

Layout Shapes

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

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

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

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

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

About Layout Shapes

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

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

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

Layout Shapes

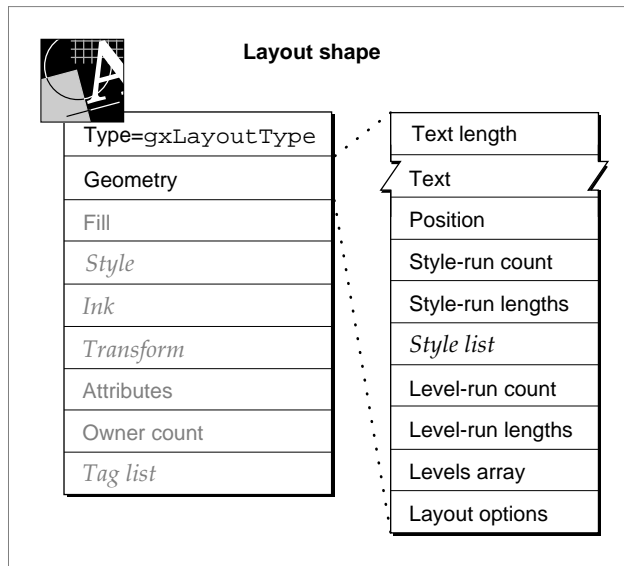
A layout shape generally describes a single line of text. It is not suited for large blocks of text, such as a paragraph. However, you can use QuickDraw GX functions to break up a paragraph into several layouts that your application can work with.

Properties of the Layout Shape

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

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

Figure 5-1 Geometry of a layout shape



The geometry of a layout shape contains these elements:

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

Layout Shapes

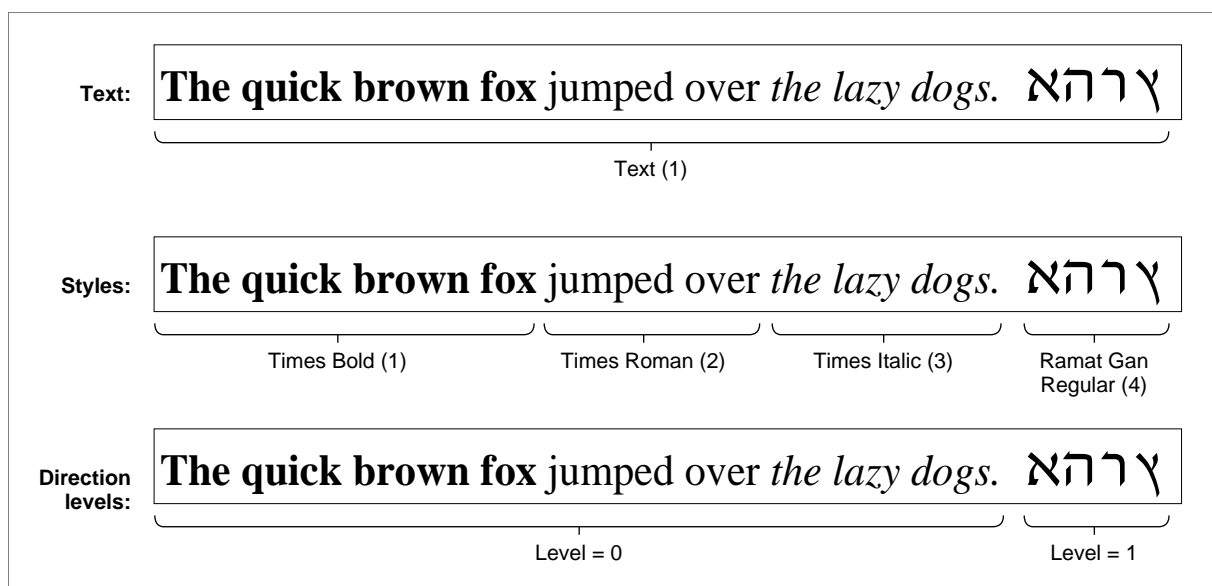
- **Style list.** An array of references to the style objects for each of the style runs. If the layout shape has only one style run or is `nil`, its style list may be empty, and the one style object may be referenced instead in the `style` property of the shape object itself.
- **Level-run count.**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

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

Runs in a Layout Shape

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

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



Text Runs

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

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

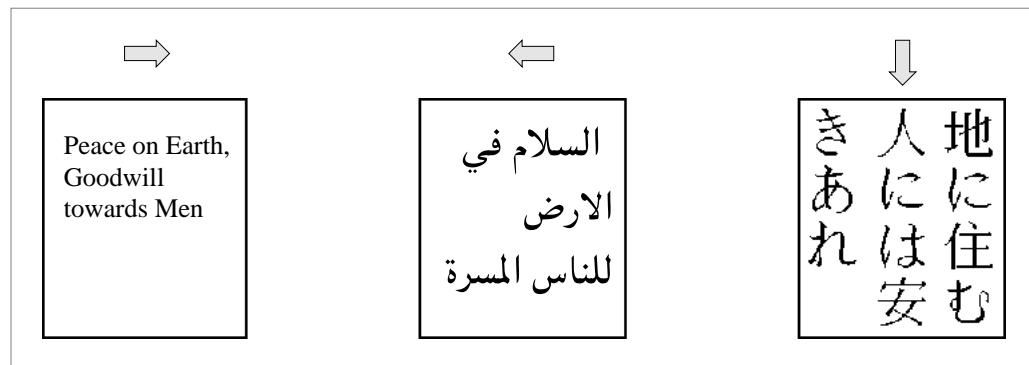
QuickDraw GX assumes that the character codes of layout shape’s source text are always stored in **input order**—the order in which characters are typically entered when the user creates a document. (Note that sometimes the term *phonetic order* is used as a synonym for *input order*.) This may be different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

Note

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

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

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



Layout Shapes

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

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

Style Runs

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

Note

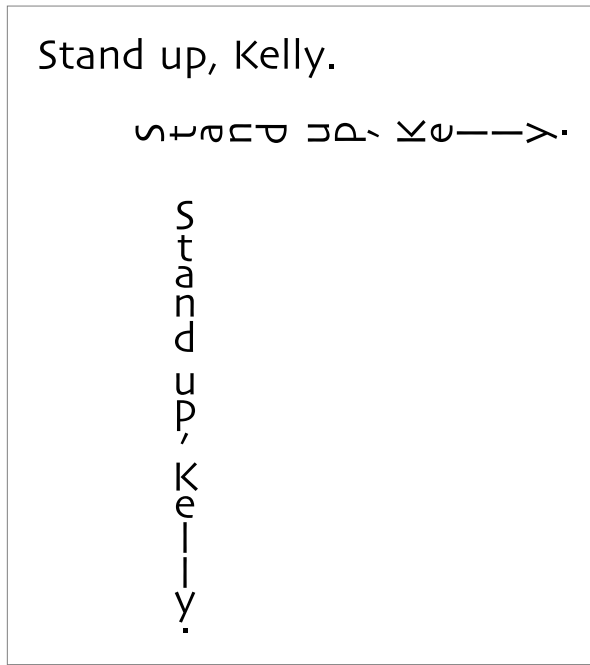
If the geometry of a layout shape contains the style object or objects used with the layout shape, the function `GXSetShapeStyle`, which affects only the style object referenced in the style property of the shape, has no effect on the look of the layout shape. To change the styles of the shape, you must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

