

This chapter describes the relationship between character-code position in a layout shape's source text and glyph position in its corresponding display text. Your application uses this information to draw carets, to highlight ranges of text, and to hit-test (to convert from display-text location to source-text location).

Read the information in this chapter if you create layout shapes containing text that the user can select or edit. You do not need the information here if you create only static (unalterable) lines of text. Also, if your application creates only simple text that is better represented by a text shape or glyph shape, you do not need the information in this chapter.

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

The chapter starts by describing how QuickDraw GX defines locations in source text and in display text of any typographic shape. The chapter next explores how the definition of text locations affects caret handling, highlighting, and hit-testing for layout shapes. It then describes how to use QuickDraw GX functions to

- draw perpendicular or angled carets in single-direction or mixed-direction text
- highlight single-direction or mixed-direction text in two different ways
- hit-test any text in a layout shape
- analyze glyphs for direction and for relationship to source-text position

About Carets, Highlighting, and Hit-Testing for Layout Shapes

Editing text from a layout shape can be a complicated process. Preparing a line of text for display can involve reversing text direction, rearranging glyphs, substituting glyphs, creating ligatures, and moving or modifying glyphs for purposes of justification, kerning, typestyle changes, and so on. Editing that line involves repeating these operations for the edited text. However, QuickDraw GX helps you in this effort by eliminating your need to track the details of conversion from source text to display text and back.

This section describes the relationship between character codes in the source text of a layout shape and glyphs in its display text. It also explains how QuickDraw GX exploits that relationship to help you with caret handling, highlighting, and hit-testing.

Positioning in Source Text and Display Text

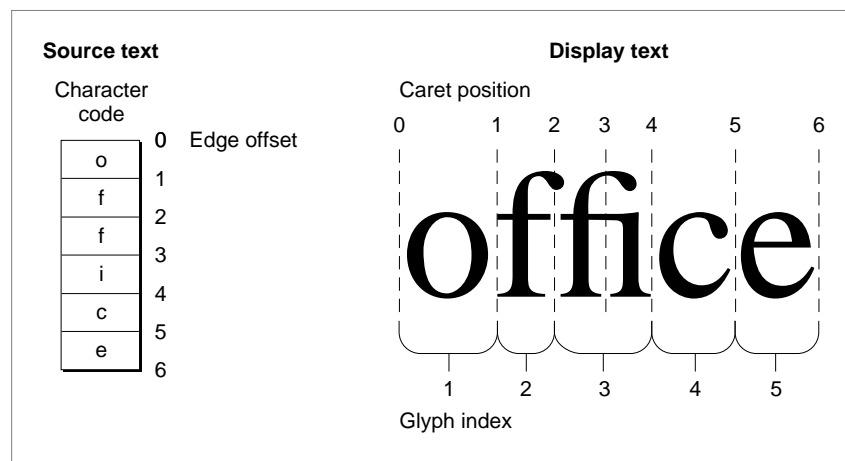
By convention, QuickDraw GX stores text in a typographic shape as a sequence of character codes, which are display-independent numeric representations of the fundamental characters that make up the text. Furthermore, these character codes are stored in input order, that is, the order in which the characters are entered from a keyboard.

When it draws text, QuickDraw GX composes glyphs from those characters and draws the glyphs in display order. Display order is uniformly left to right (or top to bottom for vertical text) and may be very different from input order.

On the screen, the sequence of glyphs in a line of text is specified by **glyph index**, a simple left-to-right (or top-to-bottom) ordering that starts with 1 for the leftmost glyph. In Figure 10-1, for example, a line of text composed of six characters in memory (on the left) is rendered onscreen with five glyphs (on the right).

In memory, the positions of character codes in the source text of a shape could also be represented by index, but they are more commonly represented by edge offset. **Edge offset** is the byte offset from the beginning of a shape's source text to a point *between* character codes in the sequence. Edge offset is zero-based. Figure 10-1 shows an edge offset of 0 marks a point just before the first byte in the source text; an offset of 1 marks a point between the first and second bytes, and so on.

Figure 10-1 Positioning conventions for source text and display text



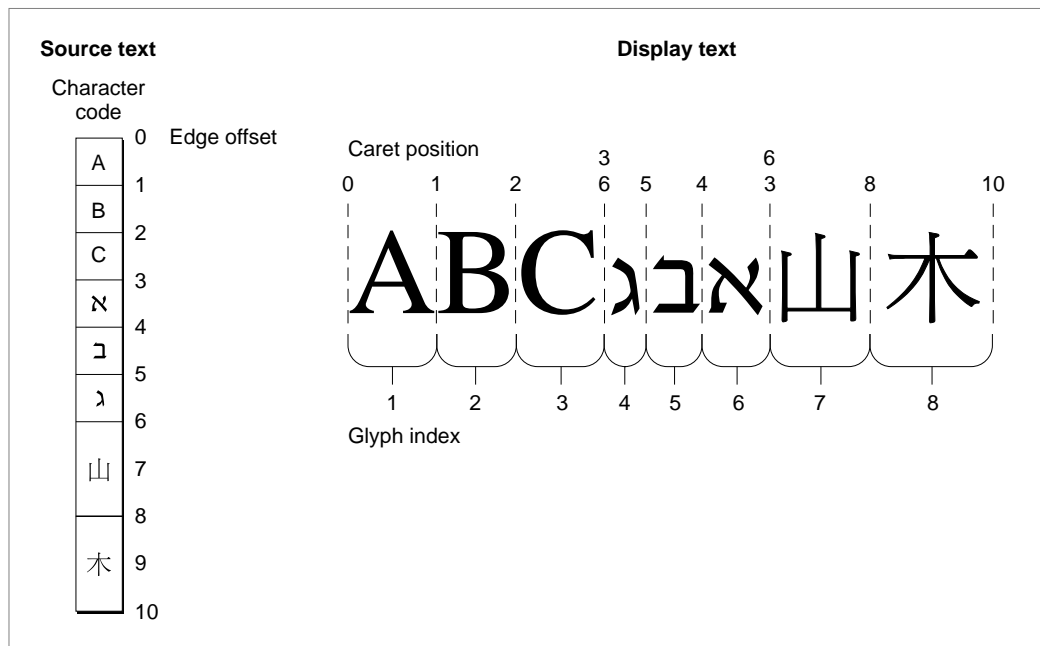
Edge offset is slightly different from the typical notion of byte offset into a buffer, in that a given edge offset is not associated with a unique byte value. An edge offset could refer to either the byte preceding it or the byte following it. This concept is useful because edge offset relates directly to **caret position**, a location on screen that is typically between glyphs. (A **caret** is a vertical or slanted bar, appearing at a caret position in the display text, that marks the point at which text is to be inserted or deleted.) Each caret position relates directly to an edge offset; you can see in Figure 10-1 that the caret positions in the display text are numbered according to their corresponding edge offsets in the source text. A caret at any of the caret positions in Figure 10-1 means an insertion point in the source text at the corresponding edge offset.

The text in Figure 10-1 is fairly simple; it is single-direction text, and all character codes are 1 byte long. One complication is that the "f" and "i" characters are combined upon display into the "fi" ligature. Because the ligature is a single glyph, there is no longer a

one-to-one correspondence between characters and glyphs. Also, there is one possible caret position (at edge offset 3) that is within a glyph rather than at its edge. For the purposes of drawing carets, highlighting, and hit-testing, QuickDraw GX permits you either to allow or disallow caret positions within complex glyphs such as ligatures.

Figure 10-2 shows a slightly more complex example of a line of text from a layout shape. In this layout, three characters of Roman text are followed by three Hebrew characters, in turn followed by two Chinese characters. Note that, because Hebrew text is read from right to left, the order of the Hebrew glyphs on the screen is the reverse of the order of their character codes in the source text. (The line direction, or dominant text direction, of this layout shape is left to right. If it were right to left, the order of the displayed characters would have been somewhat different. See the chapter “Layout Line Control” in this book for a complete discussion of how line direction and different levels of direction runs can cause complications in line layout.)

Figure 10-2 Edge offsets and glyph indexes in mixed-direction text



The reversal of the Hebrew text causes some complications in the relationship between caret position and edge offset. Note in particular what happens at edge offset 3, the boundary (in the source text) between the Roman “C” and the Hebrew “א.” In memory, it is a single point that could allow for inserting a Roman character after the “C” or a Hebrew character before the “א.” On screen, however, that insertion point splits into two caret positions: one to the right of (that is, after) the “C,” and one to the right of (that is, before) the “א.”

Layout Carets, Highlighting, and Hit-Testing

Within the run of Hebrew text, caret positions increase leftward, as would be expected for right-to-left text. And at the boundary between the Hebrew text and the Chinese text (here written horizontally and left to right), the additional change in direction means that edge offset 6 also converts to two caret positions onscreen: one to the left of (that is, after) the “א,” and one to the left of (that is, before) the “山.”

In general, each direction boundary in the source text of a layout shape maps to two different caret positions in the display text, and each direction boundary in the display text maps to two different edge offsets in the source text. This indeterminacy causes complications in caret drawing, highlighting, and hit-testing, as discussed further in subsequent sections of this chapter. However, note that QuickDraw GX handles most of the complications for you.

Figure 10-2 illustrates a second important complication. Note that the Chinese character codes in the source text are each 2 bytes long. Because edge offsets are byte offsets, each subsequent location between Chinese characters has an offset value that is 2 higher than its predecessor. In Figure 10-2, edge offsets (and caret positions) 7 and 9 are invalid—each refers to a point inside a single character.

IMPORTANT

If you are going to support 2-byte characters in your text handling, remember that successive valid edge offsets in the buffer that holds your source text may differ by either 1 or 2. Even in 2-byte languages such as Chinese, Japanese, and Korean, some character codes are only 1 byte long. You cannot assume a single storage size for characters. QuickDraw GX provides functions, such as the `GXGetOffsetGlyphs` function on page 10-56, that help you determine the sizes of character codes. ▲

Caret Handling

A **caret** is a symbol that indicates where onscreen the next text-editing operation will take place. The caret is commonly represented by a blinking vertical bar (|) in horizontal text, a horizontal bar in vertical text, and a slanted bar in italic or otherwise slanted text.

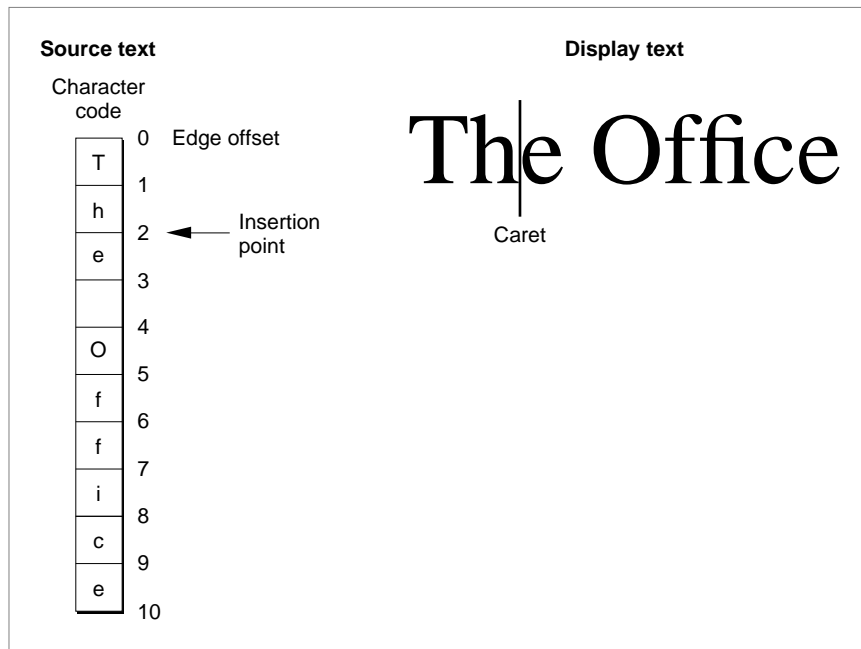
Note

In Macintosh text processing, the caret is not the same as the cursor. The **cursor** is a small icon (often an arrow or I-beam shape) that “shadows” the movement of the mouse or other pointing device. The cursor is controlled by the system, although your application can change its appearance; the caret is controlled entirely by your application. When the user clicks or holds the mouse button down while the cursor is positioned within a line of text, your application sets the caret position or extends the highlight to the cursor position. Otherwise, these positions are independent of each other. ♦

The caret is the onscreen representation of the insertion point. The **insertion point** is that point in the source text where character codes will be added or deleted when the next editing operation occurs. An insertion point is specified by a single edge offset; the caret

occupies the onscreen caret position that corresponds to that edge offset. Figure 10-3 shows the basic relationship between insertion point, edge offset, and caret position for single-direction text.

Figure 10-3 Insertion point and caret

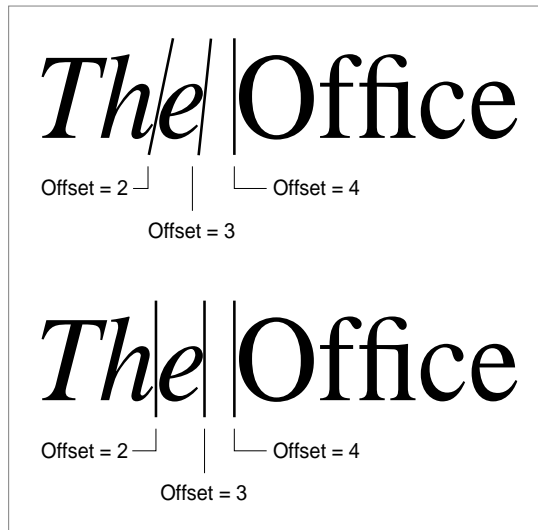


Straight and Angled Carets

When the caret appears in italic text or text that has an intrinsic angle (for instance, ITC Zapf Chancery®), you may wish, for the user's convenience, to display an angled (slanted) caret rather than a straight one. QuickDraw GX supports this capability by using data present in fonts that identifies the intrinsic font angle. Your application can disable this feature, if desired, on a per-run basis.

You can specify that a caret can be straight or angled, meaning that it is either perpendicular to the baseline or at an angle to the baseline that equals the slant of the glyphs between which the caret is drawn. If you choose an angled caret, QuickDraw GX calculates the proper angle; at boundaries of text with different slant, QuickDraw GX calculates an average angle for the caret.

In Figure 10-4, for example, the word "The" is italic, whereas the subsequent characters are not. The angled caret is parallel to the slant of the italic text at the caret position whose edge offset is 2; it has only half the slant at the caret position whose edge offset is 3; and it is vertical at caret positions with edge offsets of 4 and greater.

Figure 10-4 Angled and straight carets in single-direction text

Note also from Figure 10-4 that if you specify a straight (vertical) caret, it remains perpendicular to the baseline even in italic text.

