QuickDraw Text

This chapter describes the text-handling components of QuickDraw. You can use the QuickDraw text routines to measure and draw text ranging in complexity from a single glyph to a line of justified text containing multiple languages and styles. In addition to measuring and drawing text, the QuickDraw text routines also help you to determine which characters to highlight and where to position the caret to mark the insertion point. These routines translate pixel locations into byte offsets and vice versa.

Read this chapter if you are writing an application that draws static text in a box, such as a dialog box, or draws and manipulates text of any length in one or more languages. Before you use the routines described in this chapter, read the chapter "Introduction to Text on the Macintosh" in this book. To understand the concepts and routines described in this chapter, you must be familiar with the other parts of QuickDraw described in *Inside Macintosh: Imaging*.

Read this chapter along with the chapter "Font Manager," in this book, because of the close relationship between QuickDraw and the Font Manager. For help in understanding the tasks involved in text layout, refer to the chapters "Text Utilities" and "Script Manager," also in this book.

This chapter explains how to set up the text-drawing environment and lay out and draw text, including how to

- draw and measure a single character, a text segment, or a line of text
- determine where to break a line of text
- determine the order in which to draw text segments for a line of text containing multiple styles and mixed directions
- eliminate trailing space characters
- distribute extra space throughout a line of text to justify it appropriately for the script system
- draw and measure scaled text
- identify caret positions for marking an insertion point and highlighting text

About QuickDraw Text

Text on the Macintosh is graphical. This section provides an overview of how to draw text using the text-handling components of QuickDraw. These routines let you direct how the text is to be rendered and drawn, while insulating your application from the low-level implementation details.

Whether for onscreen display or to be printed, you always draw text in the context of a graphics port. To draw the text, QuickDraw displays the bitmap of each glyph on the display device. Although QuickDraw displays the text, you define how the text is to be rendered by setting the text-drawing parameters in the graphics port record. Text rendering is the process of portraying the text according to its character attributes, such as the font, font size, and style. You use the character attribute information associated with the text to set up the drawing environment each time you draw a segment of text that begins a new style run. A **style run** is a sequence of text that is all in the same script system, font, size, and style.

QuickDraw routines let you accept keyboard input or gain access to existing text stored in memory. In general, the tasks that you need to perform to draw text on the Macintosh are easier if your store the text as a simple sequence of character codes separate from all the character attribute information that describes how QuickDraw is to render the stored text. (For an example of how to define data structures to store the character attribute information, you can look at the TextEdit data structures used for this purpose; see the chapter “TextEdit” in this book.)

Graphics Ports and Text Drawing

You draw text on the Macintosh in the current graphics port according to the graphics environment defined by the graphics port record. A **graphics port** defines where and how graphic and text drawing operations are to take place. QuickDraw treats the graphics port information as its primary set of global information.

You can define many graphics ports on the screen, each with its own complete drawing environment, and easily switch between them. Because QuickDraw always draws in the current graphics port, it is essential that you keep track of which one this is.

Each graphics port is tied to a window. To draw in the graphics port of a window, you first need to make the port the current one. (You do this using the `SetPort` procedure, described in *Inside Macintosh: Imaging*.) The window whose port you want to draw in does not have to be active or the frontmost window. QuickDraw draws to the **current graphics port** identified by `SetPort`. You can draw to a background window or an inactive window by making its port the current one.

There are two types of graphics ports: the original version (`GrafPort`) that supports mainly black-and-white drawing with some rudimentary color capabilities and the color graphics port (`CGrafPort`), which supports all of the characteristics of the original graphics port, plus additional features including color facilities.

Both types of graphics port records contain fields that specify the colors to be used for the foreground (`fgColor`) and the background (`bkColor`) of a glyph. You can think of the **foreground** as the pixels that constitute the glyph, and the **background** as the pixels that surround the glyph. In terms of a black-and-white screen, the foreground pixels of a glyph are black, and the surrounding background pixels are white.

The original graphics port provides eight colors—black, white, red, green, blue, cyan, magenta, and yellow; however, on a black-and-white screen nonwhite colors appear as black. A color graphics port provides a wide range of possible colors that allow you to portray all aspects of the user interface in color, including the representation of text. Both types of graphics ports maintain the fractional horizontal pen position, so that a series of text-drawing calls accumulates the fractional position. For the color graphics port, this value is maintained in a graphics port record field. For the original graphics port, this value is maintained in a `grafGlobal`, which is reset whenever you reposition the pen.

There is only one QuickDraw text-handling procedure that requires a color graphics port, `CharExtra`. (Although you can call `CharExtra` for an original graphics port without causing the system to crash, `CharExtra` produces no result.) You can use all the other QuickDraw text routines with either an original graphics port or a color graphics port.

Fields in the graphics port record determine which font QuickDraw is to use to portray the text, the font style, the font size, and how the bits forming the glyph are to be placed in the bit image. You control how the text is to be rendered by setting each of these fields *before* you measure or draw a segment of text that begins a new style run. QuickDraw provides procedures that let you set each field. To ensure future compatibility, you should always use these procedures rather than directly modify a field. You use the appropriate QuickDraw procedure to set the graphics port field for the style run to be drawn, if the current value of a field differs from the characteristic that you want QuickDraw to use. The following sections describe what each of these field values represents.

Font, Font Style, and Font Size

This section provides an overview of how QuickDraw and the Font Manager interact to provide the font that you specify in the graphics port to be used to render the text.

The Font Manager keeps track of detailed font information, such as the glyphs' character codes, whether fonts are fixed-width or proportional, and which fonts are related to each other by name. When you make a call to QuickDraw to measure or draw text, QuickDraw passes the font request, including the font's size and style that you have set in the current graphics port, to the Font Manager, and the Font Manager satisfies the request as best as possible, returning to QuickDraw a bitmap of the glyph of the font, along with some information that QuickDraw uses for stylistic variation and layout. When QuickDraw receives the bitmap, it transfers the bitmap to the screen. If necessary, QuickDraw first scales the bitmap, or applies stylistic variation to it if the requested style was not intrinsic to the font.

The Macintosh supports two types of fonts: bitmapped and outline. A **font** is a complete set of glyphs in a specific typeface and style—and in the case of bitmapped fonts, a specific size. Outline fonts consist of outline glyphs in a particular typeface and style with no size restriction. The Font Manager can generate thousands of point sizes from the same TrueType outline font. For example, a single outline Courier font can produce Courier 10-point, Courier 12-point, and Courier 200-point. (You can read more about these two types of fonts and the relationship between QuickDraw and the Font Manager in the chapter "Font Manager" in this book. How the Font Manager responds to a QuickDraw font request is also explained in detail in the chapter "Font Manager," and summarized later in this chapter.)

When multiple fonts of the same typeface are present in system software, the Font Manager groups them into font families. Each font in a font family can be bitmapped or outline. Bitmapped fonts in the same family can be different styles or sizes.

A font has a name and a *font family ID* number. A font name is usually the same as the typeface from which it is derived, such as Courier. If an intrinsic font is not in plain style, its style becomes part of the font's name, for example, Courier Bold. A font family ID is a resource ID for a font family that identifies the font and also reveals the script system to which the font belongs. When you set the graphics port font field (`txFont`) for a style run, you specify the font family ID. The font family ID identifies to the Font Manager both the font and the script system to be used.

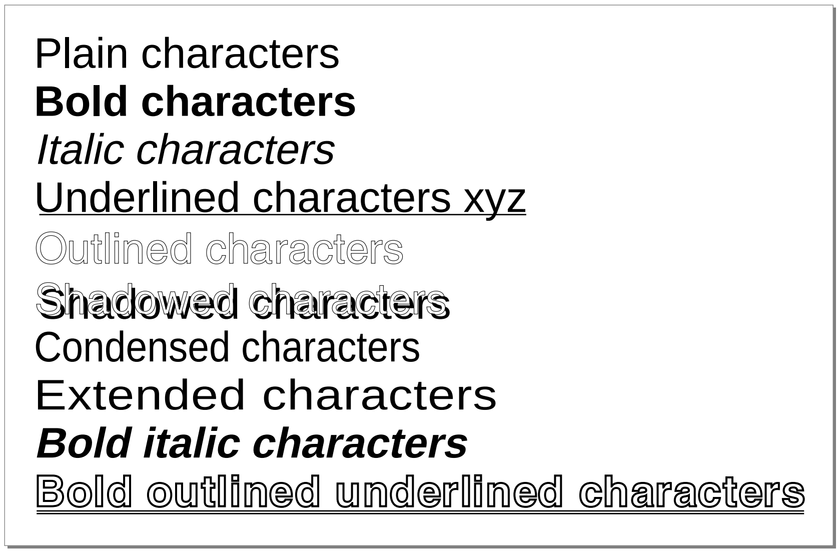
Some fonts are designed and supplied with stylistic variations integral to the font. If the Font Manager does not return a font with the requested style integral to the font design, QuickDraw applies the style. A font designer can design a font in a specific style, such as Courier Bold, or QuickDraw can add styles, such as bold or italic, to bitmaps.

A **style** is a specific variation in the appearance of a glyph that can be applied consistently to all the glyphs in a typeface. A font is described as *plain* when no styles are specified for it. The styles that QuickDraw supports include bold, italic, underline, outline, shadow, extend, and condense.

When QuickDraw requests a font in a specific style, such as Courier Bold, if the Font Manager has the font whose design includes the style, the Font Manager returns that font to QuickDraw; QuickDraw does not need to apply the stylistic variation when drawing the font, in this case. If the Font Manager does not have the font with the stylistic variation intrinsic to it, the Font Manager returns the plain font to QuickDraw, and QuickDraw applies the style when drawing the glyphs. When QuickDraw requests a font with multiple styles, if the Font Manager does not have a font with all of the styles intrinsic to it, but it has a font with one intrinsic style, the Font Manager returns that font, and QuickDraw applies the additional style or styles when drawing the glyphs. The Font Manager does not apply stylistic variations to a font.

Figure 3-1 illustrates the styles that QuickDraw supports as applied to the Helvetica font. There are many other stylistic variations not explicitly supported by QuickDraw, such as strikethrough, that you can implement.

Figure 3-1 Stylistic variations



Plain characters
Bold characters
Italic characters
Underlined characters xyz
Outlined characters
Shadowed characters
Condensed characters
Extended characters
Bold italic characters
Bold outlined underlined characters

You can specify stylistic variations alone or in combination. (Certain styles may be disabled in some script systems.) Most combinations usually look good only for large font sizes. Here are the results of specifying any of the styles that QuickDraw supports:

- Bold increases the thickness of a glyph. It causes each glyph to be repeatedly drawn one bit to the right for extra thickness.
- Italic adds an italic slant to the glyphs. Glyph bits above the base line are skewed right; bits below the base line are skewed left.
- Underline draws a line below the base line of the glyphs. If part of a glyph descends below the base line (as does the y shown in the fourth line of text in Figure 3-1), generally, the underline isn't drawn through the pixel on either side of the descending part. However, when printing to a PostScript™ LaserWriter printer, the line is drawn through the descenders.
- Outline makes a hollow, outlined glyph rather than a solid one. If you specify bold along with outline, the hollow part of the glyph is widened.
- Shadow also makes an outlined glyph, but the outline is thickened below and to the right of the glyph to achieve the effect of a shadow. If you specify bold along with shadow, the hollow part of the glyph is widened.
- Condense affects the horizontal distance between all glyphs, including spaces. Condense decreases the distance between glyphs by the amount that the Font Manager determines is appropriate.
- Extend affects the horizontal distance between all glyphs, including spaces. Extend increases the distance between glyphs by the amount that the Font Manager determines is appropriate.

The style underline draws the underline through the entire text line, from the pen starting position through the ending position, plus any offsets from the font or italic kerning. QuickDraw text clips the right edge of the underline to the ending pen position, causing outlined or shadowed underlines to match imperfectly when you draw text in sections. If the underlined text is outlined or shadowed, the ends aren't capped, that is, consecutively drawn pieces of text maintain a continuous underline.

Note that the outline and shadow styles cause the outline and shadow of the glyph to be drawn in the foreground color. The inside of the glyph, if drawn at all, is drawn in the background color.

Transfer Modes

A **transfer mode** specifies the interaction between what is to be drawn with what already exists on the screen. When you draw text, QuickDraw uses the foreground and background colors to determine how the text to be drawn, called the source, interacts with text already drawn in the current graphics port, called the destination. You define how this interplay is to occur by specifying a transfer mode, which is a value consisting of two parts. The first part is the kind of transfer mode. It specifies whether the graphic to be drawn is a pattern or text. The second part is the operation. It is a Boolean value that defines the type of interaction that is to occur, resulting in the text display.

There are two basic kinds of transfer modes in QuickDraw: pattern (`pat`), which is used to draw lines or shapes in a pattern, and source (`src`), which is used to draw text. There are four basic types of operations, totaling eight including their opposites. They are: `Copy`, `Or`, `Xor`, and `Bic`. In addition to the basic operations, there are arithmetic drawing mode operations designed specifically for use with color.

When you draw text, for each bit in the text, the corresponding bit in the destination bitmap is identified, the specified Boolean operation is performed on the pair of bits, and the resulting bit is stored into the destination bit image. The basic operations produce the following results.

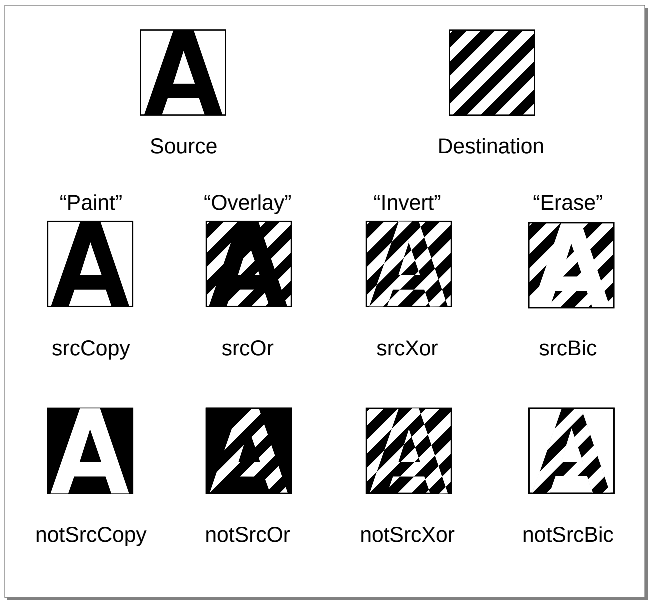
The `Copy` operation replaces the pixels in the destination with the pixels in the source, painting over the destination.

For black-and-white images, the `Or`, `Xor`, and `Bic` operations leave the destination pixels under the white part of the source unchanged. These operations differ in how they affect the pixels under the black part.

- `Or` replaces those pixels with black pixels, overlaying the destination with the black part of the source; it combines the destination with the source.
- `Xor` inverts the pixels under the black part. (The `Xor` mode inverts black in the source image at all destination depths, including 16-bit and 32-bit direct pixels.)
- `Bic` (bit clear) erases the pixels under the black part, leaving it white.

Figure 3-2 shows how each of the basic transfer modes affects the source and destination images resulting in what is displayed on the screen.

Figure 3-2 Effect of the basic transfer modes for black-and-white images



These transfer modes work with color images as follows:

- `Copy` replaces the destination with the colored source.
- `Or` mode results in the source image, regardless of the destination depth.
- `Bic` mode causes the foreground color in the source image to erase, resulting in the background color in the destination image.
- `Xor` inverts the foreground color in the source image, but not the background color, at all destination depths, including 16-bit and 32-bit direct pixels. (Inversion is not well defined for color pixels.)

The initial transfer mode for drawing text is `srcOr`. This text drawing mode is recommended for all applications because it uses the least memory and draws the entire glyph in all cases. The `srcOr` mode only affects other parts of existing glyphs if the glyphs overlap.

Note
The center of shadowed or outlined text is drawn in a graphics port in `srcBic` transfer mode if text mode is `srcOr`, for compatibility with old applications. (For color graphics ports, the center isn't drawn at all.) This allows black text with a white outline on an arbitrary background. ◆

QuickDraw Text, Script Systems, and Other Managers

Although QuickDraw provides the routines that are pivotal to drawing text on the Macintosh, it uses the services of other managers including the Font Manager and the Script Manager. To draw text consisting of multiple lines and mixed directions, you also need to use routines that belong to these managers, as well as some Text Utilities services. This section describes the relationship between QuickDraw and the Script Manager. It also provides an overview of the line-layout and text-drawing processes. For specific discussion of the routines that you use to perform the tasks inherent in these processes, see "Measuring and Drawing Lines of Text" on page 3-29.

When you draw text using QuickDraw, the Script Manager interacts with QuickDraw to provide the script-specific support. To do this, the Script Manager needs to know which script you are using. It determines this from the font that you specify in the graphics port `txFont` field. For example, if the font is Geneva, the font script is Roman. The script specified by the current graphics port font is referred to as the **font script.**

Although you can use most QuickDraw routines with all script systems, some QuickDraw routines entail restrictions. For example, you use one QuickDraw procedure to draw the glyph of a single character in a 1-byte script system, but you must use a different procedure to draw the glyph of a single character in a 2-byte script or a script system that contains zero-width characters. Some script systems contain fonts that have only 1-byte characters and some script systems contain fonts that have a mix of 1-byte and 2-byte characters. Some fonts have zero-width characters; these are usually overlapping diacritical marks which typically follow the base character in memory. With 2-byte characters, all but the first (high-order, low-address) byte are measured as zero width.

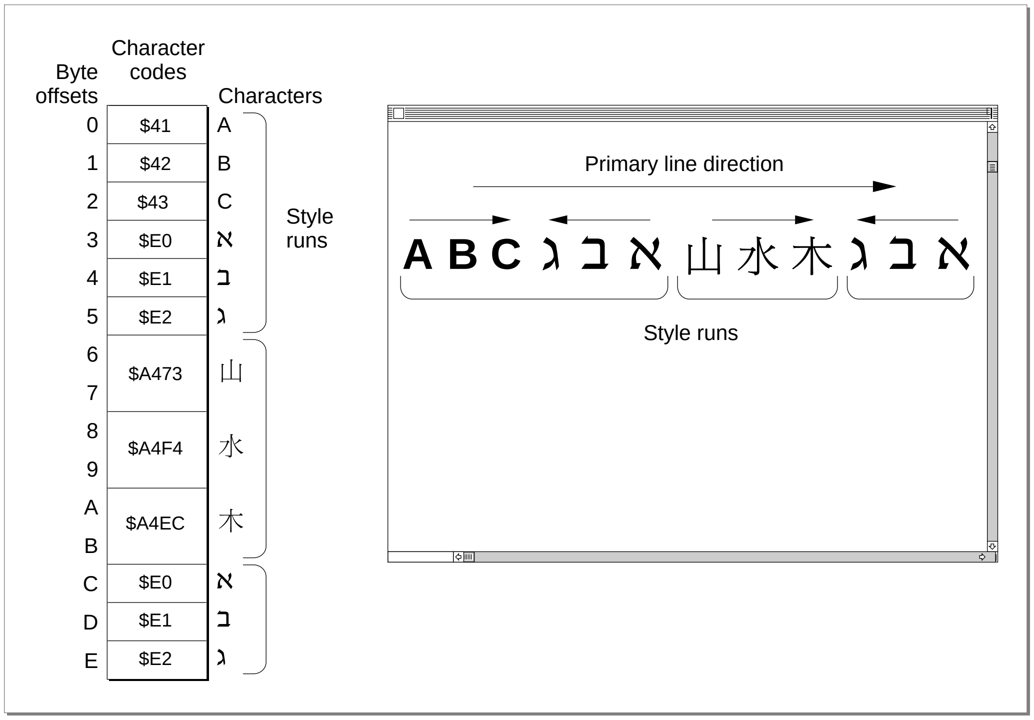
Most fonts, whatever script system they belong to, contain Roman characters, typically consisting of the 128 low-ASCII character set. The inclusion of Roman characters within another script system allows the user to enter Roman text without having to switch script systems. For script systems whose text has a left-to-right direction, such as Roman and Japanese, the direction of the text is uniform within a single style run. However, a single script system that portrays text read from right to left, such as Hebrew or Arabic, can also contain left-to-right text, such as numbers within the language of the script system or Roman-based text such as English. A single style run can also contain *bidirectional text*.

Some script systems that include the 128 low-ASCII character set include an **associated font** that is used to portray these characters. Use of an associated font is handled by the script management system without requiring any action on the part of your application. The way QuickDraw treats Roman space characters within a script system that supports bidirectional text differs from how it handles them otherwise. This behavior is explained later in this chapter in relation to eliminating trailing spaces from the end of a line.

For those script systems that support it, the existence of bidirectional text in a text range does not violate the concept of a single style run because QuickDraw uses the same text-related values in the graphics port record fields to draw all the glyphs of the entire segment of text; you do not need to change any of these values in order to draw the complete segment of bidirectional text as a single style run.

Figure 3-3 shows mixed Hebrew and Chinese text on a single line. There are three style runs. Only the first style run includes bidirectional text.

Figure 3-3 Multiple style runs on a single line



For all script systems, you measure and draw text a single style run at a time, whether the text consists of a single character, a Pascal string, or a segment of characters. A ***text segment,*** as used in this chapter, means the portion of a style run that you may pass to a single QuickDraw call. It may be a complete style run or any portion of a style run, as long as it fits on a single line. If a style run extends across a line break, you must make separate calls for the separate segments of the style run.

Whether you draw the glyph of a single character or a line of text, it is up to you to track where the text begins, both in terms of vertical and horizontal screen position and offset into your text stream. With the help of either a Font Manager function (`OutlineMetrics`) or a QuickDraw procedure (`GetFontInfo`), you can assess the line height based on the measurements of the script system font, and the associated font, if one exists, used to render the text, then determine the vertical screen position.

In most cases, you also need to know the width of the display area where the text is to be drawn. For a line of text, you can think of this area as the display line. A **display line** is the horizontal length in pixels of the screen area where you draw a line of text; the left and right ends of the display line constitute its left and right margins. You define the display line length in pixels and uses this value to determine how much text will fit on the display line.

You specify where QuickDraw is to begin drawing by setting the current pen location of the graphics port. Within a single line of text, QuickDraw takes care of correctly advancing the pen position after it draws each glyph or text segment.

For text that exceeds a single display line, you must control where a line ends and the next one begins. For unidirectional text, this task essentially constitutes the line layout process. For mixed-directional text, the order in which you display the style runs may be different from their storage order. In this case, you also need to determine the drawing order in the line layout process.

To draw a line of text, you can loop through the text, laying it out first, then loop through the drawing process. A line-layout loop measures the text and determines where to break it. In most cases, you can use a Text Utilities function (`StyledLineBreak`) for this purpose.

To lay out justified text, a loop needs to include several additional steps that determine how to distribute the extra space among the text of the line. This process entails eliminating trailing spaces from the end of the line, then distributing the remaining extra space among the text. How the distribution of extra space is expressed throughout a line of text is dependent on the script system. For example, some script systems add additional width to space characters that are used as word delimiters; some script systems, which use connecting glyphs, stretch certain glyphs to encompass the additional width. See the next section, “Text Formatting and Justification,” for more information.

Before you call a QuickDraw measuring routine, you need to set the graphics port text-related fields to those of the style run that the text is part of. You set these fields only for each new style run.

Once you have laid out a line of text, drawing it is fairly simple. An application can have a text-drawing loop that positions the pen at the beginning of a new line, sets the text-related fields of the current graphics port to the text characteristics for that style run if the text string begins a new style run, then draws the text, using one of the QuickDraw drawing routines to draw aligned text, justified text, or scaled text.

Text Formatting and Justification

When you lay out text, you can change its width and its alignment. You change the width of text to format it for special purposes, or to justify the text to fit a display area or a given line. To justify text, you spread or condensed it so that any white space is distributed evenly throughout the display area or line.

You can draw text that is aligned with either the right margin of the display area or line, which produces ragged-left text, or the left margin, which produces ragged-right text. You align text by positioning the pen appropriately so that the first glyph of the text line is flush against the margin.

There are several ways to change the width of text. You can

- use the QuickDraw justification routines that measure and draw text, automatically changing the width of the text appropriately for each script system
- set the graphics port `txFace` field to condense or extend the text
- set the graphics port `spExtra` and `chExtra` fields to narrow or widen space and nonspace characters by a specific number of pixels

You can even justify text that includes special formatting. For example, you can extend or condense the width of space and nonspace characters, while justifying the text line overall.

Of these methods, the easiest way to justify text for all script systems is to use the QuickDraw justification routines. These routines handle the script system requirements for your application. For example, because the text of some script systems, such as Arabic and Devanagari, is drawn as connected glyphs, the justification routines do not add width to or remove it from nonspace characters.

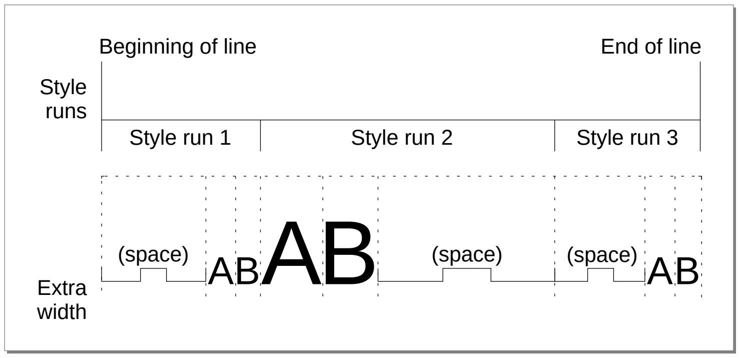
The justification routines assume that a slop value specified in pixels is to be distributed throughout the text. The **slop value** is the difference between the width of the text and the width of the display area or line. You can pass the justification routines a positive or negative slop value. To extend text to fit the display area or line, you specify a positive slop value. To justify a line of text more smoothly by condensing it when it only slightly exceeds the display area or line length, you can use a negative slop value.

