QuickDraw Text Reference

(Not Recommended)

Carbon > Text & Fonts

2006-07-13

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

QuickDraw Text Reference (Not Recommended)

Framework: ApplicationServices/ApplicationServices.h

Declared in QuickdrawText.h

Important: The information in this document is obsolete and should not be used for new development.

Overview

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.

To understand the routines described in this document, it is helpful to be familiar with the other parts of QuickDraw. It is also helpful to be familiar with the Font Manager, because of the close relationship between QuickDraw and the Font Manager. To understand the tasks involved in text layout, you should also be acquainted with the Text Utilities and the Script Manager.

Carbon supports the majority of QuickDraw Text.

Functions by Task

Determining the Caret Position, and Selecting and Highlighting Text

`CharToPixel` (page 18) Deprecated in Mac OS X v10.4

Returns the screen pixel width from the left edge of a text segment to the glyph of the character whose byte offset you specify. (Deprecated. Use ATSUI instead.)

`HiliteText` (page 28) Deprecated in Mac OS X v10.4

Finds all the characters between two byte offsets in a text segment whose glyphs are to be highlighted. (Deprecated. Use ATSUI instead.)

`PixelToChar` (page 34) Deprecated in Mac OS X v10.4

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. (Deprecated. Use ATSUI instead.)

Drawing Text

`DrawChar` (page 21) Deprecated in Mac OS X v10.4

Draws the glyph for a single 1-byte character at the current pen location in the current graphics port. (Deprecated. Use ATSUI or Quartz instead.)

`DrawJustified` (page 22) Deprecated in Mac OS X v10.4

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. (Deprecated. Use ATSUI instead.)

`DrawString` (page 24) Deprecated in Mac OS X v10.4

Draws the text of the specified Pascal string at the pen location in the current graphics port (`GrafPort` or `CGrafPort`). (Deprecated. Use ATSUI or Quartz instead.)

`DrawText` (page 25) Deprecated in Mac OS X v10.4

Draws the specified text at the current pen location in the current graphics port. (Deprecated. Use ATSUI or Quartz instead.)

`StandardGlyphs` (page 40) Deprecated in Mac OS X v10.4

This obsolete function doesn't do anything in Mac OS X. (Deprecated. Use ATSUI to render Unicode text.)

`StdText` (page 40) Deprecated in Mac OS X v10.4

Draws text from an arbitrary structure in memory. (Deprecated. Use ATSUI or Quartz instead.)

`stdtext` (page 41) Deprecated in Mac OS X v10.4

Draws text from an arbitrary structure in memory. (Deprecated. Use ATSUI or Quartz instead.)

Laying Out a Line of Text

`GetFormatOrder` (page 27) Deprecated in Mac OS X v10.4

Determines the display order of style runs for a line of text containing multiple style runs with mixed directions. (Deprecated. Use ATSUI instead.)

`PortionLine` (page 37) Deprecated in Mac OS X v10.4

Determines the correct proportion of extra space to apply to the specified style run in a line of justified text; that is, how to distribute the total slop value for a line among the style runs on that line. (Deprecated. Use ATSUI instead.)

`VisibleLength` (page 51) Deprecated in Mac OS X v10.4

Calculates the length, in bytes, of a given text segment, excluding trailing white space. (Deprecated. Use ATSUI instead.)

Measuring Text

`CharWidth` (page 20) Deprecated in Mac OS X v10.4

Returns the width (horizontel extension), in pixels, of the specified character. (Deprecated. Use ATSUI instead.)

`MeasureJustified` (page 30) Deprecated in Mac OS X v10.4

Calculates, for text that is expanded, condensed, or scaled, the onscreen width in pixels from the left edge of the text segment to the glyph of the character. (Deprecated. Use ATSUI instead.)

`MeasureText` (page 32) Deprecated in Mac OS X v10.4
: Calculates the width of the character's glyph in pixels from the left edge of the text segment. (Deprecated. Use ATSUI instead.)

`StdTxMeas` (page 41) Deprecated in Mac OS X v10.4
: Measures the width of scaled or unscaled text. (Deprecated. Use ATSUI instead.)

`StringWidth` (page 43) Deprecated in Mac OS X v10.4
: Returns the length, in pixels, of the specified text string. (Deprecated. Use ATSUI instead.)

`TextWidth` (page 48) Deprecated in Mac OS X v10.4
: Returns the length, in pixels, of the specified text. (Deprecated. Use ATSUI instead.)

Setting Text Characteristics

`CharExtra` (page 17) Deprecated in Mac OS X v10.4
: Specifies, for a color graphics port (`CGrafPort`), the number of pixels by which to widen (or narrow) the glyphs of each nonspace character in a style run. (Deprecated. Use ATSUI instead.)

`GetFontInfo` (page 26) Deprecated in Mac OS X v10.4
: Returns information about the current graphics port's font, taking into account the style and size in which the glyphs are to be drawn. (Deprecated. Use ATSUI instead.)

`SpaceExtra` (page 39) Deprecated in Mac OS X v10.4
: 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. (Deprecated. Use ATSUI instead.)

`TextFace` (page 46) Deprecated in Mac OS X v10.4
: Sets the style of the font in which the text is to be drawn in the current graphics port. (Deprecated. Use ATSUI or Quartz instead.)