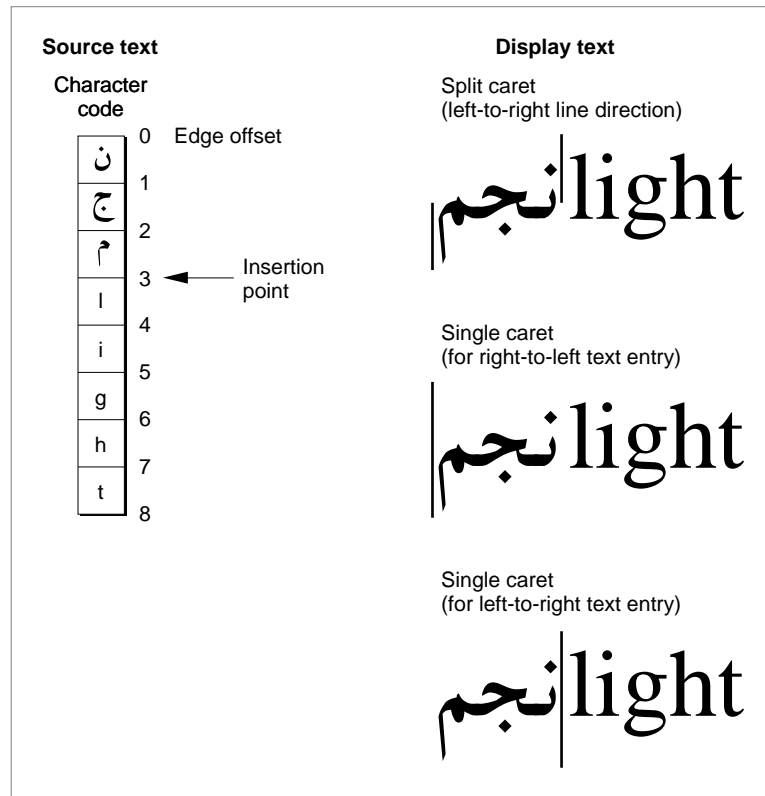
Drawing an angled cursor

You may also want to slant the cursor when it passes over areas of slanted text. QuickDraw GX provides a function to help you do that efficiently. See “Drawing the Cursor at the Correct Angle Within a Given Area” beginning on page 10-22. ♦

Split and Single Carets

At direction boundaries in mixed-direction text, a single insertion point in memory can require two caret positions onscreen, one for text entry in each direction. QuickDraw GX supports two different methods for handling that situation: a split caret or a single caret. Figure 10-5 shows examples of both caret types for an insertion point at the boundary between Arabic and Roman text.

A **split caret**, or dual caret, is the preferred caret shape provided by QuickDraw GX for use with mixed-direction text. A split caret consists of a high caret and a low caret, each measuring half the line’s height. The two separate half-carets appear only when the insertion point is at the boundary between two direction runs in a line of text. The high (dominant) caret is displayed at the caret position for insertion of text whose direction corresponds to the line direction (the dominant direction for the whole line of text). The low caret is displayed at the caret position for insertion of text whose direction is counter to the line direction. When the caret position is unambiguous (not on a direction boundary), the primary and secondary carets are at the same position, so the user sees one caret.

Figure 10-5 Split caret and single carets at a direction boundary in mixed-direction text

In Figure 10-5 (top), a split caret appears when the insertion point is at the direction boundary at edge offset 3. Because the line direction for this example is left to right, the high caret appears to the left of the space character before the word “light”, to allow insertion of Roman characters before the space character. The low caret appears at the far left of the line, to allow insertion of Arabic characters after the “م”.

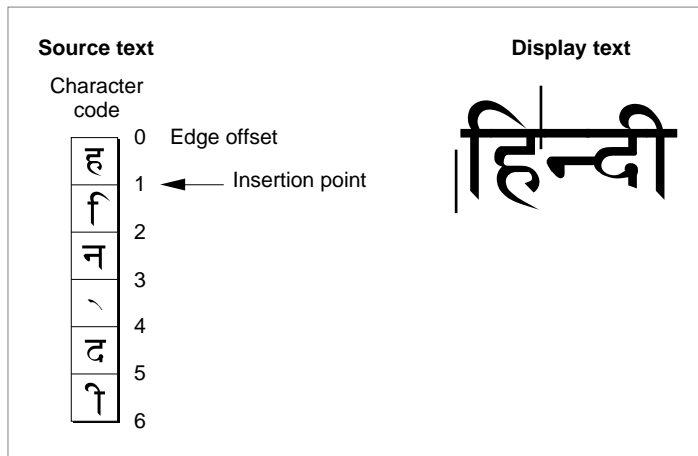
A **single caret** can also be used in mixed-direction text. It is either a **left-to-right caret** or a **right-to-left caret**, which refers not to any difference in appearance but to difference in location. When the caret position is unambiguous (not on a direction boundary), it is a standard text-insertion caret. At a direction boundary, it is also a single caret that appears at the place where the next text insertion or deletion will occur, given the application’s specification of text direction.

In Figure 10-5 (middle and bottom), a single caret appears at either of two positions when the insertion point is at the direction boundary at edge offset 3. If the current input text direction is left to right (corresponding to Roman text entry), the caret appears to the left of the “l” character before the word “light”. If the current direction is right-to-left (corresponding to Arabic text entry), the caret appears at the far left of the line. Note that switching input directions repeatedly without entering any characters causes the caret to jump between the two caret positions. Note that it is your responsibility to monitor the user’s choice of keyboard or language and set the caret type appropriately.

Split carets and linguistic rearrangement

Split carets can arise in one other case: linguistic (Indic-style) rearrangement. In this case, all the text is uniformly left to right, but because the visual order of certain glyphs on the line is different from the input order of the character codes in the source text, the returned caret is split. The high caret is located at the caret position with the numerically higher edge offset of the two for that character; see Figure 10-6. ♦

Figure 10-6 Split caret with linguistically rearranged glyphs

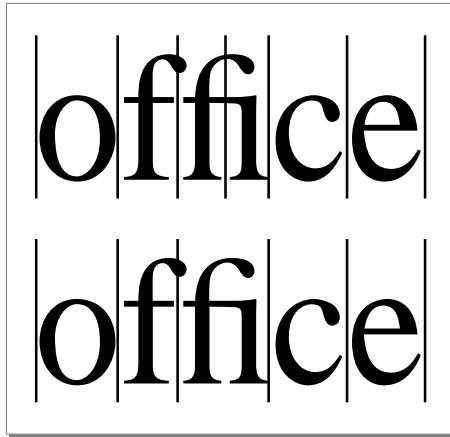
**Carets and vertical text**

Vertical text is never reordered nor linguistically rearranged; its display order (top to bottom) is always the same as its input order. Therefore, text direction is always (the equivalent of) left to right, and split carets cannot occur. ♦

Caret Position and Split Ligatures

For defining caret positions, you can treat ligatures as single, indivisible glyphs or you can split them into subareas representing each of their component characters. If you want ligatures to be treated as single glyphs, you set the `gxNoLigatureSplits` flag in the run controls structure of the style run containing the ligatures. If you clear the `gxNoLigatureSplits` flag, QuickDraw GX allows caret positions within the ligature, as defined by the font. By default, the flag is cleared.

Figure 10-7 shows two examples of carets drawn at all valid caret positions in the word “office”. In the upper example, ligature splits are permitted; in the lower example, they are not.

Figure 10-7 Caret positions with and without ligature splits

For an example of the effect of setting and clearing the `gxNoLigatureSplits` flag, see “Positioning the Caret Within Ligatures” beginning on page 10-24. Run controls are explained in the chapter “Layout Styles” in this book.

Note

Whether or not you permit ligature splits, editing should still occur one character at a time, not one glyph at a time. Thus, if the caret were positioned to the right of an “fi” ligature, a single backspace should not delete the whole ligature; it should delete only the “i”, leaving an “f” glyph in place of the ligature. ♦

Arrow Keys and Caret Movement

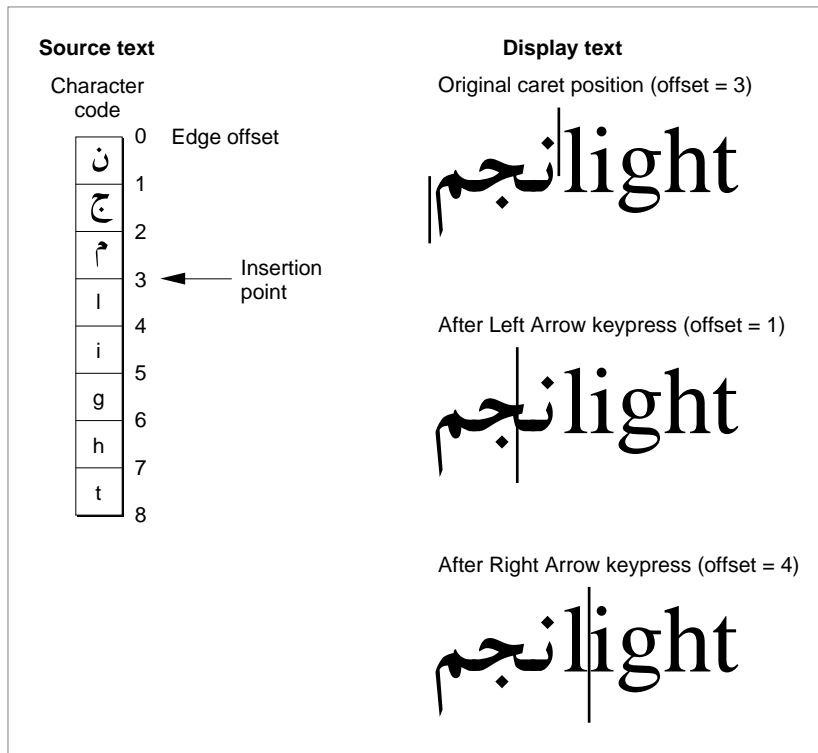
When the user presses the Right Arrow key, the caret should move one character to the right in horizontal text. When the user presses the Left Arrow key, the caret should move one character to the left. If your layout shape has ligatures, mixed 1-byte and 2-byte characters, mixed-direction text, or text that has undergone linguistic rearrangement, determining the edge offset of the next caret position is not always obvious.

Figure 10-8 shows a line of mixed Roman and Arabic text in which the dominant direction is left to right. Suppose that the current insertion point is at edge offset 3 in the source text. That offset represents a direction boundary; the high caret marks the point for inserting Roman text, and the low caret marks the point for inserting Arabic text. If the user presses the Left Arrow key, the high caret moves one space to the left. That puts the caret across the direction boundary, and corresponds to moving several characters backward in the (Arabic) source text, which puts the new edge offset at 1. For split carets, the high caret should follow the arrows.

Layout Carets, Highlighting, and Hit-Testing

If instead the user presses the Right Arrow key, the high caret moves one space to the right. That action puts the caret across the direction boundary and corresponds to moving one character forward in the source text. The new edge offset is therefore 4.

Figure 10-8 Moving the caret with Left and Right Arrow keys



If the display text contains ligatures, caret movement in response to the pressing of an arrow key can depend on whether caret positions within ligatures are valid. See "Caret Position and Split Ligatures" on page 10-10.

QuickDraw GX provides functions that allow you to determine the proper edge offset of the new insertion point when the user moves the caret position with the arrow keys. See "Positioning the Caret in Response to Arrow Keypresses" beginning on page 10-22.

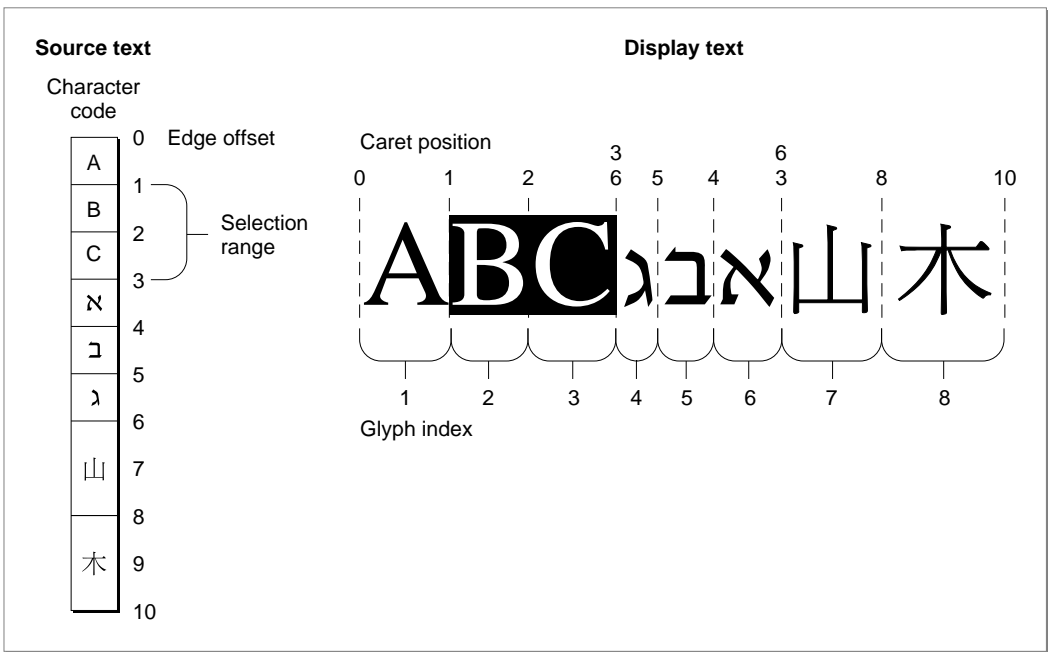
Highlighting

Highlighting is the display of portions of a line of text in inverse video or with a colored background. Just as a caret is the onscreen representation of an insertion point in source text, highlighting is the onscreen representation of a selection range in source text.

A **selection range** is the contiguous sequence of characters in memory that the next editing operation (deletion or replacement) will affect. The onscreen glyphs corresponding to those characters are commonly highlighted. The characters in a selection range are always contiguous in memory, but their glyphs are not necessarily contiguous on screen.

Like insertion points and carets, selection ranges and highlighting are described by edge offsets and caret positions. Figure 10-9 shows the simplest example of highlighting. The text represents a layout shape with mixed Roman, Hebrew, and Chinese text in which the dominant direction is left to right. A selection range from edge offsets 1 to 3 yields a simple, contiguous highlighting rectangle. The rectangle encloses two glyphs that correspond exactly to the two characters in the selection range.