How the justification routines distribute this extra space within a style run depends on the script system.

- For Roman script systems, text justification is performed by altering the size of the space characters. You can think of this as interword spacing. (Every space in a style run is allocated the same amount of extra width and thus is the same size, whether or not it is at the beginning or end of the line or the style run.)
- For Arabic, the justification routines insert extension bar glyphs between joined glyphs and widen space characters to fill any remaining gaps.
- For scripts that don’t use spaces to delimit words, these routines usually modify the intercharacter spacing to achieve justification.

Figure 3-4 shows a line of text in the Roman script system containing three style runs and how extra space is distributed among the space characters within a style run.

Figure 3-4 Justification of Roman text



To correctly handle spacing between multiple style runs on a line, the justification routines take a parameter that specifies the position of the style run on the line. The style run position parameter is meaningful only for those script systems that use intercharacter spacing for justification. For all other script systems, the parameter exists for future extensibility. Although the style run position parameter is not used, for example, for justifying text in the Roman script system, to allow for future compatibility, you should always specify the appropriate value for it for all calls that take it.

For those script systems that do use intercharacter spacing, space between styles is allocated differently depending upon whether the style is leftmost, rightmost, or between two other style runs. For example, if a style run occurs at the beginning or end of a line, extra space is *not* added to the outer edge of the outermost glyph, whereas if a style run is interior to a line, all of the glyphs of the text are treated the same: extra space is allocated to both sides of every glyph including those at either end of the style run.

Note
The text justification routines do not automatically eliminate trailing spaces from the last style run on the line. However, QuickDraw provides a routine (`VisibleLength`) that does not include trailing spaces in the byte count of the last style run on the line. ◆

If you do not want to justify a range of text, you can change the width of the text for onscreen display, for example, to format an advertisement by setting the graphics port `spExtra` and `chExtra` fields to an amount by which space and nonspace characters are to be widened or narrowed. If you use `SpaceExtra` and `CharExtra` to widen or narrow text, you are responsible for handling them properly for the script system.

The original graphics port does not have a `chExtra` field, so you can only change the width of nonspace characters if you use a color graphics port. Although line breaks are maintained, spacing defined by these values is not preserved when you print to a LaserWriter printer.

Scaling

Text scaling is the process of changing glyphs from one size or shape to another. This section discusses two kinds of scaling: implicit and explicit. This section also summarizes how the Font Manager handles scaling requirements when scaling is disabled. The chapter “Font Manager” in this book describes disabling scaling in greater detail.

Implicit scaling is performed automatically by QuickDraw when the Font Manager cannot supply a bitmapped font in the size that you request. In this case, the Font Manager returns to QuickDraw a bitmapped font that is the closest approximation, along with scaling factors. QuickDraw uses these values to scale the text when drawing it. This process is transparent to your application. Because the Font Manager can always satisfy a font request completely when outline fonts are installed, no scaling is necessary.

Explicit scaling is performed in essentially the same way as implicit scaling, but you specify how the text is to be scaled. Several QuickDraw routines include parameters that let your application specify (explicitly) how text is to be scaled. You might want to scale text explicitly, for example, to create unusual glyph shapes, or to increase or decrease the size of text when a user clicks in a zoom box. You can use the high-level QuickDraw justification routines to explicitly scale text. Alternatively, you can use the low-level standard measuring and drawing routines, referred to as *bottleneck* routines. See “Using Scaled Text” on page 3-44. To explicitly scale text, you specify values that let you stretch or shrink a glyph horizontally or vertically. You can change a glyph from a familiar point size to one that is unusual—for example, a glyph that is 12 points high but as wide as the entire page.

The same rules apply to the interaction between the Font Manager and QuickDraw whether scaling is implicit or explicit. However, for explicit scaling, QuickDraw passes the scaling factors that you specify as routine parameters to the Font Manager in an input record (`FMInput`) along with the standard information, which includes the font family ID number, the size, and the stylistic variation of the font request. Taking the requested scaling factors into account, the Font Manager follows a standard path looking for an available font that best satisfies the request, and returns the bitmap information to QuickDraw via an output record (`FMOutput`), which contains a handle to the requested font resource and, among other information, the scaling factors that now apply, if any. The returned scaling factors describe how QuickDraw is to draw the text to fulfill the input scaling factors request.

If you use the low-level bottleneck procedure or the higher-level justification procedure to *draw* the scaled text and the Font Manager returns scaling factors to be applied to the text, QuickDraw applies the additional scaling.

The low-level bottleneck *measuring* function lets you specify scaling factors in reference parameters. If only bitmapped fonts are installed and a font does not exist that matches the scale you specify, the Font Manager uses the font that best approximates the request, and measures the text using that font. The Font Manager returns scaling factors in the reference parameters, along with the width of the text based on the supplied font. In this case, QuickDraw does not apply the necessary additional scaling to the text to give you the correct text measurement including scaling. To measure the text correctly, you need to apply the additional scaling to the text width of the font that the Font Manager returns.

For example, suppose only bitmapped fonts are installed and you request a point size of 24 with a horizontal scaling factor of 2/1 and a vertical scaling factor of 1/1. The Font Manager returns the most optimal matching font that it has, which is 12, say, with a horizontal scaling factor of 4/1 and a vertical scaling factor of 2/1. Now, you must apply these scaling factors to the text width and height metrics in the 12-point font to get the correct text measurement.

You can use the Font Manager `SetFScaleDisable` procedure to enable or disable font scaling of bitmapped glyphs. When you disable scaling, the Font Manager finds the closest, smaller-sized font to the one that you request, and adjusts the width table associated with the font to match the requested size. As a result, the height of the glyphs is smaller than you requested, but the spacing compensates for it. When scaling is disabled, the Font Manager returns 1/1 scaling in response to the request.

For a complete discussion of how the Font Manager determines which font to return to QuickDraw to satisfy a font request, see the chapter "Font Manager" in this book. This chapter also describes the `SetFScaleDisable` procedure and the width table.

Carets and Highlighting

Highlighting a selection range and marking the insertion point with a caret both involve converting offsets of characters in a text buffer to pixel locations on a display screen. This task is prerequisite to both drawing a caret and highlighting text. See the chapter "Introduction to Text on the Macintosh" in this book for a discussion of the conventions underlying the relationship of a character at a byte offset to a caret position for unidirectional text and text at a direction boundary.

When the text is unidirectional, performing these tasks is uncomplicated because storage order and display order are the same. For unidirectional text, a caret position always falls between the corresponding glyphs of these characters—on the leading edge of one and the trailing edge of the other. When the text is bidirectional, it can contain characters that occur on direction boundaries; although the characters are stored contiguously in memory, the leading edge of one character's glyph does not constitute the trailing edge of the other in display order. Consequently, two physically separate caret positions exist on the display screen, one associated with each glyph.

There are a number of situations in which you need to know a caret position, and they fall within two categories: drawing a caret to mark the insertion point, and using a caret position to denote an endpoint for highlighting a text selection. For a discussion of marking an insertion point with either a single caret or a dual caret, and caret movement with arrow keys, see the chapter "Introduction to Text on the Macintosh."

You need to know the caret positions marking the endpoints of a text selection to highlight it when the user selects either a word or a range of text, and for other features that the application supports, such as a search operation. Generally, you know the byte offsets of the characters that begin and end a selection range for tasks such as search operations. However, when the user clicks in or selects a range of text to be highlighted, usually you first need to convert the pixel locations marking the cursor locations to the corresponding characters' byte offsets in memory, and then convert the characters' byte offsets to caret positions.To encompass all of the characters within the text segment to be highlighted, you use caret positions that mark endpoints which include the beginning and ending characters of the text.

On a black-and-white screen, highlighting a selection is simple; white pixels turn black and vice versa. In a color environment, the inversion of multibit pixel values usually yields many different colors, which is unsuitable for highlighting text. To highlight text rendered in color, QuickDraw lets you specify a highlight value that it uses instead of the current graphics ports background color. Generally the user sets the highlighting color, but your application can change the color. When you use highlight mode, all pixel values of the current background color are replaced with the value of the highlighting color.

Using QuickDraw Text

This section describes how to

- initialize QuickDraw
- set up the text drawing environment
- specify the text characteristics, such as the font, style, spacing, and transfer mode
- measure and draw text ranging from a single character to a line containing multiple styles and mixed directions

Preparing to Use QuickDraw

The QuickDraw text-handling routines rely on both QuickDraw and the Script Manager. Therefore, before you call any of these routines, you need to determine what versions of QuickDraw and the Script Manager are installed, and initialize QuickDraw. For more information about determining the version of the Script Manager, see the chapter "Script Manager" in this book.

Determining the Version and Initializing QuickDraw

To determine the current version of QuickDraw, you call the `Gestalt` function with the `gestaltQuickdrawVersion` selector. The `gestaltQuickdrawVersion` selector returns a 2-byte value indicating the version of QuickDraw currently present. The high-order byte of that number represents the major revision number, and the low-order byte represents the minor revision number. These are the currently defined values for the QuickDraw selector.

Constant	Value
`gestaltOriginalQD`	`$000`
`gestaltOriginalQD1`	`$001`
`gestalt8BitQD`	`$100`
`gestalt32BitQD`	`$200`
`gestalt32BitQD11`	`$210`
`gestalt32BitQD12`	`$220`
`gestalt32BitQD13`	`$230`

Gestalt returns a 4-byte value in its response parameter; the low-order word contains QuickDraw version data. In that low-order word, the high-order byte gives the major revision number and the low-order byte the minor revision. Major revisions currently defined are the original QuickDraw, the original Color QuickDraw, and the current 32-Bit QuickDraw with direct-pixel capability.

Values having a major revision number of 1 or 2 indicate that Color QuickDraw is available in either the 8-bit or 32-bit version. These results do not, however, indicate whether a color monitor is attached to the system. You need to use high-level QuickDraw routines to obtain that information.

Many Macintosh applications don't care what version of QuickDraw is available on the user's system: they don't use color at all, use only the basic QuickDraw color model, or specify all their colors abstractly, in RGB form. If your application does depend on a specific version of QuickDraw, you can check the version at run time and adapt to make best use of the available hardware (or at least inform the user gracefully that your program's graphics needs aren't being met).

For more information about the `Gestalt` function, see the chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.

Initialize QuickDraw at the beginning of your program before any other parts of the Toolbox. To do so, call the `InitGraf` procedure. For more information about the `InitGraf` procedure, see *Inside Macintosh: Imaging*.

Setting Up the Text-Drawing Environment

You draw text in the current graphics port. You create a distinct graphical environment for every window on the screen by specifying values for the graphics port. Each graphics port has its own complete drawing environment—including its own coordinate system, drawing location, font set, and character attributes.

Because your application can have more than one window open at the same time, QuickDraw routines access the data structures within the current graphics port only. You must keep track of the current graphics port and identify it to QuickDraw when you change windows.

You can use the QuickDraw `SetPort` procedure, which operates on both types of graphics ports (`GrafPort` and `CGrafPort`), to identify the current graphics port. You use the global variable `thePort` to indicate the current port. In the following example, `SetPort` identifies the port pointed to by `thePort` as the current one.

```
SetPort(thePort);
```

For more information about the `SetPort` procedure, see *Inside Macintosh: Imaging*.

Each time you draw text in a window's graphics port, you need to set the text-related fields of the graphics port to the characteristics of the text that you want to draw, if they differ from the current ones. A graphics port record contains three fields that determine how text is drawn—the font, style, and size of glyphs—and one that specifies how it will be placed in the bit image, the transfer mode. In addition to these fields, a graphics port record contains two fields that let you specify character widths to define how text is to be formatted on a line.

Specifying Text Characteristics

Each time you measure or draw text that begins a new style run and whose characteristics differ from those of the current graphics port, first you need to set the graphics port text-related fields to match those of the text. Here is how the text-related graphic port fields are initialized:

Field	Value	Explanation
`txFont`	`0`	System font
`txFace`	`[]`	Plain style of the current font
`txSize`	`0`	Size of the system font used to draw text
`spExtra`	`0`	Standard width of the space character for the font
`chExtra`	`0`	Standard width of nonspace characters for the font
`txMode`	`srcOr`	Combines the destination with the source

Do not modify any of these fields directly. Instead, always use the QuickDraw routines to change their values: `TextFace`, `TextFont`, `TextMode`, `TextSize`, `SpaceExtra`, and `CharExtra`. Using these routines ensures that your application will benefit from any future improvements to QuickDraw.

Listing 3-1 shows a simple sequence of QuickDraw calls. These routines set the current port, set the graphics port text fields, then draw a text string. The calls render the text in 12-point Geneva font using the styles bold and italic, and widen the spaces between words by 3 pixels to format the text. QuickDraw text-handling procedures that set these fields are discussed later in this section.

Listing 3-1 Using QuickDraw to set the graphics port text-related fields

```
SetPort(thePort);
TextFont(geneva);
TextFace([bold, italic]);
TextSize(12);
SpaceExtra(3);
```

If you must directly change the values of any of the graphics port fields, call the QuickDraw `PortChanged` procedure to notify QuickDraw of the change after you modify the field. For more information about `PortChanged`, see the QuickDraw chapters in *Inside Macintosh: Imaging*.

Setting the Font

You use the `TextFont` procedure to set the font for the text. The value that you specify for this field is either the font family ID or a predefined constant.

If you know the font name, you can get the font family ID by calling the `GetFNum` procedure, passing it the font name. You can get a font's name if it has a font family ID by calling the Font Manager `GetFontName` procedure. For more information about these procedures and the predefined font constants, see the chapter "Font Manager" in this book.

If you do not know either the font family ID or the name of the font, you can use the Resource Manager's `GetIndResource` function followed by the `GetResInfo` procedure to determine the fonts that are available and what their names and IDs are. See the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox* for more information.

The values 0 and 1 have special significance. When a graphics port is created, the `txFont` field is initialized to 0, which specifies the system font. This is the font that the system uses to draw text in system menus and system dialog boxes. You can use a font that is defined by the system—to do so, it sets this field to 1; 1 always specifies the application font.

Note

Do not use the font family ID 0 or the constant `'chicago'` to specify the Chicago font because the ID can vary on localized systems. To specify the Chicago font, use the following calls.

```
GetFNum('Chicago', theNum);
TextFont(theNum);
```

The variable `theNum` is an integer. ◆

Storing a font name in a document

Always store a font name, rather than its font family ID, in a document to avoid problems that can arise because IDs are not unique—many font families share the same font family ID—or because one font family may have different IDs on different computer systems. ◆

You use `TextFont` to set the `txFont` field of the current graphics port for a new style run that uses a font different from the current one, and in response to a user's actions, for example, when a user selects a new font from a menu.

Note

Whenever a user changes the keyboard script, you are responsible for setting the `txFont` field to the new font, so that the keyboard script and the font script are synchronized. ◆

Modifying the Text Style

When you create a graphics port, the `txFace` field value is initially an empty set (`[]`), which specifies the plain style of the current font.

To change the text style, you call `TextFace`, using any combination of the following constants to specify the text style: `bold`, `italic`, `underline`, `outline`, `shadow`, `condense`, and `extend`. In Pascal, you specify the value or values within square brackets. For example:

```
TextFace([bold]);          {bold}
TextFace([bold,italic]);   {bold and italic}
```

You can also add another style to the current text style, or remove a style. For example:

```
TextFace(thePort^.txFace+[bold]); {existing style plus bold}
TextFace(thePort^.txFace-[bold]); {existing style minus bold}
```

To reset the style to plain, you specify an empty set. For example:

```
TextFace([]);  {plain text}
```

If you want to restore the existing value after you draw the text in another style, save it before you call `TextFace`. For a description of how QuickDraw renders text in each of these styles, see "Font, Font Style, and Font Size" on page 3-5.

Changing the Font Size

When you create a graphics port, the value of the `txSize` field is 0, which specifies the size of the font to be used to draw system text, such as menus. The size of the system font is usually 12 points. You use the `TextSize` procedure to set the `txSize` field of the current graphics port to the font's point size. Text drawn on the QuickDraw coordinate plane can range from 1 point to 32,767 points.

Changing the Width of Characters

When you create a graphics port, the `spExtra` and `chExtra` fields are set to 0, which specifies the standard width for space and nonspace characters in the font. You change the width of space characters and nonspace characters using the `SpaceExtra` and `CharExtra` procedures, respectively.

Widening or narrowing space and nonspace characters lets you meet special formatting requirements that are not satisfied by simply justifying the text. If you want to change only the width of the space characters in a line of text for onscreen typographical formatting, you can set the `spExtra` field value before you draw each style run, narrowing spaces in some style runs and widening those in others. To change the width of nonspace characters, either extending them or narrowing them, you set the `chExtra` field value before you draw a style run.

You use the `SpaceExtra` procedure to set the `spExtra` field of the current graphics port to the number of pixels to be added to or subtracted from the standard width of the space character in the style run. (A value specified in the `spExtra` field is ignored by script systems that do not use space characters, so don't to set it for 2-byte script systems that use only intercharacter spacing.) The text measuring and drawing routines apply the `spExtra` field pixel value to every space in the text string, regardless of whether the space occurs at the beginning or the end of a style run or between words within a style run. You can use the `SpaceExtra` procedure, for both a color graphics port (`CGrafPort`) and an original graphics port (`GrafPort`).

You use the `CharExtra` procedure to set the `chExtra` field of the current graphics port to the number of extra pixels to be added to or subtracted from the width of all nonspace characters in a style run. Because only the color graphics port record has a `chExtra` field, use of `CharExtra` is limited to color graphics ports. The measuring and drawing routines apply the pixel value that you set in the `chExtra` field to the right side of the glyph of each nonspace character.

You can use `SpaceExtra` and `CharExtra` together, for example, to format text consisting of multiple style runs with different fonts in order to create a smooth visual effect by causing the fonts to measure the same or proportionally.

Note
Although printing on a LaserWriter preserves the line's endpoints, it alters the line layout in between. Any formatting internal to the line that you set through `SpaceExtra` and `CharExtra` is lost when you print. ◆

If you do not want to use the justification routines to draw justified text, you can justify a line of text using `SpaceExtra` and `CharExtra` to widen each glyph (space and nonspace characters) by the same amount of pixels for onscreen display. Here is how you do this:

1. Determine the slop value to be applied to the text to justify it.
   - Measure the width in pixels of each style run in the line of text using `TextWidth`.
   - Sum the values.
   - Subtract the total from the display line length.
2. Count the total number of characters (both space characters and nonspace characters) that the text contains.
3. Divide the slop value by the number of characters minus 1. Round the slop value to a whole number.
4. Call the `SpaceExtra` procedure, passing it the result of step 3.
5. Call the `CharExtra` procedure, passing it the result of step 3.
6. Call `DrawText` or `DrawString` to draw each style run on the line.

Use of `CharExtra` entails some restrictions. You cannot use intercharacter spacing for 1-byte complex script systems or 1-byte simple script systems that include zero-width characters, such as diacritical marks, because of the way extra width is applied to a glyph. For example, for 1-byte simple script systems with diacritical marks, intercharacter space is added to all glyphs separating the diacritical mark from the glyph of the character.

Using Fractional Glyph Widths

Fractional glyph widths are measurements of a glyph's width that can include fractions of a pixel. Using fractional glyph widths allows QuickDraw to place glyphs on the screen in a manner that will closely match the eventual placement of glyphs on a page printed by a LaserWriter. Fractional glyph widths make it possible for the LaserWriter printer to print with better spacing. You can use the Font Manager's `SetFractEnable` procedure to turn the use of fractional glyph widths on or off.

Because screen glyphs are made up of whole pixels, QuickDraw cannot draw a fractional glyph. To compensate, QuickDraw rounds off the fractional parts resulting in uneven spacing between glyphs and words. Although the text is somewhat distorted on the screen, it is correctly proportioned and shows no distortion when printed on a page using a LaserWriter.

However, to avoid screen distortion, your application can disable the use of fractional widths. A consequence of this it that placement of text on the printed page is less than optimal. For more information about fractional glyph widths, see the chapter "Font Manager" in this book.

Specifying the Transfer Mode

The value of the current graphics port transfer mode (`txMode`) field determines the way glyphs are placed in the bit image and how the text is to appear. It defines the way text to be drawn interacts with text and graphics already drawn. (When a glyph is drawn, QuickDraw does a bit-by-bit comparison based on the mode and stores the resulting bits into the bit image.) You set the text mode by calling the `TextMode` procedure.

By default, the transfer mode field is set to `srcOr`, which specifies that text to be drawn should overlay the existing graphics. The `srcOr` transfer mode produces the best results for drawing text because it writes only those bits that make up the actual glyph. In most situations, when drawing text with the basic transfer modes, you should use only `srcOr` or `srcBic`; all other modes can result in clipping of glyphs by adjacent glyphs. For example, the `Copy` operation paints over what already exists on the destination, replacing it entirely.

Basic Transfer Mode Operations

For each type of drawing mode, there are four basic kinds of operations: `Copy`, `Or`, `Xor`, and `Bic` (bit clear). For Color QuickDraw, there are additional arithmetic drawing mode operations designed specifically for use with color. They are discussed later in this section, and fully in *Inside Macintosh: Imaging*.

The transfer mode operation determines how the text is to be displayed: for each bit in the item to be drawn, the corresponding bit in the destination bitmap is identified, the specified Boolean operation is performed on the pair of bits, and the resulting bit is stored into the destination bit image. When you work with color pixels, transfer modes produce different results on indexed and direct devices.

In addition to drawing the entire glyph in all cases, the `srcOr` mode is recommended for all applications because it uses the least memory. The `srcOr` mode only affects other parts of existing glyphs if the glyphs overlap. In `srcOr` mode the color of the glyph is determined by the foreground color.

The maximum stack space required for a text drawing operation can be considerable. Text drawing uses a minimum amount of stack if the following conditions are true:

- The transfer mode is `srcOr`.
- The foreground color is black.
- The destination of the text is contained within a rectangular portion of the region of the graphics port that is actually visible on the screen.
- The text is not scaled.
- The text does not have to be italicized, outlined, or shadowed by QuickDraw.

Otherwise, the amount of stack space required to draw all of the text at once depends most on the size and the width of the text and the depth of the destination.

If QuickDraw can't get enough stack space to draw an entire text segment at once, it draws the segment in pieces. This can produce unusual results in modes other than `srcOr` or `srcBic` if some of the glyphs overlap because of kerning or italicizing. If the mode is `srcCopy`, overlapping glyphs are clipped by the last drawn glyph. If the mode is `srcXor`, pixels where the glyphs overlap are not drawn at all. If the mode is one of the arithmetic modes, the arithmetic rules are followed, ignoring that the destination may include part of the text being drawn.

The stack space required for a drawing operation in Color QuickDraw is roughly estimated by the following calculation.

```
(text width) * (text height) * (font depth) / (8 bits per byte) + 3K
```

Pixel depth normally equals the screen depth. If the amount of stack space available is small (less than 3.5K), QuickDraw instead uses a pixel depth of 1, which is slow, but uses less stack space.

For the original QuickDraw, the required stack space is roughly estimated by the following calculation.

```
(text width) * (text height) / (8 bits per byte) + 2K
```

Arithmetic Transfer Mode Operations

Arithmetic transfer modes calculate pixel values by adding, subtracting, or averaging the RGB components of the source and destination pixels. The arithmetic modes change the destination pixels by performing arithmetic operations on the source and destination pixels.

Each drawing routine converts the source and destination pixels to their RGB (red, green, and blue) components, performs an operation on each pair of components to produce a new RGB value for the destination, and then assigns the destination to a pixel value close to the calculated RGB value. The arithmetic drawing modes are `addOver`, `addPin`, `subOver`, `subPin`, `adMax`, `adMin`, and `blend`. To specify an arithmetic mode, you pass the operation to be used to the `TextMode` procedure. For example, the following calls save the current state of the text mode field, then set it to the transfer mode blend.

```
oldTextMode := theport.^txMode;
TextMode(blend);
```

The arithmetic modes were designed for use with color. They are most useful for 8-bit color, but they work on 4-bit and 2-bit color also. When the destination bitmap is one bit deep, the mode reverts to the basic transfer mode that best approximates the arithmetic mode requested. For more information about arithmetic mode operations, see the QuickDraw chapters in *Inside Macintosh: Imaging*.

Note
To help make color work well on different screen depths, Color QuickDraw does some validity checking of the foreground and background colors. If your application is drawing to a `CGrafPort` with a depth equal to 1 or 2, and if the RGB values of the foreground and background colors aren't the same, but both of them map to the same pixel value, then the foreground color is inverted. This ensures that, for instance, red text drawn on a green background doesn't map to black on black. ◆

The grayishTextOr Transfer Mode

You can use the text drawing mode, `grayishTextOr`, to draw dimmed text on the screen. It is especially useful for displaying disabled user interface items. If the destination device is color and `grayishTextOr` is the transfer mode, QuickDraw draws with a blend of the foreground and background colors. If the destination device is black and white, the `grayishTextOr` mode dithers black and white. **Dithering** is a technique that creates the effect of additional colors, if the destination device is color. If the destination device is black-and-white, dithering creates the effect of levels of gray.

Note that `grayishTextOr` is not considered a standard transfer mode because currently it is not stored in pictures, and printing with it is undefined. (It does not pass through the QuickDraw bottleneck procedure.) The following calls show how to use `grayishTextOr`. They save the current state of the text mode field, then set it to `grayishTextOr`.

```
oldTextMode := theport.^txMode;
TextMode(grayishTextOr);
```

Text Mask Mode

You can add the `mask` constant to another transfer mode to cause only the glyph portion of the text to be applied in the current transfer mode to the destination. If the text font contains more than one color or if the drawing mode is an arithmetic mode, the mask mode causes only the portion of the glyphs not equal to the background to be drawn.

