPriorities
};
```

```
typedef unsigned char gxJustificationPriority;
```

Width Delta Structure

```
typedef struct {
    Fixed          beforeGrowLimit;
    Fixed          beforeShrinkLimit;
    Fixed          afterGrowLimit;
    Fixed          afterShrinkLimit;
    gxJustificationFlags growFlags;
    gxJustificationFlags shrinkFlags;
} gxWidthDeltaRecord;
```

Justification Flags

```
enum {
    gxOverridePriority          = 0x8000,
    gxOverrideLimits           = 0x4000,
    gxOverrideUnlimited         = 0x2000,
    gxUnlimitedGapAbsorption    = 0x1000,
    gxJustificationPriorityMask = 0x000F
    gxAllJustificationFlags    = gxOverridePriority |
                                gxOverrideLimits |
                                gxOverrideUnlimited |
                                gxUnlimitedGapAbsorption |
                                gxJustificationPriorityMask
};
```

```
typedef unsigned short gxJustificationFlags;
```

Priority Justification Override Structure

```
typedef struct {
    gxWidthDeltaRecord  deltas[gxNumberOfJustificationPriorities];
} gxPriorityJustificationOverride;
```

Glyph Justification Override Structure

```
typedef struct {
    gxGlyphcode         glyph;
    gxWidthDeltaRecord  override;
} gxGlyphJustificationOverride;
```

Functions

Manipulating Baselines

```
void GXGetStyleBaselineDeltas
    (gxStyle baseStyle, gxBaselineType baseType,
     gxBaselineDeltas returnedDeltas);
```

Measuring Line Span

```
void GXGetLayoutSpan    (gxShape layout, Fixed *lineAscent,
                        Fixed *lineDescent);

void GXSetLayoutSpan    (gxShape layout, Fixed lineAscent,
                        Fixed lineDescent);
```

Breaking Lines

```
gxByteOffset GXGetLayoutBreakOffset
    (gxShape layout, gxByteOffset startOffset,
     Fixed lineWidth, long hyphenationCount,
     const gxByteOffset hyphenationPoints[],
     boolean *startIsStaked,
     gxByteOffset *priorStake,
     gxByteOffset *nextStake);

Fixed GXGetLayoutRangeWidth (gxShape layout, gxByteOffset startOffset,
                             gxByteOffset endOffset,
                             gxShape supplementaryText);

gxShape GXNewLayoutFromRange (gxShape layout, gxByteOffset startOffset,
                             gxByteOffset endOffset,
                             const gxLayoutOptions *layoutOptions,
                             gxShape supplementaryText);
```

Overriding the Behaviors of Justification Priorities

```
long GXGetStyleRunPriorityJustOverride
    (gxStyle source,
     gxPriorityJustificationOverride
     *priorityJustificationOverride);

void GXSetStyleRunPriorityJustOverride
    (gxStyle target,
     const gxPriorityJustificationOverride
     *priorityJustificationOverride);

long GXGetShapeRunPriorityJustOverride
    (gxShape source,
     gxPriorityJustificationOverride
     *priorityJustificationOverride);
```

```
void GXSetShapeRunPriorityJustOverride
    (gxShape target,
     const gxPriorityJustificationOverride
     *priorityJustificationOverride);
```

Overriding the Justification Behaviors of Individual Glyphs

```
long GXGetStyleRunGlyphJustOverrides
    (gxStyle source,
     gxGlyphJustificationOverride
     glyphJustificationOverrides[]);

void GXSetStyleRunGlyphJustOverrides
    (gxStyle target, long count,
     const gxGlyphJustificationOverride
     glyphJustificationOverrides[]);

long GXGetShapeRunGlyphJustOverrides
    (gxShape source,
     gxGlyphJustificationOverride
     glyphJustificationOverrides[]);

void GXSetShapeRunGlyphJustOverrides
    (gxShape target, long count,
     const gxGlyphJustificationOverride
     glyphJustificationOverrides[]);
```