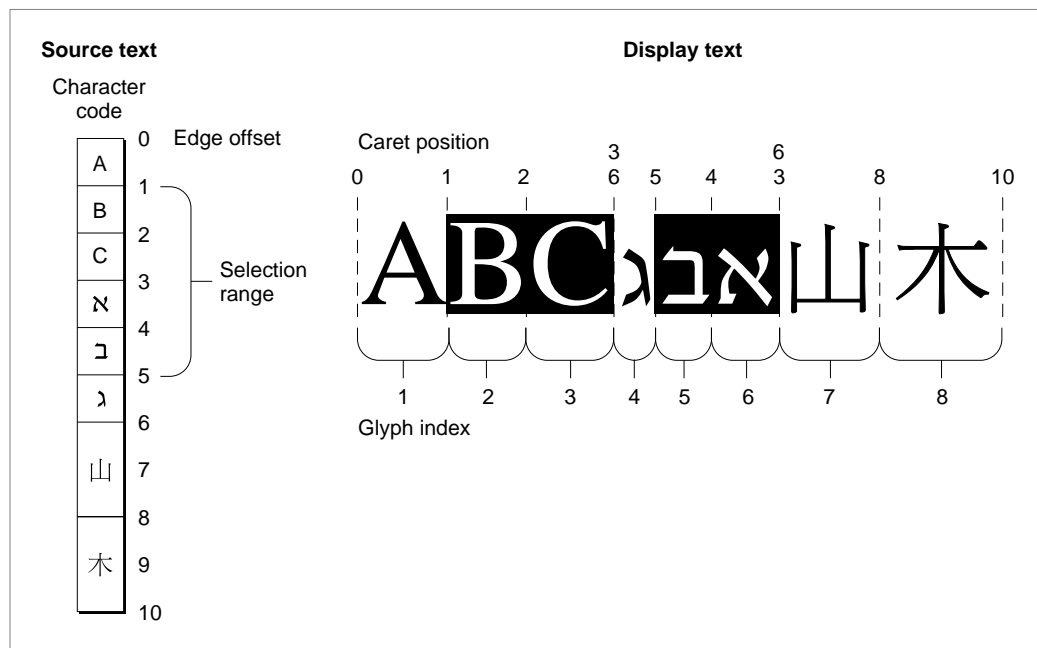
Figure 10-9 Highlighting in single-direction text



Visually Discontiguous and Contiguous Highlighting

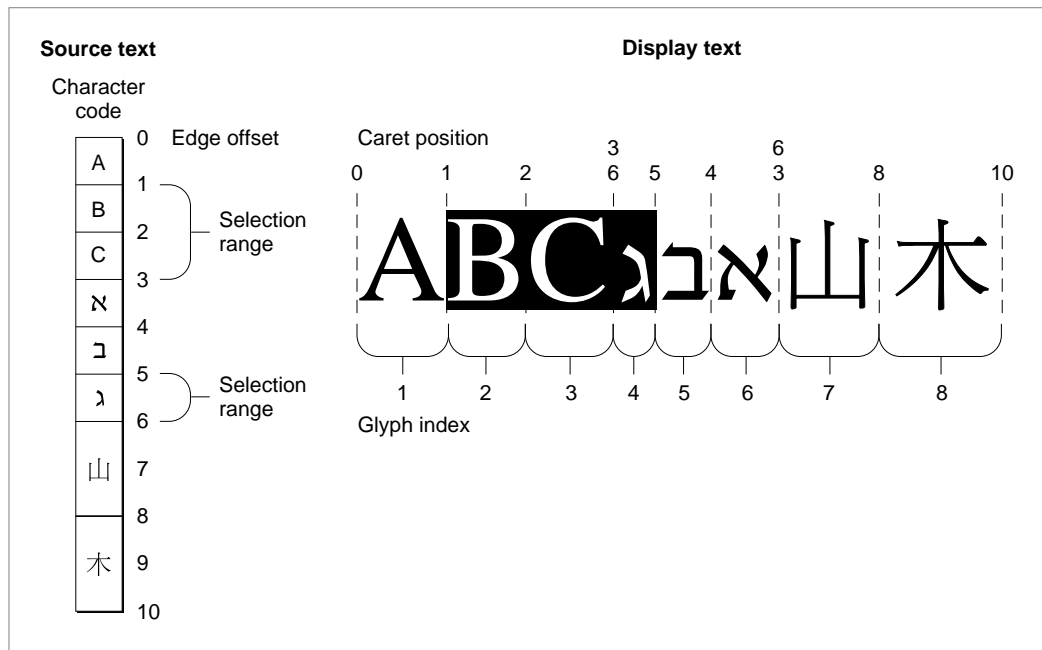
Complications arise in highlighting when a selection range crosses direction boundaries. In Figure 10-10, which represents the same layout shape as Figure 10-9, the selection range extends from offset 1 to offset 5. The equivalent highlighted area on the screen consists of two separate rectangles. Note that the highlighted glyphs still correspond exactly to the characters in the selection range. In more complicated layout shapes with many direction boundaries and complex levels of direction runs, a single selection range can become many discontiguous highlighted areas on the screen.

Figure 10-10 Discontiguous visual highlighting in mixed-direction text



The kind of highlighting shown in Figure 10-9 and Figure 10-10 is the most typical, and QuickDraw GX takes care of calculating the contiguous or discontiguous highlighting shapes that represent any selection range that you specify. However, for display simplicity, you can also specify that a single visually contiguous highlighting shape be used between two offsets, even if it crosses direction boundaries. Figure 10-11 shows an example of this visually contiguous highlighting, using the same layout shape and the same edge offsets as used by the selection range in Figure 10-10.

Note that the selection range represented by the highlighting in Figure 10-11 is *not* equivalent to the selection range represented in Figure 10-10, even though the same edge offsets apply in each case. (For contiguous highlighting, leading-edge information,

Figure 10-11 Contiguous visual highlighting in mixed-direction text

as discussed below under “Hit-Testing,” is also required.) In general, when contiguous highlighting crosses direction boundaries in text, the selection range is discontinuous and does *not* correspond exactly to the characters between the two edge offsets represented.

Caret Angle and Tiled Highlighting

Just as QuickDraw GX supports angled carets within text that has slanted glyphs, it also allows you to slant the edges of highlighted areas in the same way. If you specify angled carets, the edges of highlighted areas will also be angled, where appropriate. In that case, your highlighting areas may be parallelograms or even trapezoids, instead of rectangles. See, for example, Figure 10-19 on page 10-27.

It is usually important to make all possible highlighted areas in a line of text unique. Every character’s highlighted area should be disjoint from all other characters’ areas, with the union of all characters’ highlighted areas being exactly equal to the entire line’s highlight area. In that case, there are no gaps or overlaps between adjacent highlight areas, and there is never any uncertainty in the interpretation of a hit-test for any point within the area of the line. Highlighting that meets these criteria is called **tiled highlighting**.

If you specify straight highlighting (by specifying straight carets), the highlighted areas calculated by QuickDraw GX are always tiled. If you specify angled highlighting, the highlighted areas calculated by QuickDraw GX are almost always tiled; only in cases of extreme slant with superscripts or subscripts is tiling not maintained.

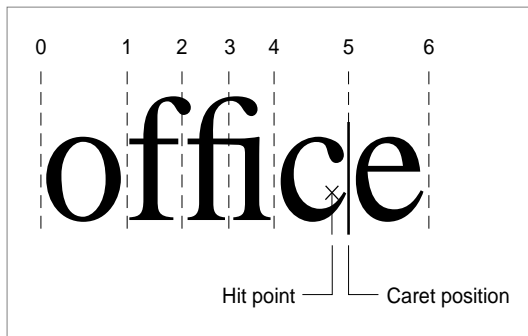
Hit-Testing

Hit-testing is the process of converting a location within a line of display text into an edge offset in the source text of that line. Its purpose is to allow you to convert user actions on displayed glyphs into editing operations on the characters of a typographic shape.

An important concept for hit-testing is that of leading and trailing edges. A glyph's **leading edge** is the edge of a glyph that is encountered first when reading text of that glyph's language. Its **trailing edge** is the edge encountered last. For glyphs of left-to-right text, the leading edge is the left edge; for glyphs of right-to-left text, the leading edge is the right edge.

Figure 10-12 shows the basics of hit-testing. The hit point (×) may represent, for example, the location of a mouse click. In response, your application needs to find the correct edge offset for subsequent text insertion and then draw a caret at the proper caret position.

Figure 10-12 Hit point and caret position in hit-testing



When used for hit-testing layout shapes, QuickDraw GX tells you which edge offset corresponds to the hit point and whether the hit was on the leading edge or the trailing edge of the hit glyph. In Figure 10-12, the hit point is within the area of the glyph “c” and closer to its trailing edge than its leading edge. The logical place to draw the caret is thus between glyphs “c” and “e.” In this case, QuickDraw GX returns an edge offset of 5 and a leading-edge value of `false`. You can then pass that edge offset back to another QuickDraw GX function to obtain the correct caret to draw. QuickDraw GX returns a polygon shape of the right size, angle, and position (caret position 5).

If the hit point had been within glyph “e,” near its leading edge, the returned edge offset would still have been 5 (because the proper caret position would still be 5), but the leading-edge value would have been `true`.

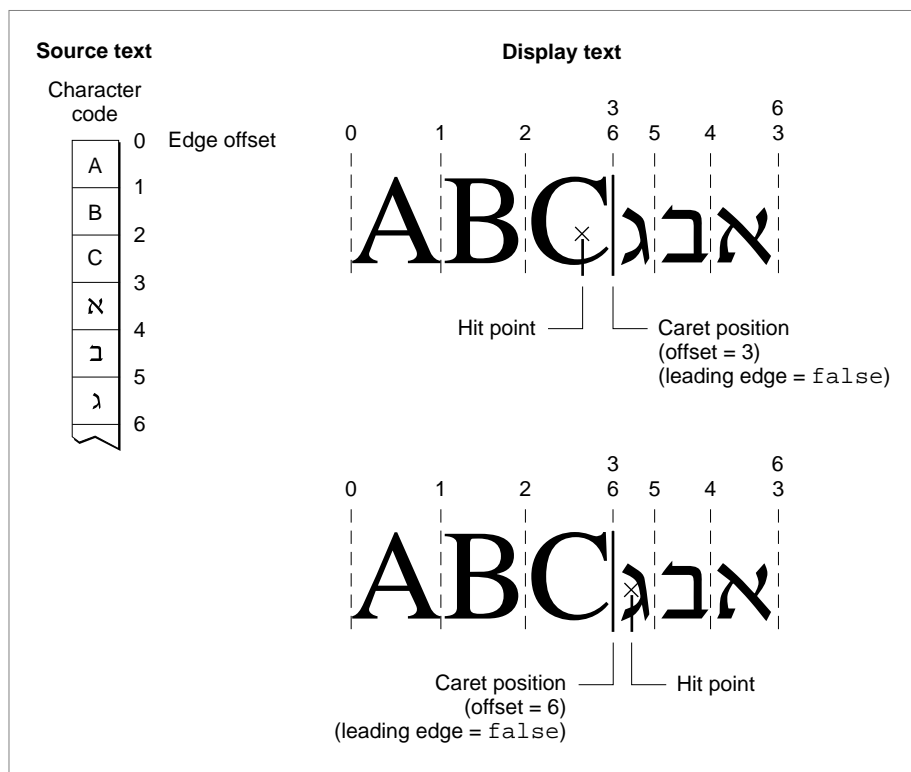
In single-direction text, the leading-edge value is not needed, but at direction boundaries in mixed-direction text both the offset and the leading-edge value are needed to know what kind of text insertion is expected—and where—in the source text. For a given

offset, if the leading edge value is `true`, the text to be inserted has the direction of the source-text character *following* the edge offset. If the leading edge value is `false`, the text to be inserted has the direction of the source-text character *preceding* the edge offset.

For example, Figure 10-13 shows mixed Roman and Hebrew text from the same layout shape as shown in Figure 10-11 on page 10-15. Neighboring hit points on either side of a direction boundary give two different results:

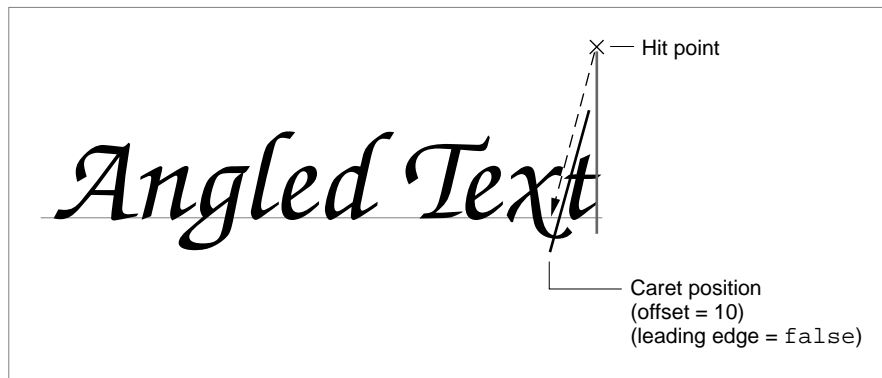
- The first hit point yields an edge offset of 3, which could mean insertion of either Roman text after the “c” or Hebrew text before the “א”. The leading-edge designation of `false` means that Roman text is to be entered at offset 3.
- The second hit point is close to the first but yields an offset of 6. That offset could mean insertion of either Hebrew text after the “א” or left to right text before the character starting at offset 6. The leading-edge designation of `false` means that in this case Hebrew text is to be entered at offset 6.

Figure 10-13 Hit-testing in mixed-direction text



For angled text, the hit point is projected to the baseline, parallel to the caret angle, to determine the correct edge offset. As shown in Figure 10-14, the caret projection to the baseline can yield a completely different caret position than a vertical projection would.

Figure 10-14 Projecting the hit point to the baseline



QuickDraw GX returns other information besides the offset and leading-edge value when you use it for hit-testing layout shapes. You can use the information to tell how close the hit point was to either edge of the hit glyph, what the edge offset of either side of the hit glyph is, whether or not the hit point was actually within the area of the text line, and what area around the hit point would yield the same edge offset. See “Performing Hit-Testing” beginning on page 10-28 for more information and examples.

Using Carets, Highlighting, and Hit-Testing With Layout Shapes

This section describes how to use QuickDraw GX functions to draw carets, to highlight, to hit-test, and to analyze glyphs for their direction and for their relationship to characters in a layout shape’s source text.

Drawing Carets

QuickDraw GX provides functions that let you locate and draw a caret correctly in the display text of any layout shape. This section shows you how to draw single and split carets in simple and complex text, and how to change the caret position when the user presses the Right or Left Arrow key.

Getting the Caret Shape

The `GXGetLayoutCaret` function takes an edge offset in a layout shape and returns a shape that represents a text caret. Determining the caret's shape means determining its position on the screen, its angle in italic or otherwise slanted text, and its height (to correspond to the text size). In accordance with QuickDraw GX typographic conventions, the caret is a single line when positioned between two characters of like directionality, and possibly a split caret (a pair of short lines at different places) at the boundaries between text of opposing direction.

The caret returned by `GXGetLayoutCaret` has all those properties automatically specified. Because the caret is a shape object, it has associated style, ink, and transform objects. The style and ink have the same default properties as those for any polygon shape. You can specify details of a caret's appearance, such as its thickness, color, and transfer mode, by modifying its style or ink objects. The caret shape's transform object is always identical to the layout shape's transform, so that the caret matches any moved, scaled, or rotated text.

The `GXGetLayoutCaret` function generates a split caret at direction boundaries if you specify `gxSplitCaretType` for the `caretType` parameter. When italic text occurs in a line containing mixed directions and the caret is split, the top and bottom portions of the caret may have different slants.

Listing 10-1 is a partial listing of a test function that creates and draws a layout shape twice, with the text specified in the string `myString` (of byte length `len`). The function then determines the caret shape for each edge offset in the layout shape's source text and draws the caret. The first time, it specifies `gxHighlightAverageAngle` in `GXGetLayoutCaret` to ensure that if the text is angled, the caret will be angled also; the second time, it specifies `gxNoCaretAngle` to make the caret vertical regardless of the text slant. (Specifying `gxSplitCaretType` ensures that two carets are drawn for caret positions at direction boundaries in mixed-direction text.)

Listing 10-1 Drawing angled and straight carets at all caret positions

```
char          *myString = "Angled Text";
gxStyle       myStyle;
gxPoint       myPoint;
gxShape       caret, highlight, layout;
short         i, len = 0;
gxStyle       myStyle;

.
.
.

len = strlen(myString);
myStyle = NewLayoutStyle((char *)
                        "\pZapf Chancery Medium Italic", ff(50),
                        0, nil, nil, 0, nil);
```

Layout Carets, Highlighting, and Hit-Testing

```

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

for (i = 0; i <= len; i++)
{
    caret = GXGetLayoutCaret(layout, i,
                             gxHighlightAverageAngle,
                             gxSplitCaretType, nil);
    GXDrawShape(caret);
    GXDisposeShape(caret);
}
GXMoveShape(layout, 0, ff(75));
GXDrawShape(layout);

for (i = 0; i <= len; i++)
{
    caret = GXGetLayoutCaret(layout, i,
                             gxNoCaretAngle,
                             gxSplitCaretType, nil);
    GXDrawShape(caret);
    GXDisposeShape(caret);
}

```

The result is two lines of display text with carets drawn at each caret position, as shown in Figure 10-15.