The arithmetic transfer modes apply the glyph's background to the destination; this can lead to undesirable results if you draw the text in pieces. QuickDraw draws the leftmost part of a piece of text on top of a previous piece if the font kerns to the left. Using `maskMode` in addition to these modes causes only the foreground part of the glyph to be drawn.

Because the rightmost glyph is clipped, to kern to the right in text mask mode, you should use `srcOr`, or add trailing spaces. The following call sets the transfer mode to blend with mask mode.

```
TextMode(blend + mask);
```

Transparent Transfer Mode

The `transparent` mode replaces the destination pixel with the source pixel when the source pixel isn't equal to the background color. For a complete description of the transparent mode, see the QuickDraw chapters in *Inside Macintosh: Imaging.*

The arithmetic transfer modes apply the glyph's background to the destination; this can produce undesirable results if you draw the text in pieces. QuickDraw draws the leftmost part of a piece of text on top of a previous piece if the font kerns to the left. If you use the mask mode (`maskMode`) in addition to these modes, QuickDraw draws only the foreground part of the glyph. Because the rightmost glyph is clipped, to kern to the right in text mask mode, you should use `srcOr`. For an explanation of kerning, see the chapter "Font Manager" in this book.

Transfer Modes and Multibit Fonts

Multibit fonts can have a specific color. Some transfer modes may not produce the desired results with a multibit font. However, the arithmetic mode and transparent mode work equally well with single bit and multibit fonts.

Unlike single bit fonts, multibit fonts draw quickly in `srcOr` mode only if the foreground is white. Single bit fonts draw quickly in `srcOr` mode only if the foreground is black. Grayscale fonts produce a spectrum of colors, rather than just the foreground and background colors.

Measuring and Drawing Single Segments of Text

This section describes how to draw a single glyph or a series of glyphs that share the same font and character attributes. Because you usually measure text before you draw it to determine if it fits on the display area, this section describes the measuring and drawing routines in pairs. These pairs are `CharWidth` and `DrawChar` to measure and draw the glyph of a single character, `StringWidth` and `DrawString` to measure and draw Pascal strings, and `TextWidth` and `DrawText` to measure and draw text segments.

You are responsible for tracking and specifying the memory location and the character attributes of the text to be drawn. This is true whether you are working with a single glyph, a Pascal string, or a text string. For a single glyph, you pass the character code to the procedure; for a Pascal string, you pass the string.

Note
Before you call a QuickDraw measuring or drawing routine, you need to set the graphics port text-related fields to those of the character. ◆

Individual Glyphs

You measure text a character at a time by calling the `CharWidth` function, and you draw text a character at a time by calling the `DrawChar` procedure. These routines only work with 1-byte simple script systems.

Although this section describes how to use these routines, you should understand their limitations, and avoid using `CharWidth` and `DrawChar` for applications drawing sequences of more than one character. Nevertheless, you may want to use `DrawChar` for special purposes, such as including a glyph in a book's index, to show a single glyph as it exists apart from contextual transformations.

The `CharWidth` function takes into account all of the text characteristics defined in the current graphics port, so make sure these values reflect the attributes that you intend to draw with. You can draw a sequence of individual glyphs by placing the pen at the beginning of the leftmost glyph, and making a succession of calls to `DrawChar`.

Always use `CharWidth` to measure the width of a sequence of glyphs that you intend to draw using `DrawChar`, instead of using `StringWidth`; `StringWidth` and `DrawString` accumulate fractional portions, while `CharWidth` and `DrawChar` do not. Making successive calls to `CharWidth` to measure a segment of text and calling `StringWidth` to measure the same segment of text produce different results.

In general, you should measure and draw text in segments, rather than as individual glyphs. In Roman fonts, if you measure fractional-width glyphs singly using `CharWidth`, you can get incorrect results, because `CharWidth` doesn't accumulate fractional-width positions. Also, it takes longer to measure the widths of several glyphs one at a time than it does to measure them together using `TextWidth`.

For contextual 1-byte fonts, `CharWidth` and `DrawChar` do not correctly measure or draw ligatures, reversals, or other contextual forms. You cannot use `CharWidth` and `DrawChar` for 2-byte fonts, because they take a 1-byte character code as a parameter.

Pascal Strings

You can call the `StringWidth` function to measure the screen pixel width of a Pascal string to determine how many glyphs will fit on the screen, and you can call the `DrawString` procedure to draw a Pascal string with the text characteristics of the current graphics port. The `DrawString` procedure accumulates the fractional portion as it draws each glyph and positions the next glyph correctly.

You cannot use the `DrawString` and `StringWidth` routines to draw or measure a Pascal string that is justified or explicitly scaled. If you want to do this, you must separate the string from its length byte, and call `MeasureJustified` or `StdTxMeas` to measure it, and then `DrawJustified` to draw it, passing the text and the text length separately. Note that a Pascal string is limited to 255 characters. The following code fragment shows how to adjust for the length byte.

```
myTextPtr := Ptr(Ord(@myString) + 1);
myTextLength := Ord(myString[0]);
```

Text Segments

You can call the `TextWidth` function to measure a segment of text to see if it fits on a single line; if it does, then you can the `DrawText` procedure to draw it.

You pass `DrawText` a pointer to the text buffer, the byte offset into the text buffer of the first character to be drawn, and the length of the text segment you want to draw. QuickDraw draws the text at the current pen position with the text characteristics of the current graphics port.

Measuring and Drawing Lines of Text

This section describes how to lay out and draw a line of text consisting of a single style run or multiple style runs. A line of text all in the same font, script, and character attributes constitutes a single style run. A new style run begins when any of these textual characteristics change. QuickDraw relies on the construct of style runs to track these text attribute changes throughout a line of text. Before you measure or draw a text segment that constitutes a new style run, you need to set the text-related graphics port fields for that style run.

This section also describes how to draw text lines that are right or left aligned, or justified. Finally, it explains how to draw explicitly scaled text, whether the lines of text are justified or not.

To draw a line of text, you first need to lay it out. If the text does not contain mixed directions, the text layout process consists of a single step: determining where to break the line. If the text contains mixed directions, the order in which you display the style runs may be different from their storage order, so you also need to determine the drawing order.

Moreover, if you want to draw a line of justified text, the process entails additional steps: you need to determine the total amount of extra pixels that remains to be distributed throughout the line of text and how to distribute these extra pixels throughout the style runs.

If you want to draw a line of text that is not justified, you can position the pen according to its alignment. You align text by positioning the pen appropriately so that the first glyph of the text line is flush against the display line's margin: at the left margin for left-aligned text, or at the right margin for right-aligned text.

Your application loops through these steps for each style run and each line of text that it measures and draws, and it needs to track the text in memory as it proceeds through each loop. Each time you measure or draw a text segment, you need to pass the beginning byte offset and its length to the QuickDraw routine. Before you call a QuickDraw measuring or drawing routine, you need to set the graphics port text-related fields to reflect the new style run's values.

These steps summarize the line layout and drawing process:

1. Determine where to break the line.
2. Determine the display order of the style runs (mixed-directional text).
3. Eliminate trailing spaces (justified text).
4. Calculate the slop value (justified text).
5. Distribute the slop (justified text).
6. Position the pen.
7. Draw the text.

Each step covers the basics, plus any additional information you need to know to perform the step for justified or scaled text. The following sections elaborate these steps.

Determining Where to Break the Line

For text that spans multiple lines, you are responsible for controlling where a text line starts and ends. To determine where to break a line of text, first you need to know the screen pixel width of the display line. Then, taking into account all the text characteristics, you need to assess how much of the text you can display on the line, and the appropriate point to break the text.

You should always break a line on a word boundary. To allow for text in different languages, use the QuickDraw and the Text Utilities routines that identify the appropriate place to break a line in any script system.

The two routines you use for unscaled text are the `StyledLineBreak` and `TextWidth` functions; `StyledLineBreak` is described in the chapter Text Utilities in this book. Each time you call `StyledLineBreak` for a style run, first you need to set the graphics port text characteristics for that style run.

Saving the screen pixel width of each style run

To draw justified text, you need to determine the amount of extra pixels to allocate to each style run in the text line. To determine this value, you need to know the screen pixel width of each style run. You can avoid having to measure the width of each style run twice in the text-layout process by using the `textWidth` parameter of the `StyledLineBreak` function to get and save the screen pixel width of each style run. The `StyledLineBreak` function maintains the value of the `textWidth` parameter, which you initially set to the length of the display line. When you call `StyledLineBreak` for each style run in the script run, it decrements this value by the width of the style run. You can calculate the screen pixel width of a style run by subtracting the current value of `textWidth` from the display line length each time through your `StyledLineBreak` loop, and save the value to be used later. ◆

If you do not want to use `StyledLineBreak`, you can use the `TextWidth` function to measure each style run, adding the returned values until the sum exceeds the display line length. You can next use the Text Utilities `FindWordBreaks` procedure on the last style run to identify the ending location of the appropriate word in the style run, then break the text accordingly.

If a space character occurs at a line's end, and more space characters follow it in memory, `StyledLineBreak` breaks the line after the final space character in memory. This obviates the need for you to check for space characters in memory, when you lay out the next line of text. However, if you do not use `StyledLineBreak`, you need to check for space characters at the beginning of a line of text, and increment the memory pointer beyond these space characters. You can use the `CharacterType` function to identify space characters. For more information about `CharacterType`, see the chapter "Script Manager" in this book.

Listing 3-2 calculates line breaks using `StyledLineBreak`. The procedure sets the local variables, the display line length, and a value to control the line-breaking loop. Then it iterates through the text for each style run, setting the graphics port text fields for the style run and calling `StyledLineBreak` to identify where to break the line.

Listing 3-2 Calling `StyledLineBreak` to identify where to break the text line

```
PROCEDURE MyBreakTextIntoLines (window: WindowPtr);
VAR
   thetextPtr: Ptr;
   thetextLength: LongInt;
   pixelWidth: Fixed;
   textOffset: LONGINT;
   StartOfLine: Point;
   index: Integer;
   tempRect: Rect;
   lineData: myLineArray;
   lineIndex: Integer;
   theBreakCode: StyledLineBreakCode;
   theStartOffset: Integer;
BEGIN
   thetextPtr := gText.textPtr;
   index := 0;
   SetPort(window);
   tempRect := window^.portRect;
   InsetRect(tempRect, 4, 4);
   SetPt(StartOfLine, 10, 4);
   MoveTo(StartOfLine.h, StartOfLine.v);
      {Set up our local flags and variables.}
   lineIndex := 0;   {This is the index into the line data.}
```

```
index := 0;       {This is the index into the style data.}
WITH gText.runData[index] DO
BEGIN
   thetextPtr := Ptr(ORD(gText.textPtr) + runStart);
   thetextLength := gText.textLength;
   theStartOffset := 0;
END;

   {Start walking through the textblock.}
REPEAT
      {For the first style run in a line, textOffset must be non-zero.}
   textOffset := -1;
      {smBreakOverFlow means that the whole style run fits on the }
      { line with space remaining. The routine uses that condition to }
      { control the loop.}
   theBreakCode := smBreakOverflow;
      {StyledLineBreak expects the width of the display area }
      { to be expressed as a fixed value.}
   pixelWidth := BSL((tempRect.right - tempRect.left), 16);
   WITH gText DO
      BEGIN
         WHILE (theBreakCode = smBreakOverflow) DO
            BEGIN WITH gText.runData[index] DO
               BEGIN
                     {set the port}
                  TextFont(font);
                  TextFace(face);
                  TextSize(size);
                     {call StyledLineBreak to break the line}
                  theBreakCode := StyledLineBreak(thetextPtr,
                                 thetextLength, theStartOffset,
                                 runEnd, 0, pixelWidth, textOffset);
                     {now remember the information returned}
      lineData[lineIndex].textStartOffset := theStartOffset;
            {remember the beginning of this run}
      lineData[lineIndex].textLength := textOffset - theStartOffset;
            {and the length of this run length}
      lineData[lineIndex].styleIndex := index;
            {since the style information is global, just remember }
            { the index to that information}
      lineIndex := lineIndex + 1;
   END;
```

```
         {If textoffset == the end of the run, increment the }
         { rundata index and set theStartOffset to be the beginning }
         {of the next run.}
      IF (textOffset = gText.runData[index].runend) THEN
         BEGIN
            index := index + 1;
            theStartOffset := gText.runData[index].runStart;
         END
      ELSE
            {If textOffset <> the end of the run, the routine splits }
            { a run, so set theStartOffset appropriately}
         theStartOffset := theStartOffset + (textOffset - theStartOffset);
            {If there is more text, reset the offset value }
            { returned by StyledLineBreak}
         IF textOffset = thetextLength THEN LEAVE
               {if textOffset == the textLength there is no more}
               { text, so jump out of the loop}
            ELSE
               textOffset := 0;
                  {we haven't found the line break yet, }
                  { textOffset must be zero for all runs after }
                  { the first in a line}
   END; {of while loop}
END;
```

If the text is explicitly scaled, you cannot use `StyledLineBreak` to determine where to break the line. This is because the `StyledLineBreak` function does not accept scaling factors. To determine where to break a line of scaled text, you can directly call the routines that `StyledLineBreak` uses. The section “Using Scaled Text” on page 3-44 describes these steps.

Determining the Display Order for Style Runs

Now that you know where to break the line, you need to determine the display order of the style runs that constitute the line when the text contains mixed directions; if your text does not contain mixed directions, you can skip this step.

You draw style runs in their display order, which may be different from how the text is stored if it contains mixed directions. (For more information about storage order and display order, see the chapter “Introduction to Text on the Macintosh” in this book.) To determine the correct order, use `GetFormatOrder`; this procedure returns the order, from left to right, in which to draw the style runs on a line.

To use `GetFormatOrder`, you must have organized your style runs sequentially in storage order. You pass `GetFormatOrder` the numbers of the first and last style runs on the line, and the primary line direction of the text to be drawn. If you do not explicitly define the primary line direction, you can base it on the value of the `SysDirection` global variable. (The `SysDirection` global variable is set to –1 if the system direction is right to left, and 0 if the system direction is left to right.)

You pass `GetFormatOrder` a pointer to an application-defined Boolean function that calculates the correct direction for each style run and a pointer to an application-defined information block, containing font and script information, that the Boolean function uses to determine the style run direction. The `GetFormatOrder` procedure calls your Boolean function for each style run on the line.

Listing 3-3 shows an example Boolean function that calculates the line direction of a style run. Here is the type declaration for the `MyLineDrawingInfo` records, which are created as the application calculates line breaks:

```
TYPE MyLineDrawingInfo =
RECORD
   textPtr: Ptr;
   textLength: Integer;
   styleIndex: Integer:
END;

MyLineArray = ARRAY[0 ..MaxNumberofStyleRuns] OF
    MyLineDrawingInfo;
myLineDrawingInfoPtr = ^MyLineDrawingInfo;
```

The `styleIndex` field of each record of type `MyLineDrawingInfo` points to an entry in an array of style run records of type `MyStyleRun`. A style run record contains style information including font, size, style, and scaling factors. The declaration for `MyStyleRun` follows:

```
MyStyleRun =
RECORD
   runStart: Integer;
   runEnd: Integer;
   size: Integer;
   font: Integer;
   face: Style;
   numer: Point;
   denom: Point;
END;

{This sample program uses a static array to store style run }
{ information. Typically, a program would use a dynamic }
{ data structure.}
```

The MyDirectionProc function checks the font of the style run to determine if the font belongs to a right-to-left script system. If it does, the function returns TRUE; otherwise it returns FALSE. When GetFormatOrder calls MyDirectionProc, it passes an integer identifying the style run and a pointer to an application-defined parameter block. In Listing 3-3, the pointer indicates the MyLineArray array. The MyDirectionProc function uses the style run identifier and the size of a MyLineDrawingInfo record to find the right MyLineDrawingInfo record in the array. It uses the styleIndex field of the MyLineDrawingInfo record to locate the right style run record in the array of MyStyleRun records. The font field of the style run record contains a font family ID that the function uses to determine the script code. The function calls the Script Manager with the script code to determine the script direction

Listing 3-3 An application-defined run direction function called by GetFormatOrder

```
FUNCTION MyDirectionProc(theFormat: Integer;
                           myDirectionParam: Ptr):Boolean;
   VAR
      scriptCode:    Integer;
      p:             myLineDrawingInfoPtr;
      offset:        LongInt;
BEGIN
   offset := SIZEOF(MyLineDrawingInfo) * theFormat;
   p := myLineDrawingInfoPtr(ORD(myDirectionParam) + offset);
   scriptCode := FontToScript(gText.runData[p^.styleIndex].font);
   IF Boolean(GetScriptVariable(scriptCode, smScriptRight)) = TRUE
      THEN MyDirectionProc := TRUE
   ELSE
      MyDirectionProc := FALSE;
END;
```

You reference the format order array to determine the display order when you draw the text. To draw a line of text that is not justified, after you determine the display order of the style runs, you can position the pen and draw the text. (To draw a line of text that is not justified, see “Drawing the Line of Text” on page 3-42.) To draw text that is scaled, you can skip ahead to “Using Scaled Text” on page 3-44. If you are drawing a line of justified text, you must complete some additional steps before positioning the pen and actually drawing the text. These steps are described in the next three sections.

Listing 3-4 shows an application-defined procedure that declares a format order array; it passes a pointer to that array when it calls `GetFormatOrder` for a line of text containing style runs with mixed directions. Using the information that `GetFormatOrder` returns in the format order array, the procedure iterates through the style runs in display order, setting the graphics for each style run, then drawing the text.

Listing 3-4 Determining the style run display order and drawing the line

```
PROCEDURE MyGetDisplayOrderAndDrawLine (theLineData: myLineArray; VAR
                                        index: Integer);
VAR
   FormatOrderArray: ARRAY[0..kMaximumNumberOfStyleRuns] OF Integer;
   I: Integer;
BEGIN
   GetFormatOrder(FormatOrderPtr(@FormatOrderArray), 0, index,
                  Boolean(GetSysDirection), @DirectionProc, @theLineData);
      {we know the display order, now we are ready to draw}
   FOR I := FormatOrderArray[0] TO FormatOrderArray[index] DO
      BEGIN
      {Set the port.}
            WITH gText.runData[theLineData[i].styleIndex] DO
               BEGIN
                  TextFont(font);
                  TextFace(face);
                  TextSize(size);
               END;
            DrawText(theTextPtr, lineData[i].textStartOffset,
                     theLineData[i].textLength);
         END;
         index := 0;
            {we found a line, so bump the index into the line data}
   END;
```

Eliminating Trailing Spaces (for Justified Text)

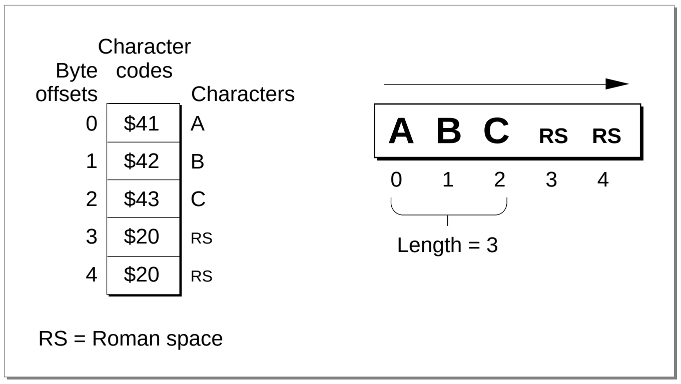
If you are justifying text, after you know the line break and display order for your text, you must determine the total amount of extra pixels that remain to be distributed throughout the line, and how to spread the extra pixels throughout the style runs. To get the correct total number of extra pixels, first you need to eliminate any trailing spaces from the last style run in memory order.

The `VisibleLength` function returns the length in bytes of the style run minus any trailing spaces; this is the byte length that you must use for the style run in any calculations necessary to determine the line layout, and for drawing the text.

The `VisibleLength` function behaves differently for various script systems. For simple script systems, such as Roman and Cyrillic, and 2-byte script systems, such as Japanese, `VisibleLength` does not include in the byte count it returns trailing spaces that occur at the display end of the text segment. For 2-byte script systems, `VisibleLength` does not count them, whether they are 1-byte or 2-byte space characters.

Figure 3-5 shows that `VisibleLength` eliminates trailing spaces at the right end of Roman text when the primary line direction is left to right. However, if you change the primary line direction, `VisibleLength` assumes the left end as the display end of the text, and does not eliminate the spaces on the right.

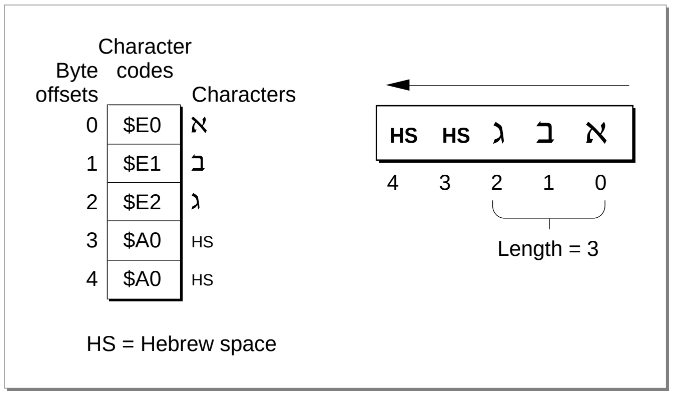
Figure 3-5 Calling `VisibleLength` for a Roman style run



For 1-byte complex script systems, `VisibleLength` does not include in the byte count that it returns spaces whose character direction is the same as the primary line direction. The primary line direction is determined by the `SysDirection` global variable. By default, the value of `SysDirection` is the direction of the system script. You can change this value. Some word processors that allow users to change the primary line direction, for example, to create a document entirely in English when the system script is Hebrew, change the value of `SysDirection`. If you modify the `SysDirection` global value, be sure to first save the original value and restore it before your program terminates.

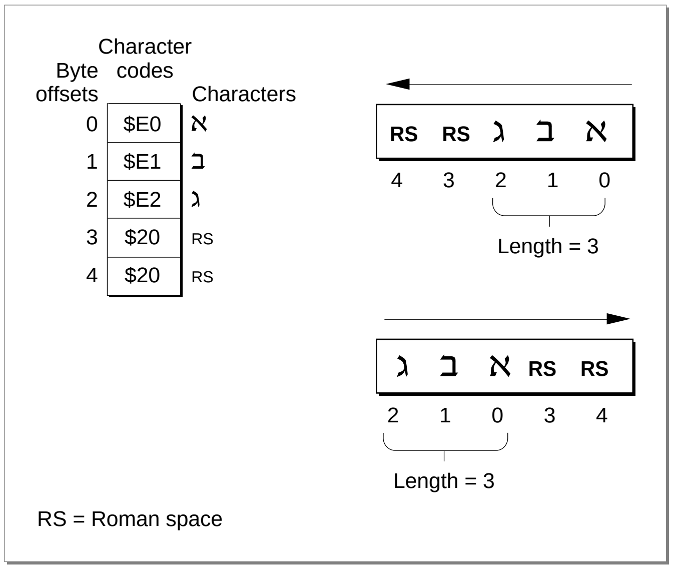
Figure 3-6 shows a Hebrew style run with trailing spaces at the left end, which `VisibleLength` eliminates. In this case, the primary line direction is right to left; if it were left to right, `VisibleLength` would return 5.

Figure 3-6 Calling `VisibleLength` for a Hebrew style run



For those 1-byte complex script systems that support bidirectional text, Roman spaces take on a character direction based on the primary line direction. If the Roman spaces then fall at the end of the text, `VisibleLength` does not include them in the returned byte count. Figure 3-7 shows Roman spaces following Hebrew text in storage order. In the first part of the example, the Roman spaces take on the primary line direction of right to left, and follow the Hebrew text in display order also. Because they fall at the end of the display line, `VisibleLength` does not count them. The second part of the example shows what happens when the primary line direction is changed to left to right: the Roman spaces fall at the end of the line again and are not counted.

Figure 3-7 Calling `VisibleLength` for Hebrew text with Roman space characters



Calculating the Slop Value (for Justified Text)

To draw justified text, after you eliminate trailing spaces from the line's last style run in memory order, you need to determine the amount of remaining extra space to be distributed throughout the line. This is called the *slop value*; it is measured in pixels, and is the difference between the width of the text and the width of the display line. After you add the widths of all the style runs on the line to get the total width of the text, you can use the following statement to determine the slop value. Be sure to use the value returned by `VisibleLength` for the last style run in memory order when you add the style run widths.

```
TotalSlop := DisplayLineLength – SumofStyleRunWidths
```

If you saved the screen pixel width of each style run when you called `StyledLineBreak`, then all you need to do is sum the total of the style run widths and subtract that total from the display line length to get the total slop value.

If you did not use `StyledLineBreak` or you did not save the screen pixel width of each style run when you called `StyledLineBreak`, first measure the width of each style run, using `TextWidth`, then add the widths to get the total. (Remember that each time you measure a style run, first you need to set the text-related graphics port fields for that style run.)

The `TextWidth` function returns the width in pixels of a style run. You pass `TextWidth` the number of bytes of the text to be measured. For script systems containing 2-byte characters, be certain that you pass the correct number of bytes; 2-byte script systems can contain a mix of one-byte and two-byte characters.

For scaled text, you cannot use the `TextWidth` function to get the screen pixel width of a style run. Instead, you can call the `StdTxMeas` function, which accepts and returns scaling parameters. Whenever you call `StdTxMeas` directly, first you must check the graphics port `grafProc` field to determine if the bottleneck routines have been customized, and if so, use the customized version. See "Low-Level QuickDraw Text Routines" on page 3-98 for more information.