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

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

Vertical Text in a Layout Shape

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

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

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

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

Font Features

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

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

Direction-Level Runs

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

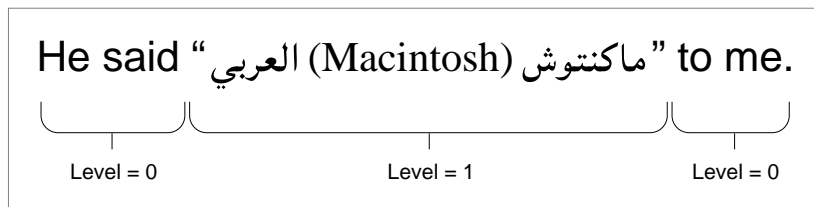
To help QuickDraw GX determine the display order of the characters in a layout shape containing text running in multiple directions, you can create direction-level runs by supplying a **direction-level** code to each block of text that runs in a different direction. The direction-level code determines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

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

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

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

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



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

Layout Options

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

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

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

Width

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

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

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

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

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

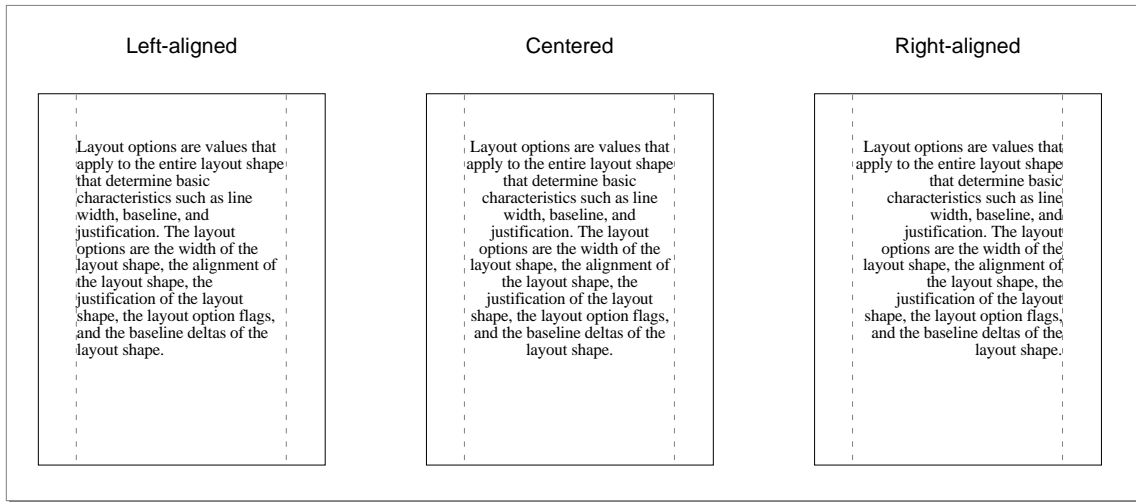
Alignment

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

Layout Shapes

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

Figure 5-6 Types of alignment



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

Figure 5-7 Use of the `flush` field



Layout Shapes

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

Justification

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

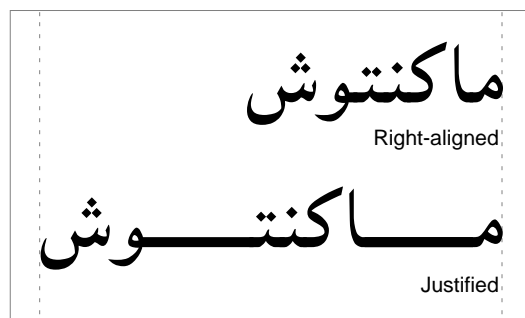
QuickDraw GX justifies Roman text primarily by adjusting the white space between words and glyphs, as shown in Figure 5-8. Note that white space is added not only between words but also between glyphs. Also notice that ligatures such as “fi” and “ffi” can be broken during justification.

Figure 5-8 Alignment and justification in English



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

Figure 5-9 Alignment and justification in Arabic



Layout Shapes

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

Figure 5-10 Use of the `just` field

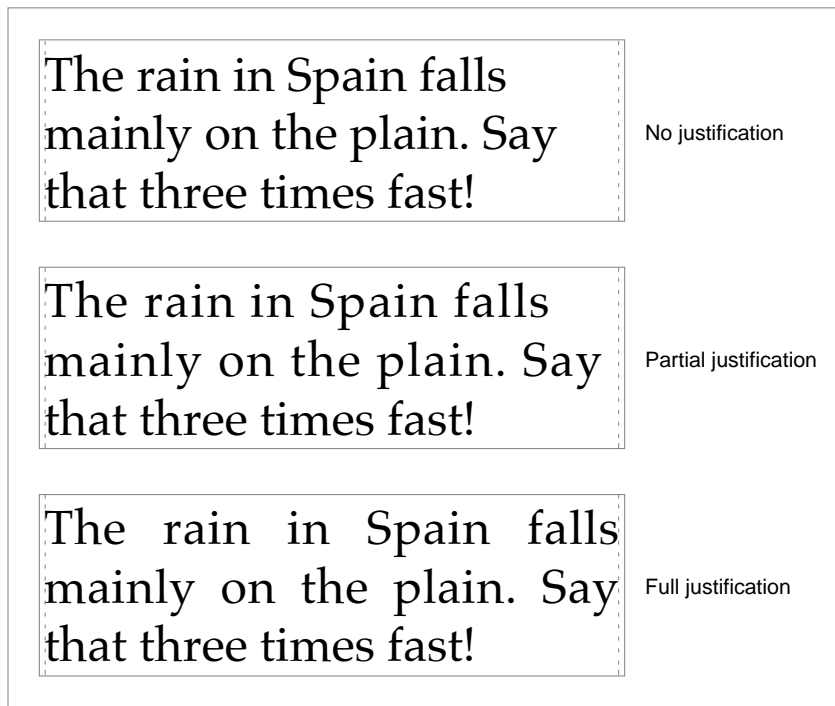


Figure 5-11 shows how the `just` field and `flush` (alignment) field of the layout options structure interact with values ranging from 0.0 to 1.0. Note that when the user chooses full justification—that is, when the `just` field equals 1.0—the value in the `flush` field has no effect.

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

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

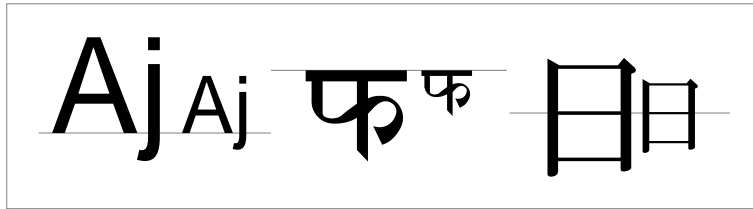


Baselines

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

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

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



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

Flags

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

The layout options flags can have the following settings:

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

The Default Layout Shape

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

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

Using Layout Shapes

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

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

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

Creating and Drawing a Layout Shape

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

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

Layout Shapes

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

Listing 5-1 Creating and drawing a layout shape

```

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

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

```

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

```

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

```

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

Creating a Layout Shape With Multiple Style Runs

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

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

Layout Shapes

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

Listing 5-2 Creating a line containing multiple style runs

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

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

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

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

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

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

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

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

Figure 5-13 A layout shape with multiple style runs

Positioning a Layout Shape

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

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

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

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

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

Changing Parts of an Existing Layout Shape

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

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

Layout Shapes

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

Changing Text in a Layout Shape

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

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

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

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

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

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

```
gxShape myLayout;  
char *originalText = "ABC";  
char *newText = " DEF";  
short originalLen, newLen;  
gxPoint position = {ff(100), ff(100)};  
  
originalLen = strlen(originalText);  
myLayout = GXNewLayout(1, &originalLen, (void *)&originalText,  
                        0, nil, nil,  
                        0, nil, nil,  
                        nil, &position);
```

Layout Shapes

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

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

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

Inserting a Typographic Shape Into a Layout Shape

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

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

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

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

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

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

Layout Shapes

```

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

GXDrawShape(myLayout);

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

```

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

Extracting a Layout Shape From Part of an Existing Layout Shape

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

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

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

```

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

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

newLayout = GXGetLayoutShapeParts(myLayout, 7, 15, nil);

GXDrawShape(newLayout);

GXDisposeShape(myLayout);
GXDisposeShape(newLayout);

```

Setting Layout Options

The layout options structure is described in detail on page 5-29. Values for layout options flags are described on page 5-30.