Figure 10-15 Drawing all possible caret positions in a layout shape's text



Listing 10-2 is a code fragment that demonstrates how to specify different caret types for a given offset in a layout shape. The code first creates a new layout shape with two style runs (one in Roman text and one in Arabic text; the text runs are specified in `textPtrs`, their lengths are specified in `runLengths`, their style objects in `styleArray`). It then draws the layout three times (starting at `myPoint`), each time drawing a caret at the screen position corresponding to edge offset 4 in the source text—the boundary between the Roman and Arabic text.

Listing 10-2 Drawing three different types of caret at one edge offset

```
/* create and draw a layout shape with two style runs */
layout = GXNewLayout(2, runLengths, (void *)textPtrs,
                    2, runLengths, styleArray,
                    0, nil, nil,
                    nil, &myPoint);
GXDrawShape(layout);

/* first draw a split caret at offset 4 */
caret = GXGetLayoutCaret(layout, 4, gxHighlightStraight,
                        gxSplitCaretType, nil);
GXDrawShape(caret);

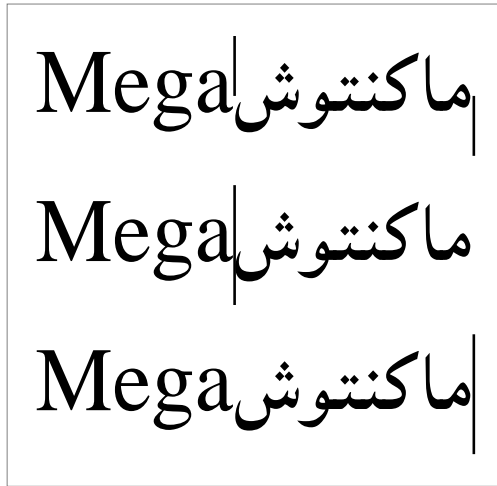
/* move the layout shape downward and redraw */
GXMoveShape(layout, 0, ff(72));
GXDrawShape(layout);

/* next draw a left-to-right caret at offset 4 */
GXGetLayoutCaret(layout, 4, gxHighlightStraight,
                gxLeftRightKeyboardCaret, caret);
GXDrawShape(caret);

/* move the layout shape downward and redraw */
GXMoveShape(layout, 0, ff(72));
GXDrawShape(layout);

/* finally draw a right-to-left caret at offset 4 */
GXGetLayoutCaret(layout, 4, gxHighlightStraight,
                gxRightLeftKeyboardCaret, caret);
GXDrawShape(caret);
```

Figure 10-16 shows the results of executing this code. The layout is drawn with a split caret, then with a left-to-right caret, and finally with a right-to-left caret. Your application chooses which caret to draw during program execution; if you choose a single caret, you should choose the one that corresponds to the current direction (left to right or right to left) for text input by the user.

Figure 10-16 Drawing different caret types at a single edge offset

The `GXGetLayoutCaret` function is described on page 10-44. The `gxCaretType` enumeration is described on page 10-41. The `GXNewLayout` function is described in the chapter “Layout Shapes” in this book.

Drawing the Cursor at the Correct Angle Within a Given Area

In addition to controlling the shape, location, and style of the caret, you may also want to modify the cursor, as mouse movements cause it to pass over different parts of your text. For example, you may want to slant the standard Macintosh “I-beam” cursor when it passes over italic text.

You could do this by making repeated calls to `GXGetLayoutCaret` as the cursor moves, and using the resulting shape to calculate the proper angle for the cursor. The `GXGetCaretAngleArea` function returns the angle for the caret, and therefore for the cursor, corresponding to a given point in the display text. The function also returns the tracking area for that angle—the contiguous area in which the angle stays the same.

The `GXGetLayoutCaret` and `GXGetCaretAngleArea` functions are described on page 10-44 and page 10-46, respectively.

Positioning the Caret in Response to Arrow Keypresses

When the user presses the Right Arrow key, the caret should move one position to the right; when the user presses the Left Arrow key, the caret should move one position to the left. You can use the `GXGetRightVisualOffset` and `GXGetLeftVisualOffset` functions to determine where to place the caret when either the Right or Left Arrow key is pressed.

These functions take an edge offset in the source text and return an offset that you can pass to `GXGetLayoutCaret` (page 10-44) to get a caret at the next position. The `GXGetRightVisualOffset` function returns the edge offset corresponding to the next

Layout Carets, Highlighting, and Hit-Testing

caret position to the right (or down); `GXGetLeftVisualOffset` returns an edge offset corresponding to the next caret position to the left (or up). If you are using a split caret with your text, `GXGetRightVisualOffset` and `GXGetLeftVisualOffset` return a edge offset corresponding to the position next to the dominant (high) caret.

Listing 10-3 is a portion of a key-down event handler for a layout-editing library. The listing shows the response of the handler to arrow keypresses. In the left-arrow case, the handler calls `GXGetLeftVisualOffset` to get the offset for the next caret position, defines the selection range as that edge offset, and then highlights (draws the caret).

The function in Listing 10-3 uses the library-defined function and `ShowHighlight` to redraw the highlighting or caret, and `NewSelection` to update the library's layout data structures.

Listing 10-3 A key-down handler using `GXGetRightVisualOffset` and `GXGetLeftVisualOffset`

```
void LayoutEditKey(LayoutEditHandle handle, char key)
{
    switch (key)
    {
        case leftArrow:
        case rightArrow:
        {
            .
            .
            .

            if (key == leftArrow)
                newCaret = GXGetLeftVisualOffset(layout->layout,
                                                    layout->selectionRanges.ranges[0].minOffset);
            else
                newCaret = GXGetRightVisualOffset( layout->layout,
                                                    layout->selectionRanges.ranges[0].maxOffset);

            NewSelection(layout, newCaret, newCaret);
            ShowHighlight(layout);
            break;
        }

        case backSpace:
        {
            .
            .
            .
    }
```

Layout Carets, Highlighting, and Hit-Testing

The `GXGetRightVisualOffset` and `GXGetLeftVisualOffset` functions are described on page 10-47 and page 10-48, respectively.

Positioning the Caret Within Ligatures

You can specify that caret positions within a ligature are not to be permitted, meaning that caret-drawing, highlighting, and hit-testing must consider that ligatures are indivisible glyphs. You do this by setting the `gxNoLigatureSplits` flag in the run controls structure of the style run containing the ligatures.

Listing 10-4 draws the same line of text twice, first without specifying any value for `gxNoLigatureSplits`, and then with the flag set. It first creates the layout shape at the location `myPoint`. The style object of the shape uses the run controls structure `runControls`.

Listing 10-4 Preventing ligature splits for caret positioning

```
void LigatureSplits(WindowPtr sampleWindow)
{
    char          *myString = "flat fin";
    short         i, len = 0;
    gxPoint       myPoint;
    gxRunControls runControls;
    gxShape       caret, layout;
    gxStyle       myStyle;
    .
    .
    .
    myStyle = NewLayoutStyle((char *) "\pTimes Roman", ff(60),
                           0, nil, nil, 0, nil);

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

    for (i = 0; i < (len + 1); i++)
    {
        caret = GXGetLayoutCaret(layout, i,
                                gxHighlightAverageAngle,
                                gxSplitCaretType, nil);

        GXDrawShape(caret);
        GXDisposeShape(caret);
    }
}
```

Layout Carets, Highlighting, and Hit-Testing

```

runControls.flags = gxNoLigatureSplits;
GXSetStyleRunControls(myStyle, &runControls);
GXMoveShape(layout, 0, ff(75));
GXDrawShape(layout);

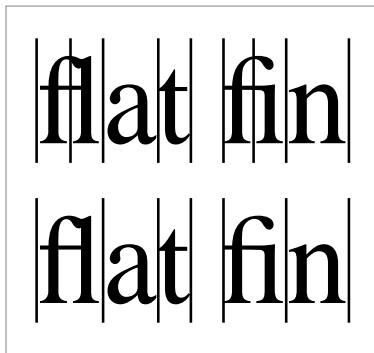
for (i = 0; i < (len + 1); i++)
{
    caret = GXGetLayoutCaret(layout, i,
                                gxHighlightAverageAngle,
                                gxSplitCaretType, nil);

    GXDrawShape(caret);
    GXDisposeShape(caret);
}
.
.
.

```

Figure 10-17 shows the results of executing the code in Listing 10-4. In the upper line, carets are drawn in the middle of the “fl” and “fi” ligatures; in the lower line, there are no valid caret positions within the ligatures.

Figure 10-17 All caret positions drawn with (upper) and without (lower) ligature splits



Drawing Highlighting

Highlighting single-direction text in a layout shape is straightforward. You call the `GXGetLayoutHighlight` function, passing it the two edge offsets between which you want the highlight to appear; the function returns a rectangular or trapezoidal shape that corresponds in size and location to the highlighted area. The with-stream edges of the highlight shape (the right and left edges, in horizontal text) are perpendicular or slanted, as appropriate, depending on whether the text is slanted and whether you have specified angled highlighting.

Layout Carets, Highlighting, and Hit-Testing

When you draw the highlight, you most typically use the highlight color specified by the user and the `gxHighlightMode` transfer mode.

The following code fragment creates and draws a layout shape with five style runs, one of which has a right-to-left text direction. It then highlights the area between edge offsets 4 and 13, all of which is within left-to-right text. In the call to `GXGetLayoutHighlight`, it specifies `gxHighlightAverageAngle` to slant the edge of the highlight that abuts slanted text. This code uses the library functions `SetShapeCommonTransfer` and `SetShapeCommonColor` to set up the ink object for the highlight shape.

```
layout = GXNewLayout( 5, textLengths, (void *) textRuns,
                    5, textLengths, textStyles,
                    0, nil, nil,
                    &layoutOptions, &posn);

GXDrawShape (layout);

/* make a highlight shape from offset 4 to offset 13 */
highlight = GXGetLayoutHighlight (layout, 4, 13,
                                gxHighlightAverageAngle, nil);
SetShapeCommonTransfer (highlight, gxHighlightMode);
SetShapeCommonColor (highlight, gxWhite);
GXDrawShape(highlight);
```

Figure 10-18 shows the results of the highlighting.

Figure 10-18 Contiguous highlighting from offsets 4 to 13 in single-direction text



The wicked fast ماكنتوش lives.

Highlighting Discontiguously in Mixed-Direction Text

In mixed-direction text, highlighting is just as straightforward as in single-direction text, although it can be more complex in appearance, and there are two possible highlights you may want to generate. The `GXGetLayoutHighlight` function returns a highlighted area that corresponds exactly to the selection range defined by the two edge offsets. The highlighting can be visually discontiguous when the selection range crosses direction boundaries.

The following code fragment highlights the selection range between edge offsets 4 and 19—crossing a direction boundary—in the same layout shape created in the previous

fragment. Like the previous example, this code calls `GXGetLayoutHighlight` and specifies `gxHighlightAverageAngle`.

```
/* make a highlight shape from offset 4 to offset 19 */
highlight = GXGetLayoutHighlight (layout, 4, 19,
                                gxHighlightAverageAngle, nil);
SetShapeCommonTransfer (highlight, gxHighlightMode);
SetShapeCommonColor (highlight, gxWhite);
GXDrawShape(highlight);
```

Figure 10-19 shows the results of the highlighting. The selection range appears as two separate highlighted trapezoids.

Figure 10-19 Discontiguous highlighting from offsets 4 to 19 in mixed-direction text



Highlighting Contiguously in Mixed-Direction Text

The highlight shape returned by `GXGetLayoutHighlight` corresponds exactly to the selection range defined by the two offsets passed to it. However, the discontiguous nature of the highlighting in mixed-direction text may confuse some users, especially when the highlighting is dynamic—that is, when the highlighting is continually drawn and redrawn as the user moves the cursor through the text while holding down the mouse button.

You can obtain a simpler shape for highlighting by calling the `GXGetLayoutVisualHighlight` function. The `GXGetLayoutVisualHighlight` function always returns a single, visually contiguous highlighted area representing the visual range between the caret positions and leading-edge states of the two specified offsets, even when the text has mixed directions. Because the highlighted area is visually contiguous, the selection range it defines does not always correspond exactly to the range between the two offsets in the source text. If the user performs an editing operation (such as deleting, cutting, or pasting) on a part of the text that is highlighted continuously, you must be sure to replace the characters in the source text that represent the highlighted glyphs exactly, and not simply all the characters between the offsets used to generate that highlight.

The following code fragment again highlights the area between edge offsets 4 and 19 in the same layout shape created in the previous fragments. Unlike the previous examples, however, this code calls `GXGetLayoutVisualHighlight` instead of

Layout Carets, Highlighting, and Hit-Testing

`GXGetLayoutHighlight`. (Also, unlike the code that calls `GXGetLayoutHighlight`, this code must specify whether to highlight from the leading edge or trailing edge of the glyph corresponding to each offset.)

```
/* make contiguous highlighting from offset 4 to offset 19 */
highlight = GXGetLayoutVisualHighlight (layout,
                                         4, true, 19, false,
                                         gxHighlightAverageAngle, nil);
SetShapeCommonTransfer (highlight, gxHighlightMode);
SetShapeCommonColor (highlight, gxWhite);
GXDrawShape(highlight);
```

Figure 10-20 shows the results of the executing this code. Note that, even though the offsets passed to the highlighting functions are the same, some glyphs that are not highlighted in Figure 10-19 are highlighted Figure 10-20. Conversely, some glyphs that are highlighted in Figure 10-19 are not highlighted Figure 10-20.

Figure 10-20 Contiguous highlighting from offsets 4 to 19 in mixed-direction text



The *wicked fast* ماكنتوش *lives.*

Providing Dynamic Highlighting

Dynamic highlighting is the process of continually drawing and redrawing the highlighted area as the user moves the cursor through the text while holding down the mouse button. Dynamic highlighting works in conjunction with hit-testing, described in the next section.

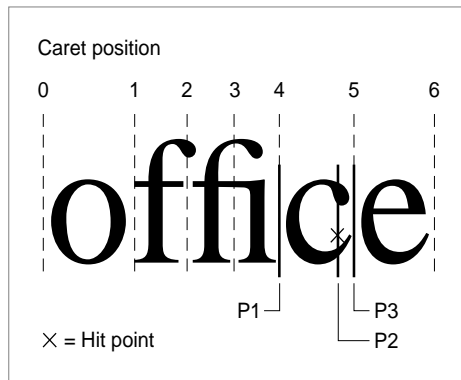
If your application uses `GXHitTestLayout` for hit-testing, your hit-test function can perform dynamic highlighting by making repeated calls to `GXHitTestLayout` (and `GXGetLayoutHighlight` or `GXGetLayoutVisualHighlight`) as long as the mouse button is held down. Listing 10-5 on page 10-31 shows an example of dynamic highlighting. (That example does not show the call to `GXGetLayoutHighlight`; that call occurs in the application-defined function `NewSelectionAndHighlight`.)

Performing Hit-Testing

You can use the `GXHitTestLayout` function to determine the source-text offset corresponding to a hit anywhere in the display text of a layout shape. The function returns (a) the edge offset corresponding to the closest edge of the glyph beneath the hit point and (b) the edge offset of the other edge of that same glyph.

For example, in Figure 10-21, the source text for the word “office” has edge offsets 0 through 6, and the display text consists of five glyphs (including the “fi” ligature). Above the glyphs, the numbers denote the edge offsets equivalent to the glyph edges.

Figure 10-21 `GXHitTestLayout` example



As you read the following sections, assume a hit occurs at the spot marked “X”, above point P2, near the trailing edge of the “c.” The next section discusses how `GXHitTestLayout` interprets the significance of the hit.

Layout Hit Info Structure

The `GXHitTestLayout` function returns information in a layout hit info structure, is described on page 10-43. That structure gives all the relevant information about the hit.