Using a negative slop value
You can pass the justification routines a positive or negative slop value. Word processing programs can use a negative slop value to justify a line of text more smoothly by condensing it, when it only slightly exceeds the display line length. ◆

Allocating the Slop to Each Style Run (for Justified Text)

Once you have assessed the total amount of slop to be distributed throughout the line of text, you need to determine the portion to apply to each style run. When you draw a style run that is part of a line of justified text, you pass this number as the value of the slop parameter.

To determine the actual number of pixels for each style run, first you determine the percentage of slop to attribute to the style run, and then apply that percentage to the total slop to get the number of pixels. To get the percentage of slop for a style run, you compute what percentage each portion is of the sum of all portions.

The following steps summarize this process:

1. Call `PortionLine` for each style run on the line. The `PortionLine` function returns a "magic number" which is the correct proportion of extra space to apply to a style run.
2. Add the returned values together.
3. For each style run, divide the value returned by `PortionLine` for that style run by the sum of the values returned for all of the style runs on the line.
4. For each style run, multiply the result of step 3 by the total slop value for the line.

For example, suppose that there are three style runs on a line: style run A, style run B, and style run C. The total slop = 11; the line needs to be widened by 11 pixels to be justified. If you call `PortionLine` for each of the style runs, it produces the following results:

Style run	Value returned by `PortionLine`
A	5.4
B	7.3
C	8.2

Summing the three values together produces a total of 20.9. Now you need to convert the values into percentages by dividing each by the total. This produces the following results:

Style run	Proportion	Percentage of total
A	5.4/20.9	25.84%
B	7.3/20.9	34.93%
C	8.2/20.9	39.23%

The final step is to multiply the total slop value—11 pixels—by each percentage and round off to compute the actual number of pixels (slop) allocated to each style run. (To correct for the roundoff error, add the remainder to the pixels for the final style run.) This produces these results:

Style run	Amount of slop (in pixels)
A	3
B	4
C	4

Listing 3-5 provides a code fragment that illustrates how you can use the `PortionLine` function to do this. The application-defined `MyCalcJustAmount` routine expects an array of the following type of records.

```
Type RunRecord =
      Record
         tPtr:          Ptr;              {ptr to the text}
         tLength:       LongInt;          {length of run}
         tFace:         style;            {txFace of run}
         tFont:         Integer;          {font family ID}
         tSize:         Integer;          {pt size}
         tPlaceOnLine:  JustStyleCode;
         tnumer,tdenom: Point;            {scaling factors}
         tJustAmount:   Fixed;            {this value }
                                          { calculated here}
      END;
RunArray = ARRAY[1..MaxRuns] OF RunRecord;
```

The `MyCalcJustAmount` routine also takes as a parameter a count of the total number of records that the array contains. Finally, the extra screen pixel width to be distributed is passed in as the `TotalPixelSlop` parameter. The routine calculates the amount of slop to be allocated to each run, and assigns that value to the field `tJustAmount`.

Listing 3-5 Distributing slop value among style runs

```
PROCEDURE MyCalcJustAmount(rArray: RunArray; NRuns: Integer;
                           TotalPixelSlop: Integer);
VAR
   I:                   Integer;
   TotalSlopProportion: Fixed;
   PixelSlopRemainder:  Fixed;

BEGIN
{Find the proportion for each run, temporarily storing}
{ it in the tJustAmount field, and sum the }
{ returned values in TotalSlopProportion.}
   TotalSlopProportion := 0;
   FOR I := 1 TO NRuns DO
      WITH rArray[I] DO BEGIN
         {Set the graphics port’s fields for each style run}
         { to this style run}
         TextFace(tFace);
         TextFont(tFont);
         TextSize(tSize);
         tJustAmount :=
PortionLine(tPtr,tLength,tstyleRunPosition,tnumer,tdenom);
         TotalSlopProportion := TotalSlopProportion + tJustAmount;
         END;
{ Normalize the slop to be allocated to each run }
{ ( runportion / totalportion), and then convert that value to }
{ UnRounded Pixels: }
{ (runportion / totalportion) * TotalPixelSlop ).}
   PixelSlopRemainder := Fixed(TotalPixelSlop);
      IF NRuns > 1 THEN
         FOR I := 1 TO NRuns-1 DO
            WITH rArray[I] DO BEGIN
               {Use the FixRound routine to round this value.}
               tJustAmount := FixMul(FixDiv(tJustAmount,
                           TotalSlopProportion),TotalPixelSlop);
               PixelSlopRemainder := PixelSlopRemainder -
                                       tJustAmount;
               END;
      rArray[NRuns].tJustAmount := PixelSlopRemainder;
END;
```

Drawing the Line of Text

Once you have laid out a line of text, drawing it is fairly simple. Your application's text-drawing routine needs to loop through the text, following these steps:

1. To position the pen correctly at the beginning of a new line, set the `pnLoc` graphics port field to the local coordinates representing the point where you want to begin drawing the text. You use the QuickDraw `MoveTo` or `Move` procedure to reposition the pen. (For more information about `Move` or `MoveTo`, see the QuickDraw chapters in *Inside Macintosh: Imaging*.) Within a line of text, after you draw a text segment, QuickDraw increments the pen location for you and positions the pen appropriately for the next text segment.
2. Before you draw each text run, set the text-related fields of the current graphics port to the text characteristics for that style run, if the text segment begins a new style run.
3. Draw the text segment.
   - If your text is not justified, use `DrawText` or `StdText` to draw it. The `StdText` procedure also allows you to draw scaled text that is not justified. If you use `StdText`, first you must determine whether the standard routine has been customized. If so, you must use the customized version. For more information, see "Low-Level QuickDraw Text Routines" on page 3-98.

Listing 3-4 on page 3-36 shows how to draw the text using `DrawText` after determining the display order.

- □ If your text is justified, use the `DrawJustified` procedure to draw it. This procedure takes a parameter, `styleRunPosition`, that identifies the location of the style run in the line of text. You must specify the same value for this parameter that you used for it when you called the `PortionLine` function for this style run. The `DrawJustified` and `PortionLine` routines also take `numer` and `denom` parameters for scaling factors. For unscaled text, specify values of 1, 1 for both of these parameters.

4. After you draw each text segment, increment the pointer in memory to the beginning of the next text segment to be drawn.

To position the pen horizontally, remember that QuickDraw always draws text from left to right:

- ■ For left-aligned text, position the pen at the left margin of the display line.
- ■ For right-aligned text, indent the pen from the left margin by the difference between the display line length and the total width of all the style runs. If you have set a `CharExtra` value, after you sum the total width of all the style runs, subtract the value that you passed to `CharExtra` from the total so that the rightmost character will be flush against the right margin.
- ■ For justified text, set the pen at the left margin.

To determine the vertical coordinate of the pen position when you draw lines of text rendered in varying fonts and styles, you need to assess the required line height for each new style. You base this on the style run that requires the greatest number of vertical pixels. You can use the `GetFontInfo` procedure, which fills a record with information describing the current font's ascent, descent, and the width measurements of the largest glyph in the font, and leading. You can determine the line height by adding the values of these fields. For outline fonts, you can use the `OutlineMetrics` function to get the font measurements. For more information about `OutlineMetrics`, see the chapter "Font Manager" in this book. Listing 3-6 shows how to call `GetFontInfo`, and use the information it returns in the font information record to determine the line height.

Listing 3-6 Calling `GetFontInfo` to determine the line height

```
FUNCTION MyGetMaximumLineHeight (VAR mylineData: myLineArray;
                                     lastStyleIndex: Integer): Integer;
   VAR
      info: fontInfo;
      I: Integer;
      ignore: Integer;
      maxHeight: Integer;
   BEGIN
      maxHeight := 0;
      FOR i := 0 TO lastStyleIndex DO
         WITH gText.rundata[mylineData[i].styleIndex] DO
      BEGIN
         {set the grafport up}
         TextFont(font);
         TextFace(face);
         TextSize(size);
         {Get the vertical metrics}
         GetFontInfo(info);
         {If this style run is taller than any others measured, }
         { remember the height.}
            WITH info DO
               IF (ascent + leading > maxHeight) THEN
                  maxHeight := ascent + leading;
      END;
   MyGetMaximumLineHeight := maxHeight;
END;
```

Using Scaled Text

This section describes how to determine where to break a line of scaled text. Then it describes how to draw scaled text, whether aligned or justified.

You cannot call `StyledLineBreak` for scaled text. To determine where to break a line of scaled text, you can directly call the routines that `StyledLineBreak` uses. The `StyledLinedBreak` function uses the `PixelToChar` function to locate the byte offset that corresponds to the pixel location marking the end of the display line.

The primary use of `PixelToChar` is to locate a caret position associated with a mouse-down event. For this purpose, the `PixelToChar` function reorders the text when the text belongs to a right-to-left script system; this ensures that `PixelToChar` returns the correct byte offset associated with the pixel location of a mouse-down event.

If right-to-left text is reordered when you use `PixelToChar` to determine where to break a line, it returns the wrong byte offset. For right-to-left text, the end of a line in memory order can occur either at the left end of a display line or in the middle of one. To get the correct result, `StyledLineBreak` turns off reordering before it calls `PixelToChar`. Your application must also do this.

You can define a routine that turns off reordering if the font's script system is right to left, and call your routine just before you call `PixelToChar`. Remember to restore reordering after you have determined where to break the line.

Listing 3-7 shows an application-defined routine that turns off reordering of text in a right-to-left script system. It tests to determine whether the reordering bit is on or off so that the application can restore it to its current state, then it clears the reordering bit (`smsfReverse`), and sets the script flag with the `SetScriptVariable` function. See the chapter "Script Manager" in this book for more information.

Listing 3-7 Turning off reordering of right-to-left text before calling `PixelToChar` for line-breaking

```
FUNCTION MySetReordering(font: integer): Boolean
   VAR
      flags: LongInt;
      err:  OSErr;
BEGIN
   flags := GetScriptVariable(smCurrentScript, smScriptFlags);
   MySetReordering := BTST(flags, smsfReverse);
   BCLR(flags, smsfReverse);
   err := SetScriptVariable(smCurrentScript, smScriptFlags,
                                 flags);
END;
```

Here are the steps you take to determine where to break a line of scaled text:

1. Call `PixelToChar` to determine the byte offset that corresponds to the pixel location where you want to break the line. You pass the pixel location of the end of the display line to `PixelToChar` as the value of the `pixelWidth` parameter. The `PixelToChar` function returns the byte offset corresponding to the pixel location of the end of the display line, if the corresponding byte offset falls with the style run that you call `PixelToChar` for.

   If the byte offset corresponding to this pixel location does not fall within the style run, on return the `widthRemaining` parameter contains the number of pixels from the right edge of the text string for which you called `PixelToChar` to the end of the display line. You can loop through your text, calling `PixelToChar` for each style run until you encounter the byte offset that corresponds to the pixel location of the end of the display line.

2. Call the Text Utilities `FindWordBreaks` procedure with an `nbreaks` parameter of –1 to determine the boundaries of the word containing the byte offset that corresponds to the pixel location of the end of the display line. If the byte offset that `PixelToChar` returns is the beginning boundary or interior to the word, you should break the text before this word, or after the preceding word.

3. If the byte offset that falls at the end of the display line is a space character, you should check to determine if there are succeeding space characters in memory; `StyledLineBreak` does this. You can use the Script Manager’s `CharType` function for this purpose. If there are additional space characters, increment the text pointer beyond them in memory to determine the starting offset for the next line of text.

The steps that you follow to draw scaled text are the same as those for unscaled text, described under "Drawing the Line of Text" on page 3-42. However, you perform some steps differently for scaled text. The steps are summarized here, and the differences are elaborated.

1. Set the `pnLoc` graphics port field to the local coordinates representing the point to begin drawing the text. You use `StdTxMeas` to get the font metrics for scaled text in order to determine the line height, instead of using `GetFontInfo`, which doesn't support scaling. Using the information that `StdTxMeas` returns, you can scale the vertical metrics. Listing 3-8 shows one way to do this.
2. Before you draw each style run, set the text-related fields of the current graphics port to the text characteristics for that style run. This step is the same as for drawing unscaled text.
3. Use `DrawJustified` or `StdText` to draw the scaled text a style run at a time. To draw scaled text that is not justified, you call `StdText` or you can call `DrawJustified` and pass in `onlyStyleRun` for the `styleRunPosition` parameter.

Listing 3-8 shows how to measure scaled text using `StdTxMeas`, and use the information returned in the font information (`fontInfo`) record to determine the line height. The `gText` global variable is initialized before the routine is called.

The application of which this routine is a part stores style runs in a text block, which is defined by the `TextBlock` data type.

```
TextBlock = RECORD
   textPtr: Ptr;
   textLength: Integer;
   runData: StyleRunArray;
END;
```

Listing 3-8 Using `StdTxMeas` to get the font metrics for determining the line height of scaled text

```
FUNCTION MyGetMaximumLineHeight(VAR lineData: LineArray;
                        lastStyleIndex: Integer): Integer;
   VAR
   info: fontInfo;
   i: Integer;
   ignore:Integer;
   MaxHeight:Integer;
   localNumer, localDenom: Point;
   size, font: Integer;
   face: Style;
```

```
BEGIN
   MaxHeight := 0;
   FOR i := 0 TO lastStyleIndex DO
    WITH gText.runData[lineData[i].styleIndex] DO BEGIN
   {Set up the graphics port}
      TextFont(font);
      TextFace(face);
      TextSize(size);
   {measure the text}
      localNumer := numer;
      localDenom := denom;
      ignore := StdTxMeas(lineData[i].textlength,
       lineData[i].textPtr, localNumer, localDenom, info);
   {scale the vertical metrics based on the StdTxMeas }
         { returned values of numer and denom}
      info.ascent := FixRound(FixMul(BSL(info.ascent, 16),
               FixRatio(localNumer.v, localDenom.v)));
      info.leading := FixRound(FixMul(BSL(info.leading, 16),
               FixRatio(localNumer.v, localDenom.v)));
      WITH info DO
         IF (ascent + leading > maxHeight) THEN
            maxHeight := ascent + leading;
   END;
```

Drawing Carets and Highlighting

This section discusses how to determine a caret position to be used to mark an insertion point or endpoint for highlighting a range of text. This section describes how to

- determine the byte offset of a character whose glyph is closest to an onscreen pixel location where a mouse-down event occurred
- determine a caret position from a byte offset, and draw a caret to mark an insertion point
- locate the endpoints for a selection range in order to highlight it, using the byte offsets at characters that begin and end the segment of text to be highlighted

For a discussion of the conventions underlying the relationship of a character at a byte offset to a caret position for unidirectional text and text at a direction boundary, see the treatment of caret handling and highlighting in the chapter “Introduction to Text on the Macintosh” in this book.

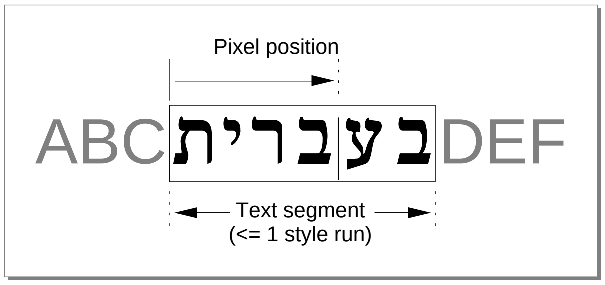
Generally, an application draws and blinks the caret in an active document window from its idle-processing procedure in response to a null event. If your application uses TextEdit, you can call the `TEIdle` procedure to do this. If your application does not use TextEdit, you are responsible for drawing and blinking the caret.

You should check the keyboard script and change the onscreen pixel location where you draw the caret, if necessary, to synchronize the caret with the keyboard script. The caret marks the insertion point where the next character is to be entered, and when the user changes the keyboard script, the caret location can change. (For more information, see "Synchronizing the Caret With the Keyboard Script" on page 3-59.)

You call `PixelToChar` from within a loop that iterates through the style runs on a line of text until you locate the byte offset of the character associated with the input pixel location. Once you have the byte offset, you call `CharToPixel` to get the pixel location of the caret position. If you already have the byte offset, you do not need to call `PixelToChar`. The `CharToPixel` function returns the length in pixels from the left edge of the text segment to the caret position corresponding to that character. (The text segment that you pass to `CharToPixel` can be a complete style run or the portion of a style run that fits on the line.)

Once you have the pixel location of the caret position within the context of the text segment, you must convert it to a pixel location relative to the entire display line's left margin. To get the correct display line pixel location, you lay out the line of text, measuring the screen pixel width of each style run from left to right up to the text segment that contains the caret position, then add the screen pixel width of the caret position to the sum of all the preceding style runs. Once you have the pixel location relative to the display line's left margin, you can draw the caret. Figure 3-8 shows Hebrew text between two runs of English text on a line. `CharToPixel` and `PixelToChar` recognize the pixel location in the Hebrew text relative to the left edge of the Hebrew style run, although the left margin of the display line begins with the English text.

Figure 3-8 What pixel position means for `CharToPixel` and `PixelToChar`



Converting an Onscreen Pixel Location to a Byte Offset

You need to find the byte offset and the text direction of the character that corresponds to a glyph onscreen in order to display the caret correctly. You need this information to mark an insertion point with a caret, select words, determine the endpoints for highlighting a range of text, and determine where to break a line of text. You can use the `PixelToChar` function to get this information.

The `PixelToChar` function returns a byte offset and a Boolean value. The Boolean flag tells you whether the input pixel location is on the leading edge or the trailing edge of the glyph.

- When the input pixel location is on the leading edge of the glyph, `PixelToChar` returns the byte offset of that glyph's character and a `leadingEdge` flag of `TRUE`. (If the glyph represents multiple characters, it returns the byte offset of the first of these characters in memory.)
- When the input pixel location is on the trailing edge of the glyph, `PixelToChar` returns the byte offset of the first character in memory *following* the character or characters represented by the glyph, and a `leadingEdge` flag of `FALSE`.
- When the input pixel location is *before* the leading edge of the first glyph in the displayed text segment, `PixelToChar` returns the byte offset of the first character in the text segment and a `leadingEdge` flag of `FALSE`.
- When the input pixel location is *after* the trailing edge of the last glyph in the displayed text segment, `PixelToChar` returns the next byte offset in memory, the one after the last character in the text segment, and a `leadingEdge` flag of `TRUE`.

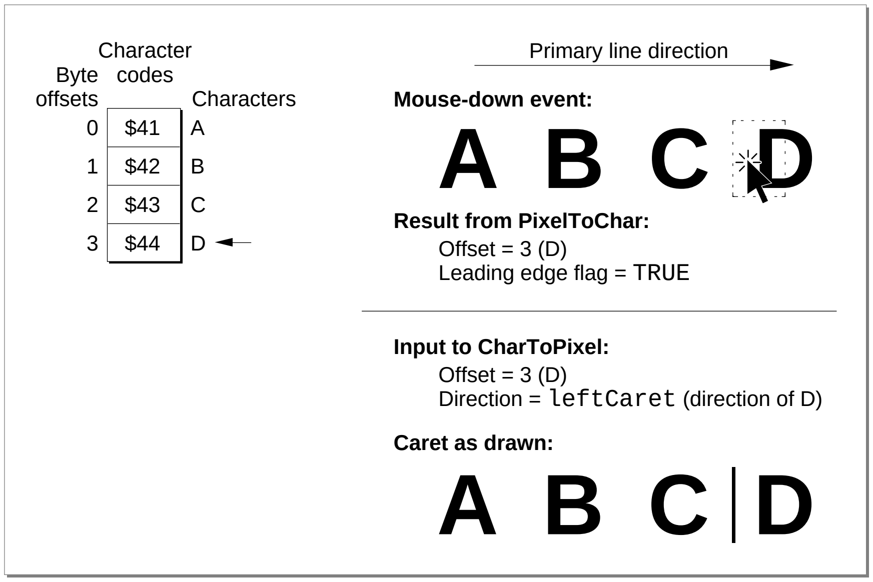
If the primary line direction is left to right, *before* means to the left of all the glyphs for the characters in the text segment, and *after* means to the right of all these glyphs. If the primary line direction is right to left, before and after hold the opposite meanings.

Finding a Caret Position and Drawing a Caret

Once you have a byte offset, you need to convert it to a caret position. The `PixelToChar` and `CharToPixel` functions work together to help you determine a caret position. You use the byte offset that `PixelToChar` returns as input to `CharToPixel`. The `CharToPixel` function requires a `direction` parameter to determine whether to place the caret for text with a left-to-right or right-to-left direction. You base the value of the `direction` parameter on the `leadingEdge` flag that `PixelToChar` returns.

When a mouse-down event in text occurs, if `PixelToChar` returns a `leadingEdge` flag of `TRUE`, you pass `CharToPixel` the text direction of the character whose byte offset `PixelToChar` returns. Figure 3-9 illustrates a simple case. The user clicks on the leading edge of the glyph of character D; `PixelToChar` returns byte offset 3 and a `leadingEdge` flag of `TRUE`. You then call `CharToPixel`, passing it byte offset 3 and a `direction` parameter of `leftCaret`, based on the text direction of the character D. The `CharToPixel` function returns the pixel location equivalent to the caret position; now you can draw the caret as shown, on the leading edge of D.

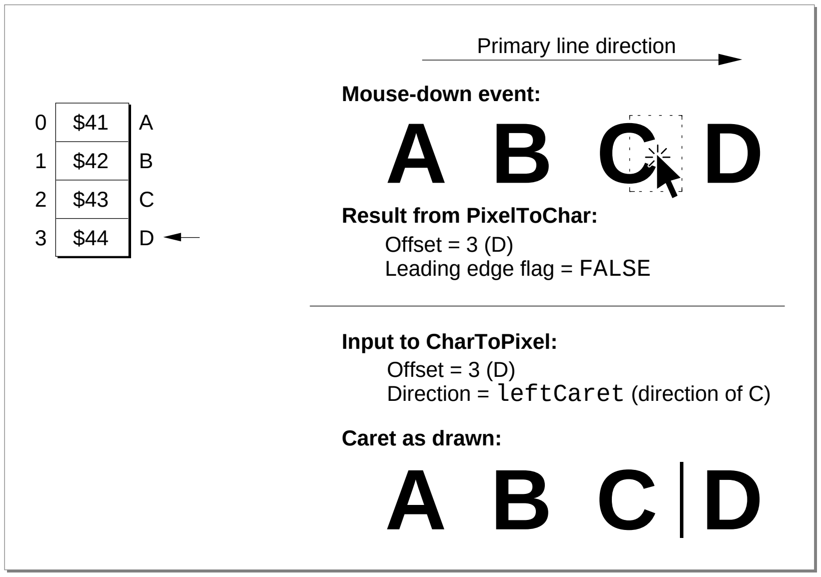
Figure 3-9 Caret position for a leading-edge mouse-down event



If `PixelToChar` returns a `leadingEdge` flag of `FALSE`, it returns the *next* byte offset in memory, not the one on whose trailing edge the mouse-down event occurred. You still base the value of the `direction` parameter on the character of the glyph at whose trailing edge the mouse-down event occurred, but this character is the one in memory that is *before* the byte offset that `PixelToChar` returned.

Figure 3-10 illustrates this for the same simple case. The user clicks on the trailing edge of the glyph of character C; `PixelToChar` returns byte offset 3, the byte offset of the next character (D) in memory, and a `leadingEdge` flag of `FALSE`. You then call `CharToPixel`, passing it byte offset 3 and a direction parameter of `leftCaret`, based on the text direction of the character C. The `CharToPixel` function returns the pixel location equivalent to the caret position; now the application can draw the caret as shown, on the trailing edge of C, which is the same position as the leading edge of D.

Figure 3-10 Caret position for a trailing-edge mouse-down event



When a character falls on a direction boundary, the case is more complicated. In display order, a direction boundary can occur on the trailing edges of two glyphs, the leading edges of two glyphs, or at the beginning or end of a text segment. The same rules apply for calling `PixelToChar` and `CharToPixel`, but the results can be different.

Figure 3-11 shows what happens when the user clicks on the leading edge of the glyph †, whose character falls on a direction boundary; `PixelToChar` returns a `leadingEdge` flag of `TRUE` and a byte offset of 3. You pass this byte offset and a direction of `rightCaret`, the text direction for *Hebrew,* to `CharToPixel`. The `CharToPixel` function returns the caret position on the leading edge of †, and you draw the caret there.