`TextFont` (page 46) Deprecated in Mac OS X v10.4
: Sets the font of the current graphics port in which the text is to be rendered. (Deprecated. Use ATSUI or Quartz instead.)

`TextMode` (page 47) Deprecated in Mac OS X v10.4
: Sets the transfer mode for drawing text in the current graphics port. (Deprecated. Use ATSUI or Quartz instead.)

`TextSize` (page 48) Deprecated in Mac OS X v10.4
: Sets the font size for text drawn in the current graphics port to the specified number of points. (Deprecated. Use ATSUI or Quartz instead.)

Truncating Strings and Breaking Lines

`StyledLineBreak` (page 44) Deprecated in Mac OS X v10.4
: Returns the proper location to break a line of text, taking into account script and language considerations, making use of tables in the string-manipulation (`'itl2'`) resource in its computations. (Deprecated. Use ATSUI instead.)

`TruncString` (page 50) Deprecated in Mac OS X v10.4
: Ensures that a Pascal string fits into the specified pixel width, by truncating the string as necessary. This function makes use of the current script and font. (Deprecated. Use CFString instead.)

`TruncText` (page 50) Deprecated in Mac OS X v10.4

Ensures that a text string fits into the specified pixel width, by truncating the string as necessary. This function makes use of the current script and font. (Deprecated. Use CFString instead.)

Working With Universal Procedure Pointers

`DisposeStyleRunDirectionUPP` (page 21) Deprecated in Mac OS X v10.4

Disposes of a universal procedure pointer (UPP) to a style run direction callback. (Deprecated. Use ATSUI to handle style runs.)

`InvokeStyleRunDirectionUPP` (page 29) Deprecated in Mac OS X v10.4

Calls your style run direction callback. (Deprecated. Use ATSUI to handle style runs.)

`NewStyleRunDirectionUPP` (page 34) Deprecated in Mac OS X v10.4

Creates a new universal procedure pointer (UPP) to a style run direction callback. (Deprecated. Use ATSUI to handle style runs.)

Callbacks

StyleRunDirectionProcPtr

Defines a pointer to a style run direction callback function that calculates, for a style run identified by number, the direction of that style run.

```
typedef Boolean (*StyleRunDirectionProcPtr)
(
    short styleRunIndex,
    void * dirParam
);
```

If you name your function `MyStyleRunDirectionProc`, you would declare it like this:

```
Boolean StyleRunDirectionProcPtr (
    short styleRunIndex,
    void * dirParam
);
```

Parameters

`styleRunIndex`

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.

Return Value

A Boolean value that is `TRUE` for right-to-left text direction, `FALSE` for left-to-right.

Discussion
To fill the ordering array (type `FormatOrder`) for style runs on a line, the `GetFormatOrder` function calls `MyStyleRunDirectionCallback` for each style run numbered from `firstFormat` to `lastFormat`. `GetFormatOrder` passes `MyStyleRunDirectionCallback` 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 `MyStyleRunDirectionCallback` function should be able to determine the style run direction.

You should store your style run information in a way that makes it convenient for `MyStyleRunDirectionCallback`. One obvious way to do this is to declare a structure 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 structures in an array. When the time comes for `GetFormatOrder` to fill the ordering array, `MyStyleRunDirectionCallback` can consult the style run array for direction information for each of the numbered style runs in turn.

For more information, see `GetFormatOrder` (page 27).

When you provide the Component Manager with a pointer to your function, you should use a universal procedure pointer (UPP). The definition of the UPP data type for your file identification function is as follows:

```
typedef (StyleRunDirectionProcPtr) StyleRunDirectionUPP;
```

Before using your style run direction callback function, you must first create a new universal procedure pointer to it, using the `NewStyleRunDirectionUPP` function, as shown here:

```
StyleRunDirectionUPP MyStyleRunDirectionUPP;

MyStyleRunDirectionUPP = StyleRunDirectionUPP(&MyStyleRunDirectionCallback)
```

You then pass `MyStyleRunDirectionUPP` to the function `GetFormatOrder`. If you wish to call your own callback function, you can use the `InvokeStyleRunDirectionUPP` function:

```
direction = InvokeStyleRunDirectionUPP(styleRunIndex, &paramInfo,
MyStyleRunDirectionUPP)
```

When you are finished using your callback function, you should dispose of the universal procedure pointer associated with it, using the `DisposeStyleRunDirectionUPP` function.

```
DisposeStyleRunDirectionUPP(MyStyleRunDirectionUPP);
```

Availability
Available in Mac OS X v10.0 and later.

Declared In
`QuickdrawText.h`

Data Types

FontInfo

Contains font metric information.

```
struct FontInfo {
    short ascent;
    short descent;
    short widMax;
    short leading;
};
typedef struct FontInfo FontInfo;
```

Fields

`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.

Discussion

The `FontInfo` data type defines a font information structure. The `GetFontInfo` (page 26) function uses the font information structure 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 structure contains the ascent, the descent, the width of the largest glyph, and the leading for a given font. The `StdTxMeas` (page 41) function also uses a structure of type `FontInfo` to return information about the current font.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`QuickdrawText.h`

FormatOrder

Contains and array of display orders for style runs.

```
typedef  FormatOrder[1];
```

Discussion

The `GetFormatOrder` (page 27) function fills the supplied format order array with the display order of each style run.