In Figure 10-21, the offset closest to the hit is 5, so in this case the value of the `hitSideOffset` field in the `layoutHitInfo` structure is 5. This value specifies the edge offset corresponding to the caret position between the glyphs “c” and “e.” The value of the `nonHitSideOffset` field is 4, which specifies the other side of the “c” glyph. If the hit had occurred on the left side of the “e” glyph, the value of `hitSideOffset` would still have been 5, but the value of `nonHitSideOffset` would have been 6.

In this example, the value of the `leadingEdge` field in the `layoutHitInfo` structure is `false`, because the hit occurred nearest the trailing edge of the glyph. If the hit had occurred on the left side of the glyph, or if the glyph had right-to-left directionality and the hit was on the right side, the `leadingEdge` field would have been `true`.

The `layoutHitInfo` structure contains two with-stream distances as well. In this example, the value of the `firstPartialDist` field is the left-side partial distance: the horizontal distance from the point P1 to the point P2 in this case. The value of `lastPartialDist` is the right-side partial distance: the horizontal distance from the point marked P2 to the point marked P3 in this case.

Layout Carets, Highlighting, and Hit-Testing

The `layoutHitInfo` structure also contains, in the `inLoose` field, an indication of whether or not the hit was actually within the area of the layout shape. In Figure 10-21, the value of the `inLoose` field is `true`, because the value of the hit point specifies that the hit occurred within the layout's area. If the hit point had been at the same horizontal position but above or below the line of text, all the returned information would have been the same, except that the value of `inLoose` would have been `false`. The `GXHitTestLayout` function projects the hit point to the baseline of the text so that your application can highlight the text dynamically, even if the mouse cursor strays out of the area of the layout shape while the mouse button is held down. (In multiline text, if the cursor strays far enough out of the layout shape's area, you would instead extend the dynamic highlight into the layout shapes that represent the other lines the cursor is passing over.)

If the hit had occurred in the middle of the “fi” ligature, the values of the two offsets returned in the `layoutHitInfo` structure would depend on whether the ligature is treated as a single glyph or as two partial glyphs; see “Caret Position and Split Ligatures” on page 10-10. If you had specified that ligatures are to be treated as single glyphs, and if the hit had occurred in the “fi” ligature in Figure 10-21, `GXHitTestLayout` would return either 2 or 4 as the value of `hitSideOffset`, depending on whether the hit point was closer to the “f” portion or the “i” portion of the ligature. If you had specified that ligatures are to be split, and the hit were near the center of the ligature, the value of `hitSideOffset` would be 3.

Mouse Tracking Area

The `GXHitTestLayout` function returns a modification of a shape that you pass it. The shape defines the mouse tracking area for the returned edge offset (`hitSideOffset`). The mouse tracking area is that area in the display text of the layout shape for which hits will yield the same edge offset. For repositioning the caret or for providing dynamic highlighting, for example, you can use the mouse tracking area to minimize the need for calls to `GXHitTestLayout`. In other words, you need to call `GXHitTestLayout` to get a new edge offset only when the mouse moves outside of this tracking area. You do not need to pass a shape if you don't wish to do mouse tracking in this way.

Sample Hit-Test Function

Listing 10-5 is a hit-test function from a layout-shape editing library. It demonstrates the use of `GXHitTestLayout` to convert mouse clicks to offsets for the purpose of drawing carets and highlighting. It also performs dynamic highlighting while the user holds down the mouse button.

This function refers to the layout shape with a pointer (`layout`), and uses the library-defined functions `GetShapeViewPort` (to get the layout's view port), `NewSelectionAndHighlight` (to update the selection range and highlight shape), and `DisposeShapeAt` (to dispose of shapes).

Listing 10-5 Using the GXHitTestLayout function

```

void LayoutEditClick(LayoutEditHandle handle, gxPoint hitDown)

/* lock the layout edit handle, initialize variables */
{
    LayoutEditPtr    layout = LockEditHandle(handle);
    gxPoint          lastPoint = hitDown;
    SelectionOffset  firstHitOffset, lastHitOffset;
    gxLayoutHitInfo  hitInfo;
    boolean          oldIsCaret, newIsCaret = true;
    gxShape          diffHighlight = nil, oldHighlight = nil;
    gxViewPort layoutViewPort = GetShapeViewPort(layout->layout);

    /* get the offset for the hit point */
    GXHitTestLayout(layout->layout, &hitDown,
                    gxHighlightAverageAngle, &hitInfo, nil);
    firstHitOffset = lastHitOffset =
        (SelectionOffset) hitInfo.hitSideOffset;

    /* erase the old highlight (stored in layout edit structure) */
    GXDrawShape(layout->highlight);

    /* make the selection a caret at firstHitOffset */
    NewSelectionAndHighlight(layout, firstHitOffset,
                            firstHitOffset);
    GXDrawShape(layout->highlight);

    /*
       Recompute the selection and the highlight as long as the
       mouse button is still down.
    */
    while (Button())
    {
        GXGetViewPortMouse(layoutViewPort, &hitDown);

        /* continue if the mouse hasn't moved */
        if (hitDown.x == lastPoint.x &&
            hitDown.y == lastPoint.y) continue;

        lastPoint = hitDown;
        GXHitTestLayout(layout->layout, &hitDown,
                        gxHighlightAverageAngle, &hitInfo, nil);
    }
}

```

Layout Carets, Highlighting, and Hit-Testing

```

/* continue if the selection hasn't changed */
if (hitInfo.hitSideOffset == lastHitOffset) continue;

/* save the old highlight and calculate the new one */
oldIsCaret = newIsCaret;
lastHitOffset = (SelectionOffset) hitInfo.hitSideOffset;
newIsCaret = (lastHitOffset == firstHitOffset);

if (oldIsCaret != newIsCaret)
/*
    If it has changed from a caret to a highlight
    or vice versa, redraw the entire highlight or caret.
*/
{
    GXDrawShape(layout->highlight);          /* erase old */
    NewSelectionAndHighlight(layout, firstHitOffset,
                            lastHitOffset);
    GXDrawShape(layout->highlight);          /* draw new */
}
else
/*
    Otherwise, to reduce flicker, draw only the difference
    between the new and the old highlight.
*/
{
    oldHighlight = GXCopyToShape(oldHighlight,
                                layout->highlight);
    NewSelectionAndHighlight(layout, firstHitOffset,
                            lastHitOffset);
    diffHighlight = GXCopyToShape(diffHighlight,
                                layout->highlight);
    GXExcludeShape(diffHighlight, oldHighlight);
    GXDrawShape(diffHighlight);              /* draw difference */
}
}
DisposeShapeAt(&diffHighlight);
DisposeShapeAt(&oldHighlight);
}

```

The layout hit info structure is described on page 10-43.

Analyzing Glyphs

You can use the functions described in this chapter to analyze several aspects of the relationship between a glyph and its characters. You can determine the directionality of a glyph, you can determine the edge offsets corresponding to both edges of a ligature, you can convert a glyph index to an edge offset, and you can convert an edge offset to a glyph index.

Determining the Direction of a Glyph

To determine the direction of a glyph in a complex layout shape that has mixed-direction text and possibly several levels of direction runs, you can call the `GXHitTestLayout` function for the left side of the glyph and test the value of the `leadingEdge` flag in the `gxLayoutHitInfo` structure returned by the function. If the value of the flag is `true`, the glyph is left to right; otherwise it is right to left. (Vertical text is always considered to be left to right by QuickDraw GX.)

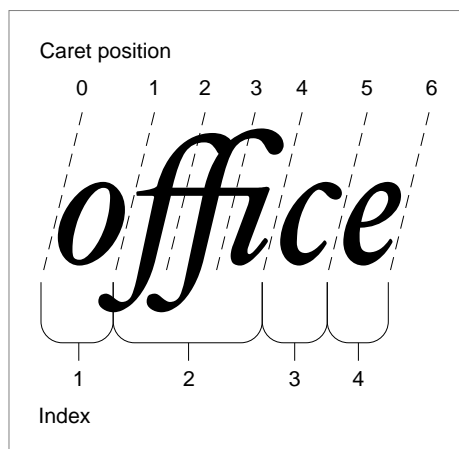
You can use the `GXGetLayoutGlyphs` function to obtain the location of any glyph in the layout shape, and pass that information to `GXHitTestLayout` to get the flag value.

Determining the Offsets for Each Edge of a Ligature

You can use the `GXGetCompoundCharacterLimits` function to find the bounding offsets equivalent to the leading and trailing edges of a character. The function is especially useful for determining the bounding offsets of a ligature, which may represent several characters.

For example, the text of a layout shape containing the word “office” is displayed in Figure 10-22. The source text has edge offsets 0 through 6, the equivalent caret positions for which are shown in the display text. If the “f,” “fi,” and “i” characters are represented by a single “ffi” ligature, then the display text itself consists of only four glyphs, with glyph indices as shown.

Figure 10-22 Caret positions and glyph indexes for one display version of the word “office”



Layout Carets, Highlighting, and Hit-Testing

You pass the `GXGetCompoundCharacterLimits` function a trial offset: an edge offset bordering on or interior to the glyph of interest. The `GXGetCompoundCharacterLimits` function returns a minimum offset and a maximum offset representing the bounding edges of that glyph. If the trial offset is between glyphs in the display text, `GXGetCompoundCharacterLimits` notifies you of that condition and considers both bounding glyphs as a single glyph for the purposes of computing minimum and maximum offset.

For example, if you call `GXGetCompoundCharacterLimits` for the text in the layout shape shown in Figure 10-22, passing it in turn each possible offset, the function returns the following results:

Trial offset	Minimum offset	Maximum offset	On boundary?
0	0	1	true
1	0	4	true
2	1	4	false
3	1	4	false
4	1	5	true
5	4	6	true
6	5	6	true

Finding the Equivalent Glyphs to an Offset in the Source Text

You can directly convert an edge offset into an identification of an individual glyph or glyphs in the display text of a layout shape. This conversion is useful, for instance, if you want to substitute one glyph for another or to perform a graphics operation on a glyph at a particular offset.

The function shown in Listing 10-6 locates and draws a box around the glyph whose trailing edge corresponds to edge offset 6 in the source text of a layout shape. It uses the `GXGetOffsetGlyphs` function for that purpose. Depending on information in the run-features array of the style object associated with the layout shape—which in this case controls which ligatures may be formed—that one offset may correspond to different glyphs. Listing 10-6 draws the layout shape and highlights the glyph at offset 6 three times, one for each of three different ligature settings.

The listing first creates the layout shape at the location `myPoint`, and uses the library-defined function `NewLayoutStyle` to create a style object. The length of the string is `len`.

Listing 10-6 Converting an edge offset to a glyph index

```
char                *myString = "affected";
gxLayoutOffsetState offsetState;
gxRectangle         boundingBoxes[20];
gxRunFeature        runFeature[3];
```


Layout Carets, Highlighting, and Hit-Testing

```

gxShape          layout;
short            len;
gxStyle          myStyle;
unsigned short   firstGlyph, secondGlyph;
.
.
.

    /* set up the style object and run features array */
    myStyle = NewLayoutStyle((char *) "\pHoefler Text", ff(36), 0,
                           nil, nil, 0, nil);
    runFeature[0].featureType = ligaturesType;
    runFeature[0].featureSelector = requiredLigaturesOffSelector;
    runFeature[1].featureType = ligaturesType;
    runFeature[1].featureSelector = commonLigaturesOffSelector;
    runFeature[2].featureType = ligaturesType;
    runFeature[2].featureSelector = rareLigaturesOffSelector;
    GXSetStyleRunFeatures(myStyle, 3, runFeature);

/* create and draw the layout with no ligatures */
layout = GXNewLayout(1, &len, (void *) &myString,
                    1, &len, &myStyle,
                    0, nil, nil,
                    nil, &myPoint);
GXDrawShape(layout);

/*
    Draw a frame around the glyph whose trailing edge corresponds
    to edge offset 6. In this case, it's glyph 6 (index = 6).
*/
GXGetOffsetGlyphs(layout, 6, false, &offsetState,
                  &firstGlyph, &secondGlyph);
GXGetGlyphMetrics(layout, nil, boundingBoxes, nil);
GXDrawRectangle(boundingBoxes + firstGlyph - 1,
                gxClosedFrameFill);
GXDisposeShape(layout);

/* reinitialize the style; this time allow normal ligatures */
runFeature[0].featureSelector = requiredLigaturesOnSelector;
runFeature[1].featureSelector = commonLigaturesOnSelector;
GXSetStyleRunFeatures(myStyle, 3, runFeature);
myPoint.y += ff(50);

```

Layout Carets, Highlighting, and Hit-Testing

```

/* re-create and redraw the layout a second time */
layout = GXNewLayout(1, &len, (void *) &myString,
                    1, &len, &myStyle,
                    0, nil, nil,
                    nil, &myPoint);
GXDrawShape(layout);

/*
   Draw a frame again at the same offset and trailing edge;
   now it encloses a ligature whose glyph index is 5.
*/
GXGetOffsetGlyphs(layout, 6, false, &offsetState,
                  &firstGlyph, &secondGlyph);
GXGetGlyphMetrics(layout, nil, boundingBoxes, nil);
GXDrawRectangle(boundingBoxes + firstGlyph - 1,
                gxClosedFrameFill);
GXDisposeShape(layout);

/* Re-initialize the style; include all the optional ligatures */
runFeature[2].featureSelector = rareLigaturesOnSelector;
GXSetStyleRunFeatures(myStyle, 3, runFeature);
myPoint.y += ff(50);

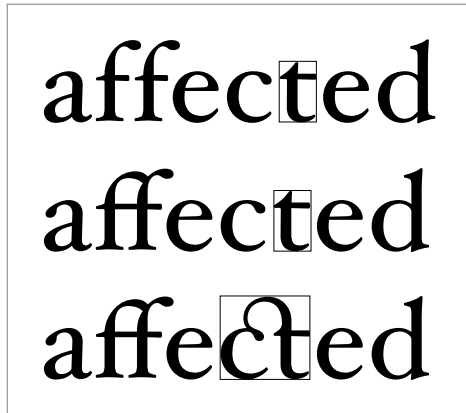
/* re-create and redraw the layout a third time */
layout = GXNewLayout(1, &len, (void *) &myString,
                    1, &len, &myStyle,
                    0, nil, nil,
                    nil, &myPoint);
GXDrawShape(layout);

/*
   Draw the frame once more at the same offset and trailing edge;
   now it encloses the ligature whose glyph index is 4.
*/
GXGetOffsetGlyphs(layout, 6, false, &offsetState,
                  &firstGlyph, &secondGlyph);
GXGetGlyphMetrics(layout, nil, boundingBoxes, nil);
GXDrawRectangle(boundingBoxes + firstGlyph - 1,
                gxClosedFrameFill);
.
.
.

```

Figure 10-23 shows the results of executing the function in Listing 10-6. The same edge offset results in a glyph with a different glyph index being framed in each case.

Figure 10-23 Using `GXGetOffsetGlyphs` to locate glyphs corresponding to known offsets



The function in Listing 10-6 uses the glyph index and a call to `GXGetGlyphMetrics` for the purpose of getting the rectangle shape to draw around the proper glyph in each case. If, after obtaining the glyph index, you need the actual glyph code that identifies the glyph within its font, you can call the `GXGetLayoutGlyphs` function. The `GXGetLayoutGlyphs` function is described in the chapter “Layout Shapes” in this book. The `GXGetGlyphMetrics` function is described in the chapter “Typographic Shapes” in this book.