Setting the Width of a Layout Shape

You can set the width of a layout shape by setting the `width` field of the layout options structure. This field specifies the desired width, in typographic points, of the line when you draw justified or right-aligned text. However, the width of the line and the width of the (unjustified) layout shape itself may be different, and if the line is *not* justified or right-aligned, the `width` field is ignored, unless the unjustified width of the layout would exceed the specified width, in which case the layout is squeezed to fit (see Table 5-1).

Also, if the line contains glyphs with large negative side bearings, hanging punctuation, or optically aligned edges, the final width of the displayed layout shape may be different from the value specified here.

For more information and examples on line measurement and the use of the `width` field for justification, see the chapter “Layout Line Control” in this book.

Setting the Alignment of a Layout Shape

You can set the alignment of a layout shape by changing the value of the `flush` field in the layout options structure. A value of 0 specifies left alignment, a value of 1 specifies right alignment, and fractional values between 0 and 1 position the text proportional distances from the left and right margins.

Listing 5-6 creates a layout shape containing the text “A line of text”. It then aligns the string at five different positions by changing the value of the `flush` field. This code uses library functions `InitializeRunControls` and `InitializeLayoutOptions` to initialize the data structures, and `NewLayoutStyle` to create a style object.

Listing 5-6 Altering the alignment of a layout shape

```
char   *myString = "A line of text";
gxLayoutOptions layoutOptions;
gxLine myLine;
static gxPoint myPoint = {ff(30), ff(50)};
gxShape layout;
short len;
gxStyle myStyle;

len = strlen(myString);

InitializeLayoutOptions(&layoutOptions);
layoutOptions.width = ff(500);
```

Layout Shapes

```

/* The initial alignment of the layout shape is set to 0.0. */
layoutOptions.flush = 0;

myLine.first.x = myLine.last.x = myPoint.x;
myLine.first.y = 0;
myLine.last.y = ff(1000);
GXDrawLine(&myLine);

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

myStyle = NewLayoutStyle((char *) "\pTimes Roman", ff(50), 0,
                        nil, nil, 0, nil);

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

/* The alignment of the layout shape is set to 0.25. */
layoutOptions.flush = fract1 / 4;
GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
&layoutOptions, nil);
GXMoveShape(layout, 0, ff(75));
GXDrawShape(layout);

/* The alignment of the layout shape is set to 0.5. */
layoutOptions.flush = fract1 / 2;
GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
&layoutOptions, nil);
GXMoveShape(layout, 0, ff(75));
GXDrawShape(layout);

/* The alignment of the layout shape is set to 0.75. */
layoutOptions.flush = 3 * (fract1 / 4);
GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
&layoutOptions, nil);
GXMoveShape(layout, 0, ff(75));
GXDrawShape(layout);

```

Layout Shapes

```

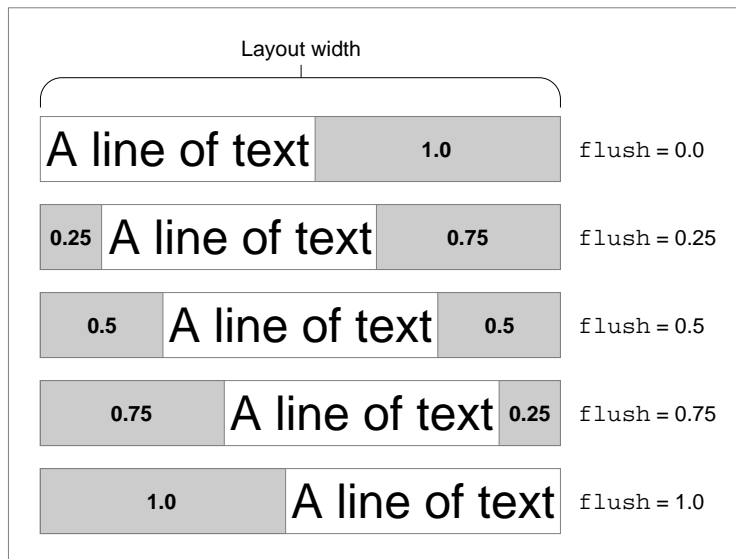
/* Here the alignment of the layout shape is set to 1.0. */
layoutOptions.flush = fract1;
GXSetLayout(layout, 0, nil, nil, 0, nil, nil, 0, nil, nil,
&layoutOptions, nil);
GXMoveShape(layout, 0, ff(75));
GXDrawShape(layout);

GXDisposeShape(layout);
GXDisposeStyle(myStyle);

```

Figure 5-14 shows the output of Listing 5-6. The numbers in the gray boxes specify what fraction of the gap appears on either side of the text.

Figure 5-14 Changing the alignment of a layout shape



See “Alignment” on page 5-11 for more information.

Justifying Text in a Layout Shape

You can control the justification of a layout shape by setting the value of the `just` field in the layout options structure. A value of 0 specifies no justification, a value of 1 specifies full justification, and fractional values between 0 and 1 distribute the extra space proportionally. The `width` field in the layout options structure defines the text width for justification.

Justification is easy to specify, but its internal workings are complex, powerful, and controlled by settings in the style object for each style run. For more information on and examples of the QuickDraw GX justification process, see the chapter “Layout Line Control” in this book.

Getting Glyph Information From a Layout Shape

In a layout shape, there can be a difference between the number of character codes in the source text and the number and order of glyphs displayed, because the layout shape can automatically form ligatures or other contextual forms. (The other typographic shapes, text and glyph, have a one-to-one correspondence between the number of character codes and the number of glyphs displayed.)

You can get information about the character codes stored in the shape using the `GXGetLayout` or `GXGetLayoutParts` function, as described in “Changing Parts of an Existing Layout Shape” on page 5-20. However, you can also get information about the glyphs in a layout shape using the `GXGetLayoutGlyphs` function, described on page 5-45.

The difference between the information returned by the `GXGetLayoutGlyphs` function and by the `GXGetLayout` function can be illustrated as follows. Take, for example, the layout shape “office”, which has a source text of six character codes but is displayed using only four glyphs because of the “ffi” ligature. If you look at the contents of the `text` parameter returned by the `GXGetLayout` function, you will see six character codes. The `GXGetLayoutGlyphs` function returns only four glyph codes—and all four may be different from the glyph codes that individually correspond to the original character codes, depending on whether alternate forms are used. Likewise, the style runs are different: there is a different count for character codes than for glyph codes.

The `GXGetLayoutGlyphs` function treats the layout shape as though it were a glyph shape—that is, it returns information about the advance bits, positions, and tangents arrays, although the layout shape does not explicitly contain any of these arrays.

The tangents array gives every entry a default value of (1.0,0.0). However, if you set the `gxVerticalText` text attribute in the layout shape, the array contains individual tangents for each glyphs.