Availability

Available in Mac OS X v10.0 and later.

Declared In

`QuickdrawText.h`

StyleRunDirectionUPP

Defines a universal procedure pointer to a style run direction callback.

```
typedef StyleRunDirectionProcPtr StyleRunDirectionUPP;
```

Discussion
For more information, see the description of the `StyleRunDirectionProcPtr` (page 8) callback function.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`QuickdrawText.h`

Constants

Caret Direction Constants

Specify a caret position.

```
enum {
    leftCaret = 0,
    rightCaret = -1,
    kHilite = 1
};
```

Constants

`leftCaret`
Place caret for left-to-right text direction.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`rightCaret`
Place caret for right-to-left text direction.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`kHilite`
Specifies that the caret position should be determined according to the primary line direction, based on the value of `SysDirection`.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

Discussion
You can use these constants to specify a value for `direction`, as used in the `CharToPixel` (page 18) function.

Truncation Status Values

Returned as result codes for the functions `TruncString` and `TruncText`.

```
enum {
    notTruncated = 0,
    truncated = 1,
    truncErr = -1,
    smNotTruncated = 0,
    smTruncated = 1,
    smTruncErr = -1
};
```

Constants

`notTruncated`
Specifies that truncation is not necessary.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`truncated`
Specifies that truncation was performed.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`truncErr`
Specifies a general error occurred.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`smNotTruncated`
Specifies that truncation is not necessary. This is obsolete.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`smTruncated`
Specifies that truncation was performed. This is obsolete.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`smTruncErr`
Specifies a general error occurred. This is obsolete.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

Style Line Break Values

Specify a line break.

```
typedef SInt8 StyledLineBreakCode;
enum {
    smBreakWord = 0,
    smBreakChar = 1,
    smBreakOverflow = 2
};
```

Constants

`smBreakWord`

Available in Mac OS X v10.0 and later.

Declared in `QuickdrawText.h`.

`smBreakChar`

Available in Mac OS X v10.0 and later.

Declared in `QuickdrawText.h`.

`smBreakOverflow`

Available in Mac OS X v10.0 and later.

Declared in `QuickdrawText.h`.

Obsolete Caret Placement Values

Specify where to place a caret.

```
enum {
    smLeftCaret = 0,
    smRightCaret = -1,
    smHilite = 1
};
```

Constants

`smLeftCaret`

Specifies to place caret for left block. This is obsolete.

Available in Mac OS X v10.0 and later.

Declared in `QuickdrawText.h`.

`smRightCaret`

Specifies to place caret for right block. This is obsolete.

Available in Mac OS X v10.0 and later.

Declared in `QuickdrawText.h`.

`smHilite`

Specifies the direction is `TESysJust`. This is obsolete.

Available in Mac OS X v10.0 and later.

Declared in `QuickdrawText.h`.

Style Run Position Constants

Specify style run positions.

```
typedef short JustStyleCode;
enum {
    onlyStyleRun = 0,
    leftStyleRun = 1,
    rightStyleRun = 2,
    middleStyleRun = 3,
    smOnlyStyleRun = 0,
    smLeftStyleRun = 1,
    smRightStyleRun = 2,
    smMiddleStyleRun = 3
};
```

Constants

`onlyStyleRun`
This is the only style run on the line.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`leftStyleRun`
This is the leftmost of multiple style runs on the line.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`rightStyleRun`
This is the rightmost of multiple style runs on the line
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`middleStyleRun`
The line and this one is interior: neither leftmost nor rightmost.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`smOnlyStyleRun`
This is obsolete.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`smLeftStyleRun`
This is obsolete.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`smRightStyleRun`
This is obsolete.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`smMiddleStyleRun`
This is obsolete.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

Discussion
Use one of the following constants (defined as type `JustStyleCode`) in the `styleRunPosition` parameter for `PortionLine` (page 37) `DrawJustified` (page 22), `MeasureJustified` (page 30), `CharToPixel` (page 18), and `PixelToChar` (page 34).

txFlag Constants

Specify constants for `txFlags`.

```
enum {
    tfAntiAlias = 1 << 0,
    tfUnicode = 1 << 1
};
```

Constants

`tfAntiAlias`
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`tfUnicode`
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

Discussion
These used to be the pad field after `txFace`.

Truncation Positions

Specify where to truncate a string.

```
typedef  TruncCode;
enum {
    truncEnd = 0,
    truncMiddle = 0x4000,
    smTruncEnd = 0,
    smTruncMiddle = 0x4000
};
```

Constants

`truncEnd`
Truncate at end.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`truncMiddle`
Truncate in middle.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`smTruncEnd`
Truncate at end. This is obsolete.
Available in Mac OS X v10.0 and later.
Declared in `QuickdrawText.h`.

`smTruncMiddle`

Truncate in middle. This is obsolete.

Available in Mac OS X v10.0 and later.

Declared in `QuickdrawText.h`.

Deprecated QuickDraw Text Reference (Not Recommended) Functions

A function identified as deprecated has been superseded and may become unsupported in the future.