Figure 3-11 Caret position for a leading-edge mouse-down event at a direction boundary

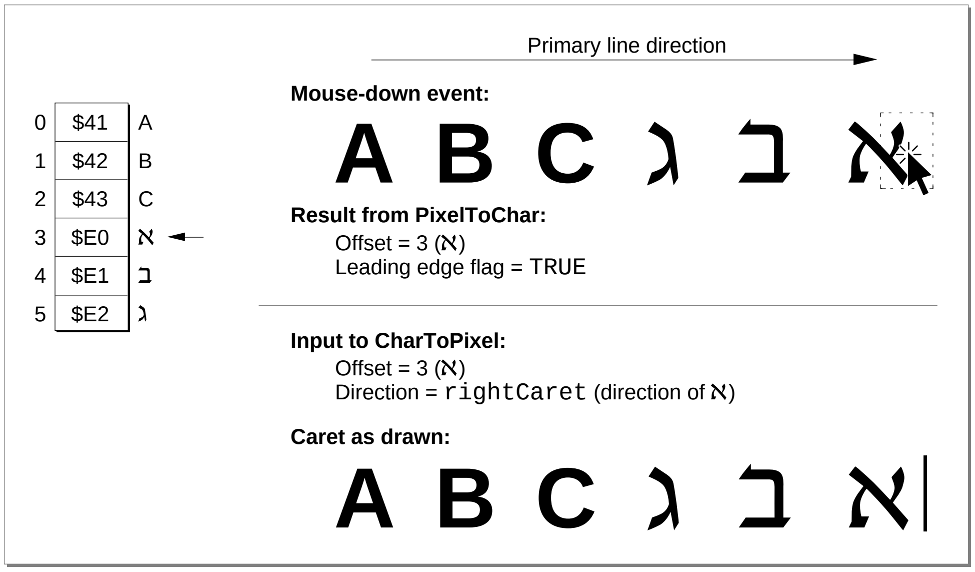
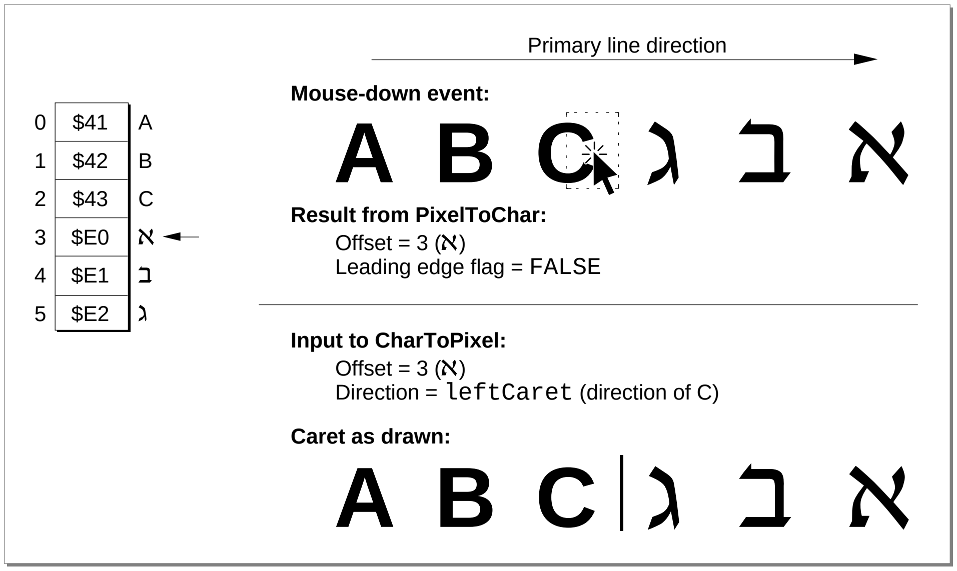


Figure 3-12 shows what happens when the user clicks on the trailing edge of the glyph C (byte offset 2). The `PixelToChar` function returns byte offset 3 (the Hebrew character †) and a `leadingEdge` flag of `FALSE`. You pass this byte offset and a `direction` parameter of `leftCaret`, the text direction for *English,* to `CharToPixel`. In this case, `CharToPixel` returns a caret position on the trailing edge of C, which is where you draw the caret.

Figure 3-12 Caret position for a trailing-edge mouse-down event at a direction boundary

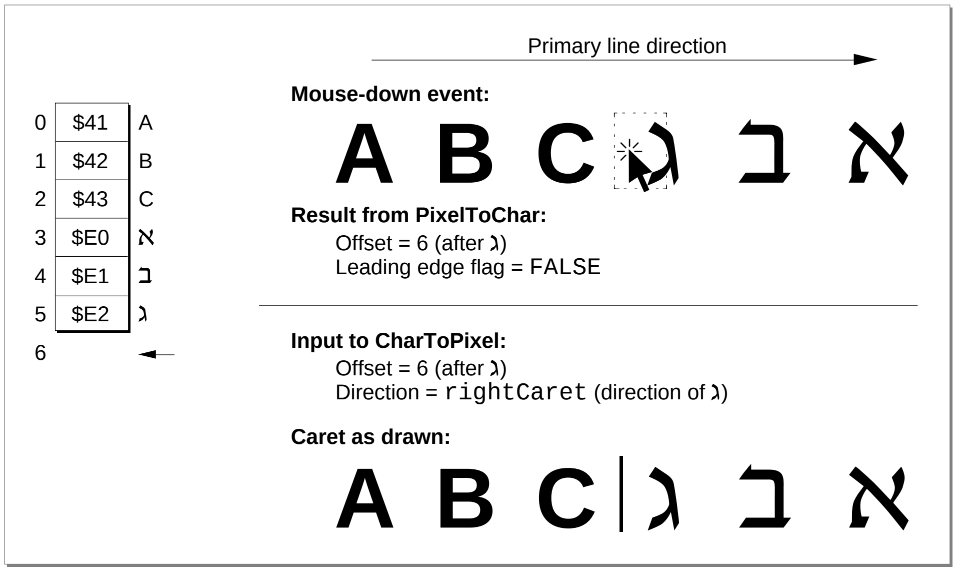


Using a dual caret

If your application is configured to use a dual caret, you must call `CharToPixel` twice to draw the caret. For example, in Figure 3-12, you would call it once with a `leftCaret` direction and again with a `rightCaret` direction, both times for byte offset 3. You always draw the high (primary) caret at the caret position obtained when the direction parameter equals the primary line direction. For more information, see the discussion of caret positions at direction boundaries in the chapter “Introduction to Text on the Macintosh” in this book. ◆

Figure 3-11 and Figure 3-12 show how one offset can yield two caret positions. Figure 3-13, when compared with Figure 3-12, shows how two offsets can yield one caret position. In Figure 3-13, the user clicks on the trailing edge of the glyph ¶. The `PixelToChar` function returns byte offset 6 and a `leadingEdge` flag of `FALSE`. (Although there is no character code associated with byte offset 6, it is the memory position of the next character to be entered.) You then call `CharToPixel`, passing it byte offset 6 and a `direction` parameter of `rightCaret`, the text direction for Hebrew. The `CharToPixel` function returns the pixel location on the trailing edge of the glyph ¶.

Figure 3-13 Caret position for a trailing-edge mouse-down event at a direction boundary

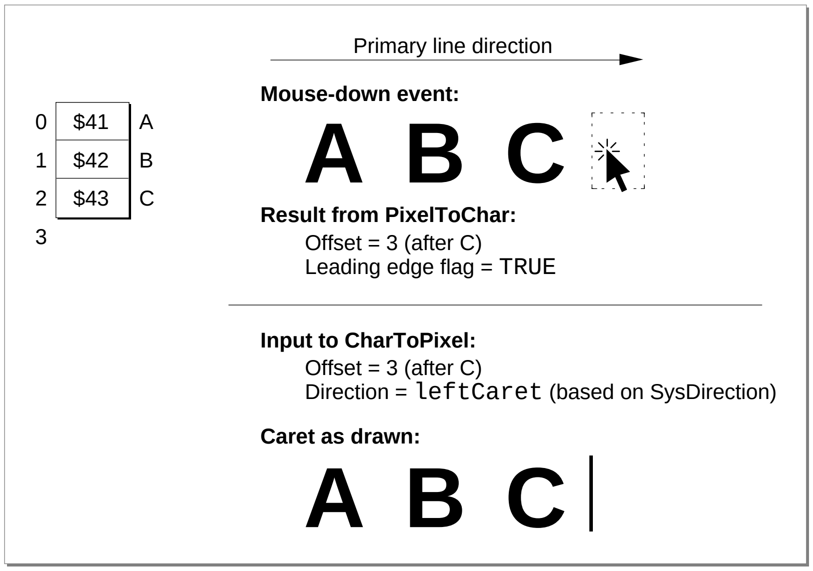


There is one additional complication that occurs at the ends of a text segment that is the only style run on a line, and at the outer end of a text segment that is the rightmost or leftmost style run on a line. Again, the rules for calling `PixelToChar` and `CharToPixel` are the same. Here is how they are interpreted for these cases. If a user clicks the mouse *before* the text segment that is at the beginning of a line, `PixelToChar` returns a leading edge value of `FALSE` and a byte offset of 0. (The first character of a text segment that you pass to `PixelToChar` is always at byte offset 0.)

If a user clicks the mouse *after* the text segment that is at the end of a line, `PixelToChar` returns a leading edge value of `TRUE` and the next byte offset in memory, following the last character in the text segment.

Figure 3-14 shows what happens when a mouse-down event occurs beyond the last glyph of the text segment. The `PixelToChar` function returns byte offset 3 and a `leadingEdge` flag of `TRUE`. You pass this byte offset and a `direction` parameter of `leftCaret` to `CharToPixel`. In this case, the direction parameter is based on the value of `SysDirection` because there isn't a character in memory associated with byte offset 3. The `CharToPixel` function returns a caret position on the trailing edge of C, which also marks the insertion point of the next character to be entered. This is where you draw the caret.

Figure 3-14 Caret position for a mouse-down event beyond the last glyph of the text segment



Listing 3-9 is a sample routine that converts mouse clicks to caret positions for drawing the caret or for highlighting a selection range. It determines a text offset (`charLoc`) from a mouse-down position and turns it into caret positions or ends of highlighting rectangles (`leftSide`, `rightSide`). It tracks the mouse and dynamically draws highlighting as the cursor is moved across the text. The routine calls `HiliteText` to determine selection ranges. It calls `CharacterType` to determine the primary and secondary caret positions for mixed-directional text. It draws the caret or highlighting rectangles by calling the application routine `MyAddSelectionArea`.

Listing 3-9 Drawing the caret and highlighting a selection range

```
PROCEDURE MyDoTextClick(w: WindowPtr; where: POINT;
                        cmdKeyIsDown, shiftKeyIsDown,
                        optionKeyIsDown: BOOLEAN);
VAR
   txLineH:             TextLineHandle;
   horizontalPosition:  FIXED;
   leadingEdge:         BOOLEAN;
   widthRemaining:      FIXED;
   charLoc:             INTEGER;
   selectionOffsets:    OffsetTable;
   c:                   INTEGER;
   leftSide, rightSide: INTEGER;
```

```
      prevMouseLoc:        POINT;
      direction:           INTEGER;
BEGIN
   txLineH := TextLineHandle(GetWRefCon(w));    {get the text}
   IF txLineH <> NIL THEN BEGIN
      LockHandleHigh(txLineH);

      WITH txLineH^^ DO BEGIN
         {initialize character offsets to invalid values}
         IF NOT shiftKeyIsDown THEN
            leftOffset := -1;
         rightOffset := -1;

         {initialize mouse position to invalid values}
         SetPt(prevMouseLoc, kMaxInteger, kMaxInteger);
         {track mouse and display text selection or caret }
         REPEAT
            IF DeltaPoint(where, prevMouseLoc) <> 0 THEN BEGIN
               {mouse has moved:}
               prevMouseLoc := where;

               {adjust mouse position relative to lineStart, }
               { convert mouse position's INTEGER to FIXED, }
               { assume style run position doesn’t matter, }
               { assume no scaling (1: 1 ratio)}
               charLoc := PixelToChar(@textBuffer, textLength, 0,
                           BitShift(where.h - lineStart.h, 16),
                           leadingEdge, widthRemaining,
                           smOnlyStyleRun,
                           POINT(kOneToOneScaling),
                           POINT(kOneToOneScaling));

               IF charLoc <> rightOffset THEN BEGIN
                  {character location has changed:}
                  IF leftOffset = -1 THEN
                     {anchor position hasn't been set yet:}
                     leftOffset := charLoc;  {set anchor position}
                  rightOffset := charLoc;    {save new caret pos.}

                  {erase previous selection; note that it }
                  {would be more optimal to erase only the }
                  { difference between old and new selection}
                  MyDeleteSelectionAreas(w, txLineH);
```

```
{now get the selection ranges to highlight}
HiliteText(@textBuffer, textLength, leftOffset,
            rightOffset, selectionOffsets);

{check whether a range of text is selected, }
{ or if it’s only an insertion point}
IF selectionOffsets[0].offFirst <>
selectionOffsets[0].offSecond THEN BEGIN
   {it’s a selection range:}
   c := 0;        {offsetPairs are zero-based}
   REPEAT
      leftSide := CharToPixel(@textBuffer,
               textLength, 0,
               selectionOffsets[c].offFirst,
               smHilite, smOnlyStyleRun,
               POINT(kOneToOneScaling),
               POINT(kOneToOneScaling));
      rightSide := CharToPixel(@textBuffer,
               textLength, 0,
               selectionOffsets[c].offSecond,
               smHilite, smOnlyStyleRun,
               POINT(kOneToOneScaling),
               POINT(kOneToOneScaling));

      {put rectangle ends in right order}
      IF rightSide < leftSide THEN
         SwapIntegers(leftSide, rightSide);

      {now draw the rectangle}
      MyAddSelectionArea(txLineH, leftSide,
                     lineStart.v - caretHeight,
                     rightSide,lineStart.v,
                     TRUE);
      c := c + 1;
   UNTIL (selectionOffsets[c].offFirst =
   selectionOffsets[c].offSecond) OR (c = 3);
END
ELSE BEGIN
   {it’s a caret position, not a range:}
   { calculate caret and draw it}

   {position of caret depends on character's }
   { direction; call CharacterType to find it}
```

```
IF BAND(CharacterType(@textBuffer,
selectionOffsets[0].offFirst),
smCharRight) <> 0 THEN
   direction := smRightCaret
ELSE
   direction := smLeftCaret;
leftSide := CharToPixel(@textBuffer,
               textLength, 0,
               selectionOffsets[0].offFirst,
               direction, smOnlyStyleRun,
               POINT(kOneToOneScaling),
               POINT(kOneToOneScaling));
{if user has specified dual caret, call }
{ CharToPixel again with the opposite }
{ value for the direction parameter}
IF documentSettings.useDualCaret THEN BEGIN
   IF direction = smRightCaret THEN
      direction := smLeftCaret
   ELSE
      direction := smRightCaret;
   rightSide := CharToPixel(@textBuffer,
               textLength, 0,
               selectionOffsets[0].offFirst,
               direction, smOnlyStyleRun,
               POINT(kOneToOneScaling),
               POINT(kOneToOneScaling));
END
ELSE
   rightSide := leftSide;

IF leftSide = rightSide THEN
   {it’s only a single caret:}
   MyAddSelectionArea(txLineH, leftSide,
               lineStart.v - caretHeight,
               leftSide + kCaretWidth,
               lineStart.v, TRUE)
ELSE BEGIN
   {it’s a split-caret: assume upper caret }
   { is left-to-right text, lower caret is
   { right-to-left text}
   IF direction = smRightCaret THEN BEGIN
      {rightSide is right-to-left: }
      { use upper caret for leftSide}
```

```
                           MyAddSelectionArea(txLineH, leftSide,
                                       lineStart.v - caretHeight,
                                       leftSide + kCaretWidth,
                                       lineStart.v -
                                       (caretHeight DIV 2), TRUE);
                           MyAddSelectionArea(txLineH, rightSide,
                                       lineStart.v -
                                       (caretHeight DIV 2),
                                       rightSide + kCaretWidth,
                                       lineStart.v, TRUE);
                        END
                        ELSE BEGIN
                           {rightSide is left-to-right: }
                           { use lower caret for leftSide}
                           MyAddSelectionArea(txLineH, rightSide,
                                       lineStart.v - caretHeight,
                                       rightSide + kCaretWidth,
                                       lineStart.v -
                                       (caretHeight DIV 2), TRUE);

                           MyAddSelectionArea(txLineH, leftSide,
                                       lineStart.v -
                                       (caretHeight DIV 2),
                                       leftSide + kCaretWidth,
                                       lineStart.v, TRUE);
                        END;
                     END;
                  END
               END;
            END;

            GetMouse(where);
         UNTIL NOT WaitMouseUp;
      END;

      HUnlock(Handle(txLineH));
   END
END;                             {MyDoTextClick}
```

Synchronizing the Caret With the Keyboard Script

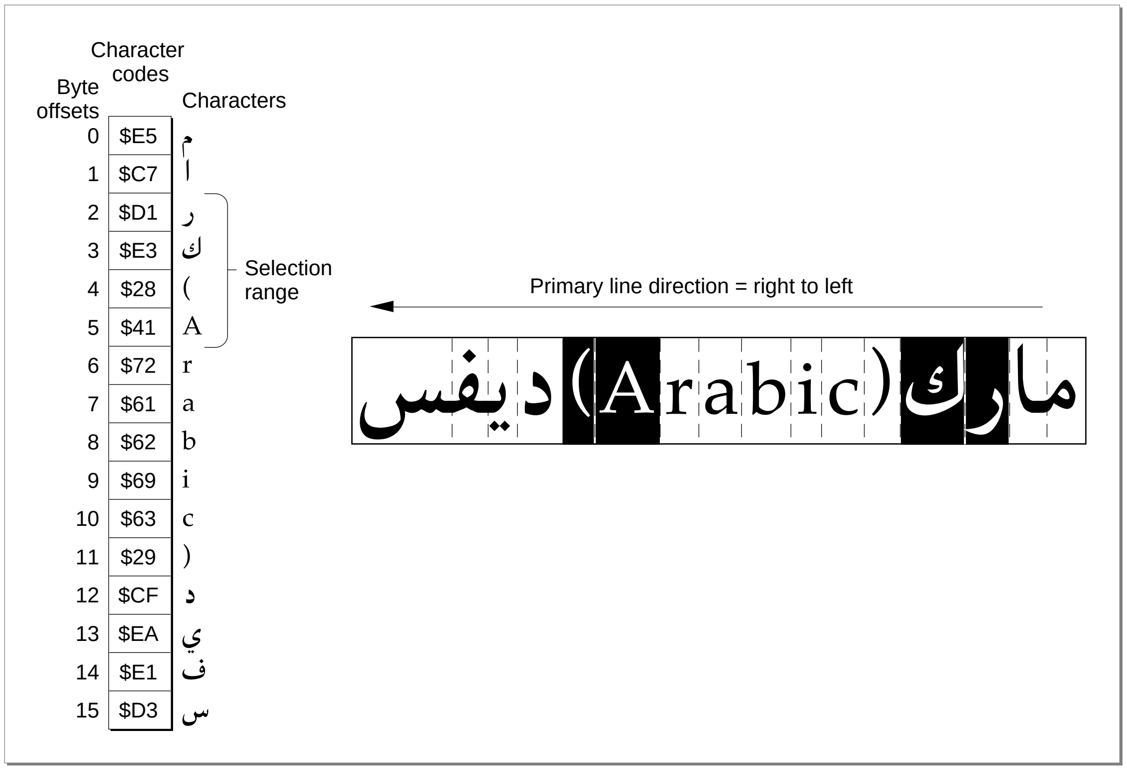
If the user changes the keyboard script, you can call the `CharToPixel` function to determine the caret position, specifying the `direction` parameter based on the keyboard script. However, the user may change the keyboard script between the time you draw and erase the caret. You can save the position where you drew the caret, then invert (erase) at that position again. To do this, save the direction of the keyboard script, the screen pixel width, or even the whole rectangle.

Highlighting a Text Selection

To display a selection range, you typically highlight the text. This process entails converting the offsets to their display screen pixel locations, and then calling the `InvertRect` procedure to display the text selection in inverse video or with a colored or outlined background.

When a range of text to be highlighted is unidirectional, it is contiguous in both memory order and display order; the highlighted text constitutes a single range. When the text is bidirectional, however, it can contain characters that occur on direction boundaries. Although the characters are stored contiguously in memory, the leading edge of one character's glyph does not constitute the trailing edge of the other in display order. A range of mixed-directional text that is contiguous in memory can produce up to three physically separate ranges of displayed text to be highlighted. For example, Figure 3-15 shows two separate ranges of highlighted text whose characters are contiguous in memory.

Figure 3-15 Highlighting mixed-directional text



To highlight a selection range, you need the beginning and ending byte offsets of the selected text. From these offsets, you determine one or more pairs of offsets of the displayed text. Once you have the pairs of offsets, you determine the pixel locations that mark the beginning and the end of the displayed text of each pair. You can include the following steps in the inner loop of your highlighting routine to determine these values.

1. You call `HiliteText` to get the individual pairs of byte offsets that encompass the onscreen ranges of text to be highlighted. The `HiliteText` procedure always returns three pairs of offsets. This is because if a text selection contains mixed-directional text, it can consist of up to three distinct ranges of text when displayed. For unidirectional text, `HiliteText` returns one pair that contains the beginning and ending byte offsets whose text is to be highlighted, and two pairs that each include the same numbers. You can ignore any pair of duplicate numbers.
2. Using the offset pairs that `HiliteText` returns, you convert each byte offset of a pair to its equivalent onscreen pixel location. You call `CharToPixel` once each for the beginning and ending offsets of a pair. You might call `CharToPixel` up to 6 times. You must pass the `CharToPixel` function a direction parameter of `hilite`, which signals it to use the primary line direction to determine the correct caret position. When you specify `hilite`, `CharToPixel` returns the correct caret position for the glyph based on the text direction of its character.

Once you have the pixel locations corresponding to the ends of each range of text, you must convert them to display line pixel locations that are relative to the line’s left margin. (The `CharToPixel` function returns the pixel location relative to the left edge of the text range for which you called it.) If you saved the line layout information and you have the screen pixel widths of the preceding style runs in display order, you can sum the widths of these style runs and add the screen pixel width that `CharToPixel` returns to the total. You must do this for the beginning and ending pixel locations that mark the text. If you did not save the screen pixel widths of the preceding style runs on the display line, you must lay out the text line again to get these values. When you have the pixel locations relative to left margin of the display line, you can highlight the text.

For text that is rendered in black and white, you call the `InvertRect` procedure to highlight each distinct text range; the background color is exchanged with the foreground color. For text that is rendered in color, all pixel values of the current background color are replaced with the value of the highlighting color.

Generally, the user chooses the highlighting color from the Color control panel, and the application uses this color. However, you can reset this color using the QuickDraw `HiliteColor` procedure. If a monitor is black-and-white and a highlighting color is specified, the highlighting color reverts to black.

Before you call `InvertRect` for colored text, first you must clear the `HiliteMode` low-memory global. By default the highlight mode bit of the low-memory global variable is set to 1. You clear it by setting it to 0. After you highlight the text, you don’t need to reset the bit; `InvertRect` resets it automatically.

The easiest way to clear the highlight mode bit is to call the Toolbox Utilities' `BitClr` procedure, for example:

```
BitClr(Ptr(HiliteMode), pHiliteBit);
```

just before calling `InvertRect` using `srcXor` mode. (Do not alter the other bits in `HiliteMode`.)

Note
Routines that formerly used `Xor` inversion, such as `InvertRect` and the text drawing routines, will use highlight mode if the `hilite` bit is clear. ◆

From assembly language, you must call the `pHiliteBit` selector for highlight mode when you use the `BitClear` trap: `BCLR` must use the assembly-language equate `hiliteBit`. For example:

```
BCLR #hiliteBit, hiliteMode
```

Customizing QuickDraw's Text Handling

The QuickDraw bottleneck routines are procedures that perform the fundamental tasks associated with QuickDraw drawing operations. For each type of object that QuickDraw can draw, including text, there is a low-level routine which the higher-level routines call that actually performs the operation. These low-level routines, called bottlenecks because so many of the higher-level routines use them, carry out the actual work of measuring and drawing text.

The QuickDraw text routines use two of the bottleneck routines extensively—one to measure text (`StdTxMeas`) and one to draw it (`StdText`). Most of the high-level QuickDraw text routines call the low-level routines. The use of bottleneck routines provides flexibility to QuickDraw and applications that need to alter or augment the basic behavior of QuickDraw.

The graphics port record contains a field (`grafProcs`) which is set by default to `NIL`, indicating that QuickDraw should use the standard low-level bottleneck routines. You can modify this field to point to a record, `QDProcs`, which holds the addresses of customized routines for QuickDraw to use instead of the standard ones. For more information about the `QDProcs` record, see the QuickDraw chapters in *Inside Macintosh: Imaging*.

You can set some of the fields of this record to point to the standard bottleneck routines, and some to point to your customized routines. Your customized bottleneck routine can augment the standard bottleneck routine by calling it directly, either before or after performing its own operations, or it can replace a standard routine. If you replace either of the two standard bottleneck routines used for measuring (`StdTxtMeas`) and drawing

(StdText) text, the routines you install must have the same calling sequences as the standard routines. See "Low-Level QuickDraw Text Routines" on page 3-98 for these routines and their parameters. For the major discussion of how to customize the QuickDraw bottleneck routines, see *Inside Macintosh: Imaging*.

Note
Before replacing a bottleneck routine, consider the possibility that to do so could jeopardize the future compatibility of the application. If you replace either StdTxMeas or StdText, you change the behavior of the high-level routines that call them. ◆

You can also customize QuickDraw's text drawing and measuring capabilities by writing high-level routines that do additional processing, but call the standard bottleneck routines.

Note
If you need to call either StdText or StdTxMeas directly, you must first check the graphics port grafProc field to determine whether the bottleneck routines have been customized, and if so, you must call the customized routine instead of the standard one. The bottleneck routines are always customized for printing. ◆

Text in QuickDraw Pictures

This section describes aspects of how text is stored in picture files, including related limitations and restrictions, such as the following:

- The grayishTextOr transfer mode is not stored in pictures files, and therefore you cannot use it for printing.
- Inside a picture definition, DrawText cannot have a byteCount greater than 255.

Fonts

Whenever you record text in a picture file, QuickDraw stores the name of the current font and uses it when playing back the picture. This is true for pictures drawn in both the original and color graphics ports. The opcode that QuickDraw uses to save this information is $002C. Here is its data type:

```
PictFontInfo   =  Record
                     length:  Integer; {length of data in bytes}
                     fontID:  Integer; {ID in the source system}
                     fontName:STR255;
                  End;
```

QuickDraw saves this information only one time for each font used in a picture. The code in Listing 3-10 generates a picture file containing the font information that Listing 3-11 shows.

Listing 3-10 Generating a picture file with font information

```
GetFNum ('Venice', theFontID); {Set a font before opening PICT}
TextFont (theFontID);

pHand2   := OpenPicture (pictRect);
      MoveTo(20,20);
      DrawString(' Better be Venice');

      GetFNum('Geneva', theFontID);
      TextFont(theFontID);
      MoveTo(20,40);
      DrawString('Geneva');

      GetFNum('Geneva', theFontID);
      TextFont(theFontID);
      MoveTo(20,60);
      DrawString('Geneva');
   ClosePicture;
```

When QuickDraw plays back a picture, it uses the font family ID (`fontID`) as a reference into the list of font names which are used to set the correct font on the target system.