Converting a Layout Shape Into a Glyph Shape

You can convert a layout shape into a text shape, a glyph shape, or any other type of shape.

As with all typographic shapes, you cannot convert any type of geometric shape (point, line, rectangle, and so on) into a layout shape. You can include, in the layout shape’s style list, a text face that is patterned. Text faces are described in the chapter “Typographic Styles” in this book.

You can use the `GXPrimitiveShape` function, described in *Inside Macintosh: QuickDraw GX Graphics*, to convert a layout shape into a glyph shape. This conversion lets you take advantage of the special capabilities of a glyph shape. For instance, you may want to alter the tangents or positions of the individual glyphs in the shape.

Keep in mind that, after conversion, the resulting glyph shape looks exactly the same as the layout shape, but its internal data may be very different. For example, the order of character codes in the source text is the same as the display order of the glyphs in the glyph shape, which may not have been the case with the original layout shape.

Layout Shapes

In addition, you cannot restore the original layout shape by converting the glyph shape back into a layout shape. However, you can use the glyph shape for clipping, dashing, and patterns.

Also keep in mind that, after conversion, the resulting glyph shape may not retain alignment settings, justification settings, and other information originally in the layout shape. This information is lost, for instance, if you alter the positions of glyphs in the displayed shape by calling `GXSetGlyphTangents` and other functions described in the chapter “Glyph Shapes” in this book.

Note

You can also use the `GXSetShapeType` function to convert a layout shape into a glyph shape, although the resulting shape may not look the same. The `GXSetShapeType` function does not preserve all layout functionality in the destination shape, whereas the `GXPrimitiveShape` function does. ♦

Layout Shapes Reference

This section describes the constants and data types you use with layout shapes, as well as the basic functions you need to create a layout shape, change the information stored in a layout shape, and retrieve information from a layout shape.

Functions that relate to other aspects of layout shapes are described in the following chapters in this book:

- The chapter “Layout Styles” describes functions you can use to override the kerning and justification behavior and manipulate the special typographic features of a layout shape.
- The chapter “Layout Line Control” describes functions you can use to measure and break lines in a layout shape.
- The chapter “Layout Carets, Highlighting, and Hit-Testing” describes functions you can use to create and manipulate carets and highlighted sections of a layout shape.

Constants and Data Types

This section describes the constants and data types that you use when creating layout shapes.

Layout Options Structure

The layout options structure contains information that is relevant to the entire line of text rather than to any individual text run, style run, or direction-level run in the shape.

```
typedef struct {
    Fixed                width;
    fract                flush;
    fract                just;
    gxLayoutOptionsFlags flags;
    gxLineBaselineRecord *baselineRec;
} gxLayoutOptions;
```

Field descriptions

width The desired width of the line, measured in points (72 per inch) in geometry space. If you want the layout shape to be fully left-aligned and not justified, you do not need this field and you can set it to 0. See “Width” on page 5-10.

flush The alignment of the text, based on the layout shape’s position and the line width as specified in the `width` field. Alignment is a continuously varying, fractional value. A value of 0 means the text is to be left-aligned; its left or top edge coincides with the shape’s position. A value of 1 means the text is to be right-aligned; its right or bottom edge coincides with the shape position plus the text width. Intermediate values place the text between the left and right edges. (A value of 0.5 centers the text.) The following constants are available for specifying typical values in the `flush` field:

```
#define gxFlushLeft      0
#define gxFlushCenter    (fract1/2)
#define gxFlushRight     fract1
```

just The degree of justification in the line, defined as a continuously varying fractional value between 0 and 1. A value of 0 means no justification. A value of 1 means full justification between the edges defined by the layout position and the sum of the layout position and the `width` field. Intermediate values cause a fractional amount of the extra white space to be taken up by justification. The following constants are available for specifying typical values in the `just` field:

```
#define gxNoJustification 0
#define gxFullJustification fract1
```

flags Flag values that describe certain aspects of the entire layout shape. Values for the `flags` field are described in the next section, “Layout Options Flags.”

Layout Shapes

baselineRec An array of distances, in points (72 per inch), from the primary baseline for this layout shape to other baseline types. If you fill in this structure manually, you need to fill in only those values that correspond to the set of baselines present on the line. If you specify `nil` for this value, the line uses the Roman baseline, and all text is aligned to it. For more information on baselines and baseline alignment, see the chapter “Layout Line Control” in this book.

The layout options structure is used by the functions `GXNewLayout` (page 5-31), `GXGetLayout` (page 5-34), `GXSetLayout` (page 5-36), and `GXDrawLayout` (page 5-33).

Layout Options Flags

The layout options flags allow you to set certain characteristics of the layout shape as a whole. You set these flags through the `flags` field of the layout options structure.

```
#define gxNoLayoutOptions      0
#define gxLineIsDisplayOnly    0x00000001
```

Flag descriptions

`gxNoLayoutOptions`

Indicates that no layout option flags are used in this layout shape.

`gxLineIsDisplayOnly`

Indicates that the layout shape will be displayed but not edited in any way. If this bit is set, the shape is not edited, and QuickDraw GX does not recalculate any changes to the shape, such as the addition of ligatures or kashidas. This allows QuickDraw GX to display the layout shape faster and make the shape smaller.

Functions

This section describes functions for creating and drawing layout shapes, getting information from layout shapes, and editing layout shapes.

Some functions in this section use a byte offset parameter type. The `gxByteOffset` data type defines a byte offset into the text stored in the layout shape.

```
typedef long gxByteOffset;
```

The number of bytes is not necessarily equal to the number of character codes or glyph codes.

Creating and Drawing Layout Shapes

When you create a layout shape, you can supply it with different numbers of text runs, style runs, and direction-level runs. The `GXNewLayout` function creates a new layout shape for you to use; the `GXDrawLayout` function creates, draws, and disposes of a layout shape with a single call.

Layout Shapes

Note that you can also create and draw a layout shape by using the `GXNewShape` and `GXDrawShape` functions, described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

GXNewLayout

You can use the `GXNewLayout` function to create a new layout shape.

```
gxShape GXNewLayout(long textRunCount,
                    const short textRunLengths[],
                    const void *text[],
                    long styleRunCount,
                    const short styleRunLengths[],
                    const gxStyle styles[],
                    long levelRunCount,
                    const short levelRunLengths[],
                    const short levels[],
                    const gxLayoutOptions *layoutOptions,
                    const gxPoint *position);
```

`textRunCount`

The number of text runs supplied (the number of entries in the `text` parameter).

`textRunLengths`

An array containing the byte length of each text run (the length of each entry in the `text` parameter).

`text`

An array of pointers to runs of text. The text from these runs is concatenated, in order, to make up the source text of the layout shape.

`styleRunCount`

The number of style runs in the layout shape. (The number of entries in the `styles` parameter.)

`styleRunLengths`

An array containing the byte length of each style run.

`styles`

The style list: an array of references to the style objects associated with the layout shape, one for each style run. If you pass `nil` for this parameter, QuickDraw GX assigns the default layout style object to the layout shape and leaves the style list empty. If you pass a non-`nil` value for this parameter, then any `nil` entries in the array also refer to the shape's style.

`levelRunCount`

The number of direction-level runs in this layout shape (the number of entries in the `levels` parameter).

`levelRunLengths`

An array containing the byte length of each direction-level run.