Deprecated in Mac OS X v10.4

CharExtra

Specifies, for a color graphics port (`CGrafPort`), the number of pixels by which to widen (or narrow) the glyphs of each nonspace character in a style run. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
void CharExtra (
   Fixed extra
);
```

Parameters

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.

Discussion

The `CharExtra` function sets the value of the `chExtra` field of the color graphics port structure. 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` function 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 functions use the value in this field when an application calls them to measure or draw text. The `CharExtra` function is available only for color graphic ports.

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.

To ensure future compatibility and benefit from any enhancements, always use this function to modify the `chExtra` field, rather than directly change the field value.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In
`QuickdrawText.h`

CharToPixel

Returns the screen pixel width from the left edge of a text segment to the glyph of the character whose byte offset you specify. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
short CharToPixel (
   Ptr textBuf,
   long textLength,
   Fixed slop,
   long offset,
   short direction,
   JustStyleCode styleRunPosition,
   Point numer,
   Point denom
);
```

Parameters

textBuf

A pointer to the beginning of the text segment.

textLength

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.

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 smuch 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.

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.

This parameter specifies the position of the style run on the display line. It 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, and 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.

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.

See "Style Run Position Constants" (page 13) for a list of the constants you can supply.

numer

A point giving the numerator for the horizontal and vertical scaling factors.

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). 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.

denom

A point giving the denominator for the horizontal and vertical scaling factors.

Return Value

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

Discussion

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.

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 functions for this style run.

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.

Special Considerations

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

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`QuickdrawText.h`

CharWidth

Returns the width (horizontel extension), in pixels, of the specified character. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
short CharWidth (
   CharParameter ch
);
```

Parameters

ch

The character whose width is to be measured.

Return Value

The width (horizontel extension), in pixels, of the specified character.

Discussion
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`.

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`.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`QuickdrawText.h`

DisposeStyleRunDirectionUPP

Disposes of a universal procedure pointer (UPP) to a style run direction callback. (Deprecated in Mac OS X v10.4. Use ATSUI to handle style runs.)

```
void DisposeStyleRunDirectionUPP (
   StyleRunDirectionUPP userUPP
);
```

Parameters
userUPP

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.

Declared In
`QuickdrawText.h`

DrawChar

Draws the glyph for a single 1-byte character at the current pen location in the current graphics port. (Deprecated in Mac OS X v10.4. Use ATSUI or Quartz instead.)

```
void DrawChar (
   CharParameter ch
);
```

Parameters

ch

The character code whose glyph is to be drawn.

Discussion

The `DrawChar` function 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.

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.

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`.

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.

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.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`QuickdrawText.h`

DrawJustified

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. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
void DrawJustified (
   Ptr textPtr,
   long textLength,
   Fixed slop,
   JustStyleCode styleRunPosition,
   Point numer,
   Point denom
);
```

Parameters

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.

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.

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. Pass the value assessed for this style run based on the proportion returned for it from `PortionLine`.

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.

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. Be sure to pass the same value that you pass to `PortionLine`.

This parameter specifies the position of the style run on the display line. It 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, and 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.

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.

See "Style Run Position Constants" (page 13) for a list of the constants you can supply.

numer

A point giving the numerator for the horizontal and vertical scaling factors.

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). 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. Be sure to pass the same value that you pass to `PortionLine`.

denom

A point giving the denominator for the horizontal and vertical scaling factors. Be sure to pass the same value that you pass to `PortionLine`.

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

The `DrawJustified` function 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.

The `DrawJustifie` d function 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.

Special Considerations
The `DrawJustified` function may move memory; do not call this function at interrupt time.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`QuickdrawText.h`

DrawString

Draws the text of the specified Pascal string at the pen location in the current graphics port (`GrafPort` or `CGrafPort`). (Deprecated in Mac OS X v10.4. Use ATSUI or Quartz instead.)

```
void DrawString (
   ConstStr255Param s
);
```

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

Discussion
The `DrawString` function 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.

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.

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` (page 17).

For contextual script systems, `DrawString` substitutes the proper ligatures, reversals, and compound characters as needed. Inside a picture definition, `DrawString` can't have a `byteCount` greater than 255.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Related Sample Code
HideMenuBar
Simple DrawSprocket

Declared In
`QuickdrawText.h`

DrawText

Draws the specified text at the current pen location in the current graphics port. (Deprecated in Mac OS X v10.4. Use ATSUI or Quartz instead.)

```
void DrawText (
    const void * textBuf,
    short firstByte,
    short byteCount
);
```

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

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.

Discussion

The `DrawText` function 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.

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.

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.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`QuickdrawText.h`

GetFontInfo

Returns information about the current graphics port's font, taking into account the style and size in which the glyphs are to be drawn. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
void GetFontInfo (
   FontInfo *info
);
```

Parameters

info

Pointer to a font information structure that contains the font measurement information, in integer values.

Discussion

The `GetFontInfo` function 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` function is similar to the Font Manager's `FontMetrics` function, except that the `GetFontInfo` function returns integer values. See `FontInfo` (page 9) for a description of the structure and its fields.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`QuickdrawText.h`

GetFormatOrder

Determines the display order of style runs for a line of text containing multiple style runs with mixed directions. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
void GetFormatOrder (
   FormatOrderPtr ordering,
   short firstFormat,
   short lastFormat,
   Boolean lineRight,
   StyleRunDirectionUPP rlDirProc,
   Ptr dirParam
);
```

Parameters

ordering
A pointer to a format order array, with (`lastFormat - firstFormat + 1`) entries. The function fills the array 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.