Finding the Equivalent Offset to a Glyph in the Display Text

You can directly convert a glyph index (plus leading edge information) into an edge offset in the source text of a layout shape. You may need this conversion, for instance, to obtain the character code for a particular glyph, or to substitute one character for another in a layout shape’s source text.

The function shown in Listing 10-7 locates the character associated with glyph 17 in a layout shape’s display text and then highlights that character’s glyph. First it calls the `GXGetGlyphOffset` function to get the offset for the leading edge of glyph 17 and then calls `GXGetLayoutHighlight` to highlight the region from that offset to the next. Like the function in Listing 10-6 on page 10-34, this function sets values in the run features array of the layout shape’s style object to control which ligatures may be formed. Listing 10-7 draws the layout shape and highlights glyph 17 three times, once for each of three different ligature settings.

This listing draws the initial layout shape at the location `myPoint`, and uses the library-defined functions `NewLayoutStyle` (to create a style object), `SetShapeCommonTransfer` (to assign a special transfer mode for fast highlighting).

Listing 10-7 Converting a glyph index to an edge offset

```

boolean      leadingEdge, wasRealCharacter;
char         *myString = "fly-fishing aeronaut";
gxByteOffset offset;
gxRunFeature runFeature[3];
gxShape      highlight, layout;
short        len = 0;
gxStyle      myStyle;
.
.
.
len = strlen(myString);

/* set up the style object and run features array */
myStyle = NewLayoutStyle((char *) "\pTimes Roman", ff(36),
                        0, nil, nil, 0, nil);
runFeature[0].featureType = ligaturesType;
runFeature[0].featureSelector = requiredLigaturesOffSelector;
runFeature[1].featureType = ligaturesType;
runFeature[1].featureSelector = commonLigaturesOffSelector;
runFeature[2].featureType = ligaturesType;
runFeature[2].featureSelector = diphthongLigaturesOffSelector;
GXSetStyleRunFeatures(myStyle, 3, runFeature);

/* create and draw the layout, with no ligatures */
layout = GXNewLayout(1, &len, (void *) &myString,
                    1, &len, &myStyle,
                    0, nil, nil,
                    nil, &myPoint);
GXDrawShape(layout);

/*
   Get the offset for left edge of glyph 17 ( = "n"),
   then highlight from that offset to the next offset.
   (This simple example assumes a 1-byte character code.)
*/
GXGetGlyphOffset(layout, 17, true, &offset,
                 &leadingEdge, &wasRealCharacter);
highlight = GXGetLayoutHighlight(layout, offset, offset + 1,
                                gxHighlightAverageAngle, nil);
SetShapeCommonTransfer(highlight, gxHighlightMode);
GXDrawShape(highlight);
GXDisposeShape(layout);
GXDisposeShape(highlight);

```

Layout Carets, Highlighting, and Hit-Testing

```

/* reinitialize the style; this time allow normal ligatures */
runFeature[0].featureSelector = requiredLigaturesOnSelector;
runFeature[1].featureSelector = commonLigaturesOnSelector;
GXSetStyleRunFeatures(myStyle, 3, runFeature);
myPoint.y += ff(50);

/* re-create and redraw the layout a second time */
layout = GXNewLayout(1, &len, (void *) &myString,
                    1, &len, &myStyle,
                    0, nil, nil,
                    nil, &myPoint);
GXDrawShape(layout);

/* locate and highlight glyph 17 again; this time it's "u" */
GXGetGlyphOffset(layout, 17, true, &offset,
                &leadingEdge, &wasRealCharacter);
highlight = GXGetLayoutHighlight(layout, offset, offset + 1,
                                gxHighlightAverageAngle, nil);
SetShapeCommonTransfer(highlight, gxHighlightMode);
GXDrawShape(highlight);
GXDisposeShape(layout);
GXDisposeShape(highlight);

/* reinitialize the style; allow all optional ligatures */
runFeature[2].featureSelector = diphthongLigaturesOnSelector;
GXSetStyleRunFeatures(myStyle, 3, runFeature);
myPoint.y += ff(50);

/* re-create and redraw the layout a third time */
layout = GXNewLayout(1, &len, (void *) &myString,
                    1, &len, &myStyle,
                    0, nil, nil,
                    nil, &myPoint);
GXDrawShape(layout);

/* locate and highlight glyph 17 once more; this time it's "t" */
GXGetGlyphOffset(layout, 17, true, &offset,
                &leadingEdge, &wasRealCharacter);
highlight = GXGetLayoutHighlight(layout, offset, offset + 1,
                                gxHighlightAverageAngle, nil);
SetShapeCommonTransfer(highlight, gxHighlightMode);
GXDrawShape(highlight);

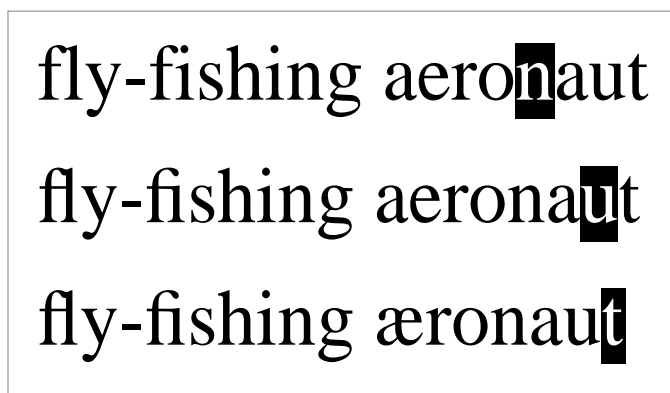
```

Layout Carets, Highlighting, and Hit-Testing

```
GXDisposeShape(layout);
GXDisposeShape(highlight);
.
.
.
```

Figure 10-24 shows the results of executing the function Listing 10-7. Note that in each case, glyph 17 corresponds to a different offset in the source text.

Figure 10-24 Using `GXGetGlyphOffset` to locate a glyph's character



Layout Carets, Highlighting, and Hit-Testing Reference

This section provides reference information to the constants, data structures, and functions that allow you to

- draw carets and highlighting properly in the display text of layout shapes
- hit-test the display text of layout shapes
- convert between edge offsets in the source text and glyph indexes in the display text of layout shapes

Constants and Data Types

This section describes the `gxHighlightType`, `gxCaretType`, and `gxLayoutOffsetState` enumerations, and the `gxLayoutHitInfo` structure.

Highlighting Type

The `gxHighlightType` enumeration controls the shape that QuickDraw GX returns for carets and highlighting in a layout shape.

```
enum {
    gxHighlightStraight      = 0,
    gxHighlightAverageAngle = 1
};
typedef unsigned long gxHighlightType;
```

Constant descriptions

`gxHighlightStraight`

The highlighting is perpendicular to the baseline.

`gxHighlightAverageAngle`

The highlighting is slanted at the angle specified by the font for slanted highlights, or at the average of the two angles at the boundary between text of different slants.

Note

The run control flag `gxNoCaretAngle` can override the effects of the highlight type selected. However, for basic layout highlighting, selecting a highlight type using the `gxHighlightType` data type is sufficient. For more information about run control flags, see the chapter “Layout Styles” in this book. ♦

Caret Type

The `gxCaretType` enumeration controls the kind of caret that QuickDraw GX returns via the `GXGetLayoutCaret` function (page 10-44).

```
enum {
    gxSplitCaretType          = 0,
    gxLeftRightKeyboardCaret  = 1,
    gxRightLeftKeyboardCaret  = 2
};
typedef unsigned long gxCaretType;
```

Constant descriptions

`gxSplitCaretType`

The preferred caret to use at all times. If all the text is unidirectional, the caret always appears as one piece. It appears as two partial carets at direction boundaries. When split, the high caret appears at the caret location for insertion of text in the dominant direction; the low caret appears at the caret location for insertion of text in the opposite direction.

Layout Carets, Highlighting, and Hit-Testing

`gxLeftRightKeyboardCaret`

A single caret that appears at the proper location for left-to-right text entry.

`gxRightLeftKeyboardCaret`

A single caret that appears at the proper location for right-to-left text entry.

In general, you should always use a split caret. The other two types of carets should be used only when providing a choice of directionality for mixed-direction text. Note that, if you select a single caret type, you must always synchronize it with the direction (left to right or right to left) associated with the user's current text-entry direction.

Layout Offset State

The layout offset state is one of the values returned by the `GXGetOffsetGlyphs` function (page 10-56). It gives an indication of where the specified edge offset lies in relation to adjacent characters in the source text.

```
enum {
    gxOffset8_8           = 0,
    gxOffset8_16          = 1,
    gxOffset16_8          = 2,
    gxOffset16_16         = 3,
    gxOffsetInvalid       = 4,
    gxOffsetInsideLigature = 0x8000
};

typedef unsigned short gxLayoutOffsetState;
```

Constant descriptions

<code>gxOffset8_8</code>	The specified offset corresponds to the edge of a glyph and is the boundary between two 8-bit characters.
<code>gxOffset8_16</code>	The specified offset corresponds to the edge of a glyph and is the boundary between an 8-bit character code and a (following) 16-bit character code.
<code>gxOffset16_8</code>	The specified offset corresponds to the edge of a glyph and is the boundary between a 16-bit character code and a (following) 8-bit character code.
<code>gxOffset16_16</code>	The specified offset corresponds to the edge of a glyph and is the boundary between two 16-bit character codes.
<code>gxOffsetInvalid</code>	The specified offset is interior to a 16-bit character code.
<code>gxOffsetInsideLigature</code>	The specified offset corresponds to the interior of a ligature glyph. This value can be returned in addition to another <code>gxLayoutOffsetState</code> value.

At offsets marking the beginning and end of the source text for the line, at which there is only one bounding character, the layout offset state has the value `gxOffset8_8` or `gxOffset16_16`, depending on the size of the bounding character code.

Layout Hit Info Structure

The layout hit info structure (type `gxLayoutHitInfo`) contains the information that is returned by the `GXHitTestLayout` function (page 10-54).

```
typedef struct {
    Fixed          firstPartialDist;
    Fixed          lastPartialDist;
    gxByteOffset   hitSideOffset;
    gxByteOffset   nonHitSideOffset;
    boolean        leadingEdge;
    boolean        inLoose;
} gxLayoutHitInfo;
```

Field descriptions

<code>firstPartialDist</code>	The (with-stream) distance from the left or top edge of the hit glyph to the actual hit point.
<code>lastPartialDist</code>	The (with-stream) distance from the right or bottom edge of the hit glyph to the actual hit point.
<code>hitSideOffset</code>	The edge offset corresponding to one edge of the hit ligature (the edge that is closer to the hit point). For example, you do not get the offset from the other edge of the ligature. However, see also the discussion of the <code>gxNoLigatureSplits</code> flag (below).
<code>nonHitSideOffset</code>	The edge offset corresponding to the other edge of the hit glyph (the edge that is farther from the hit point). However, see also the discussion of the <code>gxNoLigatureSplits</code> flag (below).
<code>leadingEdge</code>	A Boolean value specifying whether the hit occurred closer to the leading edge of the hit glyph (<code>true</code>), or closer to its trailing edge (<code>false</code>).
<code>inLoose</code>	A Boolean value specifying whether the hit occurred within the highlighted area. This value is <code>true</code> if the hit occurred within the highlighted area of the shape as a whole. In general, hits outside of this area would not be considered hits within the text of the layout shape. Nevertheless, even when the value of <code>inLoose</code> is <code>false</code> , the layout hit info reflects the correct projection of the hit point to the baseline in the layout shape. You can therefore use <code>GXHitTestLayout</code> to perform dynamic highlighting, even when the mouse drifts outside the line of text being highlighted.

Layout Carets, Highlighting, and Hit-Testing

There is an interaction between the `hitSideOffset` and `nonHitSideOffset` fields and the state of the `gxNoLigatureSplits` run controls flag (described in this book in the chapter “Layout Styles”) associated with the style run in which the hit occurred. If the hit occurred on a glyph that is a ligature (encompassing more than one character in the source text), the `gxNoLigatureSplits` flag controls whether just the two outermost offsets of the ligature are returned (flag is set) or whether the appropriate intermediate offsets are generated (the default case, where the flag is clear).

Functions

This section describes the QuickDraw GX functions that allow you to

- manipulate carets
- highlight text
- perform hit-testing
- convert between characters and glyphs

Manipulating Carets in a Layout Shape

The functions described in this section allow you to obtain a caret shape, determine the area within which a given caret shape is valid, and move the caret position properly when an arrow key is pressed.

GXGetLayoutCaret

You can use the `GXGetLayoutCaret` function to obtain a shape that describes the caret for a given edge offset in the source text of a layout shape.

```
gxShape GXGetLayoutCaret(gxShape layout, gxByteOffset offset,
                        gxHighlightType highlightType,
                        gxCaretType caretType, gxShape caret);
```

<code>layout</code>	A reference to the layout shape whose caret you need to draw.
<code>offset</code>	The edge offset defining the insertion point in the source text.
<code>highlightType</code>	The angle of caret to use (perpendicular or oblique), of type <code>gxHighlightType</code> .
<code>caretType</code>	The type of caret to use (single or split), of type <code>gxCaretType</code> .
<code>caret</code>	A reference to a shape object. You may supply an existing caret shape here for <code>GXGetLayoutCaret</code> to reuse; if you pass <code>nil</code> for this parameter, <code>GXGetLayoutCaret</code> allocates a new shape to return in its function result.

function result The shape describing the caret for the insertion point specified by the `offset` parameter. If you pass an existing shape in the `caret` parameter, `GXGetLayoutCaret` modifies the shape as necessary and returns it; otherwise, `GXGetLayoutCaret` returns a new shape.

DESCRIPTION

The `GXGetLayoutCaret` function returns a shape that you can use to draw a caret with correct form and locations in the display text of a layout shape, given the edge offset value that you pass to the function. In simple horizontal single-direction text, `GXGetLayoutCaret` returns a caret shape that is a vertical bar between two glyphs. In italic text, the caret is angled (if you specify the oblique highlight type). At direction boundaries in mixed-direction text (and in text that has undergone linguistic rearrangement), the caret is split into two separate halves (if you specify a split caret in the `caretType` parameter). If the input edge offset corresponds to an interior point in a ligature, the resulting caret is located on the edge of the ligature if the `gxNoLigatureSplits` flag (in the run controls structure of the style run that includes the offset) is set; otherwise, the caret is located at a point within the ligature.

If you pass `nil` for the `caret` parameter, `GXGetLayoutCaret` creates a new caret shape and returns it as the function result. You can also pass an existing caret shape to save QuickDraw GX the overhead of disposing of one shape and creating another. If you pass an existing shape, `GXGetLayoutCaret` does not change the shape's fill or any of its style or ink values. If `GXGetLayoutCaret` creates a new shape, however, it sets the shape fill to frame fill. If the layout shape has a transform, the caret shape this function returns has the same transform.

The `highlightType` parameter controls the angle of the with-stream edges of the highlighting shape (the left and right edges for horizontal text). If the highlight type is `gxHighlightStraight`, the highlighting shape has edges that are perpendicular to the baseline, and the highlighting is always tiled (contiguous and nonambiguous across boundaries of text with different slant). If the highlight type is `gxHighlightAverageAngle`, the angle of the edge of the highlighting area is the average of the slants of the two glyphs on either side of the edge. In this case also, highlighting is usually tiled; highlighting is not tiled only in cases of extreme slant coupled with superscripts or subscripts. (In such a case a triangular highlighting area may result.)

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`parameter_out_of_range`

SEE ALSO

For a description of how carets appear in a layout shape, see “Drawing Highlighting” on page 10-25. Caret types are described on page 10-41. Highlighting types are described on page 10-41.

For examples of the use of the `GXGetLayoutCaret` function, see page 10-19.