Layout Shapes

<code>levels</code>	The levels array: an array of nested direction levels that control the dominant text direction within the layout shape. If pass <code>nil</code> for this parameter, QuickDraw GX assumes that the layout shape has an overall dominant direction of left to right.
<code>layoutOptions</code>	A pointer to a layout options structure. If you specify <code>nil</code> for this parameter, the default values are: left-aligned, unjustified, horizontal text on a Roman baseline.
<code>position</code>	The position of the baseline in the geometry coordinates. If you specify <code>nil</code> for this parameter, <code>GXNewLayout</code> sets the position to (0.0,0.0).

function result A reference to the newly created layout shape.

DESCRIPTION

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

Although this function creates a new layout shape, it does not create new style, ink, or transform objects. The new layout shape returned by `GXNewLayout` contains references to the default style, ink, and transform objects.

Most of the parameters to `GXNewLayout` are optional; if you set them to `nil`, QuickDraw GX sets the layout shape's equivalent properties to the default values, which are no text, no styles, and no levels.

ERRORS, WARNINGS, AND NOTICES

Errors

`out_of_memory`
`count_is_less_than_zero`
`parameter_out_of_range`
`inconsistent_parameters`

SEE ALSO

To create a new layout shape without specifying an initial geometry, see the description of the `GXNewShape` function in the chapter "Shape Objects" in *Inside Macintosh: QuickDraw GX Objects*.

GXDrawLayout

You can use the `GXDrawLayout` function to create, draw, and dispose of a layout shape with one call.

```
void GXDrawLayout(long textRunCount, const short textRunLengths[],
                  const void *text[], long styleRunCount,
                  const short styleRunLengths[],
                  const gxStyle styles[], long levelRunCount,
                  const short levelRunLengths[],
                  const short levels[],
                  const gxLayoutOptions *layoutOptions,
                  const gxPoint *position);
```

`textRunCount`

The number of text runs supplied (the number of entries in the `text` parameter).

`textRunLengths`

An array containing the byte length of each text run (the length of each entry in the `text` parameter).

`text`

An array of pointers to runs of text. The text from these runs is concatenated, in order, to make up the source text of the (temporary) layout shape.

`styleRunCount`

The number of style runs in the layout shape. (The number of entries in the `styles` parameter.)

`styleRunLengths`

An array containing the byte length of each style run.

`styles`

The style list: an array of references to the style objects associated with the (temporary) layout shape, one for each style run. If you pass `nil` for this parameter, QuickDraw GX assigns the default layout style object to the layout shape and leaves the style list empty. If you pass a non-`nil` value for this parameter, then any `nil` entries in the array also refer to the shape's style.

`levelRunCount`

The number of direction-level runs in this layout shape (the number of entries in the `levels` parameter).

`levelRunLengths`

An array containing the byte length of each direction-level run.

`levels`

The levels array: an array of nested direction levels that control the dominant text direction within the layout shape. If you specify `nil` for this parameter, QuickDraw GX assumes that the layout shape has an overall dominant direction of left to right.

`layoutOptions`

A pointer to a layout options structure. If you specify `nil` for this parameter, the default values are: left-aligned, unjustified, left to right horizontal text on a Roman baseline.

Layout Shapes

position The position of the baseline in the geometry coordinates. If you specify `nil` for this parameter, `GXNewLayout` sets the position to (0.0,0.0).

DESCRIPTION

The `GXDrawLayout` function creates, draws, and then disposes of a layout shape. You may want to use this function if you do not need to store a layout shape or draw the shape more than once.

ERRORS, WARNINGS, AND NOTICES**Errors**

`out_of_memory`
`count_is_less_than_zero`
`parameter_out_of_range`
`inconsistent_parameters`

SEE ALSO

The `GXNewLayout` function, described on page 5-31, creates a new layout shape that you can store and draw more than once.

You can use the `GXDrawShape` function, described in *Inside Macintosh: QuickDraw GX Objects*, to draw an existing layout shape.

Getting and Setting the Geometry of a Layout Shape

When you retrieve information about the text, style, and direction-level runs from a layout shape, you can get an entire array from the geometry—for instance, the complete style list—using the `GXGetLayout` function. You can change an entire array in the geometry—for instance, all of the text runs in the shape—using the `GXSetLayout` function.

GXGetLayout

You can use the `GXGetLayout` function to get all the information from the geometry of a layout shape.

```
long GXGetLayout(gxShape layout, void *text, long *styleRunCount,
                short styleRunLengths[], gxStyle styles[],
                long *levelRunCount, short levelRunLengths[],
                short levels[], gxLayoutOptions *layoutOptions,
                gxPoint *position);
```


Layout Shapes

<code>layout</code>	A reference to the layout shape whose information you need.
<code>text</code>	A pointer to a space for a text string. On return, the string contains all of the text from the layout shape (as a single text run).
<code>styleRunCount</code>	A pointer to a long value. On return, the value is the number of style runs in the shape (the number of entries in the style list).
<code>styleRunLengths</code>	An array of short values. On return, the array contains the byte length of each style run in the layout shape.
<code>styles</code>	An array of style-object references. On return, the array contains the style list for the layout shape.
<code>levelRunCount</code>	A pointer to a long value. On return, the value is the number of direction-level runs in this layout shape (the number of entries in the <code>levels</code> parameter).
<code>levelRunLengths</code>	An array of short values. On return, the array contains the byte length of each direction-level run.
<code>levels</code>	An array of short values. On return, the array is the levels array for the layout shape: an array of nested direction levels that control the dominant text direction within the layout shape.
<code>layoutOptions</code>	A pointer to the <code>gxLayoutOptions</code> structure. On return, the structure contains the layout options for the layout shape.
<code>position</code>	A pointer to a <code>gxPoint</code> value. On return, this parameter contains the position of the layout shape in geometry coordinates.

function result The number of bytes of text returned in the `text` parameter.

DESCRIPTION

The `GXGetLayout` function returns all the information from the geometry of the specified layout shape. If you specify `nil` for any parameter, `GXGetLayout` does not return that information.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`illegal_type_for_shape`
`index_is_less_than_zero`
`parameter_out_of_range`
`inconsistent_parameters`

Layout Shapes

SEE ALSO

To change the geometry of an existing layout shape by replacing an entire array in the geometry, use the `GXSetLayout` function, described next.

To get a portion of one or more arrays from the geometry of an existing layout shape, use the `GXGetLayoutParts` function, described on page 5-38. To change the geometry of an existing layout shape by replacing a portion of one or more of its arrays, use the `GXSetLayoutParts` function, described on page 5-40.

To get a portion of the geometry of an existing layout shape and put that portion into another layout shape, use the `GXGetLayoutShapeParts` function, described on page 5-42. To change the geometry of an existing layout shape by inserting the geometry of another typographic shape, use the `GXSetLayoutShapeParts` function, described on page 5-44.

GXSetLayout

You can use the `GXSetLayout` function to assign a new text array, style list, direction-levels array, or other property in the geometry of a layout shape.

```
void GXSetLayout(gxShape layout, long textRunCount,
                const short textRunLengths[], const void *text[],
                long styleRunCount,
                const short styleRunLengths[],
                const gxStyle styles[], long levelRunCount,
                const short levelRunLengths[],
                const short levels[],
                const gxLayoutOptions *layoutOptions,
                const gxPoint *position);
```

<code>layout</code>	A reference to the layout shape whose properties you want to set.
<code>textRunCount</code>	The number of text runs supplied (the number of entries in the <code>text</code> parameter).
<code>textRunLengths</code>	An array containing the byte length of each text run (the length of each entry in the <code>text</code> parameter).
<code>text</code>	An array of pointers to runs of text. The text from these runs is collated, in order, to make up the new source text of the layout shape.
<code>styleRunCount</code>	The number of style runs to put in the layout shape. (The number of entries in the <code>styles</code> parameter.)
<code>styleRunLengths</code>	An array containing the byte length of each style run.
<code>styles</code>	The new style list for the layout shape.