Upon completion of the call, the `FormatOrder` (page 10) array contains the numbers identifying the style runs in display order.

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 a callback function that calculates the correct direction, given the style run identifier. The `GetFormatOrder` function calls the application-defined `rlDirProc` function for each identifier from `firstFormat` to `lastFormat`.

This function returns `TRUE` for right-to-left text direction and `FALSE` for left-to-right. Given `dirParam` and a style run identifier, the callback function should be able to determine the style run direction. For more information, see `StyleRunDirectionProcPtr` (page 8).

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 function.

Discussion

The `GetFormatOrder` function 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` function 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.)

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`QuickdrawText.h`

HiliteText

Finds all the characters between two byte offsets in a text segment whose glyphs are to be highlighted. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
void HiliteText (
   Ptr textPtr,
   short textLength,
   short firstOffset,
   short secondOffset,
   OffsetTable offsets
);
```

Parameters

textPtr

A pointer to a buffer that contains the text to be highlighted.

textLength

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.

Discussion
The `HiliteText` function 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` function 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 to be a set of three offset pairs.

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.

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.

Special Considerations

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

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`QuickdrawText.h`

InvokeStyleRunDirectionUPP

Calls your style run direction callback. (Deprecated in Mac OS X v10.4. Use ATSUI to handle style runs.)

```
Boolean InvokeStyleRunDirectionUPP (
   short styleRunIndex,
   void *dirParam,
   StyleRunDirectionUPP userUPP
);
```

Parameters

userUPP

Return Value
A `Boolean` value that indicates whether the callback was invoked successfully.

Discussion
You should not need to use the function `InvokeStyleRunDirectionUPP` as the system calls your style run direction callback for you.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.

Declared In
`QuickdrawText.h`

MeasureJustified

Calculates, for text that is expanded, condensed, or scaled, the onscreen width in pixels from the left edge of the text segment to the glyph of the character. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
void MeasureJustified (
   Ptr textPtr,
   long textLength,
   Fixed slop,
   Ptr charLocs,
   JustStyleCode styleRunPosition,
   Point numer,
   Point denom
);
```

Parameters

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.

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.

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.

This parameter specifies the position of the style run on the display line. It 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, and 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.

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.

See "Style Run Position Constants" (page 13) for a list of the constants you can supply.

numer

A point giving the numerator for the horizontal and vertical scaling factors.

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). 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.

denom

A point giving the denominator for the horizontal and vertical scaling factors.

Discussion

The `MeasureJustified` function is similar to the `MeasureText` function, except that it is used to find the pixel location of a character's glyph in text that is expanded or condensed. The function 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` function 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` function 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 additional information about `MeasureJustified`, contact Developer Technical Support.

The `MeasureJustified` function 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.

Special Considerations

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

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`QuickdrawText.h`

MeasureText

Calculates the width of the character's glyph in pixels from the left edge of the text segment. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
void MeasureText (
   short count,
   const void *textAddr,
   void *charLocs
);
```

Parameters

count

The number of bytes (as opposed to characters) to be measured. 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.

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. 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.

Discussion

Provides an array version of the `TextWidth` function. The `MeasureText` function calculates the onscreen pixel width of the glyph of each character, beginning from the left edge of the text segment. The function 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.

Because this function measures text in the font, style, and size of the current graphics port, you need to call it once for each individual style run in any line of text that contains multiple style runs.

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.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`QuickdrawText.h`

NewStyleRunDirectionUPP

Creates a new universal procedure pointer (UPP) to a style run direction callback. (Deprecated in Mac OS X v10.4. Use ATSUI to handle style runs.)

```
StyleRunDirectionUPP NewStyleRunDirectionUPP (
   StyleRunDirectionProcPtr userRoutine
);
```

Parameters

userRoutine

Return Value

See the description of the `StyleRunDirectionUPP` data type.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Declared In

`QuickdrawText.h`

PixelToChar

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. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
short PixelToChar (
   Ptr textBuf,
   long textLength,
   Fixed slop,
   Fixed pixelWidth,
   Boolean *leadingEdge,
   Fixed *widthRemaining,
   JustStyleCode styleRunPosition,
   Point numer,
   Point denom
);
```

Parameters

textBuf

A pointer to the start of the text segment.

textLength

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.

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.

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

Pointer to 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

Pointer to a location that, 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, the location 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.

This parameter specifies the position of the style run on the display line. It 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, and 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.

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.

See "Style Run Position Constants" (page 13) for a list of the constants you can supply.

numer

A point giving the numerator for the horizontal and vertical scaling factors.

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). 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.

denom

A point giving the denominator for the horizontal and vertical scaling factors.

Return Value
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.

Discussion
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.

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 functions 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 `PixelToCha` r. Remember to restore reordering after you have determined where to break the line.

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.