The `gxNoLigatureSplits` flag is described with the run controls structure in the chapter “Layout Styles” in this book.

GXGetCaretAngleArea

You can use the `GXGetCaretAngleArea` function to get (a) the angle needed to draw a cursor in italic text as well as (b) the area in the display text within which that angle is valid.

```
gxShape GXGetCaretAngleArea(gxShape layout,
                             const gxPoint *hitPoint,
                             gxHighlightType highlightType,
                             gxShape caretArea,
                             short *returnedRise,
                             short *returnedRun);
```

layout A reference to the layout shape whose caret-angle information you need.

hitPoint A pointer to a point structure that contains the location for which you need the caret angle. The location is in local (view port) coordinates.

highlightType The kind of caret (perpendicular or oblique) that would be drawn in this text, of type `gxHighlightType`.

caretArea A reference to a shape object. You may supply an existing caret-area shape here for `GXGetCaretAngleArea` to reuse; if you pass `nil` for this parameter, `GXGetCaretAngleArea` allocates a new shape to return in its function result.

returnedRise A pointer to a short value. On return, it contains the vertical component of the caret angle.

returnedRun A pointer to a short value. On return, it contains the horizontal component of the caret angle.

function result The shape describing the caret area for the point specified by the `hitPoint` parameter. The caret area is the area of the display text within which the caret or cursor can move and retain the same angle. If you pass an existing shape in the `caretArea` parameter, `GXGetCaretAngleArea` modifies the shape as necessary and returns it; otherwise, `GXGetCaretAngleArea` returns a new shape.

DESCRIPTION

The `GXGetCaretAngleArea` function helps you draw the cursor (the small icon, commonly an “I-beam” shape or arrow, that moves with mouse movements) at the proper angle for any slanted or italic text. The function provides two kinds of information. First, it returns the angle of the caret (and therefore the cursor) for caret positions in the vicinity of the hit point. Second, it returns a shape describing the area within which the cursor can move and retain the given angle.

The shape returned by `GXGetCaretAngleArea` can be used as a cursor-tracking area. As long as the cursor does not move outside of it, the shape of the cursor need not change. Thus, you can minimize the calls you need to make to `GXGetCaretAngleArea` or `GXGetLayoutCaret` as the user moves the cursor across the text.

If you pass `nil` for the `caretArea` parameter, `GXGetCaretAngleArea` creates a new caret-area shape and returns it as the function result. You can also pass an existing caret-area shape to save QuickDraw GX the overhead of disposing of one shape and creating another. If the layout shape has a transform, the caret-area shape this function returns has the same transform.

ERRORS, WARNINGS, AND NOTICES**Errors**

`shape_is_nil`
`parameter_out_of_range`

SEE ALSO

Highlight types are described on page 10-41. The `GXGetLayoutCaret` function is described in the previous section.

GXGetRightVisualOffset

You can use the `GXGetRightVisualOffset` function to determine the edge offset corresponding to the next caret position to the right (or downward, for vertical text) in a layout shape.

```
gxByteOffset GXGetRightVisualOffset(gxShape layout,
                                     gxByteOffset currentOffset);
```

`layout` A reference to the layout shape whose right visual offset you need.

`currentOffset` The edge offset in the source text corresponding to the current caret position in the display text.

function result The edge offset in the source text corresponding to the next rightward (or downward) caret position in the display text.

DESCRIPTION

The `GXGetRightVisualOffset` function determines the next caret position to the right (or down) in the display text of a layout shape. This is where the caret would move if the user pressed the Right or Down Arrow key. The function takes into account the text direction (left to right or right to left). It also considers whether the text has undergone linguistic rearrangement. If the caret passes through a ligature, the resulting offset depends on whether the `gxNoLigatureSplits` flag is set. If so, the caret position passes entirely across the ligature; if not, the caret position can be interior to the ligature.

If a split caret moves across a direction boundary, this function describes the movement of the high (dominant) caret only.

SPECIAL CONSIDERATIONS

This function may not always work correctly for right-to-left carets (type `gxRightToLeftKeyboardCaret`) at direction boundaries in mixed-direction text.

ERRORS, WARNINGS, AND NOTICES**Errors**

`shape_is_nil`
`parameter_out_of_range`

SEE ALSO

For an example of how to use this function, see Listing 10-3 on page 10-23.

Caret types are described on page 10-41.

The `gxNoLigatureSplits` flag is described with the run controls structure in the chapter “Layout Styles” in this book.

The complementary function `GXGetLeftVisualOffset` is described next.

GXGetLeftVisualOffset

You can use the `GXGetLeftVisualOffset` function to determine the edge offset corresponding to the next caret position to the left (or upward, for vertical text) in a layout shape.

```
gxByteOffset GXGetLeftVisualOffset(gxShape layout,
                                   gxByteOffset currentOffset);
```

`layout` A reference to the layout shape whose left visual offset you need.

`currentOffset` The edge offset in the source text corresponding to the current caret position in the display text.

function result The edge offset in the source text corresponding to the next leftward (or upward) caret position in the display text.

DESCRIPTION

The `GXGetLeftVisualOffset` function determines the next caret position to the left (or upward) in the display text of a layout shape. This is where the caret would move to if the user pressed the Left or Up Arrow key. The function takes into account the text direction (left to right or right to left). It also considers whether the text has undergone linguistic rearrangement. If the caret passes through a ligature, the resulting offset depends on whether or not the `gxNoLigatureSplits` flag is set. If so, the caret position passes entirely across the ligature; if not, the caret position can be interior to the ligature.

If a split caret moves across a direction boundary, this function describes the movement of the high (dominant) caret only.

SPECIAL CONSIDERATIONS

This function may not always work correctly for right-to-left carets (type `gxRightToLeftKeyboardCaret`) at direction boundaries in mixed-direction text.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`parameter_out_of_range`

SEE ALSO

For an example of how to use this function, see Listing 10-3 on page 10-23.

Caret types are described on page 10-41.

The `gxNoLigatureSplits` flag is described with the run controls structure in the chapter “Layout Styles” in this book.

The complementary function `GXGetRightVisualOffset` is described in the previous section.

Highlighting in a Layout Shape

The functions described in this section allow you to highlight text and, for mixed-direction text, allow you to choose whether or not to make your highlights visually contiguous.

GXGetLayoutHighlight

You can use the `GXGetLayoutHighlight` function to obtain a shape to use to draw the highlighting corresponding to a selection range in a specified layout shape. If the selection range includes text of different directions, the highlighting shape may be discontinuous.

```
gxShape GXGetLayoutHighlight(gxShape layout,
                             gxByteOffset startOffset,
                             gxByteOffset endOffset,
                             gxHighlightType highlightType,
                             gxShape highlight);
```

<code>layout</code>	A reference to the layout shape you want to highlight.
<code>startOffset</code>	The edge offset in the source text that marks the start of the selection range. If this value is the same as <code>endOffset</code> , <code>GXGetLayoutHighlight</code> generates a caret.
<code>endOffset</code>	The edge offset marking the end of the selection range. If this value is the same as <code>startOffset</code> , <code>GXGetLayoutHighlight</code> generates a caret. You can specify <code>gxSelectToEnd</code> for this parameter to select to the end of the text in the layout shape.
<code>highlightType</code>	The type of highlight to use (perpendicular or oblique), of type <code>gxHighlightType</code> .
<code>highlight</code>	A reference to a shape object. You may supply an existing highlight shape here for <code>GXGetLayoutHighlight</code> to reuse; if you pass <code>nil</code> for this parameter, <code>GXGetLayoutHighlight</code> allocates a new shape to return in its function result.
<i>function result</i>	The shape describing the highlight for the selection range specified by the <code>startOffset</code> and <code>endOffset</code> parameters. If you pass an existing shape in the <code>highlight</code> parameter, <code>GXGetLayoutHighlight</code> modifies the shape as necessary and returns it; otherwise, <code>GXGetLayoutHighlight</code> returns a new shape.

DESCRIPTION

The `GXGetLayoutHighlight` function calculates a shape corresponding to the area covered by the glyphs corresponding to all the characters between the two edge offsets `startOffset` and `endOffset`. For single-direction text this is usually a simple rectangular or trapezoidal area, but for mixed-direction text the area that is returned may be quite complex.

If you pass `nil` for the `highlight` parameter, `GXGetLayoutHighlight` creates a new highlight shape and returns it as the function result. You can also pass an existing highlight shape to save the overhead of disposing of one shape and creating another.

If either of the offsets corresponds to a point interior to a ligature, the appearance of the highlight depends on the state of the `gxNoLigatureSplits` flag in the run controls structure of the style run containing the offset. If the flag is set, the highlight extends across the entire ligature; if it is clear, only the portion of the ligature corresponding to the included characters is highlighted.

The `highlightType` parameter controls the angle of the with-stream edges of the highlighting shape (the left and right edges for horizontal text). If the highlight type is `gxHighlightStraight`, the highlighting shape has edges that are perpendicular to the baseline, and the highlighting is always tiled (contiguous and nonambiguous across boundaries of text with different slant). If the highlight type is `gxHighlightAverageAngle`, the angle of the edge of the highlighting area is the average of the slants of the two glyphs on either side of the edge. In this case also, highlighting is usually tiled. Highlighting is not tiled only in cases of extreme slant coupled with superscripts or subscripts. (In such a case a triangular highlighting area may result.)

If the layout shape has a transform, the highlight shape this function returns has the same transform.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`parameter_out_of_range`

SEE ALSO

For an example of how to use this function, see page 10-26 and page 10-27.

For a description of how highlighting applies to a layout shape, see “Drawing Highlighting” on page 10-25. Highlight types are described on page 10-41.

The `gxNoLigatureSplits` flag is described with the run controls structure in the chapter “Layout Styles” in this book.

GXGetLayoutVisualHighlight

You can use the `GXGetLayoutVisualHighlight` function to obtain a shape that describes a single, contiguous highlighting band corresponding to two edge offsets in a specified layout shape. The shape is a solid connection between the beginning and end glyphs—even across direction boundaries in the text or if it results in a visual highlight that covers discontinuous text in the backing store.

```
gxShape GXGetLayoutVisualHighlight(gxShape layout,
                                   gxByteOffset startOffset,
                                   long startLeadingEdge,
                                   gxByteOffset endOffset,
                                   long endLeadingEdge,
                                   gxHighlightType highlightType,
                                   gxShape highlight);
```

`layout` A reference to the layout shape you want to highlight.

`startOffset` The edge offset in the source text that marks the start of the selection range. If this value is the same as `endOffset`, `GXGetLayoutVisualHighlight` generates a caret.

`startLeadingEdge` If this value is `true`, the highlight starts at the leading edge of the glyph whose leading edge bounds the caret position corresponding to the `startOffset` parameter. Otherwise, the highlight starts at the trailing edge of the glyph whose trailing edge bounds the caret position corresponding to the `startOffset` parameter.

`endOffset` The edge offset in the source text that marks the end of the selection range. If this value is the same as `startOffset`, `GXGetLayoutVisualHighlight` generates a caret. You can specify `gxSelectToEnd` for this parameter to select to the end of the text in the layout shape.

`endLeadingEdge` If this value is `true`, the highlight ends at the leading edge of the glyph whose leading edge bounds the caret position corresponding to the `endOffset` parameter. Otherwise, the highlight ends at the trailing edge of the glyph whose trailing edge bounds the caret position corresponding to the `endOffset` parameter.

`highlightType` The type of highlight to use (perpendicular or oblique), of type `gxHighlightType`.

`highlight` A reference to a shape object. You may supply an existing highlight shape here for `GXGetLayoutVisualHighlight` to reuse; if you pass `nil` for this parameter, `GXGetLayoutVisualHighlight` allocates a new shape to return in its function result.

function result The shape describing the highlight for the selection range specified by the `startOffset` and `endOffset` parameters, and given the `startLeadingEdge` and `endLeadingEdge` constraints. If you pass an existing shape in the `highlight` parameter, `GXGetLayoutVisualHighlight` modifies the shape as necessary and returns it; otherwise, `GXGetLayoutVisualHighlight` returns a new shape.

DESCRIPTION

The `GXGetLayoutVisualHighlight` function calculates a shape that is a single, contiguous highlighting rectangle (or trapezoid) between the glyphs corresponding to the edge offsets `startOffset` and `endOffset` and the leading-edge states specified in `startLeadingEdge` and `endLeadingEdge`.

The shape returned by `GXGetLayoutVisualHighlight` function is similar to that returned by `GXGetLayoutHighlight`, except that the result of `GXGetLayoutVisualHighlight` always represents the continuous visual range between the two specified offsets and edges; it never consists of discontinuous visual segments, even in mixed-direction text. (For single-direction text, calling `GXGetLayoutVisualHighlight` yields the same results as calling `GXGetLayoutHighlight` for the same start and end offsets.)

This function is most useful for mixed-direction text, where only a contiguous highlight is desired. For dynamic highlighting as the user moves the cursor through the display text, contiguous highlighting may be less confusing.

If you pass `nil` for the `highlight` parameter, `GXGetLayoutVisualHighlight` creates a new highlight shape and returns it as the function result. You can also pass an existing highlight shape to save QuickDraw GX the overhead of disposing of one shape and creating another.

If either of the offsets corresponds to a point interior to a ligature, the appearance of the highlight depends on the state of the `gxNoLigatureSplits` flag in the run controls structure of the style run containing the offset. If the flag is set, the highlight extends across the entire ligature; if it is clear, only the portion of the ligature corresponding to the included characters is highlighted.

The `highlightType` parameter controls the angle of the with-stream edges of the highlighting shape (the left and right edges for horizontal text). If the highlight type is `gxHighlightStraight`, the highlighting shape has edges that are perpendicular to the baseline, and the highlighting is always tiled (contiguous and nonambiguous across boundaries of text with different slant). If the highlight type is `gxHighlightAverageAngle`, the angle of the edge of the highlighting area is the average of the slants of the two glyphs on either side of the edge. In this case also, highlighting is usually tiled; highlighting is not tiled only in cases of extreme slant coupled with superscripts or subscripts. (In such a case a triangular highlighting area may result.)

If the layout shape has a transform, the highlight shape this function returns has the same transform.

SPECIAL CONSIDERATIONS

The highlight shape returned by this function, and therefore the selection range it implies, do *not* in general correspond to the continuous set of characters between the two edge offsets used as input to the function. Where contiguous highlighting crosses direction boundaries, the resulting selection range is discontinuous and can be quite complex, possibly involving characters beyond the range of edge offsets used as input. For this reason, use of this function is not recommended.

ERRORS, WARNINGS, AND NOTICES**Errors**

shape_is_nil
parameter_out_of_range

SEE ALSO

For an example of how to use this function, see page 10-28.

For a description of how highlighting applies to a layout shape, see “Drawing Highlighting” on page 10-25. Highlight types are described on page 10-41.

The `gxNoLigatureSplits` flag is described with the run controls structure in the chapter “Layout Styles” in this book.

Hit-Testing in a Layout Shape

The function described in this section allows you to hit-test the text of any layout shape, no matter how complex. For hit-testing of shapes other than layout shapes, you can use the `GXHitTestShape` function, described in the shape objects chapter of *Inside Macintosh: QuickDraw GX Objects*.

GXHitTestLayout

You can use the `GXHitTestLayout` function to convert a view port location (representing, for example, the position of a mouse-down event) into an edge offset in the source text of a layout shape.

```
gxByteOffset GXHitTestLayout(gxShape layout,
                             const gxPoint *hitDown,
                             gxHighlightType highlightType,
                             gxLayoutHitInfo *hitInfo,
                             gxShape hitTrackingArea);
```

layout A reference to the layout shape to hit-test.

hitDown A pointer to a point structure that contains the location to test. The location is in local (view port) coordinates.