Layout Shapes

`levelRunCount`

The number of direction-level runs to put in this layout shape (the number of entries in the `levels` parameter).

`levelRunLengths`

An array containing the byte length of each direction-level run.

`levels`

The new levels array for the layout shape.

`layoutOptions`

A pointer to the layout options structure to use for this layout shape. If you specify `nil` for this parameter, the layout shape's current layout options are not changed.

`position`

The position of the baseline in the geometry coordinates. If you specify `nil` for this parameter, the layout shape's current position is not changed.

DESCRIPTION

The `GXSetLayout` function sets one or more entire field properties of a layout shape. You can change the values in any of the text, styles, or direction-levels arrays, or in the layout options or position. If you change one value having to do with the text, styles, or direction-levels components, you must send values for all the parameters having to do with that part—run count, run lengths, or the main component itself—even if those values are not changing.

If you want to add new values to existing values, you must send both the old and new values. For example, if you want to change the text "ABC" to "ABC DEF", you must send the entire string, not simply the string "DEF". (You must also send the new text length and resend the text run count. If you don't, the function returns the error `inconsistent_parameters`. In addition, you must update the style list and change the level run length.)

If you don't want to change one property of a layout (the text, styles, or direction levels), you can specify `0`, `nil`, and `nil` for the corresponding count, run lengths, and array parameters.

If you don't want to change the layout options or position, specify `nil` for the corresponding parameters.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`parameter_out_of_range`
`length_is_less_than_zero`
`index_is_less_than_zero`
`inconsistent_parameters`
`count_is_out_of_range`
`count_is_less_than_zero`

Warnings

`shape_access_not_allowed`
`shape_contains_invalid_data`

Layout Shapes

SEE ALSO

For an example of how to use the `GXSetLayout` function, see “Changing Parts of an Existing Layout Shape” beginning on page 5-20.

To get one or more complete arrays from the geometry of an existing layout shape, use the `GXGetLayout` function, described on page 5-34.

To get a portion of one or more arrays from the geometry of an existing layout shape, use the `GXGetLayoutParts` function, described next. To change the geometry of an existing layout shape by replacing a portion of one or more of its arrays, use the `GXSetLayoutParts` function, described on page 5-40.

To get a portion of the geometry of an existing layout shape and put that portion into another layout shape, use the `GXGetLayoutShapeParts` function, described on page 5-42. To change the geometry of an existing layout shape by inserting the geometry of another typographic shape, use the `GXSetLayoutShapeParts` function, described on page 5-44.

Getting and Setting Portions of a Layout Shape’s Geometry

If you want to retrieve information about only a part of the shape—for example, the part of the style list associated with the characters 9 to 15 of the shape in the source text—you can use the `GXGetLayoutParts` function. If you want to change only a portion of the geometry—for example, if you want to insert four character codes in the middle of the existing shape’s source text—you can use the `GXSetLayoutParts` function.

GXGetLayoutParts

You can use the `GXGetLayoutParts` function to get a portion of one of the various arrays of a layout shape, such as the style runs or layout options.

```
long GXGetLayoutParts(gxShape layout, gxByteOffset startOffset,
                     gxByteOffset endOffset, void *text,
                     short *styleRunCount,
                     short styleRunLengths[], gxStyle styles[],
                     short *levelRunCount,
                     short levelRunLengths[], short levels[]);
```

`layout` A reference to the layout shape whose information you need.

`startOffset` The edge offset in source text preceding the first character you want to retrieve from the layout shape.

`endOffset` The edge offset in source text following the final character you want to retrieve from the layout shape.

`text` A pointer to a text string. On return, the specified portion of the text of the layout shape.

Layout Shapes

`styleRunCount`

A pointer to a short value. On return, the number of entries in the style list.

`styleRunLengths`

An array of short values. On return, the array contains the number of bytes of text associated with each entry in the `styles` parameter.

`styles`

An array of style-object references. On return, the array is a style list associated with the `text` parameter.

`levelRunCount`

A pointer to a short value. On return, the number of entries in the `levels` parameter.

`levelRunLengths`

An array of short values. On return, the array contains the number of bytes per level associated with each entry in the `levels` parameter.

`levels`

An array of short values. On return, the array of direction-level codes associated with the text.

function result The byte count of the part of the text specified, which may not be equal to the character count or the glyph count, depending on the character codes or glyph codes used and the platform.

DESCRIPTION

The `GXGetLayoutParts` function queries a layout shape and retrieves a portion of the information from the shape, such as the style runs or layout options. For example, if a layout shape has three styles attached to it, you can retrieve one of those styles, rather than all three.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`illegal_type_for_shape`
`index_is_less_than_zero`
`parameter_out_of_range`
`inconsistent_parameters`

SEE ALSO

To change the geometry of an existing layout shape by replacing a portion of one or more of its arrays, use the `GXSetLayoutParts` function, described next.

To get one or more complete arrays from the geometry of an existing layout shape, use the `GXGetLayout` function, described on page 5-34. To change an existing layout shape by replacing an entire array in its geometry, use the `GXSetLayout` function, described on page 5-36.

Layout Shapes

To get a portion of the geometry of an existing layout shape and put that portion into another layout shape, use the `GXGetLayoutShapeParts` function, described on page 5-42. To change the geometry of an existing layout shape by inserting the geometry of another typographic shape, use the `GXSetLayoutShapeParts` function, described on page 5-44.

GXSetLayoutParts

You can use the `GXSetLayoutParts` function to change a portion of the text, styles, or direction-levels arrays in the geometry of a layout shape.

```
void GXSetLayoutParts(gxShape layout, gxByteOffset oldStartOffset,
                    gxByteOffset oldEndOffset,
                    long newTextRunCount,
                    const short newTextRunLengths[],
                    const void *newText[],
                    long newStyleRunCount,
                    const short newStyleRunLengths[],
                    const gxStyle newStyles[],
                    long newLevelRunCount,
                    const short newLevelRunLengths[],
                    const short newLevels[]);
```

<code>layout</code>	A reference to the layout shape you want to modify.
<code>oldStartOffset</code>	The edge offset in the source text preceding the first character to replace.
<code>oldEndOffset</code>	The edge offset in the source text following the last character to replace.
<code>newTextRunCount</code>	The number of text runs supplied in the <code>newText</code> parameter.
<code>newTextRunLengths</code>	An array containing the byte length of each text run in the <code>newText</code> parameter.
<code>newText</code>	An array of pointers to runs of text. If you pass <code>nil</code> for this parameter, QuickDraw GX uses the default text object.
<code>newStyleRunCount</code>	The number of styles in the style list.
<code>newStyleRunLengths</code>	An array containing the byte length of each style run.
<code>newStyles</code>	An array of references to style objects, one for each style run. If you pass <code>nil</code> for this parameter, QuickDraw GX uses the default style object. If you pass a non- <code>nil</code> value for this parameter, then any <code>nil</code> entries in the array also refer to the shape's style.

Layout Shapes

`newLevelRunCount`

The number of direction-level runs.

`newLevelRunLengths`

An array containing the byte lengths of each direction-level run.

`newLevels`

An array of nested direction levels that control text direction within the layout shape.