Special Considerations

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

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`QuickdrawText.h`

PortionLine

Determines the correct proportion of extra space to apply to the specified style run in a line of justified text; that is, how to distribute the total slop value for a line among the style runs on that line. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
Fixed PortionLine (
   Ptr textPtr,
   long textLen,
   JustStyleCode styleRunPosition,
   Point numer,
   Point denom
);
```

Parameters

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.

This parameter specifies the position of the style run on the display line. It 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, and 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.

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.

See "Style Run Position Constants" (page 13) for a list of the constants you can supply.

numer

A point giving the numerator for the horizontal and vertical scaling factors.

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). 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.

denom

A point giving the denominator for the horizontal and vertical scaling factors.

Return Value
A number that represents the portion of the slop to be applied to the style run for which it is called.

Discussion
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.

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 functions for this style run.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`QuickdrawText.h`

SpaceExtra

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. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
void SpaceExtra (
   Fixed extra
);
```

Parameters

extra

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

Discussion
The `SpaceExtra` function sets the value of the extra space (`spExtra`) field in the current graphics port structure. 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` function to set a fixed number of pixels to be added to each space character, then call `DrawText` or `DrawString`.

When you use the justification functions (`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 functions restore the original value.

For a color graphics port (`CGrafPort`), you can use `SpaceExtra` by itself or in conjunction with the `CharExtra` function 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.

To ensure future compatibility and benefit from any enhancements, always use this function to modify the `spExtra` field, rather than directly change the field value.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`QuickdrawText.h`

StandardGlyphs

This obsolete function doesn't do anything in Mac OS X. (Deprecated in Mac OS X v10.4. Use ATSUI to render Unicode text.)

Not recommended.

```
OSStatus StandardGlyphs (
   void *dataStream,
   ByteCount size
);
```

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`QuickdrawText.h`

StdText

Draws text from an arbitrary structure in memory. (Deprecated in Mac OS X v10.4. Use ATSUI or Quartz instead.)

```
void StdText (
   short count,
   const void *textAddr,
   Point numer,
   Point denom
);
```

Parameters

count
The number of bytes of text to draw.

textAddr
A memory structure containing the text to draw.

numer
Scaling numerator.

denom
Scaling denominator.

Discussion
This is QuickDraw's standard low-level function for drawing text. The `StdText` function draws text from the arbitrary structure in memory specified by the `textBuf` parameter, starting from the first byte and continuing for the number of bytes specified in the `byteCount` parameter. The `numer` and `denom` parameters specify the scaling factor: `numer.v` over `denom.v` gives the vertical scaling, and `numer.h` over `denom.h` gives the horizontal scaling factor.

You should only call this low-level function from your customized QuickDraw functions.

Special Considerations

The `StdText` function may move or purge memory blocks in the application heap; do not call this function at interrupt time.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`QuickdrawText.h`

stdtext

Draws text from an arbitrary structure in memory. (Deprecated in Mac OS X v10.4. Use ATSUI or Quartz instead.)

Modified

```
void stdtext (
   short count,
   const void *textAddr,
   const Point *numer,
   const Point *denom
);
```

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`QuickdrawText.h`

StdTxMeas

Measures the width of scaled or unscaled text. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

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

Parameters

byteCount

The number of bytes to be counted.

textAddr

A pointer to the beginning of the text in memory.

numer

Pointer to a point giving the numerator for the horizontal and vertical scaling factors. For this function, `numer` and `denom` are reference parameters. On output, these parameters contain additional scaling to be applied to the text.

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). 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.

denom

Pointer to a point giving the denominator for the horizontal and vertical scaling factors.

info

Pointer to a font information structure that describes the current font.

Return Value

The width of the text stored in memory, beginning with the first character at `textAddr` and continuing for `byteCount` bytes.

Discussion

The `StdTxMeas` function is a QuickDraw bottleneck function that the QuickDraw text-measuring functions 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.

The high-level QuickDraw text functions provide most of the functionality needed to measure and draw text. However, if you need to call `StdTxMeas` directly, you must first check the graphics port `grafProcs` field to determine whether the bottleneck functions have been customized, and if so, you must call the customized function instead of the standard one. The bottleneck functions are always customized for printing.

If the `grafProcs` field contains `NULL`, the standard bottleneck functions have not been customized. If the `grafProcs` field contains a pointer, the standard bottleneck functions have been replaced by customized ones. This pointer (of type `QDProcPtr`) points to a `QDProc` structure, which contains fields that point to the bottleneck function to be used for a specific drawing function. If the standard bottleneck function has been customized, your application needs to use the customized function indicated by the `QDProc` structure field.

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.

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.

The `StdTxMeas` function 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.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`QuickdrawText.h`

StringWidth

Returns the length, in pixels, of the specified text string. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
short StringWidth (
   ConstStr255Param s
);
```

Parameters

s

A Pascal string containing the text to be measured.

Return Value
The length, in pixels, of the specified text string.

Discussion
You should not call `StringWidth` to measure scaled text. Although `StringWidth` takes into account the graphics port structure 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.

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.

Because this function measures text in the font, style, and size of the current graphics port, you need to call it once for each individual style run in any line of text that contains multiple style runs.

The `StringWidth` function works with all script systems.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Related Sample Code
Simple DrawSprocket

Declared In
`QuickdrawText.h`

StyledLineBreak

Returns the proper location to break a line of text, taking into account script and language considerations, making use of tables in the string-manipulation (`'itl2'`) resource in its computations. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
StyledLineBreakCode StyledLineBreak (
   Ptr textPtr,
   SInt32 textLen,
   SInt32 textStart,
   SInt32 textEnd,
   SInt32 flags,
   Fixed *textWidth,
   SInt32 *textOffset
);
```

Parameters

`textPtr`

A pointer to the beginning of a script run on the current line to be broken.