Listing 3-11 A picture file with font information

```
OpCode 0x002C {9,
   "0005 0656 656E 6963 65"} /* save current font */
TxFont 'venice'
DHDVText {20, 20, " Better be Venice"}
OpCode 0x002C {9,             /* save next font name */
   "0003 0647 656E 6576 61"}
TxFont 'geneva'
DVText {20, "Geneva"}
OpCode 0x002C {11,      /* ditto       */
   "0002 084E 6577 2059 6F72 6B"}
TxFont 'geneva'               /* second Geneva does not need
                                 another $002C */
DVText {20, "Geneva"}
```

Text With Multiple Style Runs

If you used the `GetFormatOrder` procedure to determine the correct order in which to draw text consisting of multiple style runs, the style runs are stored in display order when you record the text in a picture file. To reconstruct the storage order of the text from the picture file, an application that reads the style runs into memory from the picture file must then use the `GetFormatOrder` procedure to reverse the display order to storage order. Finally, the application must write the text into memory again, following the order that `GetFormatOrder` returns.

For example, suppose you have a a text string consisting of three style runs with different text directions—A (right-to-left), B (left-to-right), and C (left-to-right)—in storage order. You number them: A=1, B=2, C=3.

You call `GetFormatOrder` for these style runs with a line direction of right to left, and it returns 2, 3, 1. When you draw the style runs in display order and record them in a picture file, they are written to the picture file in display order. Suppose that after you record the picture file, an application cuts and pastes the text. Now the style runs are written to memory in display order: B, C, A.

To get the proper storage order, you call `GetFormatOrder` again, with the same line direction that you used the first time you called `GetFormatOrder` to determine the display order, and it produces the original storage order: A, B, C.

QuickDraw Text Reference

This section describes the data structures, routines, and an application-defined routine that provide the text-handling components of QuickDraw.

In addition to the graphics port record, which all of the routines use and which is defined in the QuickDraw chapters of *Inside Macintosh: Imaging*, these routines use two additional data structures: the font information record and the `Style` data type. They are described in the “Data Structures” section.

The “Routines” section describes the QuickDraw routines that you use to define the text drawing environment, measure and draw text, and identify what glyphs to highlight and where to position the cursor in a range of text.

The constants that you use to identify the text direction and to specify where a style run occurs within a line of text are listed in the “Summary of QuickDraw Text.” The constants that you use to identify a font are listed in the chapter “Font Manager” in this book. Equivalent declarations in the C language for the declarations and routines described in this section are listed in the “C Summary.”

Data Structures

This section describes the data structures that you use to provide information to the text-handling routines of QuickDraw. The font information record returns measurement information about the font or fonts used. The `Style` data type defines the styles that you use to set the text style.

For more information about QuickDraw pictures, see the QuickDraw chapters in *Inside Macintosh: Imaging*.

The Font Information Record

The `GetFontInfo` procedure uses the font information record to return measurement information based on the font of the current graphics port. If the current font has an associated font, as do Arabic and Hebrew, `GetFontInfo` returns information based on both fonts. The font information record contains the ascent, the descent, the width of the largest glyph, and the leading for a given font. The `StdTxtMeas` function also uses a record of type `FontInfo` to return information about the current font. The `FontInfo` data type defines a font information record.

```
TYPE FontInfo = RECORD
   ascent:  Integer; {ascent}
   descent: Integer; {descent}
   widMax:  Integer; {maximum glyph width}
   leading: Integer; {leading}
   END;
```

Field descriptions

`ascent`	The measurement in pixels from the baseline to the ascent line of the font.
`descent`	The measurement in pixels from the baseline to the descent line of the font.
`widMax`	The width in pixels of the largest glyph in the font.
`leading`	The measurement in pixels from the descent line to the ascent line below it.

The Style Data Type

The `Style` data type defines the styles that you specify as values to the `TextFace` procedure to set the text style in the current graphics port's `txFace` field. QuickDraw draws the glyph in this style.

```
StyleItem   = (bold, italic, underline, outline,
               shadow, condense, extend);
   Style    = SET OF StyleItem;
```

Routines

This section describes the routines that you use to set the text characteristics of the graphics port drawing environment, measure and draw text, lay out lines of text, and determine where to position the caret and which glyphs to highlight in a range of text. It also describes two low-level routines that you can use to measure and draw text.

Four parameters that are common to a number of routines are described in detail here. These parameters are also listed and defined briefly in the each routine.

The slop Parameter

The `DrawJustified`, `MeasureJustified`, `PixelToChar`, and `CharToPixel` routines take a `slop` parameter. The value of this parameter is the number of pixels by which the width of the text segment is to be changed, after the text has been scaled. The `slop` is a signed value that specifies how much the text is to be extended or condensed. The `slop` is derived from the calculations made using the proportion returned from the `PortionLine` function for a style run. To measure or draw text that is not to be extended or condensed, pass a `slop` value of 0.

The styleRunPosition Parameter

The `PortionLine`, `MeasureJustified`, `DrawJustified`, `PixelToChar`, and `CharToPixel` routines take a `styleRunPosition` parameter. This parameter specifies the position of the style run on the display line, and is used to

- determine the proportion of total slop to apply to a style run
- measure or draw a line of justified text
- identify where to break a line of text
- determine the caret position to mark an insertion point or highlight text.

The style run position parameter is meaningful only for those script systems that use intercharacter spacing for justification. For all other script systems, the parameter exists for future extensibility. Although the style run position parameter is not used, for example, for justifying text in the Roman script system, to allow for future compatibility, you should always specify the appropriate value for it for all calls that take it.

For those script systems that do use intercharacter spacing, space between style runs may be allocated differently depending upon whether the style run is leftmost, rightmost, or between two other style runs. For example, depending on the script system, if a style run occurs at the beginning or end of a line, extra space *may not* be added to the outer edge of the outermost glyph, whereas if a style run is interior to a line, all of the glyphs of the text may be treated the same: extra space is allocated to both sides of every glyph including those at either end of the style run.

Note
The current implementations of simple script systems such as Roman and Cyrillic do not justify a line of text by changing the width of nonspace characters. Instead, they rely solely on the use of space characters: the same amount of extra width is added to (or subtracted from) every space whether the space is at the beginning or end of the line or interior to it. ◆

Use one of the following constants (defined as type `JustStyleCode`) in the `styleRunPosition` parameter.

Constant	Value	Meaning
`onlyStyleRun`	0	Only style run on the line
`leftStyleRun`	1	Leftmost of multiple style runs on the line
`rightStyleRun`	2	Rightmost of multiple style runs on the line
`middleStyleRun`	3	Interior style run: neither leftmost nor rightmost

The numer and denom Parameters

The `PortionLine`, `DrawJustified`, `MeasureJustified`, `PixelToChar`, `CharToPixel`, `StdText`, and `StdTxMeas` routines take `numer` and `denom` parameters. Both `numer` and `denom` are point values: `numer` specifies the numerator for the horizontal and vertical scaling factors, and `denom` specifies the denominator for the horizontal and vertical scaling factors. Together, these values specify the scaling factors for the text: `numer.v` over `denom.v` gives the vertical scaling (height), and `numer.h` over `denom.h` gives the horizontal scaling factors (width). For routines that take these parameters, you need to specify values for `numer` and `denom` even if you are not scaling the text. For unscaled text, you can specify scaling factors of 1, 1.

For all routines except `StdTxtMeas` that take these parameters, `numer` and `denom` are input parameters only. For `StdTxtMeas`, `numer` and `denom` are reference parameters. On output, these parameters contain additional scaling to be applied to the text. Use of the output values is explained in the description of “StdTxMeas” on page 3-99.

Setting Text Characteristics

The routines in this section set values in the text-related fields of the current graphics port (`GrafPort` or `CGrafPort`). You also use these routines to set text characteristics that vary from style run to style run. You use these routines to set the graphics port fields to values equivalent to a new style run’s text characteristics before you call other QuickDraw routines to measure and draw the text.

- The `TextFont` procedure specifies the font to be used.
- The `TextFace` procedure specifies the glyph style.
- The `TextMode` procedure specifies the transfer mode.
- The `TextSize` procedure specifies the font size.

- The `CharExtra` procedure specifies the amount of pixels by which to widen or narrow each space character in a range of text.
- The `SpaceExtra` procedure specifies the amount of pixels by which to widen or narrow each glyph other than the space characters in a range of text (`CharExtra`).

Note
To ensure future compatibility and benefit from any future enhancements, always use these routines to modify the text fields of the graphics port record, rather than directly change the field values. ◆

TextFont

The `TextFont` procedure sets the font of the current graphics port in which the text is to be rendered.

```
PROCEDURE TextFont (font: Integer);
```

`font`	The font family ID.

DESCRIPTION

The `TextFont` procedure sets the value of the graphics port text font (`txFont`) field. The initial font family ID is `0`, which represents the system font. The value that you specify for this field is either an integer or a constant. The range of integers currently defined are from `0` to `32767`. Currently, negative font family IDs are not supported, although they may be supported in the future.

For more information about `TextFont`, see "Setting the Font" on page 3-20.

SPECIAL CONSIDERATIONS

The system font and application font have different font IDs and sizes on various script systems. However, the special designators `0` and `1` always map to the system font and the application font for the system script, respectively.

TextFace

The `TextFace` procedure sets the style of the font in which the text is to be drawn in the current graphics port.

```
PROCEDURE TextFace (face: Style);
```

`face`	The style for text to be drawn in the current graphics port.

DESCRIPTION

The `TextFace` procedure sets the value for the style of the font in the text face (`txFace`) field of the current graphics port. The `Style` data type allows you to specify a set of one or more of the following predefined constants: `bold`, `italic`, `underline`, `outline`, `shadow`, `condense`, and `extend`. In Pascal, you specify the constants within square brackets. For example:

```
TextFace([bold]);          {bold}
TextFace([bold,italic]);   {bold and italic}
```

The style is set to the empty set (`[]`) by default, which specifies plain. For more information, see "Modifying the Text Style" on page 3-21.

ASSEMBLY-LANGUAGE INFORMATION

In assembly language, the style set is stored as a word whose low-order byte contains bits representing the style. The bit numbers are specified by the following global constants.

Constant	Bit	Meaning
`bold`	0	Bold style
`italicBit`	1	Italic style
`ulineBit`	2	Underlined style
`outlineBit`	3	Outlined style
`shadowBit`	4	Shadowed style
`condense`	5	Condensed style
`extendBit`	6	Extended style

If all bits are `0`, the low-order byte represents the plain glyph style.

TextMode

The `TextMode` procedure sets the transfer mode for drawing text in the current graphics port.

```
PROCEDURE TextMode (mode: Integer);
```

`mode` The transfer mode to be used to draw the text.

DESCRIPTION

The `TextMode` procedure sets the transfer mode in the graphics port `txMode` field. The transfer mode determines the interplay between what an application is drawing (the source) and what already exists on the display device (the destination), resulting in the text display.

There are two basic kinds of modes: pattern (`pat`) and source (`src`). Source is the kind that you use for drawing text. There are four basic Boolean operations: `Copy`, `Or`, `Xor`, and `Bic` (bit clear), each of which has an inverse variant in which the source is inverted before the transfer, yielding eight operations in all. Original QuickDraw supports these eight transfer modes. Color QuickDraw enables your application to achieve color effects within those basic transfer modes, and offers an additional set of transfer modes that perform arithmetic operations on the RGB values of the source and destination pixels. See the chapter "Color QuickDraw" in *Inside Macintosh: Imaging* for a complete discussion of the arithmetic transfer modes. Other transfer modes are `grayishTextOr`, `transparent` mode, and text mask mode.

Table 3-1 shows the eight basic transfer modes and their effects on the destination pixels.

Table 3-1 Effects of the basic transfer modes

Source	**Action on the destination pixel**	
	If black source	If white source
`srcCopy`	Force black	Force white
`srcOr`	Force black	Leave alone
`srcXOr`	Invert	Leave alone
`srcBic`	Force white	Leave alone
`NotSrcCopy`	Force white	Force black
`NotSrcOr`	Leave alone	Force black
`NotSrcXOr`	Leave alone	Invert
`NotSrcBic`	Leave alone	Force white

This is how color affects these transfer modes when the source pixels are either all black (all 1's) or white (all 0's).

Copy

The `Copy` mode applies the foreground color to the black part of the source (the part containing 1's) and the background color to the white part of the source (the part containing 0's), and replaces the destination with the colored source.

Or

The `Or` mode applies the foreground color to the black part of the source and replaces the destination with the colored source. The white part of the source isn't transferred to the destination. If the foreground is black, the drawing will be faster. Copying to a white background always reproduces the source image, regardless of the pixel depth.

Xor

The `Xor` mode complements the bits in the destination corresponding to the bits equal to 1 in the source. When used on a colored destination, the color of the inverted destination isn't defined.

Bic

The `Bic` mode applies the background color to the black part of the source and replaces the destination with the colored source. The white part of the source isn't transferred to the destination. The black part of the source is erased, resulting in white in the destination.

NotCopy

The `NotCopy` mode applies the foreground color to the white part of the source and the background color to the black part of the source, and replaces the destination with the colored source. It thus has the effect of reversing the foreground and background colors.

NotOr

The `NotOr` mode applies the foreground color to the white part of the source and replaces the destination with the colored source. The black part of the source isn't transferred to the destination. If the foreground is black, the drawing will be faster.

NotXor

The `NotXor` mode inverts the bits that are 0 in the source. When used on a colored destination, the color of the inverted destination isn't defined.

NotBic

The `NotBic` mode applies the background color to the white part of the source and replaces the destination with the colored source. The black part of the source isn't transferred to the destination.

The arithmetic transfer modes are `addOver`, `addPin`, `subOver`, `subPin`, `adMax`, `adMin`, and `blend`. For color, the arithmetic modes change the destination pixels by performing arithmetic operations on the source and destination pixels. Arithmetic transfer modes calculate pixel values by adding, subtracting, or averaging the RGB components of the source and destination pixels. They are most useful for 8-bit color, but they work on 4-bit and 2-bit color also. When the destination bitmap is one bit deep, the mode reverts to the basic transfer mode that best approximates the arithmetic mode requested.

The `grayishTextOr` transfer mode draws dimmed text on the screen. You can use it for black-and-white or color graphics ports. The `grayishTextOr` transfer mode is not considered a standard transfer mode because currently it is not stored in pictures, and printing with it is undefined. (It does not pass through the QuickDraw bottleneck routines.)

The transparent mode replaces the destination pixel with the source pixel if the source pixel isn't equal to the background color. This mode is most useful in 8-bit, 4-bit, or 2-bit color modes.

Note
Multibit fonts may have a specific color. Some transfer modes may not produce the desired results with a multibit font. However, the arithmetic modes, transparent mode, and hilite mode work equally well with single bit and multibit fonts. Multibit fonts draw quickly in `srcOr` mode only if the foreground is white. Single bit fonts draw quickly in `srcOr` mode only if the foreground is black. Grayscale fonts produce a spectrum of colors, rather than just the foreground and background colors. The following table shows transfer mode constants and their selectors. ◆

Table 3-2 Transfer mode constants and selectors

Transfer mode	Selector	Transfer mode	Selector
`srcCopy`	0	`addPin`	33
`srcOr`	1	`addOver`	34
`srcXor`	2	`subPin`	35
`srcBic`	3	`transparent`	36
`notSrcCopy`	4	`adMax`	37
`notSrcOr`	5	`subOver`	38
`notSrcXor`	6	`adMin`	39
`notSrcBic`	7	`grayishTextOr`	49
`blend`	32	`mask`	64

For more information about transfer modes, see the chapters "QuickDraw Drawing" and "Color QuickDraw" in *Inside Macintosh: Imaging*.

TextSize

The `TextSize` procedure sets the font size for text drawn in the current graphics port to the specified number of points.

```
PROCEDURE TextSize (size: Integer);
```

`size`	The font size in points. If you specify 0, the system font size (normally 12 points) is used.

DESCRIPTION

The `TextSize` procedure sets the font size in the text size (`txSize`) field of the current graphics port record. The initial setting is 0, which specifies that the font size of the system font is to be used. You may specify a value from 0 up to 32,767. For more information, see “Changing the Font Size” on page 3-22.

SpaceExtra

The `SpaceExtra` procedure specifies the number of pixels by which to widen (or narrow) each space in a style run to be drawn in the current graphics port.

```
PROCEDURE SpaceExtra (extra: Fixed);
```

`extra`	The amount (in pixels or binary fractions of a pixel) to widen (or narrow) each space in a style run on a line.

DESCRIPTION

The `SpaceExtra` procedure sets the value of the extra space (`spExtra`) field in the current graphics port record. The initial setting is `0`. You can pass a negative value for the `extra` parameter, but be careful not to narrow spaces so much that the text is unreadable. The value you specify is added to the width of each space character in the style run. For those script systems that do not use spaces, any value set in the extra space field is ignored. For those script systems that use spaces as delimiters, if you do not want to justify a line of text using `DrawJustified`, you can use the `SpaceExtra` procedure to set a fixed number of pixels to be added to each space character, then call `DrawText` or `DrawString`.

When you use the justification routines (`MeasureJustified`, `DrawJustified`) to measure or draw justified text, they temporarily reset the extra space value. They add to the current value of the field, if any, the amount of extra space to be added to space characters in the specified text in order to justify the text, based on calculations that take into account the slop value for the range of text and all of the text characteristics. On exit, these routines restore the original value.

For more information about `SpaceExtra`, see “Changing the Width of Characters” on page 3-22.

SPECIAL CONSIDERATIONS

For a color graphics port (`CGrafPort`), you can use `SpaceExtra` by itself or in conjunction with the `CharExtra` procedure to format a line of text in the 1-byte simple or 2-byte script systems. You should not use `CharExtra` for 1-byte complex script systems.

CharExtra

For a color graphics port (`CGrafPort`), the `CharExtra` procedure specifies the number of pixels by which to widen (or narrow) the glyphs of each nonspace character in a style run.

```
PROCEDURE CharExtra (extra: Fixed);
```

`extra`	The amount (in pixels or decimal fractions of a pixel) to widen (or narrow) each glyph other than the space character in a range of text.

DESCRIPTION

The `CharExtra` procedure sets the value of the `chExtra` field of the color graphics port record. This field contains a number that is in 4.12 fractional notation: four bits of signed integer followed by 12 bits of fraction. The `CharExtra` procedure uses the value of the `txSize` field, so you must call `TextSize` to set the font size of the text before you call `CharExtra`.

The initial setting is 0. You can pass a negative value for the `extra` parameter, but be careful not to narrow glyphs so much that the text is unreadable. The measuring and drawing routines use the value in this field when an application calls them to measure or draw text. The `CharExtra` procedure is available only for color graphic ports.

SPECIAL CONSIDERATIONS

Do not use `CharExtra` for script systems that include zero-width characters, such as diacritical marks, because intercharacter space is added to all glyphs, separating the diacritical mark from the glyph of the character. Do not use it for script systems that include contextual forms, such as ligatures or conjunct characters, which would not be represented properly were intercharacter space added to these glyphs. For example, you should not use `CharExtra` for the Devanagari or Arabic languages, whose text is drawn as connected glyphs, or with the Sonata font because it includes zero-width characters.

The 2-byte script systems use the `chExtra` field value properly.

GetFontInfo

The `GetFontInfo` procedure returns information about the current graphics port's font, taking into account the style and size in which the glyphs are to be drawn.

```
PROCEDURE GetFontInfo (VAR info: FontInfo);
```

`info`	A font information record that contains the font measurement information, in integer values.

DESCRIPTION

The GetFontInfo procedure returns the ascent, descent, leading, and width of the largest glyph of the font in the text font, size, and style specified in the current graphics port. If the script system specified by the current graphics port txFont field has an associated font, as do Hebrew and Arabic, GetFontInfo returns combined information based on both fonts. This is to accommodate text written in the Roman script when the primary script system is non-Roman. However, even if all of the text is written in a non-Roman script, if there is an associated font, GetFontInfo always bases its information on the combined fonts. You can determine the line height, in pixels, by adding the values of the ascent, descent, and leading fields.

The GetFontInfo procedure is similar to the Font Manager's FontMetrics procedure, except that the GetFontInfo procedure returns integer values. See "The Font Information Record" on page 3-66 for a description of the record and its fields.

Drawing Text

QuickDraw provides routines that allow you to draw a single character, a Pascal string, or an arbitrary sequence of text. You can also draw a text sequence made narrower or wider using these routines; this technique is commonly used to justify a line of text. These routines draw text in the font, style, and size of the current graphics port. Consequently, you can draw only a single style run at a time using these routines.

- The DrawChar procedure draws the glyph of a single 1-byte character.
- The DrawString procedure draws the text of a Pascal string.
- The DrawText procedure draws the glyphs of a sequence of characters.
- The DrawJustified procedure draws a sequence of text that is widened or narrowed by a specified number of pixels.

Whether the text to be drawn has a left-to-right direction, a right-to-left direction, or is bidirectional, QuickDraw always draws text starting at the current pen location and always advances the pen to the right by the width of the glyph or glyphs it has just drawn. Before drawing text that has a right-to-left direction, QuickDraw reorders the glyphs for display so that they can be read correctly, even though it draws them from left to right.

DrawChar

The DrawChar procedure draws the glyph for the specified character at the current pen location in the current graphics port.

```
PROCEDURE DrawChar (ch: CHAR);
```

ch	The character code whose glyph is to be drawn.

DESCRIPTION

The `DrawChar` procedure draws a single character's glyph and then advances the pen by the width of the glyph. If the glyph isn't in the font, the font's missing symbol is drawn. For more information, see "Individual Glyphs" on page 3-28.

Note
If you're drawing more than one character, it's faster to make one `DrawString` or `DrawText` call rather than a series of `DrawChar` calls. ◆

SPECIAL CONSIDERATIONS

Because it takes a single-byte value as the `ch` parameter, `DrawChar` works only for 1-byte script systems. If you want to draw the glyph of a single character in a 2-byte script, call either `DrawText`, `DrawString`, or `DrawJustified`.

However, a series of calls to `DrawChar` in a 1-byte complex script system can give incorrect results because a text string is not always a simple concatenation of a series of characters. In a contextual script, two different glyphs may be used to represent a single character in its contextual form and alone. To draw a sequence of text in a 1-byte complex script system, use `DrawText`, `DrawString`, or `DrawJustified` instead.

However, for 1-byte complex scripts, you can use `DrawChar` for special purposes, such as to include the isolated glyph of a character in a book's index, for example, to show a single glyph as it exists apart from contextual transformations.

DrawString

The `DrawString` procedure draws the specified Pascal string at the pen location in the current graphics port (`GrafPort` or `CGrafPort`).

```
PROCEDURE DrawString (s: Str255);
```

`s`	A Pascal string consisting of the text to be drawn.

DESCRIPTION

The `DrawString` procedure draws the string with its left edge at the current pen location, extending right. The final position of the pen location, after the text is drawn, is to the right of the rightmost glyph in the string. QuickDraw does not do any formatting, such as handling of carriage returns or line feeds.

Note that you can use `DrawString` only for a Pascal string containing a single style run.

Drawing text visible on the screen
QuickDraw temporarily stores on the stack all of the text you ask it to draw, even if the text is to be clipped. When drawing large font sizes or complex style variations, draw only what is visible on the screen. You can determine the number of characters whose corresponding glyphs actually fit on the screen by calling the `StringWidth` function to determine the length of the string before calling `DrawString`. ◆

If you specify values in the graphics port `spExtra` or `chExtra` fields to change the width of space or nonspace characters, `DrawString` takes these values into account.

SPECIAL CONSIDERATIONS

For right-to-left text, such as Hebrew or Arabic, QuickDraw draws the final (leftmost) glyph first, then moves to the right through all the glyphs, drawing the initial (rightmost) glyph last.

Note that you should not change the width of nonspace characters for 1-byte simple script systems with zero-width characters or 1-byte complex script systems. For more information, see "CharExtra" on page 3-75.

For contextual script systems, `DrawString` substitutes the proper ligatures, reversals, and compound characters as needed.

Note
Inside a picture definition, `DrawString` can't have a `byteCount` greater than 255. ◆

DrawText

The `DrawText` procedure draws the specified text at the current pen location in the current graphics port.

```
PROCEDURE DrawText (textBuf: Ptr; firstByte, byteCount: Integer);
```

`textBuf`	A pointer to a buffer containing the text to be drawn.
`firstByte`	An offset from the start of the text buffer (`textBuf`) to the first byte of the text to be drawn.
`byteCount`	The number of bytes of text to be drawn.

DESCRIPTION

The `DrawText` procedure draws the text with the leftmost glyph at the current pen location, extending right. After QuickDraw draws the text, it sets the pen location to the right of the rightmost glyph. For more information, see "Text Segments" on page 3-29.

Drawing text visible on the screen
QuickDraw temporarily stores on the stack all of the text you ask it to draw, even if the text is to be clipped. When drawing a range of text, it's best to draw only what is visible on the screen. If an entire text string does not fit on a line, truncate the text at a word boundary. If possible, avoid truncating within a style run. You can determine the number of characters whose glyphs actually fit on the screen by calling the `TextWidth` function before calling `DrawText`. ◆

If you specify values in the graphics port `spExtra` and `chExtra` fields to change the width of nonspace and space characters, both `TextWidth` and `DrawText` take these values into account.

SPECIAL CONSIDERATIONS

For 1-byte complex script systems, `DrawText` substitutes the proper ligatures, reversals, and compound characters as needed.