DESCRIPTION

The `GXSetLayoutParts` function sets or changes parts of the geometry of a layout shape. You can make changes to an array in the layout shape without sending the old values in addition to the new values. For example, if you want to change the text “ABC” to “ABC DEF”, send only the string “DEF”. You must also set the `oldStartOffset` and `oldEndOffset` parameters to the character offset of the last character offset, in the source text, of the original string; in the previous example, both parameters should be set to 3.

Any new values you add must be consistent with values already in the shape, unless you are discarding the old values. For example, if you change the text in the layout shape, the new text must be consistent with the values of the `newTextRunCount` and `newTextRunLengths` parameters, or you must enter new values for these parameters.

If you don’t want to change one component of a layout (text, styles, or levels), you can specify 0, `nil`, and `nil` for the corresponding count, run lengths, and array parameters.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`parameter_out_of_range`
`length_is_less_than_zero`
`index_is_less_than_zero`
`inconsistent_parameters`
`count_is_out_of_range`
`count_is_less_than_zero`

Warnings

`shape_access_not_allowed`
`shape_contains_invalid_data`

SEE ALSO

For an example of how to use the `GXSetLayoutParts` function, see “Changing Parts of an Existing Layout Shape” beginning on page 5-20.

To get a portion of one or more arrays from the geometry of an existing layout shape, use the `GXGetLayoutParts` function, described on page 5-38.

To get one or more complete arrays from the geometry of an existing layout shape, use the `GXGetLayout` function, described on page 5-34. To change the geometry of an existing layout shape by replacing an entire array in the geometry, use the `GXSetLayout` function, described on page 5-36.

Layout Shapes

To get a portion of the geometry of an existing layout shape and put that portion into another layout shape, use the `GXGetLayoutShapeParts` function, described on page 5-42. To change the geometry of an existing layout shape by inserting the geometry of another typographic shape, use the `GXSetLayoutShapeParts` function, described on page 5-44.

Extracting or Inserting Parts of a Layout Shape

If you want to get part of the geometry of a layout shape and put it into another layout shape, use the `GXGetLayoutShapeParts` function. If you want to insert the geometry of another typographic shape, whether a text shape, glyph shape, or layout shape into a layout shape, use the `GXSetLayoutShapeParts` function.

GXGetLayoutShapeParts

You can use the `GXGetLayoutShapeParts` function to extract a copy of a specified range of the geometry of a layout shape and encapsulate it in another layout shape.

```
gxShape GXGetLayoutShapeParts(gxShape layout,
                              gxByteOffset startOffset,
                              gxByteOffset endOffset,
                              gxShape dest);
```

`layout` A reference to the layout shape containing the geometry you want to use.

`startOffset` The edge offset in the source text preceding the starting character to retrieve from the layout shape.

`endOffset` The edge offset in the source text following the ending character to retrieve from the layout shape.

`dest` A reference to the layout shape that will receive the extracted geometry. If you set this parameter to `nil`, a new layout shape is returned as the function result. Even if it's not set to `nil`, the function result is still a copy of the old shape's reference.

function result A reference to the layout shape referenced by the `dest` parameter or to a new layout shape if the value of the `dest` parameter is `nil`.

DESCRIPTION

The `GXGetLayoutShapeParts` function extracts a copy of a portion of the layout shape in the `layout` parameter, including the text, styles, and levels arrays, and copies the geometry into an existing layout shape, specified by the `dest` parameter. The function returns a reference to the layout shape specified by the `dest` parameter.

Layout Shapes

or a new layout shape, if `dest` is set to `nil`. The extracted data is bounded by the `startOffset` and `endOffset` values, which refer to byte offsets, in the source text, of the original layout shape.

If you want to put the copy in an existing layout shape, put a reference to that layout shape in the `dest` parameter; otherwise, leave that parameter set to `nil`, and the function returns a reference to a new layout shape in the function result.

SPECIAL CONSIDERATIONS

The `GXGetLayoutShapeParts` function is analogous to the `GXGetShapeParts` function, described in *Inside Macintosh: QuickDraw GX Objects*. However, the `GXGetLayoutShapeParts` function uses zero-based indexing and two offsets to mark a section within a layout shape; the `GXGetShapeParts` function uses 1-based indexing, a single offset, and a count to mark a section within a shape.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`illegal_type_for_shape`
`index_is_less_than_zero`
`parameter_out_of_range`
`inconsistent_parameters`

SEE ALSO

For an example of the use of `GXGetLayoutShapeParts`, see “Extracting a Layout Shape From Part of an Existing Layout Shape” on page 5-23.

To change the geometry of an existing layout shape by inserting the geometry of another typographic shape, use the `GXSetLayoutShapeParts` function, described next.

To get one or more entire arrays from the geometry of an existing layout shape, use the `GXGetLayout` function, described on page 5-34. To change an existing layout shape by replacing an entire array in its geometry, use the `GXSetLayout` function, described on page 5-36.

To get a portion of one or more arrays from the geometry of an existing layout shape, use the `GXGetLayoutParts` function, described on page 5-38. To change the geometry of an existing layout shape by replacing a portion of one or more of its arrays, use the `GXSetLayoutParts` function, described on page 5-40.

GXSetLayoutShapeParts

You can use the `GXSetLayoutShapeParts` function to replace the geometry in a layout shape with another typographic shape's geometry.

```
void GXSetLayoutShapeParts(gxShape layout,
                           gxByteOffset startOffset,
                           gxByteOffset endOffset,
                           gxShape insert);
```

<code>layout</code>	A reference to the layout shape whose geometry you want to edit.
<code>startOffset</code>	The edge offset in the source text preceding the starting character to retrieve from the layout shape.
<code>endOffset</code>	The edge offset in the source text following the ending character to retrieve from the layout shape.
<code>insert</code>	The typographic shape whose geometry you want to insert. This shape may be a text, glyph, or layout shape. Note: the whole shape's geometry is inserted; the start and end offsets refer to the edited shape only.

DESCRIPTION

The `GXSetLayoutShapeParts` function inserts the entire text of the shape specified by the `insert` parameter into the layout shape specified by the `layout` parameter. The inserted text may have its own styles; however, you cannot add positions and tangents arrays of a glyph shape to a layout shape.

The `startOffset` and `endOffset` parameters are the starting offset and ending offset in the layout shape where you want to insert the new geometry. As long as the values of `startOffset` and `endOffset` are equal, `GXSetLayoutShapeParts` performs the insertion. If they are not equal, it replaces the old text between `startOffset` and `endOffset`.

If the value of `insert` is not a text, glyph, or layout shape, the function returns the warning `illegal_type_for_shape`.

SPECIAL CONSIDERATIONS

The `GXSetLayoutShapeParts` function is analogous to the `GXSetShapeParts` function, described in *Inside Macintosh: QuickDraw GX Objects*. However, the `GXSetLayoutShapeParts` function uses 0-based offsets to mark a section within a layout shape; `GXSetShapeParts` uses 1-based indexing, an offset, and a count to mark a section within a shape.

ERRORS, WARNINGS, AND NOTICES

Errors

```
shape_is_nil
parameter_out_of_range
inconsistent_parameters
count_is_less_than_zero
count_is_out_of_range
index_is_less_than_zero
length_is_less_than_zero
```

Warnings

```
shape_access_not_allowed
shape_contains_invalid_data
illegal_type_for_shape
```

SEE ALSO

For an example of how to use the `GXSetLayoutShapeParts` function, see “Changing Parts of an Existing Layout Shape” beginning on page 5-20.

To get a portion of the geometry of an existing layout shape and put that portion into another layout shape, use the `GXGetLayoutShapeParts` function, described on page 5-42.