`textLen`

The number of bytes in the script run on the current line to be broken.

`textStart`

A byte offset to the beginning of a style run within the script run.

When used with unformatted text, `textStart` can be 0, and `textEnd` is identical to `textLen`. With styled text, the interval between `textStart` and `textEnd` specifies a style run. The interval between `textPtr` and `textLen` specifies a script run. Note that the style runs in `StyledLineBreak` must be traversed in memory order, not in display order.

`textEnd`

A byte offset to the end of the style run within the script run.

`flags`

Reserved for future expansion; must be 0.

`textWidth`

A pointer to the maximum length of the displayed line in pixels. `StyledLineBreak` decrements this value for its own use. You are responsible for setting it before your first call to `StyledLineBreak` for a line.

`StyledLineBreak` automatically decrements the `textWidth` variable by the width of the style run for use on the next call. You need to set the value of `textWidth` before calling it to process a line.

textOffset

A pointer to the text offset value, which must be nonzero on your first call to `StyledLineBreak` for a line, and zero for subsequent calls to `StyledLineBreak` for that line. This value allows `StyledLineBreak` to differentiate between the first and subsequent calls, which is important when a long word is found (as described below).

The `textOffset` parameter must be nonzero for the first call on a line and zero for each call to the function on the line. This allows `StyledLineBreak` to act differently when a long word is encountered: if the word is in the first style run on the line, `StyledLineBreak` breaks the line on a character boundary within the word; if the word is in a subsequent style run on the line, `StyledLineBreak` breaks the line before the start of the word.

On output, `textOffset` is the count of bytes from `textPtr` to the location in the text string where the line break is to occur. When `StyledLineBreak` finds a line break, it sets the value of `textOffset` to the count of bytes that can be displayed starting at `textPtr`.

When `StyledLineBreak` is called for the second or subsequent style runs within a script run, the `textOffset` value at exit may be less than the `textStart` parameter (that is, it may specify a line break before the current style run).

Return Value

Indicates whether the function broke on a word boundary or a character boundary, or if the width extended beyond the edge of the text. See "Style Line Break Values" (page 12) for a list of the constants that can be returned.

Discussion

The function `StyledLineBreak` breaks the line on a word boundary if possible and allows for multiscript runs and style runs on a single line.

Use the `StyledLineBreak` function when you are laying out lines in an environment that may include text from multiple scripts. To use this function, you need to understand how QuickDraw draws text.

You can only use the `StyledLineBreak` function when you have organized your text in script runs and style runs within each script run. This type of text organization is used by most text-processing applications that allow for multiscript text. Use this function when you are displaying text in a screen area to determine the best place to break each displayed line.

What you do is iterate through your text, a script run at a time starting from the first character past the end of the previous line. Use `StyledLineBreak` to check each style run in the script run (in memory order) until the function determines that it has arrived at a line break. As you loop through each style run, before calling `StyledLineBreak`, you must set the text values in the graphics port structure that are used by QuickDraw to measure the text. These include the font, font size, and font style of the style run.

If the current style run is included in a contiguous sequence of other style runs of the same script, then `textPtr` should point to the start of the first style run of the same script on the line, and `textLen` should include the last style run of the same script on the line. This is because word boundaries can extend across style runs, but not across script runs.

Although the offsets are in long integer values and the widths are in fixed values for future extensions, in the current version the long integer values are restricted to the integer range, and only the integer portion of the widths is used.

`StyledLineBreak` always chooses a line break for the last style run on the line in memory order as if all whitespace in that style run would be stripped. The `VisibleLength` function, which is a QuickDraw function used to eliminate trailing spaces from a style run before drawing it, can be called for the style run that is at

the display end of a line. This leads to a potential conflict when both functions are used with mixed-directional text: if the end of a line in memory order actually occurs in the middle of the displayed line, `StyledLineBreak` assumes that the whitespace is stripped from that run, but `VisibleLength` does not strip the characters.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`QuickdrawText.h`

TextFace

Sets the style of the font in which the text is to be drawn in the current graphics port. (Deprecated in Mac OS X v10.4. Use ATSUI or Quartz instead.)

```
void TextFace (
   StyleParameter face
);
```

Parameters

face

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

Discussion
The `TextFace` function 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.

To ensure future compatibility and benefit from any enhancements, always use this function to modify the `txFace` field, rather than directly change the field value.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`QuickdrawText.h`

TextFont

Sets the font of the current graphics port in which the text is to be rendered. (Deprecated in Mac OS X v10.4. Use ATSUI or Quartz instead.)

```
void TextFont (
   short font
);
```

Parameters

font

The font family ID. 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.

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.

Discussion

The `TextFont` function sets the value of the graphics port text font (`txFont`) field. To ensure future compatibility and benefit from any enhancements, always use this function to modify the `txFont` field, rather than directly change the field value.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Related Sample Code

Simple DrawSprocket

Declared In

`QuickdrawText.h`

TextMode

Sets the transfer mode for drawing text in the current graphics port. (Deprecated in Mac OS X v10.4. Use ATSUI or Quartz instead.)

```
void TextMode (
   short mode
);
```

Parameters

mode

The transfer mode to be used to draw the text.