For right-to-left text, such as Hebrew or Arabic, QuickDraw draws the final (leftmost) glyph first, then moves to the right through all the characters, drawing the initial (rightmost) glyph last.

For 2-byte script systems, note that `byteCount` is the number of *bytes* to be drawn, not the number of glyphs. Because 2-byte script systems also include characters consisting of only 1 byte, do not simply multiply the number of characters by 2 to determine this value; you must determine and specify the correct number of bytes.

Note
Inside a picture definition, `DrawText` cannot have a `byteCount` greater than 255. ◆

DrawJustified

The `DrawJustified` procedure draws the specified text at the current pen location in the current graphics port, taking into account the adjustment necessary to condense or extend the text by the slop value, appropriately for the script system.

```
PROCEDURE DrawJustified (textPtr: Ptr; textLength: LongInt;
                         slop: Fixed;
                         styleRunPosition: JustStyleCode;
                         numer, denom: Point);
```

`textPtr` A pointer to the memory location of the beginning of the text to be drawn.

`textLength`
The number of bytes of text to be drawn.

`slop`	The amount of slop for the text to be drawn. A positive value extends the text segment; a negative value condenses the text segment.
`styleRunPosition`	The position on the line of this style run. The style run can be the only one on the line, the leftmost on the line, the rightmost on the line, or one between two other style runs.
`numer`	A point giving the numerator for the horizontal and vertical scaling factors.
`denom`	A point giving the denominator for the horizontal and vertical scaling factors.

DESCRIPTION

The `DrawJustified` procedure is similar to the `DrawText` procedure, except that you use it to draw text that is expanded or condensed by the number of pixels specified by `slop`. The `DrawJustified` procedure is most commonly used to draw a line of justified text.

The `DrawJustified` procedure draws the specified text in the font, size, and style of the current graphics port, taking into account any scaling factors, and it distributes the slop appropriately for the script system. Regardless of the line direction of the text to be drawn, you place the pen at the left edge of the line before calling `DrawJustified` for the first style run. For all subsequent style runs on that line, QuickDraw advances the pen appropriately.

If `DrawJustified` changes the width of spaces, it temporarily resets the space extra (`spExtra`) value. It adds to the current value of the field, if any, the amount of extra space to be applied to each space character within the range of text in order to justify the text, based on calculations that take into account the slop value and all of the text characteristics. On exit, `DrawJustified` restores the original value.

For the slop parameter, pass `DrawJustified` the value assessed for this style run based on the proportion returned for it from `PortionLine`. For more information, see “The numer and denom Parameters” on page 3-68.

Note
Be sure to pass the same values for `styleRunPosition` and the scaling factors (`numer` and `denom`) to `DrawJustified` that you pass to `PortionLine`. ◆

See “The styleRunPosition Parameter” on page 3-67 for a description of this parameter and the values it takes. See “The slop Parameter” on page 3-67 for more information about the `slop` parameter.

For more information about how to use `DrawJustified` in conjunction with the other routines used to prepare to draw a line of justified text, see“Measuring and Drawing Lines of Text,” beginning on page 3-29.

SPECIAL CONSIDERATIONS

The `DrawJustified` procedure works with text in all script systems. For example, to depict justified Arabic text, `DrawJustified` uses extension bars to create the additional width that is distributed as slop within a style run.

For 1-byte complex script systems, `DrawJustified` substitutes the proper ligatures, reversals, and compound characters as needed.

For 2-byte script systems that do not use space characters to delimit words, `DrawJustified` distributes the slop value in a manner appropriate to the script system. For script systems, such as Japanese, that use ideographic characters, `DrawJustified` distributes the additional screen pixel width appropriately for the text representation.

Note that `textLength` is the number of *bytes* to be drawn, not the number of characters. Because 2-byte script systems also include characters consisting of only 1 byte, do not simply multiply the number of characters by 2 to determine this value; you must determine and specify the correct number of bytes.

The `DrawJustified` procedure may move memory; do not call this procedure at interrupt time.

Measuring Text

Laying out text to determine how much of it fits on the display line entails measuring the text. QuickDraw provides five high-level routines that let you do this:

- The `CharWidth` function returns the horizontal extension of a single glyph.
- The `StringWidth` function returns the width of a Pascal string.
- The `TextWidth` function returns the width of the glyphs of a text segment.
- The `MeasureText` procedure fills an array with an entry for each character identifying the width of each character's glyph as measured from the left side of the entire text segment.
- The `MeasureJustified` procedure fills an array with an entry for each character in a style run identifying the width of each character's glyph as measured from the left side of the text segment.

These routines measure text in the font, style, and size of the current graphics port. Consequently, you need to call them once for each individual style run in any line of text that contains multiple style runs.

CharWidth

The `CharWidth` function returns the width in pixels of the specified character.

```
FUNCTION CharWidth (ch: CHAR): Integer;
```

`ch`	The character whose width is to be measured.

DESCRIPTION

The `CharWidth` function includes the effects of the stylistic variations for the text set in the current graphics port. If you change any of these attributes after determining the glyph width but before actually drawing it, the predetermined width may not be correct. For a space character, `CharWidth` also includes the effect of `SpaceExtra`. For a nonspace character, `CharWidth` includes the effect of `CharExtra`. For more information, see "Individual Glyphs" on page 3-28.

SPECIAL CONSIDERATIONS

Because it takes a single-byte value as the `ch` parameter, `CharWidth` works only for 1-byte simple script systems.

A series of calls to `CharWidth` in a contextual 1-byte font may give incorrect results, because the width of a text segment may be different from the sum of its individual character widths. In that case, to measure a line of text you should call `TextWidth`.

Do not use the `CharWidth` function for 2-byte script systems. If you want to measure the width of a single glyph in a 2-byte font, you should use `TextWidth`.

StringWidth

The `StringWidth` function returns the length in pixels of the specified Pascal string.

```
FUNCTION StringWidth (s: Str255): Integer;
```

`s`	A pascal string containing the text to be measured.

DESCRIPTION

You should not call `StringWidth` to measure scaled text. Although `StringWidth` takes into account the graphics port record settings, it does not accept scaling parameters, and therefore cannot determine the correct text width result for text to be drawn using scaling factor parameters. For more information, see "Pascal Strings" on page 3-28.

If you specify values in the graphics port `spExtra` or `chExtra` fields to change the width of space or nonspace characters, `StringWidth` takes these values into account. The `StringWidth` function works with all script systems.

TextWidth

The `TextWidth` function returns the length in pixels of the specified text.

```
FUNCTION TextWidth (textBuf: Ptr;
                    firstByte, byteCount: Integer): Integer;
```

`textBuf`	A pointer to a buffer that contains the text to be measured.
`firstByte`	An offset from `textBuf` to the first byte of the text to be measured.
`byteCount`	The number of bytes of text to be measured.

DESCRIPTION

You can use `TextWidth` to measure the screen pixel width of any text segment that has uniform character attributes. You can use it to measure the style runs in a line of text, whether you intend to draw the line using `DrawText` or `DrawJustified`. The `TextWidth` function takes into account the character attributes set in the graphics port. If you change any of these attributes after determining the text width but before actually drawing the text, the predetermined width may not be correct. For a space character, `TextWidth` also includes the effect of `SpaceExtra`. For a nonspace character, `TextWidth` includes the effect of `CharExtra`.

The `TextWidth` function works with text in all script systems because the script management system modifies the routine if necessary to give the proper results.

Note
To draw justified lines of text that include multiple style runs, you calculate the amount of extra pixels, or slop, that remains to be distributed throughout the line. This process entails measuring the screen pixel width of each style run on the line: you can use `TextWidth` for this purpose. For a complete discussion of how to use `TextWidth` to prepare to draw a line of justified text, refer to "Measuring and Drawing Lines of Text" beginning page 3-29. ◆

SPECIAL CONSIDERATIONS

For 1-byte complex script systems, `TextWidth` calculates the widths of any ligatures, reversals, and compound characters that need to be drawn.

Note that `byteCount` is the number of bytes to be measured, not the number of characters. Because 2-byte script systems also include characters consisting of only one byte, you should not simply multiply the number of characters by 2 to determine this value; you must determine and specify the correct number of bytes.

MeasureText

The `MeasureText` procedure provides an array version of the `TextWidth` function. For each character in the specified text, `MeasureText` calculates the width of the character's glyph in pixels from the left edge of the text segment.

```
PROCEDURE MeasureText (count: Integer; textAddr, charLocs: Ptr);
```

`count`	The number of bytes to be measured.
`textAddr`	A pointer to the memory location of the beginning of the text to be measured. The value of `textAddr` must point directly to the first character whose glyph is to be measured.
`charLocs`	A pointer to an application-defined array of `count` + 1 integers.

DESCRIPTION

The `MeasureText` procedure calculates the onscreen pixel width of the glyph of each character beginning from the left edge of the text segment. On return, the first element in the `charLocs` array contains 0 and the last element contains the total width of the text segment, when the primary line direction is left to right and the text is unidirectional. When the primary line direction is right to left and the text is unidirectional, the first element in the array contains the total width of the text segment, and the last element in the array contains 0. When the text is bidirectional, at a direction boundary, `MeasureText` selects the character whose direction maps to that of the primary line direction.

The `MeasureText` procedure returns the same results that an application would get if it called `CharToPixel` for each character with a direction parameter value of `hilite`. Using `MeasureText` to find the pixel location of a character's glyph is less efficient than using the `CharToPixel` function because the application must define the array pointed to by `charLocs`, and then walk the array after `MeasureText` returns the results.

For more information about `MeasureText`, contact Developer Technical Support.

SPECIAL CONSIDERATIONS

Some fonts in 1-byte script systems may have zero-width characters, which are usually overlapping diacritical marks that typically follow the base character in memory. In this case, `MeasureText` measures both the glyph of the base character (the high-order, low-address byte) and the width of the diacritical mark. The `charLoc` array includes an entry for each, but both entries contain the same value.

For 1-byte complex script systems, `MeasureText` calculates the widths of any ligatures, reversals, compound characters, and character clusters that need to be drawn. For example, for an Arabic ligature, the entry that corresponds to the trailing edge of each character that is part of the ligature is the trailing edge of the entire ligature.

Note that `count` is the number of *bytes* to be measured, not the number of characters. Because 2-byte script systems also include characters consisting of only one byte, do not simply multiply the number of characters by 2 to determine this value; you must determine and specify the correct number of bytes. For 2-byte characters, the `charLocs` array contains two entries—one corresponding to each byte—but both entries contain the same pixel-width value.

MeasureJustified

For text that is expanded, condensed, or scaled, the `MeasureJustified` procedure calculates the onscreen width in pixels from the left edge of the text segment to the glyph of the character.

```
PROCEDURE MeasureJustified (textPtr: Ptr; textLength: LongInt;
                            slop: Fixed; charLocs: Ptr;
                            styleRunPosition: JustStyleCode;
                            numer, denom: Point);
```

`textPtr`	A pointer to the memory location of the beginning of the text to be measured.
`textLength`	The number of bytes of text to be measured. The text length should equal the entire visible part of the text on a line, including trailing spaces if and only if they are displayed. Otherwise, the results for the last glyph on the line may be invalid.
`slop`	The amount of slop for the text to be drawn. A positive value extends the text segment; a negative value condenses the text segment.
`charLocs`	A pointer to an application-defined array of `textLength` + 1 integers.
`styleRunPosition`	The position on the line of this style run. The style run can be the only one on the line, the leftmost on the line, the rightmost on the line, or one between two other style runs.
`numer`	A point giving the numerator for the horizontal and vertical scaling factors.
`denom`	A point giving the denominator for the horizontal and vertical scaling factors.

DESCRIPTION

The `MeasureJustified` procedure is similar to the `MeasureText` procedure, except that it is used to find the pixel location of a character's glyph in text that is expanded or condensed.

The `MeasureJustified` procedure calculates the onscreen pixel width of the glyph of each character beginning from the left edge of the text segment, taking into account `slop` value, scaling, and style run position.

On return, the first element in the `charLocs` array contains 0 and the last element contains the total width of the text segment, when the primary line direction is left to right and the text is unidirectional. When the primary line direction is right to left and the text is unidirectional, the first element in the array contains the total width of the text segment, and the last element in the array contains 0. When the text is bidirectional, at a direction boundary, `MeasureJustified` selects the character whose direction maps to that of the primary line direction.

The `MeasureJustified` procedure returns the same results that an application would get if it called `CharToPixel` for each character with a direction parameter value of `hilite`. Using `MeasureJustified` to find the pixel location of a character's glyph is less efficient than using the `CharToPixel` function because the application must define the array pointed to by `charLocs`, and then walk the array after `MeasureText` returns the results.

The `MeasureJustified` procedure temporarily resets the space extra (`spExtra`) value, adding to the current value of the field, if any, the amount of extra space to be added to space characters in order to fully justify the text, based on calculations that take into account the slop value and all the text characteristics. On exit, `MeasureJustified` restores the original value.

Because `MeasureJustified` measures text in only the current font, style, and size, you need to call it once for each individual style run.

For more information about the scaling factors, see "The numer and denom Parameters" on page 3-68. See "The styleRunPosition Parameter" on page 3-67 for a description of the `styleRunPosition` parameter and the values it takes. See "The slop Parameter" on page 3-67 for more information about the `slop` parameter.

For additional information about `MeasureJustified`, contact Developer Technical Support.

SPECIAL CONSIDERATIONS

The `MeasureJustified` procedure works properly for text in all script systems. For 1-byte complex script systems, `MeasureJustified` calculates the widths of any ligatures, reversals, and compound characters that would need to be drawn.

Note that `textLength` is the number of *bytes* to be drawn, not the number of characters. Because 2-byte script systems also include characters consisting of only one byte, you should not simply multiply the number of characters by 2 to determine this value; the application must determine and specify the correct number of bytes.

Some 1-byte script system fonts may have zero-width characters, which are usually overlapping diacritical marks that typically follow the base character in memory. In this case, `MeasureJustified` measures both the glyph of the base character (the high-order, low-address byte) and the width of the diacritical mark. The `charLoc` array includes an entry for each, but both entries contain the same value.

For 1-byte complex script systems, `MeasureJustified` calculates the widths of any ligatures, reversals, compound characters, and character clusters that need to be drawn. For example, for an Arabic ligature, the entry that corresponds to the trailing edge of each character that is part of the ligature is the trailing edge of the entire ligature.

The `MeasureJustified` procedure may move memory; do not call this procedure at interrupt time.

Laying Out a Line of Text

In addition to the routines that measure text, QuickDraw text provides additional routines that help you perform the tasks involved in laying out a line of text.

- The `GetFormatOrder` procedure determines the display order of style runs for a line of text containing multiple style runs with mixed directions.
- The `VisibleLength` function eliminates trailing spaces from the last style run on the line.
- The `PortionLine` function determines how to distribute the total slop value for a line among the style runs on that line.

GetFormatOrder

The `GetFormatOrder` procedure determines the display order of multiple style runs with mixed directions.

```
PROCEDURE GetFormatOrder (ordering: FormatOrderPtr;
                          firstFormat: Integer;
                          lastFormat: Integer;
                          lineRight: Boolean;
                          rlDirProc: Ptr; dirParam: Ptr);
```

`ordering` A pointer to a format order array. Upon completion of the call, the format order array contains the numbers identifying the style runs in display order. This is its type declaration:

```
TYPE
   FormatOrder = ARRAY [0..0] OF Integer;
   FormatOrderPtr = ^FormatOrder;
```

`firstFormat`
A number greater than or equal to 0 identifying the first style run in storage order that is part of the line for which you are calling `GetFormatOrder`.

`lastFormat`
A number greater than or equal to 0 identifying the last style run in storage order that is part of the line for which you are calling `GetFormatOrder`.

`lineRight`	A flag that you set to `TRUE` if the primary line direction is right-to-left.
`rlDirProc`	A pointer to an application-supplied function that calculates the correct direction, given the style run identifier. The `GetFormatOrder` procedure calls the application-defined `rlDirProc` function for each identifier from `firstFormat` to `lastFormat`. The interface to this function looks like this: `FUNCTION MyRlDirProc(theFormat: Integer; dirParam: Ptr ): Boolean;` This function returns `TRUE` for right-to-left text direction and `FALSE` for left-to-right. Given `dirParam` and a style run identifier, the application-defined `rlDirProc` routine should be able to determine the style run direction.
`theFormat`	A number identifying the style run and its associated attribute information in the information block pointed to by `dirParam`.
`dirParam`	A pointer to a parameter block that contains the font and script information for each style run in the text. This parameter block is used by the application-supplied routine.

```
FUNCTION MyRlDirProc(theFormat: Integer;
                     dirParam: Ptr ): Boolean;
```

DESCRIPTION

The `GetFormatOrder` procedure helps you determine how to draw text that contains multiple style runs with mixed directions. For mixed-directional text, after you determine where to break the line, you need to call `GetFormatOrder` to determine the display order. When you call `GetFormatOrder`, you supply a Boolean function, and reference it using the `rlDirProc` parameter. This function calculates the direction of each style run identified by number. Do not call `GetFormatOrder` if there is only one style run on the line.

You must index the style runs in storage order. You pass `GetFormatOrder` numbers identifying the first and last style runs of the line in storage order and the primary line direction. The `GetFormatOrder` procedure returns to you an equivalent sequence in display order.

If you do not explicitly define the primary line direction of the text, base the `lineRight` parameter on the value of the `SysDirection` global variable. (The `SysDirection` global variable is set to `-1` if the system direction is right to left, and 0 if the system direction is left to right.)

The `ordering` parameter points to an array of integers, with (`lastFormat – firstFormat + 1`) entries. The `GetFormatOrder` procedure fills an array (the size of the number of the style runs) with the display order of each style run. On exit, the array contains a permuted list of the numbers from `firstFormat` to `lastFormat`. The first entry in the array is the number of the style run to draw first; this is the leftmost style run in display order. The last entry in the array is the number of the entry to draw last, the rightmost style run in display order. For more information about how to use the `GetFormatOrder` procedure, see “Determining the Display Order for Style Runs,” which begins on page 3-33. For more information about the `rlDirProc` function, see “Application-Supplied Routine” on page 3-100.

VisibleLength

The `VisibleLength` function calculates the length in bytes of a given text segment, excluding trailing white space.

```
FUNCTION VisibleLength (textPtr: Ptr;
                        textLength: LongInt): LongInt;
```

`textPtr`	A pointer to a text string.
`textLength`	The number of bytes in the text segment.

DESCRIPTION

The `VisibleLength` function determines how much of a style run to display, without displaying trailing spaces. You call `VisibleLength` for the last style run of a line *in memory order*. The last style run in memory order of the text constituting the line is not always the last style run in display order. For a line of unidirectional left-to-right text, the last style run in memory order is the rightmost style run in display order. For a line of unidirectional right-to-left text, the last style run in memory order is the leftmost style run in display order. However, if the text contains mixed directions, the last style run in memory order may be an interior style run in display order.

The text justification routines do not automatically exclude trailing spaces, so you pass them the value that `VisibleLength` returns as the length of the last style run in memory order.

The `VisibleLength` function behaves differently for various script systems.

- For simple script systems, such as Roman and Cryllic, and for 2-byte script systems, `VisibleLength` does not include in the byte count it returns trailing spaces that occur at the display end of the text segment. For 2-byte script systems, `VisibleLength` does not count them, whether they are 1-byte or 2-byte space characters.
- For 1-byte complex script systems, `VisibleLength` does not include in the byte count that it returns spaces whose character direction is the same as the primary line direction. For 1-byte complex script systems that support bidirectional text, Roman spaces take on a character direction based on the primary line direction. If the Roman spaces then fall at the end of the text, `VisibleLength` does not include them in the returned byte count.

Advancing the pointer in memory in response to VisibleLength

The purpose of `VisibleLength` is to trim off white space at the display end of the line. The `VisibleLength` function does not eliminate the white space by removing its character code from memory. Rather, it does not include white space characters in the count that it returns as the length of the range of text for which you call it. ◆

For more information about `VisibleLength`, see the task description “Eliminating Trailing Spaces (for Justified Text)” on page 3-36.

PortionLine

The `PortionLine` function determines the correct proportion of extra space to apply to the specified style run in a line of justified text.

```
FUNCTION PortionLine(textPtr: Ptr; textLen: LongInt;
                        styleRunPosition: JustStyleCode;
                        numer: Point; denom: Point) : Fixed;
```

`textPtr`	A pointer to the style run.
`textLen`	The number of bytes in the text of the style run.
`styleRunPosition`	The position on the line of this style run. The style run can be the only one on the line, the leftmost on the line, the rightmost on the line, or one between two other style runs.
`numer`	A point giving the numerator for the horizontal and vertical scaling factors.
`denom`	A point giving the denominator for the horizontal and vertical scaling factors.

DESCRIPTION

You use `PortionLine` in formatting a line of justified text. It helps you determine how to distribute the slop for a line among its style runs. When you know the total slop for a line of text, you need to determine what portion of it to attribute to each style run. To do this, you call the `PortionLine` function once for each style run on the line. The `PortionLine` function computes the portion of extra space for a style run, taking into account the font, size, style, and scaling factors of the style run. It returns a number that represents the portion of the slop to be applied to the style run for which it is called. You use the value that `PortionLine` returns to determine the percentage of slop that you should attribute to a style run.

To determine the percentage of slop to allocate to each style run, you compute what percentage each portion is of the sum of all portions. To determine the actual slop value in pixels for each style run, you apply the percentage to the total slop value. The following steps summarize this process:

1. Call `PortionLine` for each style run on the line.
2. Add the returned values together.
3. Calculate the percentage of the slop value for each style run using the ratio of the value returned by `PortionLine` for that style run and the total of the values returned for all of the style runs on the line.

4. Calculate the number of pixels to be added to each style run by multiplying the percentage of the slop for each style run by the total number of pixels.

For more information about the scaling factors, see "The numer and denom Parameters" on page 3-68. See "The styleRunPosition Parameter" on page 3-67 for a description of the `styleRunPosition` parameter and the values it takes.

Note
Be sure to pass the same values for `styleRunPosition` and the scaling factors (`numer` and `denom`) to `PortionLine` that you pass to any of the other justification routines for this style run. ◆

Determining the Caret Position, and Selecting and Highlighting Text

To mark an insertion point you need to know where to draw the caret. To highlight text, you need to know the caret positions that begin and end the text range. This section describes routines that you use to locate a caret position for marking an insertion point or highlighting text. You can also use the `PixelToChar` function to determine where to break a line, and the `CharToPixel` function to find the screen pixel width of a text segment.

- The `PixelToChar` function converts a pixel location associated with a glyph in a range of text to a byte offset within the style run.
- The `CharToPixel` function converts a byte offset to a pixel location. The pixel location is measured from the left edge of the style run.
- The `HiliteText` procedure returns three pairs of offsets marking the endpoints of ranges of text to be highlighted.

PixelToChar

The `PixelToChar` function returns the byte offset of a character in a style run, or part of a style run, whose onscreen glyph is nearest the place where the user clicked the mouse.

```
FUNCTION PixelToChar (textBuf: Ptr; textLen: LongInt;
                      slop: Fixed; pixelWidth: Fixed;
                      VAR leadingEdge: Boolean;
                      VAR widthRemaining: Fixed;
                      styleRunPosition: JustStyleCode;
                      numer: Point; denom: Point): Integer;
```

`textBuf`	A pointer to the start of the text segment.
`textLen`	The length in bytes of the entire text segment pointed to by `textBuf`. The `PixelToChar` function requires the context of the complete text segment in order to determine the correct value.

`slop`	The amount of slop for the text to be drawn. A positive value extends the text segment; a negative value condenses the text segment.
`pixelWidth`	The screen location of the glyph associated with the character whose byte offset is to be returned. The screen location is measured in pixels beginning from the left edge of the text segment for which you call `PixelToChar`.
`leadingEdge`	A Boolean flag that, upon completion of the call, is set to `TRUE` if the pixel location is on the leading edge of the glyph, and `FALSE` if the pixel location is on the trailing edge of the glyph. The leading edge is the left side if the direction of the character that the glyph represents is left-to-right (such as a Roman character), and the right side if the character direction is right-to-left (such as an Arabic or a Hebrew letter).
`widthRemaining`	Upon completion of the call, contains –1 if the pixel location (specified by the `pixelWidth` parameter) falls within the style run (represented by the `textLen` bytes starting at `textBuf`). Otherwise, contains the amount of pixels by which the input pixel location (`pixelWidth`) extends beyond the right edge of the text for which you called `PixelToChar`.
`styleRunPosition`	The position on the line of this style run. The style run can be the only one on the line, the leftmost on the line, the rightmost on the line, or one between two other style runs.
`numer`	A point giving the numerator for the horizontal and vertical scaling factors.
`denom`	A point giving the denominator for the horizontal and vertical scaling factors.

DESCRIPTION

You can use the information that `PixelToChar` returns for highlighting, word selection, and identifying the caret position. The `PixelToChar` function returns a byte offset and a Boolean value that describes whether the pixel location is on the leading edge or trailing edge of the glyph where the mouse-down event occurred. When the pixel location falls on a glyph that corresponds to one or more characters that are part of the text segment, the `PixelToChar` function uses the direction of the character or characters to determine which side of the glyph is the leading edge. (A glyph can represent more than one character, for example, for a ligature. Generally, if a glyph represents more than one character, all of the characters have the same text direction.)

If the pixel location is on the leading edge, `PixelToChar` returns the byte offset of the character whose glyph is at the pixel location. (If the glyph represents multiple characters, it returns the byte offset of the first of these characters in memory.) If the pixel location is on the trailing edge, `PixelToChar` returns the byte offset of the first character in memory *following* the character or characters represented by the glyph. If the pixel location is on the trailing edge of the glyph that corresponds to the last character in the text segment, `PixelToChar` returns a byte offset equal to the length of the text segment.