To get one or more entire arrays from the geometry of an existing layout shape, use the `GXGetLayout` function, described on page 5-34. To change the geometry of an existing layout shape by replacing an entire array in the geometry, use the `GXSetLayout` function, described on page 5-36.

To get a portion of one or more arrays from the geometry of an existing layout shape, use the `GXGetLayoutParts` function, described on page 5-38. To change the geometry of an existing layout shape by replacing a portion of one or more of its arrays, use the `GXSetLayoutParts` function, described on page 5-40.

Obtaining Glyph Information From a Layout Shape

You can obtain information about the glyphs in a layout shape by using the `GXGetLayoutGlyphs` function.

GXGetLayoutGlyphs

You can use the `GXGetLayoutGlyphs` function to get information about each of the glyphs in a layout shape.

```
long GXGetLayoutGlyphs(gxShape layout, gxGlyphcode glyphs[],
                      gxPoint positions[], long advance[],
                      gxPoint tangents[], long *runCount,
                      short styleRuns[], gxStyle glyphStyles[]);
```

Layout Shapes

<code>layout</code>	A reference to the layout shape whose glyph information you need.
<code>glyphs</code>	A pointer to an array of glyph codes. On return, the array contains the glyph codes for all the glyphs in the layout shape.
<code>positions</code>	An array of <code>gxPoint</code> values. On return, the array contains the positions of each of the glyphs in the layout shape.
<code>advance</code>	An array of <code>long</code> values. On return, the array contains the advance bits for the glyphs in the layout shape with the first bit on, the others off.
<code>tangents</code>	An array of <code>gxPoint</code> values. On return, the array contains the tangents for the glyphs in the layout shape. If the layout contains runs with the <code>gxVertical Text</code> flag set, the array contains the tangents for the individual glyphs; otherwise it is filled with <code>[1.0,0.0]</code> .
<code>runCount</code>	A pointer to a <code>long</code> value. On return, the value is the number of style runs in the glyph shape that is equivalent to this layout shape.
<code>styleRuns</code>	An array of <code>short</code> values. On return, the array contains the number of glyphs in each style run.
<code>glyphStyles</code>	An array of style-object references. On return, the array contains the style list for the glyph shape that is equivalent to this layout shape.

function result The number of glyphs in the shape.

DESCRIPTION

The `GXGetLayoutGlyphs` function provides access to a layout shape's glyph information without requiring you to first convert the layout shape to a glyph shape. It returns the glyph code and positioning information for each glyph, in a form analogous to the information returned for glyph shapes by the `GXGetGlyphs` function.

This function treats the layout shape as if it were a glyph shape. The `advance` and the `positions` parameters return the advance bits and positions arrays, respectively, which contain information about whether the positions stored in the positions array (one position per glyph in the shape) are absolute or relative. For layout shapes, the first bit of the advance bits array is always absolute, and the remaining bits are relative. The `tangents` parameter contains as many tangents as there are glyphs in the shape and determines the direction and scaling of the individual glyph. For more information about advance bits, positions, and tangents, see the chapter "Glyph Shapes" in this book.

Besides glyph identity and positioning, this function returns style-run information indexed by glyph (rather than by character code, as is true for other functions). See "Getting Glyph Information From a Layout Shape" on page 5-27 for a demonstration of the difference between the `GXGetLayoutGlyphs` function and the `GXGetLayout` function, which returns information based upon the character codes stored in the shape.

If you pass `nil` for a parameter, `GXGetLayoutGlyphs` does not return values in that parameter.

ERRORS, WARNINGS, AND NOTICES

Errors`shape_is_nil`**Notices (debugging version)**`glyph_tangents_have_no_effect`

SEE ALSO

This function is similar in form and purpose to the `GXGetGlyphs` function, described in the chapter “Glyph Shapes” in this book. The advance bits and tangent arrays are properties of the glyph shape, also described in the chapter “Glyph Shapes” in this book.

Summary of Layout Shapes

Constants and Data Types

```
typedef long gxByteOffset;
```

Layout Options Structure

```
typedef struct {
    Fixed          width;
    fract          flush;
    fract          just;
    gxLayoutOptionsFlags flags;
    gxLineBaselineRecord *baselineRec;
} gxLayoutOptions;

#define gxNoLayoutOptions 0
#define gxLineIsDisplayOnly 0x00000100
#define gxMaxRunLevel 15
#define gxFlushLeft 0
#define gxFlushCenter (frac1/2)
#define gxFlushRight frac1
#define gxNoJustification 0
#define gxFullJustification frac1
typedef unsigned long gxLayoutOptionsFlags;
```

Layout Shape Functions

Creating and Drawing Layout Shapes

```
gxShape GXNewLayout          (long textRunCount,
                              const short textRunLengths[],
                              const void *text[], long styleRunCount,
                              const short styleRunLengths[],
                              const gxStyle styles[], long levelRunCount,
                              const short levelRunLengths[],
                              const short levels[],
                              const gxLayoutOptions *layoutOptions,
                              const gxPoint *position);
```

Layout Shapes

```

void GXDrawLayout          (long textRunCount,
                           const short textRunLengths[],
                           const void *text[], long styleRunCount,
                           const short styleRunLengths[],
                           const gxStyle styles[], long levelRunCount,
                           const short levelRunLengths[],
                           const short levels[],
                           const gxLayoutOptions *layoutOptions,
                           const gxPoint *position);

```

Getting and Setting the Geometry of a Layout Shape

```

long GXGetLayout          (gxShape layout, void *text,
                           long *styleRunCount, short styleRunLengths[],
                           gxStyle styles[], long *levelRunCount,
                           short levelRunLengths[], short levels[],
                           gxLayoutOptions *layoutOptions,
                           gxPoint *position);

void GXSetLayout          (gxShape layout, long textRunCount,
                           const short textRunLengths[],
                           const void *text[], long styleRunCount,
                           const short styleRunLengths[],
                           const gxStyle styles[], long levelRunCount,
                           const short levelRunLengths[],
                           const short levels[],
                           const gxLayoutOptions *layoutOptions,
                           const gxPoint *position);

```

Getting and Setting Portions of a Layout Shape's Geometry

```

long GXGetLayoutParts     (gxShape layout, gxByteOffset startOffset,
                           gxByteOffset endOffset, void *text,
                           short *styleRunCount, short styleRunLengths[],
                           gxStyle styles[], short *levelRunCount,
                           short levelRunLengths[], short levels[]);

void GXSetLayoutParts     (gxShape layout, gxByteOffset oldStartOffset,
                           gxByteOffset oldEndOffset,
                           long newTextRunCount,
                           const short newTextRunLengths[],
                           const void *newText[], long newStyleRunCount,
                           const short newStyleRunLengths[],
                           const gxStyle newStyles[],
                           long newLevelRunCount,
                           const short newLevelRunLengths[],
                           const short newLevels[]);

```

Layout Shapes

Extracting or Inserting Parts of a Layout Shape

```
gxShape GXGetLayoutShapeParts  
                                (gxShape layout, gxByteOffset startOffset,  
                                gxByteOffset endOffset, gxShape dest);  
void GXSetLayoutShapeParts    (gxShape layout, gxByteOffset startOffset,  
                                gxByteOffset endOffset, gxShape insert);
```

Obtaining Glyph Information From a Layout Shape

```
long GXGetLayoutGlyphs        (gxShape layout, gxGlyphcode *glyphs,  
                                gxPoint positions[], long advance[],  
                                gxPoint tangents[], long *runCount,  
                                short styleRuns[], gxStyle glyphStyles[]);
```