Layout Carets, Highlighting, and Hit-Testing

`highlightType`

The kind of highlight used (perpendicular or oblique), of type `highlightType`.

`hitInfo`

A pointer to a layout hit info structure. On return, the structure contains the results of the hit-test. If you pass `nil` for this parameter, the hit info structure is not filled out but the function result is still valid.

`hitTrackingArea`

A reference to a shape object. On return, the shape specifies the mouse tracking area that corresponds to the specified hit point. You may supply an existing shape here for `GXHitTestLayout` to reuse. Pass `nil` for this parameter if you do not want `GXHitTestLayout` to return the hit-tracking area.

function result

The edge offset corresponding to the location where the hit occurred. This value always equals the value of the `hitSideOffset` field of `gxLayoutHitInfo` structure returned in the `hitInfo` parameter.

DESCRIPTION

The `GXHitTestLayout` function determines which part of which glyph in the display text of a layout shape is closest to a particular location. It then returns in the `hitInfo` parameter the equivalent source-text edge offset as the function result, and leading-edge information.

You must specify the test location in the local coordinates of the view port in which the text is displayed. For hit-testing of mouse-down events, you must convert mouse coordinates (such as might be returned in an event record) to view port coordinates (with a function such as `GXQDGlobalToGXLocal` or `GXGetViewPortMouse`) before you call `GXHitTestLayout`.

The `GXHitTestLayout` function also returns a mouse tracking area, which specifies the area in the display text within which the resulting edge offset is valid. For subsequent hits anywhere within that area, there may be no need to call `GXHitTestLayout` again, because the resulting edge offset will be the same. The `hitTrackingArea` parameter is modified if it is not `nil`; if it is `nil`, it is not generated.

The hit info structure (returned in the `hitInfo` parameter) provides more than just leading-edge information. It also returns the exact position of the hit point within the hit glyph, two edge offsets corresponding to the two edges of the hit glyph, and an indication of whether the hit point is actually outside the boundaries of the layout shape (such as above or below, for horizontal text).

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`

`parameter_is_nil`

`parameter_out_of_range`

SEE ALSO

For an example of how to use this function, see Listing 10-5 on page 10-31.

Coordinate systems and view ports are described in the chapter “View-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*. Functions such as `GXQDGlobalToGXLocal` and `GXGetViewPortMouse`, which convert from Macintosh coordinates to QuickDraw GX coordinates, are described in the chapter “The QuickDraw GX and the Macintosh Environment” in *Inside Macintosh: QuickDraw GX Environment and Utilities*.

The `GXHitTestShape` function, used for hit-testing shapes other than layout shapes, is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*. Typographic information returned by `GXHitTestShape` is described in the chapter “Typographic Shapes” in this book.

Highlighting types are described on page 10-41. The layout hit info structure is described on page 10-43.

Converting Between Glyphs and Characters in a Layout Shape

Functions described in this section allow you to convert glyphs to character codes and back. The `GXGetOffsetGlyphs` and `GXGetGlyphOffset` functions map back and forth between edge offsets in the source text and glyph indices in the display text of a layout shape. The `GXGetCompoundCharacterLimits` function maps the edges of a ligature glyph to offsets in the source text.

GXGetOffsetGlyphs

You can use the `GXGetOffsetGlyphs` function to determine which glyphs border the caret positions corresponding to a particular edge offset in the source text.

```
void GXGetOffsetGlyphs(gxShape layout, gxByteOffset trial,
                      long leadingEdge,
                      gxLayoutOffsetState *offsetState,
                      unsigned short *firstGlyph,
                      unsigned short *secondGlyph);
```

<code>layout</code>	A reference to the layout shape whose glyphs you want to identify.
<code>trial</code>	An edge offset in the source text. The glyph or glyphs bounding the caret positions corresponding to that offset are the glyphs to be returned.
<code>leadingEdge</code>	Specify <code>true</code> if you want the function to return in the <code>firstGlyph</code> parameter the glyph corresponding to the character code <i>following</i> the edge offset specified in the <code>trial</code> parameter; specify <code>false</code> if you want the glyph corresponding to the character code <i>preceding</i> the edge offset.

`offsetState`

A pointer to a layout offset state value. On return, it specifies the characteristics (such as the sizes of the bounding character codes and whether the offset corresponds to the interior of a glyph) of the edge offset specified in the `trial` parameter.

`firstGlyph` A pointer to a short value. On return, it contains the glyph index of the glyph corresponding to the specified values in the `trial` and `leadingEdge` parameters.

`secondGlyph`

A pointer to a short value. On return, it contains the glyph index of the other glyph corresponding to the offset specified in the `trial` parameter. In other words, it contains the glyph index that would have been returned in the `firstGlyph` parameter if the `leadingEdge` parameter had had the opposite value.

DESCRIPTION

The `GXGetOffsetGlyphs` function determines which glyph in the display text of a layout shape corresponds to the specified edge offset and leading-edge state in the source text. The function returns its result as a glyph index in the `firstGlyph` parameter. (A glyph index is the 1-based position of a glyph in the left-to-right, or top-to-bottom, display order of a layout shape's display text.)

The function also returns, in the `secondGlyph` parameter, the glyph index of the other glyph that bounds the caret position corresponding to the specified edge offset. Typically, that is the bounding glyph whose leading-edge state is opposite to that of the glyph returned in the `firstGlyph` parameter.

The `GXGetOffsetGlyphs` function also returns information on the nature of the boundary at the specified edge offset, in the form of a layout offset state value. It tells you the sizes of the bounding character codes, and whether the offset is interior to a ligature or whether it is invalid (interior to a single character code).

In the case of an edge offset that corresponds to the interior of a compound glyph such as a ligature, both `firstGlyph` and `secondGlyph` contain the same value: the index of the glyph containing the offset. In the case of an invalid edge offset (between the two bytes of a 16-bit character code), `firstGlyph` and `secondGlyph` both contain the index of the glyph corresponding to that character.

You can call this function to perform a geometric operation on the glyph corresponding to a known edge offset in the source text. You can get a copy of the glyph array that makes up the display text of a layout shape by calling the `GXGetLayoutGlyphs` function.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`parameter_out_of_range`

SEE ALSO

For an example of how to use this function, see Listing 10-6 on page 10-34.

Layout offset state values are described on page 10-42.

The complementary function `GXGetGlyphOffset` is described next.

Glyph indexes are described in the section “Positioning in Source Text and Display Text” beginning on page 10-3. The `GXGetLayoutGlyphs` function is described in the chapter “Layout Shapes” in this book.

GXGetGlyphOffset

You can use the `GXGetGlyphOffset` function to find the edge offset in the source text corresponding to a particular edge of a particular glyph in a layout shape.

```
void GXGetGlyphOffset(gxShape layout, long trial,
                     long onLeftTop, gxByteOffset *offset,
                     boolean *leadingEdge,
                     boolean *wasRealCharacter);
```

<code>layout</code>	A reference to the layout shape containing the glyph whose corresponding edge offset you need.
<code>trial</code>	A (1-based) glyph index specifying the position of the glyph in the display text.
<code>onLeftTop</code>	A Boolean value indicating whether the <code>trial</code> parameter specifies the edge offset corresponding to the left (or top, for vertical text) edge of the glyph (<code>true</code>) or the edge offset corresponding to the right (or bottom, for vertical text) edge of the glyph (<code>false</code>).
<code>offset</code>	A pointer to a caret-offset value. On return, it contains the edge offset in the source text corresponding to the glyph and edge specified in the <code>trial</code> and <code>onLeftTop</code> parameters.
<code>leadingEdge</code>	A pointer to a Boolean value. On return, the value is <code>true</code> if the specified edge of the specified glyph is the leading edge; otherwise the value is <code>false</code> .
<code>wasRealCharacter</code>	A pointer to a Boolean value. On return, the value is <code>true</code> if the specified glyph corresponds to one or more character codes in the source text; otherwise the value is <code>false</code> .

DESCRIPTION

The `GXGetGlyphOffset` function determines which edge offset in the source text of a layout shape corresponds to the specified edge of the specified glyph in the display text. You specify the input glyph by index. (A glyph index is the 1-based position of a glyph in the left-to-right, or top-to-bottom, display order of a layout shape’s display text.) The function returns its result in the `offset` parameter.

The function also returns, in the `leadingEdge` parameter, information indicating whether the specified edge of the specified glyph is its leading edge. It also returns, in the `wasRealCharacter` parameter, whether or not the specified glyph corresponds to an actual character code in the source text. For example, no character codes correspond to kashida glyphs. (If the value of `wasRealCharacter` is `false`, `GXGetGlyphOffset` returns an offset corresponding to the right side of the glyph to the left of the kashida glyph.)

You can call this function to perform an operation on the character code corresponding to a known glyph in the display text. You can get a copy of the glyph array that makes up the display text of a layout shape by calling the `GXGetLayoutGlyphs` function.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`parameter_out_of_range`

SEE ALSO

For an example of how to use this function, see Listing 10-7 on page 10-38.

The complementary function `GXGetOffsetGlyphs` is described in the previous section.

Glyph indexes are described in the section “Positioning in Source Text and Display Text” beginning on page 10-3. The `GXGetLayoutGlyphs` function is described in the chapter “Layout Shapes” in this book.

GXGetCompoundCharacterLimits

You can use the `GXGetCompoundCharacterLimits` function to find the edge offsets that correspond to the leading and trailing edges of a particular glyph (itself specified by edge offset).

```
void GXGetCompoundCharacterLimits(gxShape layout,
                                  gxByteOffset trial,
                                  gxByteOffset *minOffset,
                                  gxByteOffset *maxOffset,
                                  boolean *onBoundary);
```

<code>layout</code>	A reference to the layout shape containing the glyph you want to analyze.
<code>trial</code>	An edge offset in the source text. The glyph corresponding to that offset is the glyph to be analyzed. If the offset is between glyphs, both glyphs bounding that offset are taken into account: the glyph to be analyzed for <code>minOffset</code> is the glyph corresponding to the character preceding the <code>trial</code> offset, and the glyph to be analyzed for <code>maxOffset</code> is the glyph

Layout Carets, Highlighting, and Hit-Testing

	corresponding to the character following the <code>trial</code> offset. If the offset is at the beginning (or end) of the source text, the glyph corresponding to the first (or last) character in the text is the glyph to be analyzed.
<code>minOffset</code>	A pointer to an edge-offset value. On return, it specifies the edge offset in the source text corresponding to the leading edge of the glyph corresponding to the character specified by the <code>trial</code> parameter.
<code>maxOffset</code>	A pointer to a caret-offset value. On return, it specifies the edge offset in the source text corresponding to the trailing edge of the glyph corresponding to the character specified by the <code>trial</code> parameter.
<code>onBoundary</code>	A pointer to a Boolean value. On return, it is <code>true</code> if the edge offset specified in the <code>trial</code> parameter corresponds to a caret position between glyphs in the display text. It is <code>false</code> if the offset corresponds to a caret position in the interior of a compound glyph (such as a ligature).

DESCRIPTION

Given an edge offset in the source text of a layout shape, the `GXGetCompoundCharacterLimits` function first determines which glyph in the display text corresponds to that offset, and then returns the edge offsets corresponding to both edges of the glyph. For a compound character such as a ligature, `GXGetCompoundCharacterLimits` thus lets you determine which characters make up the ligature.

The function returns the offsets in two parameters. The value of `minOffset` is the edge offset that marks the leading edge of the glyph that includes (or precedes, if the value of `onBoundary` is `true`) the `trial` offset. The value of `maxOffset` is the edge offset that marks the trailing edge of the glyph that includes (or follows, if the value of `onBoundary` is `true`) the `trial` offset.

If the `trial` offset precedes the first character or follows the last character in the source text, `GXGetCompoundCharacterLimits` returns a value of `true` for `onBoundary`, and returns edge offsets bounding the glyph corresponding to the first or last character, respectively.

You can use `GXGetCompoundCharacterLimits` to determine whether a particular edge offset corresponds to the interior of a ligature and, if it does, what the bounding offsets of the characters making up that ligature are.

ERRORS, WARNINGS, AND NOTICES

Errors

`shape_is_nil`
`parameter_out_of_range`

SEE ALSO

For information about related functions, see the descriptions of `GXGetOffsetGlyphs` on page 10-56 and `GXGetGlyphOffset` on page 10-58.

Summary of Layout Carets, Highlighting, and Hit-Testing

Constants and Data Types

Highlighting Type

```
enum {
    gxHighlightStraight      = 0,
    gxHighlightAverageAngle = 1
};
typedef unsigned long gxHighlightType;
```

Caret Type

```
enum {
    gxSplitCaretType        = 0,
    gxLeftRightKeyboardCaret = 1,
    gxRightLeftKeyboardCaret = 2
};
typedef unsigned long gxCaretType;
```

Layout Offset State

```
enum {
    gxOffset8_8      = 0,
    gxOffset8_16     = 1,
    gxOffset16_8     = 2,
    gxOffset16_16    = 3,
    gxOffsetInvalid  = 4,
    gxOffsetInsideLigature = 0x8000
};
typedef unsigned short gxLayoutOffsetState;
```

Layout Hit Inf Structure

```
typedef struct {
    Fixed      firstPartialDist;
    Fixed      lastPartialDist;
    gxByteOffset hitSideOffset;
    gxByteOffset nonHitSideOffset;
    boolean     leadingEdge;
    boolean     inLoose;
} gxLayoutHitInfo;
```

Functions

Manipulating Carets in a Layout Shape

```

gxShape GXGetLayoutCaret      (gxShape layout, gxByteOffset offset,
                               gxHighlightType highlightType,
                               gxCaretType caretType, gxShape caret) ;

gxShape GXGetCaretAngleArea  (gxShape layout, const gxPoint *hitPoint,
                               gxHighlightType highlightType,
                               gxShape caretArea, short *returnedRise,
                               short *returnedRun);

gxByteOffset GXGetRightVisualOffset
                               (gxShape layout, gxByteOffset currentOffset);

gxByteOffset GXGetLeftVisualOffset
                               (gxShape layout, gxByteOffset currentOffset);

```

Highlighting in a Layout Shape

```

gxShape GXGetLayoutHighlight (gxShape layout, gxByteOffset startOffset,
                               gxByteOffset endOffset,
                               gxHighlightType highlightType,
                               gxShape highlight);

gxShape GXGetLayoutVisualHighlight
                               (gxShape layout, gxByteOffset startOffset,
                               long startLeadingEdge, gxByteOffset endOffset,
                               long endLeadingEdge,
                               gxHighlightType highlightType,
                               gxShape highlight);

```

Hit-Testing in a Layout Shape

```

gxByteOffset GXHitTestLayout (gxShape layout, const gxPoint *hitDown,
                               gxHighlightType highlightType,
                               gxLayoutHitInfo *hitInfo,
                               gxShape hitTrackingArea);

```

Converting Between Glyphs and Characters in a Layout Shape

```

void GXGetOffsetGlyphs      (gxShape layout, gxByteOffset trial,
                               long leadingEdge,
                               gxLayoutOffsetState *offsetState,
                               unsigned short *firstGlyph,
                               unsigned short *secondGlyph);

void GXGetGlyphOffset       (gxShape layout, long trial,
                               long onLeftTop, gxByteOffset *offset,
                               boolean *leadingEdge,
                               boolean *wasRealCharacter);

void GXGetCompoundCharacterLimits
                               (gxShape layout, gxByteOffset trial,
                               gxByteOffset *minOffset,
                               gxByteOffset *maxOffset, boolean *onBoundary);

```