When the pixel location is *before* the leading edge of the first glyph in the displayed text segment, `PixelToChar` returns a leading edge value of `FALSE` and the byte offset of the first character. When the pixel location is *after* the trailing edge of the last glyph in the displayed text segment, `PixelToChar` returns a leading edge value of `TRUE` and the next byte offset in memory, the one after the last character in the text segment. If the primary line direction is left to right, *before* means to the left of all of the glyphs for the characters in the text segment, and *after* means to the right of all these glyphs. If the primary line direction is right to left, before and after hold the opposite meanings.

You also use the value of the `leadingEdge` flag to help determine the value of the `direction` parameter to pass to `CharToPixel`, which you call to get the caret position. If the `leadingEdge` flag is `FALSE`, you base the value of the `direction` parameter on the direction of the character at the byte offset in memory that precedes the one that `PixelToChar` returns; if `leadingEdge` is `TRUE`, you base the value of the `direction` parameter on the direction of the character at the byte offset that `PixelToChar` returns. If there isn't a character at the byte offset, you base the value of the `direction` parameter on the primary line direction as determined by the `SysDirection` global variable.

You specify a value for `textLen` that is equal to the entire visible part of the style run on a line and includes trailing spaces if and only if they are displayed. They may not be displayed, for example, for the last style run in memory order that is part of the current line.

For more information about the scaling factors, see "The numer and denom Parameters" on page 3-68. See "The styleRunPosition Parameter" on page 3-67 for a description of the `styleRunPosition` parameter and the values it takes. See "The slop Parameter" on page 3-67 for more information about the `slop` parameter.

Note
Be sure to pass the same values for `styleRunPosition` and the scaling factors (`numer` and `denom`) to `PixelToChar` that you pass to any of the other justification routines for this style run. ◆

You pass `PixelToChar` a pointer to the byte offset of the character in the text buffer that begins the text segment or style run containing the character whose glyph is at the pixel location. If you do not know which style run on the display line contains the character whose glyph is at the pixel location, you can loop through the style runs until you find the one that contains the pixel location. If the style run contains the character, `PixelToChar` returns its byte offset. If it doesn't, you can use the `widthRemaining` parameter value to help determine which style run contains the glyph at the pixel location.

If you pass PixelToChar the pixel width of the display line, you can use the returned value of widthRemaining to calculate the length of a style run. The widthRemaining parameter contains the length in pixels from the end of the style run for which you call PixelToChar to the end of the display line, in this case, if the style run for which you call it does not include the byte offset whose glyph corresponds to the pixel location. You subtract the returned widthRemaining value from the screen pixel width of the display line to get the style run's length.

To truncate a line of text, you can use PixelToChar to find the byte offset of the character where the line should be broken. To return the correct byte offset associated with the pixel location of a mouse-down event when the text belongs to a right-to-left script system, the PixelToChar function reorders the text. If right-to-left text is reordered when you use PixelToChar to determine where to break a line, it returns the wrong byte offset. To get the correct result, you must turn off reordering before you call PixelToChar. Remember to restore reordering after you have determined where to break the line. See "Using Scaled Text" beginning on page 3-44 for more information.

SPECIAL CONSIDERATIONS

The PixelToChar function works with text in all script systems, and for text that is justified or not. For contextual script systems, PixelToChar takes into account the widths of any ligatures, reversals, and compound characters that were created when the text was drawn.

Because 2-byte script systems also include characters consisting of only one byte, you should not simply multiply the number of characters by 2 to determine this value; you must determine and specify the correct number of bytes.

The PixelToChar function may move memory; do not call this procedure at interrupt time.

CharToPixel

The CharToPixel function returns the screen pixel width from the left edge of a text segment to the glyph of the character whose byte offset you specify.

```
FUNCTION CharToPixel (textBuf: Ptr; textLen: LongInt;
                      slop: Fixed; offset: LongInt;
                      direction: Integer;
                      styleRunPosition: JustStyleCode;
                      numer: Point; denom: Point): Integer;
```

textBuf	A pointer to the beginning of the text segment.

`textLen`	The length in bytes of the entire text segment pointed to by `textBuf`. The `CharToPixel` function requires the context of the complete text in order to determine the correct value.
`slop`	The amount of slop for the text to be drawn. A positive value extends the text segment; a negative value condenses the text segment.
`offset`	The offset from `textBuf` to the character within the text segment whose display pixel location is to be measured. For 2-byte script systems, if the character whose position is to be measured is 2 bytes long, this is the offset of the first byte.
`direction`	This parameter specifies whether `CharToPixel` is to return the caret position for a character with a direction of left-to-right or right-to-left. A direction value of `hilite` indicates that `CharToPixel` is to use the caret position for the character direction that matches the primary line direction as specified by the `SysDirection` global variable.
`styleRunPosition`	The position on the line of this style run. The style run can be the only one on the line, the leftmost on the line, the rightmost on the line, or one between two other style runs.
`numer`	A point giving the numerator for the horizontal and vertical scaling factors.
`denom`	A point giving the denominator for the horizontal and vertical scaling factors.

DESCRIPTION

You use `CharToPixel` to find the onscreen pixel location at which to draw a caret and to identify the selection points for highlighting. The `CharToPixel` function returns the horizontal distance in pixels from the start of the range of text beginning with the byte offset at `textBuf` to the glyph corresponding to the character whose byte offset is specified by the `offset` parameter. The pixel location is relative to the beginning of the text segment, not the left margin of the display line. To get the actual display line pixel location of the glyph relative to the left margin, you add the pixel value that `CharToPixel` returns to the pixel location at the end of the previous style run (on the left) in display order. In other words, you need to know the length of the text in pixels on the display line up to the beginning of the range of text that you call `CharToPixel` for, and then you add in the screen pixel width that `CharToPixel` returns.

You specify a value for `textLen` that is equal to the entire visible part of the style run on a line and includes trailing spaces if and only if they are displayed. They may not be displayed, for example, for the last style run in memory order, which is part of the line. Do not confuse the `textLen` parameter with the `offset`, which is the byte offset of the character within the text segment whose pixel location `CharToPixel` is to return.

If you use `CharToPixel` to get a caret position to mark the insertion point, you specify a value of `leftCaret` or `rightCaret` for the `direction` parameter. You can use the value of the `PixelToChar leadingEdge` flag to determine the `direction` parameter value.

If the `leadingEdge` flag is `FALSE`, you base the value of the `direction` parameter on the direction of the character at the byte offset in memory that precedes the one that `PixelToChar` returns; if `leadingEdge` is `TRUE`, you base the value of the direction parameter on the direction of the character at the byte offset that `PixelToChar` returns. If there isn't a character at the byte offset, you base the value of the `direction` parameter on the primary line direction as determined by the `SysDirection` global variable.

You can use the following constants to specify a value for `direction`.

Constant	Value	Meaning
`leftCaret`	0	Place caret for left-to-right text direction.
`rightCaret`	–1	Place caret for right-to-left text direction.
`hilite`	1	Specifies that the caret position should be determined according to the primary line direction, based on the value of `SysDirection`.

For more information about the scaling factors, see "The numer and denom Parameters" on page 3-68. See "The styleRunPosition Parameter" on page 3-67 for a description of the `styleRunPosition` parameter and the values it takes. See "The slop Parameter" on page 3-67 for more information about the `slop` parameter.

Note
Be sure to pass the same values for `styleRunPosition` and the scaling factors (`numer` and `denom`) to `CharToPixel` that you pass to any of the other justification routines for this style run. ◆

For more information about `CharToPixel` see "Drawing Carets and Highlighting" on page 3-47.

SPECIAL CONSIDERATIONS

The `CharToPixel` function works with text in all script systems. For 1-byte contextual script systems, `CharToPixel` calculates the widths of any ligatures, reversals, and compound characters that need to be drawn.

Note that `textLen` is the number of *bytes* to be drawn, not the number of characters. Because 2-byte script systems also include characters consisting of only one byte, do not simply multiply the number of characters by 2 to determine this value; you must determine and specify the correct number of bytes.

The `CharToPixel` function may move memory; do not call this procedure at interrupt time.

HiliteText

The `HiliteText` procedure finds all the characters between two byte offsets in a text segment whose glyphs are to be highlighted.

```
PROCEDURE HiliteText (textPtr: Ptr;
                      textLen, firstOffset, secondOffset: Integer;
                      VAR offsets: OffsetTable);
```

`textPtr`	A pointer to a buffer that contains the text to be highlighted.
`textLen`	The length in bytes of the entire text segment pointed to by `textPtr`.
`firstOffset`	The byte offset from `textPtr` to the first character to be highlighted.
`secondOffset`	The byte offset from `textPtr` to the last character to be highlighted.
`offsets`	A table that, upon completion of the call, specifies the boundaries of the text to be highlighted.

DESCRIPTION

The `HiliteText` procedure returns three pairs of byte offsets that mark the onscreen ranges of text to be highlighted. This is because for bidirectional text, although the characters are contiguous in memory, their displayed glyphs can include up to three separate ranges of text.

The `HiliteText` procedure takes into account the fact that to highlight the complete range of text whose beginning and ending byte offsets you pass it, it must return byte offsets that encompass the glyphs of the first and last characters in the text segment. To determine the correct offset pairs, `HiliteText` relies on the primary line direction as specified by the `SysDirection` global variable.

Before calling `HiliteText`, you must set up an offset table (of type `OffsetTable`) in your application to hold the results. You can consider the offset table a set of three offset pairs:

```
TYPE OffPair =
   RECORD
      offFirst:   Integer;
      offSecond:  Integer;
   END;

   OffsetTable = ARRAY [0..2] of OffPair;
```

If the two offsets in any pair are equal, the pair is empty and you can ignore it. Otherwise the pair identifies a run of characters whose glyphs are to be highlighted.

SPECIAL CONSIDERATIONS

The offsets that `HiliteText` returns depend on the primary line direction as defined by the `SysDirection` global variable. If you change the value of `SysDirection`, `HiliteText` returns the offset that is meaningful according to the primary line direction for ambiguous offsets on the boundary of right-to-left and left-to-right text.

The `HiliteText` procedure may move memory; do not call this procedure at interrupt time. For more information, see "Highlighting a Text Selection" on page 3-60.

Low-Level QuickDraw Text Routines

The QuickDraw text routines use two bottleneck routines extensively—one to draw text, and one to measure it. This section describes the `StdText` procedure that is used to draw text and the `StdTxMeas` function that is used to measure text. Although the high-level QuickDraw text routines provide most of the functionality needed to measure and draw text under most circumstances, you can call these low-level routines directly when necessary. However, if you need to call either `StdText` or `StdTxMeas` directly, you must first check the graphics port `grafProc` field to determine whether the bottleneck routines have been customized, and if so, you must call the customized routine instead of the standard one. The bottleneck routines are always customized for printing.

If the `grafProcs` field contains `NIL`, the standard bottleneck routines have *not* been customized. If the `grafProcs` field contains a pointer, the standard bottleneck routines have been replaced by customized ones. A pointer (of type `QDProcsPtr`) in the `grafProc` field points to a `QDProc` record. This record contains fields that point to the bottleneck routine to be used for a specific drawing function. If the standard bottleneck routine has been customized, your application needs to use the customized routine indicated by the `QDProcs` record field.

The QuickDraw standard low-level bottleneck routines work properly for all script systems. For more information about replacing or customizing the bottleneck routines, see "Customizing QuickDraw's Text Handling" on page 3-62 and the QuickDraw chapters in *Inside Macintosh: Imaging*.

StdText

The `StdText` procedure is the standard low-level routine for drawing text. It draws text from an arbitrary structure in memory specified by `textBuf`, starting from the first byte and continuing for `count` bytes.

```
PROCEDURE StdText (count: Integer; textBuf: Ptr;
                   numer, denom: Point);
```

`count`	The number of bytes to be counted.
`textBuf`	A pointer to the beginning of the text in memory.

`numer`	A point giving the numerator for the horizontal and vertical scaling factors.
`denom`	A point giving the denominator for the horizontal and vertical scaling factors.

DESCRIPTION

The `StdText` procedure is a QuickDraw bottleneck routine that the QuickDraw text drawing routines use extensively. However, you can call the `StdText` routine directly to draw text that is scaled or unscaled. For more information about the scaling factors, see "The numer and denom Parameters" on page 3-68.

SPECIAL CONSIDERATIONS

The `StdText` procedure gives the correct results for all script systems. The `count` parameter is the number of bytes of the text to be drawn, not characters. When specifying this value, consider that 2-byte script systems also include characters consisting of only one byte.

StdTxMeas

The `StdTxMeas` function measures the width of scaled or unscaled text.

```
FUNCTION StdTxMeas (byteCount: Integer; textAddr: Ptr;
                    VAR numer, denom: Point;
                    VAR info: FontInfo): Integer;
```

`byteCount`	The number of bytes to be counted.
`textAddr`	A pointer to the beginning of the text in memory.
`numer`	A point giving the numerator for the horizontal and vertical scaling factors.
`denom`	A point giving the denominator for the horizontal and vertical scaling factors.
`info`	A font information record that describes the current font.

DESCRIPTION

The `StdTxMeas` function is a QuickDraw bottleneck routine that the QuickDraw text-measuring routines use extensively. The `StdTxMeas` function returns the width of the text stored in memory beginning with the first character at `textAddr` and continuing for `byteCount` bytes. You can call the `StdTxMeas` function directly, for example, to measure text that you want to explicitly scale, but not justify. You can also use `StdTxMeas` to get the font metrics for scaled text in order to determine the line height, instead of using `GetFontInfo`, which doesn't support scaling.

On input, you need to specify values for `numer` and `denom`, even if you are not scaling the text. You can specify 1,1 scaling factors, in this case, so that no scaling is applied. On return, `numer` and `denom` contain the additional scaling to be applied to the text. For more information about the input scaling factors, see "The numer and denom Parameters" on page 3-68.

The `StdTxtMeas` function returns output scaling factors that you need to apply to the text to get the right measurement if the Font Manager was not able to fully satisfy the scaling request. You can use the Toolbox Utilities' `FixRound` and `FixRatio` functions to help with this process. For more information, see "Using Scaled Text" on page 3-44.

SPECIAL CONSIDERATIONS

The `StdTxMeas` routine gives the correct results for all script systems. The `byteCount` parameter is the number of bytes of the text to be drawn, not characters. When specifying this value, consider that 2-byte script systems also include characters consisting of only one byte.

Application-Supplied Routine

One of the QuickDraw text routines, `GetFormatOrder`, requires an application-supplied routine, which is described in this section.

MyRlDirProc

The `MyRlDirProc` function is a callback routine that you supply for use by the `GetFormatOrder` procedure. The `MyRlDirProc` function is a Boolean function that calculates, for a style run identified by number, the direction of that style run. Your routine returns `TRUE` for right-to-left text direction, `FALSE` for left-to-right. `MyRlDirProc` is pointed to by the `rlDirProc` parameter of `GetFormatOrder`.

```
FUNCTION MyRlDirProc (theFormat: Integer;
                      dirParam: Ptr): Boolean;
```

`theFormat`	A value that identifies the style run whose direction is needed.
`dirParam`	A pointer to an application-defined parameter block that contains the font and script information for each style run in the text. The contents of this parameter block are used to determine the direction of the style run. Because of the relationship between the font family ID and the script code, the font family ID can be used to determine the text direction.

DESCRIPTION

To fill the ordering array (type `FormatOrder`) for style runs on a line, the `GetFormatOrder` procedure calls `MyRlDirProc` for each style run numbered from `firstFormat` to `lastFormat`. `GetFormatOrder` passes `MyRlDirProc` a number identifying the style run in storage order, and a pointer to the parameter information block, `dirParam`, that contains the font and style information for the style run. Given `dirParam` and a style run identifier, the application-defined `MyRlDirProc` routine should be able to determine the style run direction.

You should store your style run information in a way that makes it convenient for `MyRLDirProc`. One obvious way to do this is to declare a record type for style runs that allows you to save things like font style, font family ID, script number, and so forth. You then can store these records in an array. When the time comes for `GetFormatOrder` to fill the ordering array, `MyRlDirProc` can consult the style run array for direction information for each of the numbered style runs in turn.

For more information, see “GetFormatOrder” on page 3-87.

Summary of QuickDraw Text

Pascal Summary

Constants

```
CONST
   {CharToPixel directions}
   leftCaret   =  0;    {Place caret for left block}
   rightCaret  =  -1;   {Place caret for right block}
   hilite      =  1;    {Direction is SysDirection}

   {constants for styleRunPosition parameter in PortionLine, DrawJustified, }
   { MeasureJustified, CharToPixel, and PixelToChar}
   onlyStyleRun   =  0;    {This is the only style run on the line.}
   leftStyleRun   =  1;    {This is the leftmost of multiple style runs on }
                           { the line.}
   rightStyleRun  =  2;    {This is the rightmost of multiple style runs }
                           {  on the line.}
   middleStyleRun =  3;    {There are multiple style runs on the line }
                           {  and this one is interior; neither }
                           {  leftmost nor rightmost.}
```

Data Types

```
TYPE
   {Type declaration for GetFontInfo info VAR parameter}
   FontInfo =  RECORD
      ascent:  Integer;
      descent: Integer;
      widMax:  Integer;
      leading: Integer;
      END;
```

```
{GetFormatOrder ordering array}
FormatOrder = ARRAY  [0..0] OF Integer;
FormatOrderPtr =  ^FormatOrder;
FormatStatus   =  Integer;

{Type declaration for TextFace face parameter}
StyleItem   =  (bold,italic,underline,outline,shadow,condense,extend);
Style       =  SET OF StyleItem;
```

Routines

Setting Text Characteristics

```
PROCEDURE TextFont          (font: Integer);
PROCEDURE TextFace          (face: Style);
PROCEDURE TextMode          (mode: Integer);
PROCEDURE TextSize          (size: Integer);
PROCEDURE SpaceExtra        (extra: Fixed);
PROCEDURE CharExtra         (extra: Fixed);
PROCEDURE GetFontInfo       (VAR info: FontInfo);
```

Drawing Text

```
PROCEDURE DrawChar          (ch: CHAR);
PROCEDURE DrawString        (s: Str255);
PROCEDURE DrawText          (textBuf: Ptr; firstByte, byteCount: Integer);
PROCEDURE DrawJustified     (textPtr: Ptr; textLength: LongInt;slop: Fixed;
                             styleRunPosition: JustStyleCode;
                             numer: Point; denom: Point);
```

Measuring Text

```
FUNCTION CharWidth          (ch: CHAR): Integer;
FUNCTION StringWidth        (s: Str255) : Integer;
FUNCTION TextWidth          (textBuf: Ptr;
                             firstByte, byteCount: Integer): Integer;
PROCEDURE MeasureText       (count: Integer; textAddr, charLocs: Ptr);
PROCEDURE MeasureJustified  (textPtr: Ptr; textLength: LongInt;
                             slop: Fixed; charLocs: Ptr;
                             styleRunPosition: JustStyleCode;
                             numer: Point; denom: Point);
```

Laying Out a Line of Text

```
PROCEDURE GetFormatOrder    (ordering: FormatOrderPtr; firstFormat: Integer;
                             lastFormat: Integer; lineRight: Boolean;
                             rlDirProc: Ptr;dirParam: Ptr);
FUNCTION VisibleLength      (textPtr: Ptr; textLength: LongInt): LongInt;
FUNCTION PortionLine        (textPtr: Ptr; textLen: LongInt;
                             styleRunPosition: JustStyleCode;
                             numer: Point; denom: Point): Fixed;
```

Determining the Caret Position, and Selecting and Highlighting Text

```
FUNCTION PixelToChar        (textBuf: Ptr; textLen: LongInt;
                             slop: Fixed; pixelWidth: Fixed;
                             VAR leadingEdge: Boolean;
                             VAR widthRemaining: Fixed;
                             styleRunPosition: JustStyleCode;
                             numer, denom: Point): Integer;
FUNCTION CharToPixel        (textBuf: Ptr; textLen: LongInt;
                             slop: Fixed; offset: LongInt;
                             direction: Integer;
                             styleRunPosition: JustStyleCode;
                             numer: Point; denom: Point): Integer;
PROCEDURE HiliteText        (textPtr: Ptr;textLength, firstOffset,
                             secondOffset: Integer;
                             VAR offsets: OffsetTable);
```

Low-Level QuickDraw Text Routines

```
PROCEDURE StdText           (count: Integer; textAddr: Ptr;
                             numer, denom: Point);
FUNCTION StdTxMeas          (byteCount: Integer; textAddr: Ptr;
                             VAR numer, denom: Point;
                             VAR info: FontInfo): Integer;
```

Application-Supplied Routine

```
FUNCTION MyRlDirProc        (theFormat: Integer; dirParam: Ptr) Boolean;
```

C Summary

Constants

```
enum{
   /*CharToPixel directions*/
   leftCaret   =  0,    /*Place caret for left block*/
   rightCaret  =  -1,   /*Place caret for right block*/
   hilite      =  1,    /*Direction is SysDirection*/

   /*constants for styleRunPosition parameter in PortionLine,*/
   /*DrawJustified, MeasureJustified, CharToPixel, and PixelToChar*/
   onlyStyleRun   =  0,    /*This is the only style run on the line.*/
   leftStyleRun   =  1,    /*This is the leftmost of multiple style */
                           /*runs on the line.*/
   rightStyleRun  =  2,    /*This is the rightmost of multiple style runs */
                           /* on the line.*/
   middleStyleRun =  3,    /*There are multiple style runs on the line */
                           /* and this one is interior: neither */
                           /* leftmost nor rightmost.*/
```

Types

```
TYPE
   /*Type declaration for GetFontInfo info VAR parameter*/
   struct FontInfo {
      short ascent;
      short descent;
      short widMax;
      short leading;
   };
   typedef struct FontInfo FontInfo;

   /*GetFormatOrder ordering array*/
   typedef short FormatOrder[1];
   typedef FormatOrder *FormatOrderPtr;
   typedef short FormatStatus;

   /*Type declaration for TextFace face parameter*/
   ??StyleItem =  (bold,italic,underline,outline,shadow,condense,extend);
   Style       =  SET OF StyleItem;??
```

Routines

Setting Text Characteristics

```
pascal void TextFont        (short font);
pascal void TextFace        (short face);
pascal void TextMode        (short mode);
pascal void TextSize        (short size);
pascal void SpaceExtra      (extra: Fixed);
pascal void CharExtra       (Fixed extra);
pascal void GetFontInfo     (FontInfo *info);
```

Drawing Text

```
pascal void DrawChar        (short ch);
pascal void DrawString      (ConstStr255Param s);
pascal void DrawText        (const void *textBuf, short firstByte,
                             short byteCount);
pascal void DrawJustified   (Ptr textPtr, long textLength, Fixed slop,
                             JustStyleCode styleRunPosition,
                             Point numer, Point denom);
```

Measuring Text

```
pascal short CharWidth      (short ch);
pascal short StringWidth    (ConstStr255Param s);
pascal short TextWidth      (const void *textBuf, short firstByte,
                             short byteCount);
pascal void MeasureText     (short count, const void *textAddr,
                             void *charLocs);
pascal void MeasureJustified
                            (Ptr textPtr, long textLength, Fixed slop,
                             Ptr charLocs, JustStyleCode styleRunPosition,
                             Point numer, Point denom);
```

Laying Out a Line of Text

```
pascal void GetFormatOrder  (FormatOrderPtr ordering,
                             short firstFormat, short lastFormat,
                             Boolean lineRight,
                             Ptr rlDirProc, Ptr dirParam);
pascal long VisibleLength   (Ptr textPtr,long textLen);
pascal Fixed PortionLine    (Ptr textPtr, long textLen, JustStyleCode
                             styleRunPosition, Point numer, Point denom);
```

Determining the Caret Position, and Selecting and Highlighting Text

```
pascal short PixelToChar    (Ptr textBuf, long textLen, Fixed slop,
                             Fixed pixelWidth, Boolean *leadingEdge,
                             Fixed *widthRemaining, JustStyleCode
                             styleRunPosition, Point numer, Point denom);
pascal short CharToPixel    (Ptr textBuf, long textLen, Fixed slop,
                             long offset, short direction, JustStyleCode
                             styleRunPosition, Point numer, Point denom);
pascal void HiliteText      (Ptr textPtr, short textLength,
                             short firstOffset, short secondOffset,
                             OffsetTable offsets);
```

Low-Level QuickDraw Text Routines

```
pascal void StdText         (short count, const void *textAddr,
                             Point numer, Point denom);
pascal short StdTxMeas      (short byteCount, const void *textAddr,
                             Point *numer, Point *denom, FontInfo *info);
```

Application-Supplied Routine

```
pascal Boolean MyRlDirProc  (short theFormat,Ptr dirParam);
```

Assembly-Language Summary

Trap Macros

Trap Macro Names

Pascal name	Trap macro name
DrawJustified	_DrawJustified
MeasureJustified	_MeasureJustified
GetFormatOrder	_GetFormatOrder
VisibleLength	_VisibleLength
PortionLine	_PortionLine
CharToPixel	_CharToPixel
PixelToChar	_PixelToChar
HiliteText	_HiliteText

Trap Macros With Trap Words

Trap macro name	Trap word
`_MeasureText`	$A837
`_StdText`	$A882
`_DrawChar`	$A883
`_DrawString`	$A884
`_DrawText`	$A885
`_TextWidth`	$A886
`_TextFont`	$A887
`_TextFace`	$A888
`_TextMode`	$A889
`_TextSize`	$A88A
`_GetFontInfo`	$A88B
`_StringWidth`	$A88C
`_CharWidth`	$A88D
`_SpaceExtra`	$A88E
`_StdTxMeas`	$A8ED
`_CharExtra`	$AA23

Global Variables

`HiliteMode`	Flag used for color highlighting
`SysDirection`	System direction; the primary line direction and alignment for text
`thePort`	The currently active graphics port