Discussion

The `TextMode` function 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. Other transfer modes are `grayishTextOr`, `transparent` mode, and text mask mode.

To ensure future compatibility and benefit from any enhancements, always use this function to modify the `txMode` field, rather than directly change the field value.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`QuickdrawText.h`

TextSize

Sets the font size for text drawn in the current graphics port to the specified number of points. (Deprecated in Mac OS X v10.4. Use ATSUI or Quartz instead.)

```
void TextSize (
   short size
);
```

Parameters

size

The font size in points (0 to 32,767). The initial setting is 0, which specifies that the font size of the system font (normally 12 points) is to be used.

Discussion
The `TextSize` function sets the font size in the text size (`txSize`) field of the current graphics port structure. To ensure future compatibility and benefit from any enhancements, always use this function to modify the `txSize` field, rather than directly change the field value.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Related Sample Code
Simple DrawSprocket

Declared In
`QuickdrawText.h`

TextWidth

Returns the length, in pixels, of the specified text. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
short TextWidth (
   const void *textBuf,
   short firstByte,
   short byteCount
);
```

Parameters

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.

Return Value

The length, in pixels, of the specified text.

Discussion

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`.

Because this function measures text in the font, style, and size of the current graphics port, you need to call it once for each individual style run in any line of text that contains multiple style runs.

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

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 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.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`QuickdrawText.h`

TruncString

Ensures that a Pascal string fits into the specified pixel width, by truncating the string as necessary. This function makes use of the current script and font. (Deprecated in Mac OS X v10.4. Use CFString instead.)

```
short TruncString (
   short width,
   Str255 theString,
   TruncCode truncWhere
);
```

Parameters

width

The number of pixels in which the string must be displayed in the current script and font.

theString

The Pascal string to be displayed. On output, contains a version of the string that has been truncated (if necessary) to fit in the number of pixels specified by `width`.

truncWhere

A constant that indicates where the string should be truncated. If you supply the `truncEnd` value, characters are truncated off the end of the string. If you supply the `truncMiddle` value, characters are truncated from the middle of the string; this is useful when displaying pathnames.

See "Truncation Positions" (page 15) for a list of the constants you can supply.

Discussion

The `TruncString` function ensures that a Pascal string fits into the pixel width specified by the `width` parameter by modifying the string, if necessary, through truncation. `TruncString` uses the font script to determine how to perform truncation. If truncation occurs, `TruncString` inserts a truncation indicator, which is the ellipsis (…) in the Roman script system. You can specify which token to use for indicating truncation as the `tokenEllipsis` token type in the untoken table of a tokens (`'itl4'`) resource.

To determine the width of a string in the current font and script, use the QuickDraw `StringWidth` function.

Special Considerations

`TruncString` may move memory; your application should not call this function at interrupt time.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`QuickdrawText.h`

TruncText

Ensures that a text string fits into the specified pixel width, by truncating the string as necessary. This function makes use of the current script and font. (Deprecated in Mac OS X v10.4. Use CFString instead.)

```
short TruncText (
   short width,
   Ptr textPtr,
   short *length,
   TruncCode truncWhere
);
```

Parameters

width

The number of pixels in which the text string must be displayed in the current script and font.

textPtr

A pointer to the text string to be truncated. The text string can be up to 32 KB long.

length

On input, a pointer to a value containing the length, in bytes, of the text string to be truncated. On output, this value is updated to reflect the length of the (possibly) truncated text.

truncWhere

A constant that indicates where the text string should be truncated. You must set this parameter to one of the constants `truncEnd` or `truncMiddle`. If you supply the `truncEnd` value, characters are truncated off the end of the string. If you supply the `truncMiddle` value, characters are truncated from the middle of the string; this is useful when displaying pathnames.

See "Truncation Positions" (page 15) for a list of the constants you can supply.

Discussion

You can use the `TruncText` function to ensure that a string defined by a pointer and a byte length fits into the specified pixel width, by truncating the string in a manner dependent on the font script.

`TruncText` uses the font script to determine how to perform truncation. If truncation occurs, `TruncText` inserts a truncation indicator which is the ellipsis (…) in the Roman script system. You can specify which token to use for indicating truncation as the `tokenEllipsis` token type in the untoken table of a tokens resource.

To determine the width of a string in the current font and script, use the QuickDraw `StringWidth` function.

Special Considerations

`TruncText` may move memory; your application should not call this function at interrupt time.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`QuickdrawText.h`

VisibleLength

Calculates the length, in bytes, of a given text segment, excluding trailing white space. (Deprecated in Mac OS X v10.4. Use ATSUI instead.)

```
long VisibleLength (
   Ptr textPtr,
   long textLength
);
```

Parameters

textPtr

A pointer to a text string.

textLength

The number of bytes in the text segment.

Return Value

The length, in bytes, of a given text segment, excluding trailing white space.

Discussion

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 functions 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.

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.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`QuickdrawText.h`

Document Revision History

This table describes the changes to *QuickDraw Text Reference*.

Date	Notes
2006-07-13	Made formatting changes.
2006-07-24	Added information on deprecated functions.
2003-04-01	Added abstracts for two functions: `stdtext` (page 41) and `StandardGlyphs` (page 40).
2002-12-03	Updated formatting.

Document Revision History

Index

P

R

S

T

V